MDG Tools (V1.0) Developer's Manual

Zijian Zhou

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

Multiway Decision Graphs (MDGs) [1] have been proposed as a new kind of decision graphs to represent Register-Transfer (RT) level hardware designs. MDG-based verification techniques have been developed for combinational and sequential designs. In this document, we describe the structure of the MDG tools that we implemented and the user programming interface for developing new applications. The important implementation issues (such as structure sharing and result tables) are also addressed.

MDG tools include an MDG package that contains basic MDG operators, a reachability analysis procedure based on the abstract implicit enumeration and an application layer that implements the verification algorithms. As a prototype, MDG tools are implemented in Quintus Prolog Version 3.2.

The rest of the manual is organized as follows. In Section 2, we show the structure of MDG tools. In Section 3, we describe data structures and structure sharing techniques for graphs and terms. Section 4 -- 7 explain the programming interface of the tools. Section 8 discusses some programming tips. Appendix A gives a complete list of files in MDG tools. The subsections marked * are implementation details that are only for interested readers.

Font. The Prolog predicates and examples will be given in 11 points Helvetica font.

Further references. This manual is not self-contained. Readers should refer to the references, especially [1] and [7] for terminology and techniques about Multiway Decision Graphs. Readers are supposed to have basic Prolog knowledge. For those who only want to use the existing applications in MDG tools, they are recommended to read *MDG Tools (V1.0) User's Manual*.

2. The Structure of MDG Tools

MDG tools (V1.0) are composed of five Prolog modules which separate the different functions of code. A Prolog module encapsulates the predicates defined inside this module. Only the predicates declared "public" in the module interface are accessible by the other modules. The Prolog modules are flat, i.e., there is no model hierarchy. However, the current MDG tools logically organize the modules in three layers as shown in Figure 1.

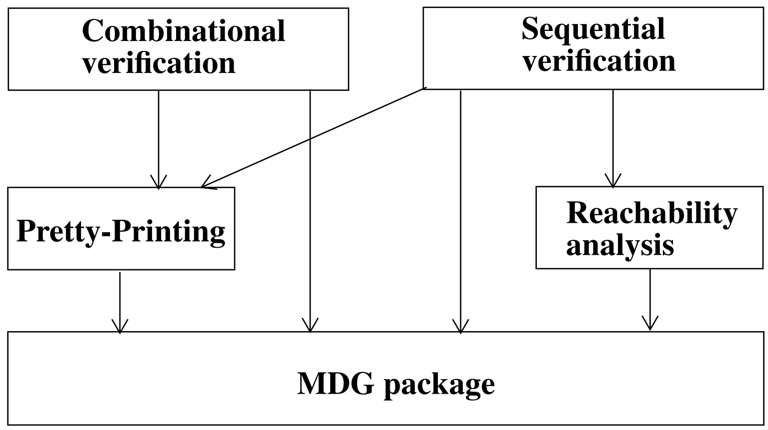

Figure 1. The structure of MDG Tools

The MDG package module is at the lowest layer. It provides a set of MDG operators and a number of utilities, which are the basic building blocks for MDG-based techniques. In the middle layer, there are two modules: (i) the printing module provides various pretty-printing procedures; (ii) the reachability analysis module implements the reachability analysis algorithm based on abstract implicit enumeration. It also contains a counterexample facility which can provide explanations (a sequence of input-state pairs leading to the faulty behavior) when the invariant being verified is violated. Currently, there are two application modules. The combinational verification module implements the equivalence checking of input-output relations of two combinational circuits using the canonicity property of MDGs. The sequential verification module implements (i) safety property checking and (ii) equivalence checking of two state machines.

3. Data Structures and Structure Sharing

In this section we describe the term structure and the graph structure used in our implementation. We also explain a structure sharing technique using Quintus Prolog library “logarr.pl” (logarithmic array).

3.1 Term Structure

A term in an MDG is defined as follows:

- A (individual or generic) constant is a term.
- A (concrete or abstract) variable is a term.
- If $a_1,...,a_n$ are terms and f is a function symbol, then $f(a_1,...,a_n)$ is a term, which is also referred to as a *compound term*.

In MDGs, sometimes, there may be many compound terms which also can be very large. Thus it is important to have compound terms share some common parts. In order to achieve term sharing, we assign a unique identifier (ID) to each term. The ID for a constant or a variable is the constant or the variable itself. The ID for a compound term is a number obtained by hashing the function symbol and the IDs of its subterms using the built-in hash function *hash_term/2*. Below we explain the different internal representations for these three kinds of terms:

- Constants:

  A constant is represented by a Prolog atom. If a constant is a character string, then the atom is the string itself. Or if a constant is an integer, the integer is converted into an atom (using built-in predicate number_atom/2 for converting).

  Example 1: Constant zero. Internal representation: 'zero'.

  Example 2: Constant number 123. Internal representation: '123'.

- Variables:

  A variable *Var* is represented by *v(Var)* where *Var* is a Prolog atom. In the case *Var* is a fresh variable, it is represented as *v(fresh(Var, Number)),* where *Var* is a Prolog atom and *Number* is a natural number.

  Example 1: variable pc. Internal representation: v(pc).

  Example 2: fresh variable x#1. Internal representation: v(fresh(x, 1)).

- Compound terms:

  A compound term is represented by *term(Func_symbol, TermId, SubTerms)* where *Func_symbol* is a function symbol, *TermId* is an integer which is the unique identifier of the term, and *SubTerms* is a list of subterms.

  Example 1: term add(x1, x2). Internal representation: term(add, ID, [v(x1), v(x2)]) where ID is a number.

  Example 2: term f(0,y,g(x#3,c,x,y)). Internal representation: term(f, ID1, ['0',v(y), term(g, ID2, [fresh(x, 3)), c,v(x), v(y)])]) where ID1 and ID2 are numbers (ID1 ≠ ID2), c is a generic constant, x#3 is a fresh variable.

3.2 Graph structure

- Leaves:

  For well-formed MDGs, leaves are represented by 'T' (True) or 'F' (False). For non-well-formed MDGs, they may have formulas as leaves. In the current package, we only allow well-formed MDGs.

- Internal node:

  An internal node is represented by the following structure:

  graph(*TopSymbolOrder, NodeKind, NodeLabel, Id, Edges, SubGraphs, SecVars*).

  TopSymbolOrder is the custom order number for the top symbol of the node label.

  NodeKind has one of the following forms according to different kind of node labels:

  xterm(*Sort*): the node is a cross-term and the target sort of the cross-operator is *Sort*.

  concvar(*Sort*): the node is a concrete variable which is of sort *Sort*.

absvar: the node is an abstract variable.

NodeLabel is the term which is the label of the node.

ID is the unique identifier of the graph.

Edges and *SubGraphs* are two corresponding lists, representing root edges and the immediate subgraphs, respectively.

SecVars is a sorted list that contains all the secondary variables in the graph.

3.3 Structure Sharing

Like OBDDs, graph sharing is an important feature of MDGs. Besides graph sharing, MDGs also need term sharing because there may be many compound terms which also can be very large.

3.3.1 Graph arrays and term arrays

In Prolog, if we assert the graphs and terms into the database, the Prolog system essentially generates copies of the structures and hence no structure sharing is possible. On the other hand, a variable in Prolog is a pointer (managed by the Prolog system) which points to a structure on the heap. With the unification of variables, i.e., two variables are bound to the same address, we can achieve structure sharing. We use two *logarithmic arrays* to index the graphs and terms, respectively. Logarithmic array is provided by the Quintus Prolog library "logarr.pl". It is an array with logarithmic accessing time complexity. Each entry of the array has an index and a pointer. The index is used to access the entry and it is also used as the unique identifier (ID) of the structure, while the pointer points to the structure on the heap. The following diagram shows the structure sharing scenario.

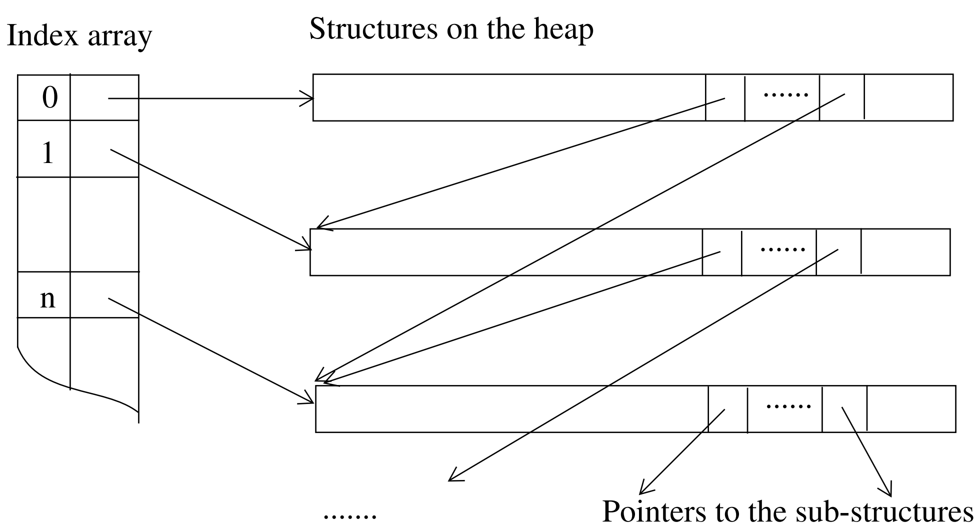

Figure 2. Structure sharing

The pointers in the index arrays are Prolog variables. They are managed by the Prolog system and cannot be manipulated by the programmer. Hence the index arrays have to be passed as incoming arguments to every predicates where graph or term processing are required. The updated arrays are returned as outcoming arguments. We use an integer *IAP* as an index array pointer to remember the next available entry in the array. We denote a pointer-array pair as (*IAP, Arr*) where *Arr* is an index array and *IAP* is a pointer. As a convention in the rest of the manual, we use, as parameters in predicates where necessary, *G1* and *E1* to denote the pointer-array pairs for current graphs and terms, and *G, E* to denote the corresponding updated index-array pairs.

The MDG algorithms use a common graph reduction table and a common term reduction table, all implemented as hash tables. They are asserted into the database when created. The utilities assemble/7 and assemble_term/7 in the MDG package are the basic procedures to create new graphs and terms.

3.3.2 Graph Reduction Table*

Each MDG has a unique ID identified by its root label, its root edges and immediate subgraphs. The ID is the entry to the graph array. The graph reduction table has an entry for each MDG that maps a *HashKey*, the sequence of label IDs of the root edges and the sequence of IDs of the immediate subgraphs to the ID of the result graph. Looking up the reduction table is accelerated by *HashKey* which is obtained by using build-in hash function:

hash_term(h(RootlabelID, EdgeIDs, SubGraphIDs), HashKey).

The graph reduction table g**ht/5** has the following structure:

ght(HashKey, RootLabelID, EdgeIDs, SubGraphIDs, ResultID).

When we are going to construct a graph with a given root label, root edge labels and immediate subgraphs, we will first look up on the reduction table to check if such a graph already exists. If an entry is found, then we look up the graph array using ResultID and let the graph point to the existing one through unification of pointers (or "assignment" like in procedural languages). If no entry is found, a new graph structure is created and a new entry is appended in the graph array. Also a new entry is asserted into the graph reduction table.

3.3.3 Term Reduction Table*

Each term also has a unique identifier ID. The ID of a constant or a variable is the constant or the variable itself. For a compound term, its ID is identified by its function symbol and its subterms. The term reduction table has an entry for each compound term that maps the function symbol of the term, the sequence of IDs of subterms to the term ID. The term reduction table **eht/4** has the following structure:

eht(HashKey, FuncSymbol, SubTermIDs, ResultID).

HashKey is obtained by using build-in hash function:

hash_term(h(FuncSymbol, SubTermIDs), HashKey).

Similarly, to construct a new term, we need to look up the term reduction table. If an entry is found, then we look up the term array for term sharing. Otherwise, a new term structure is created and a new entry is appended in the term array. Also a new entry is asserted into the term reduction table.

4. MDG Package Module

The MDG package provides a number of MDG operators and various utilities. It is packed as a Prolog module "mdgpac". In the export list of the module interface, we identify two different kind of public predicates, data and procedures. In this section, we explain the interface for each public predicate. We will also explain the result tables used in the implementations for avoiding repetitive computations.

4.1 Data Interface

Four kinds of data are required by the procedures in the MDG package:

1. Concrete sort declaration: csort_sorted(*Sort, SortedEnums*).

   Sort is a concrete sort and *SortedEnums* is its enumeration set with all the individual constants in standard symbol order. For a sort definition, there should exist only one entry.

2. Custom order declaration: order(*Label, Order*).

   For the variables and cross-operators that participate in the custom symbol ordering, they must be declared using this predicate. *Order* is a unique positive integer assigned to *Label* (a variable or a cross-op). For a label, there should exist only one entry.

3. Individual constant declaration: ind_const(*Ind_const*).

   Every individual constant appeared in the concrete sort declarations must be declared using this predicates.

4. Xterm rewriting switch: apply_xtrr(*Flag*).

   If *Flag*=yes/no, then the xterm rewriting procedure is switched on/off.

If there are user defined rewrite rules, they also have to be transformed into the following internal representations before calling the rewriting operator.

- General conditional rewrite rule: rr(*TopSymbol*, RC_S, *LHS*, *RHS*).

  xterm rewrite rule: xtrr(*TopSymbol*, RC_S, *LHS*, *RHS*).

  TopSymbol is the top symbol of *LHS*. *LHS* and *RHS* are terms (in their internal representation). RC_S is a list of pairs (RC_i, RV_i) or (P_i), where RC_i is a cross-term in its internal representation and RV_i is an individual constant, P_i is a term *not* i its internal representation.

4.2 Basic MDG Operators

4.2.1 Disjunction: disj(*Ps, Q, G1, E1, G, E*).

Mode[1]**:** disj(+, -, +, +, -, -).

Argument: *Ps* is a list of MDGs. *Q* is the result MDG.

Function: *Q* is the disjunction of *Ps*.

Data required: csort_sorted/2.

File: homeMDG/programs/mdgpac1.0/disj.pl.

4.2.2 Relational product: relp(*Ps, Us, Ren, Q, G1, E1, G, E*).

Mode: relp(+, +, +, -, +, +, -, -).

Arguments: *Ps* is a list of MDGs. *Us* is a ordered list of symbol orders. *Ren* is the renaming substitution. *Ren* = *ren(Name, Largest_VarOrd, RenSs),* where *Name* identifies a particular renaming substitution, *RenSs* is the renaming substitution given as a list of triples (*Label, NewLabel, NewOrder*), and *Largest_VarOrd* is the number of the largest *OldOrder* in the list. The triple defines a renaming substitution from *Label* to *NewLabel,* at the same time provides the order of the new label. If RenSs=[], then *Name, Largest_VarOrd* are don't-care. *Q* is the result graph.

Function: *Q* is the conjunction of *Ps* with existential quantification of variables in *Us*, and if a node label is in the *RenSs*, then replace it with the new label.

Data required: apply_xtrr/1, csort_sorted/2, ind_const/1, xtrr/3.

File: homeMDG/programs/mdgpac1.0/relp1.pl, relp2.pl, relp3.pl.

Remark: Relational product is the combination of conjunction, existential quantification and renaming substitution.

4.2.3 Conjunction: conj(*Ps, Q, G1, E1, G, E*).

Mode: conj(+, -, +, +, -, -).

Arguments: *Ps* is a list of MDGs. *Q* is the result MDG.

Function: *Q* is the conjunction of *Ps*.

Data required: apply_xtrr/1, csort_sorted/2, ind_const/1, xtrr/3.

File: homeMDG/programs/mdgpac1.0/relp1.pl, relp2.pl, relp3.pl.

Remark: this operator is equivalent to relp(Ps, [], ren(_, _, []), Q, G1, E1, G, E).

1. For a argument of a predicate, + means the argument is an incoming parameter and - an outcoming one.

4.2.4 Existential quantification: exquan(*P, Us, Q, G1, E1, G, E*).

Mode: exquan(+, +, -, +, +, -, -).

Arguments: *P* is an MDG. *Us* is a list of variables. *Q* is the result MDG.

Function: For MDG *P*, the nodes which are the variables in *Us* are existentially abstracted.

Data required: apply_xtrr/1, csort_sorted/2, ind_sonst/1, xtrr/3.

File: homeMDG/programs/mdgpac1.0/relp1.pl, relp2.pl, relp3.pl.

Remark: this operator is equivalent to relp([P], Us, ren(_, _, []), Q, G1, E1, G, E).

4.2.5 Renaming substitution: renaming(*P, Ren, Q, G1, E1, G, E*).

Mode: renaming(+, +, -, +, +, -, -).

Arguments: *P* is an MDG. *Ren* is the renaming substitution (the same as in Section 4.2.2). *Q* is the result MDG.

Function: *Q* is the result of renaming substitution of *P*.

Data required: apply_xtrr/1, csort_sorted/2, ind_const/1, xtrr/3.

File: homeMDG/programs/mdgpac1.0/relp1.pl, relp2.pl, relp3.pl.

Remark: this operator is equivalent to relp([P], [], Ren, Q, G1, E1, G, E).

4.2.6 Pruning-by-subsumption: pbys(*P, Q, R, G1, E1, G, E*).

Mode: pbys(+, +, -, +, +, -, -).

Arguments: *P* is the graph to be pruned by *Q*. *R* is the result graph.

Function: *R* is obtained by removing the paths in *P* which are subsumed by *Q*.

Data required: csort_sorted/2.

File: homeMDG/programs/mdgpac1.0/pbys.pl.

4.3 More MDG Operators

4.3.1 Difference-by-subsumption: dbys(*P, Q, R, G1, E1, G, E*).

Mode: dbys(+, +, -, +, +, -, -).

Arguments: *P* is the graph to be pruned. *Q* is an MDG whose secondary variables are considered to be implicitly quantified. *R* is the (possibly not well-formed) result graph.

Function: Take an arbitrary path of *P*, say *P'*. *R* is obtained by removing the cases in *P'* which are subsumed by *Q*.

Data required: csort_sorted/2.

File: homeMDG/programs/mdgpac1.0/dbys.pl.

4.3.2 Relational product with substitution: relp(*Ps, S, Us, Ren, Q, G1, E1, G, E*).

Mode: relp(+, +, +, -, +, +, -, -).

Arguments: *Ps* is a list of MDGs whose terms may not be necessarily concretely reduced. *S* is a substitution given as a list of pairs (*V, Term*). *Us*, *Ren* are the same as explained in Section 4.2.2. *Q* is the result graph.

Function: *Q* is the conjunction of *Ps* with existential quantification of variables in *Us*, if a primary variable is in the domain of the renaming substitution, then replace it with the new label, and if a secondary variable is in the domain of *S*, then replace it with the new term given by *S*.

Data required: apply_xtrr/1, csort_sorted/2, ind_const/1, order/2, xtrr/3.

File: homeMDG/programs/mdgpac1.0/relp1.pl, relp2.pl, relp3.pl.

Remark: Relational product with substitution is the combination of conjunction, existential quantification, substitution and renaming. It can be used to transform non-well formed MDGs (e.g., MDGs containing non-concretely-reduced terms) into well-formed MDGs with a proper set of *S*.

4.3.3 Abstraction of redundant cross-terms: abs_xterm(*P, R, G1, E1, G, E*).

Mode: abs_xterm(+, -, +, +, -, -).

Arguments: *P* is an MDG to be processed. *R* is the result MDG.

Function: *R* is the MDG resulting from abstracting redundant cross-terms in *P*.

Data required: csort_sorted/2.

File: homeMDG/programs/mdgpac1.0/abs_xterm.pl.

4.3.4 Conditional term rewriting (1): rew(*P, Q, G1, E1, G, E*).

Mode: rew(+, -, +, +, -, -).

Arguments: *P* is an MDG. *Q* is the result MDG.

Function: Rewrite MDG *P* under the conditional and the unconditional rewriting systems which are implicitly given as sets of rules described by xtrr/4 and rr/4.

Data required: csort_sorted/2, ind_const/1, order/2, xtrr/4, rr/4.

File: homeMDG/programs/mdgpac1.0/rew1.pl, rew2.pl.

4.3.5 Conditional term rewriting (2): rews(*Ps, Qs, G1, E1, G, E*).

Mode: rews(+, -, +, +, -, -).

Arguments: *Ps* is a list of MDGs. *Qs* is the result MDGs.

Function: Rewrite each MDG in *Ps* under the conditional and the unconditional rewriting systems which are implicitly given as sets of rules described by ucrr/3 and rr/4.

Data required: csort_sorted/2, ind_const/1, order/2, xtrr/4, rr/4.

File: homeMDG/programs/mdgpac1.0/rew1.pl, rew2.pl.

4.3.6 Xterm rewriting (1): xtrew(*Term, RewTerm, E1, E*).

Mode: xtrew(+, -, +, -).

Arguments. *Term* is a cross-term. *RewTerm* is the result term after rewriting.

Function: Rewrite *Term* under the conditional rewriting system which is implicitly given as a set of rules described by xtrr/4.

Data required: csort_sorted/2, ind_const/1, order/2, xtrr/4.

File: homeMDG/programs/mdgpac1.0/rew1.pl, rew2.pl.

4.3.7 Xterm rewriting (2): xtrews(*Terms, RewTerms, E1, E*).

Mode: xtrews(+, -, +, -).

Arguments. *Terms* is a list of cross-term. *RewTerms* is the result terms after rewriting.

Function: Rewrite each term in the list *Terms* under the conditional rewriting system which is implicitly given as a set of rules described by xtrr/4.

Data required: csort_sorted/2, ind_const/1, order/2, xtrr/4.

File: homeMDG/programs/mdgpac1.0/util_mdgpac.pl.

4.3.8 generalization: generalize(*P, P_id, Q, G1, E1, G, E*).

Mode: generalize(+, +, -, +, +, -, -).

Arguments. *P* is an MDG and *P_id* is its graph ID. *Q* is the result MDG.

Function: Abstract (eliminate) all the cross-terms in *P*.

Data required: non.

File: homeMDG/programs/mdgpac1.0/gene.pl

4.4 MDG Utilities

The utilities described in this section are all in homeMDG/programs/mdgpac1.0/util_mdgpac.pl. They could be helpful for the development of new applications.

4.4.1 Clear result tables: clear_res_table.

Function: retract all the entries for all the result tables used in the MDG package.

File: homeMDG/programs/mdgpac1.0/mdgpac.pl.

Remark: Before using the MDG package, the following operation “clear_res_table/0” should be applied to reset all the result tables to empty.

4.4.2 Graph assembling: assemble(*RootInfo, Edges, SubGs, Method, R, G1, E, G, E*).

Mode: assemble(+, +, +, +, -, +, +, -, -).

Arguments: *RootInfo* is a structure ‘rootinfo(*O, K, L*)’ where *O* is the order number of the root, *K* is the root kind and *L* is the root label. *Edges* and *SubGs* are the root edges and corresponding immediate subgraphs. *R* is the result graph. *Method* tells how to compute the secondary variables of *R*. It can be set to ‘noop’ or ‘union(*VSs*)’ where *VSs* is a list of variable lists. Note that *E* will not be affected in the predicate.

Function: Build an MDG according to the given root information and the root edges and immediate subgraphs. It first checks if the result graph already exists by looking up the reduction table ght/5. If so, *R* points to that graph (thus achieving graph sharing). Otherwise a new graph structure is created and entered in the graph array. Also a new entry in ht/5 is created. If *Method*=union(*VSs*), the secondary variable of *R* will simply be the ordered union of *VSs*. If *Method*=noop, then the secondary variables will be obtained by searching the graph.

Data required: none.

4.4.3 Term assembling: assemble_term(*Symbol, Terms, RTerm, E1, E*).

Mode: assemble_term(+, +, -, +, -).

Arguments: *Symbol* is a symbol. *Terms* are a list of terms. *RTerm* is the result term which has *Symbol* as its top symbol and *Terms* as its arguments.

Function: Build a term according to *Symbol* and *Terms*. First, it checks if the result term already exists by looking up the reduction table eht/4. If so, *RTerm* points to that term and thus achieves term sharing. Otherwise a new term is created.

Data required: none.

4.4.4 Retrieve graph ID (1) : get_graphId(*P, Id*).

Mode: get_graphId(+, -).

Arguments: *P* is an MDG. *Id* is the ID of the graph.

Function: Retrieve the graph ID.

Data required: none.

4.4.5 Retrieve graph ID (2) : get_graphIds(*Ps, Ids*).

Mode: get_graphIds(+, -).

Arguments: *Ps* is a list of MDGs. *Ids* is the list of IDs with respect to *Ps*.

Function: Retrieve graph IDs for a list of graphs.

Data required: none.

4.4.6 Retrieve term ID (1) : get_termId(*Term, Id*).

Mode: get_termId(+, -).

Arguments: *Term* is a term. *Id* is the ID of the term.

Function: Retrieve the term ID.

Data required: none.

4.4.7 Retrieve term ID (2) : get_termIds(*Terms, Ids*).

Mode: get_termIds(+, -).

Arguments: *Terms* is a list of terms. *Ids* is the list of IDs with respect to *Terms*.

Function: Retrieve term ID for a list of terms.

Data required: none.

4.4.8 Retrieve graph from the graph array : get_mdg_from_logarr(*Id, G, P*).

Mode: get_mdg_from_logarr(+, +, -).

Arguments: *Id* is an ID. *P* is the graph corresponding to *Id*. *G* is the graph array.

Function: Retrieve a graph *P* from the graph array according to *Id*.

Data required: none.

4.4.9 Retrieve term from term array (1): get_term_from_logarr(*Id, E, Term*).

Mode: get_term_from_logarr(+, +, -).

Arguments: *Id* is an ID. *Term* is the term corresponding to *Id*. *E* is the term array.

Function: Retrieve a term *Term* from the term array according to *Id*.

Data required: none.

4.4.10 Retrieve term from term array (2): get_terms_from_logarr(*Ids, E, Terms*).

Mode: get_terms_from_logarr(+, +, -).

Arguments: *Ids* is a list of IDs. *Terms* is a list of terms corresponding to *Ids*. *E* is the term array.

Function: Retrieve a list of terms from the term array according to *Ids*.

Data required: none.

4.4.11 Transform number to atom (1): number_to_atom(*Number, Atom*).

Mode: number_to_atom(+, -).

Arguments: *Number* is a number. *Atom* is an Prolog atom.

Function: Transform a number into an atom.

Data required: none.

4.4.12 Transform number to atom (2): numbers_to_atoms(*Numbers, Atoms*).

Mode: numbers_to_atoms(+, -).

Arguments: *Numbers* is a list of numbers. *Atoms* is a list of Prolog atoms.

Function: Transform a list of numbers into atoms.

Data required: none.

4.4.13 MDG statistics: size(*P, N, C, E*).

Mode: size(+, -, -, -).

Arguments: *P* is an MDG. *N*, *C* and *E* are the number of nodes, the number of cross-terms and the number of edges in *P*, respectively.

Function: Count the number of nodes, cross-terms and edges in an MDG.

Data required: none.

4.4.14 MDG statistics: sizes(*Ps, N, C, E*).

Mode: sizes(+, -, -, -).

Arguments: *Ps* is a list of MDGs. *N*, *C* and *E* are the number of nodes, the number of cross-terms and the number of edges for all the MDGs in *Ps*, respectively.

Function: Count the number of nodes, cross-terms and edges in a list of MDGs.

Data required: none.

4.5 Result Tables*

Result tables are used to remember the results of previous computations. Before performing an operation, it is checked whether it has already been computed by looking up a result table. If so, we simply retrieve the result without further computation. Otherwise the normal operation will be performed and its result will be asserted as a new entry in the result table.

In the MDG package, we use a number of result tables for different operations. All the result tables have three parts: a hash term *Key* for indexing, the arguments of the operation and the result. In the case that the arguments and the result are MDGs or Terms, we use their Ids instead of the graphs or terms themselves (thus to avoid copying large structures to the database). The *Key* is computed from the arguments using the built-in predicate hash_term/2.

When we start a specific operation, we have two choices. We can either reset the result table or keep it. The former method means to forget all the previous computations. It could save memory space for storing the table. The later method, on the other hand, can reuse the previous computation as much as possible.

The result tables used in the MDG package are listed below :

- **Result table for abstracting cross-terms:** abs1_xterm_rt *(Key, P_id, V, Q_id)* **.**

  P_id is the ID of the graph to be processed. *V* is a list of variables. *Q_id* is the ID of the result graph.

- **Result table for disjunction:** disj_rt(*Key, P_ids, Q_id*) **.**

  P_ids is a list of IDs for the graphs to be disjunct. *Q_id* is the ID of the result graph.

- **Result table for generalization:** gene_rt(*P_id, Q_id*).

  P_id and *Q_id* are the graph IDs of the input graph and the result graph.

- **Result table for relational product:** relp1_rt *(Key, P_ids, S, Us, RenName, Q_id)* **.**

  P_ids is a list of IDs of the input MDGs, *S* is a substitution set, *Us* is a list of variables, *RenName* is the name of the renaming substitution. *Q_id* is the Id of the result graph.

- **Result table for pruning-by-subsumption:** pbys1_rt *(Key, P_id, Q_ids, Ss, Q_id)* **.**

  P_id is the ID of the graph to be pruned. *Q_ids* is a list of graph IDs. *Ss* is a list of substitution sets. *Q_id* is the ID of the result graph.

- **Result table for term rewriting :** ctrew_rt(*Key, T_Id, Cids, V, T_Id1*) **.**

  T_Id is the ID of the term to be rewritten. *T_id1* is the ID of the result term. *Cids* is a list of pairs (*Cid, V*) where *Cid* is an ID of a cross-term and *V* is an individual constant. *Cids* represent the constraint set for the term. *V* is an argument which identifies the rewriting system. Currently it is not used.

- **Result table for cross-term rewriting :** xtrew_rt(*X_Id, X_Id1*) **.**

  X_Id is the term ID of a cross-term *X*. *X_Id1* is term ID of the normal form of *X*.

- **Result table for MDG rewriting :** rew1_rt(*Key, P_id, Cids, TermIds, QId_TermIds_s*).

  P_Id is the ID of the graph to be rewritten. *Cids*, *TermIds*, *QId_TermIds_s* are the corresponding lists for *Cs, Terms* and *Q_Terms_s* in predicate rew1/13 where graphs are replaced by their IDs.

- **Result table for substitution at top:** subst_rt *(Key, P_Id, S, Q_Id)* **.**

  P_Id is the ID of the graph to do substitution on, *S* is the substitution set, *Q_id* is the ID of the result graph.

- **Result table for cross-term ordering adjustment in the procedure of substitution at top** : subst_xterm_rt(*Key, P_Id, Label, Edge, Q_Id*).

  P_id is the ID of the graph. *Edge* is the edge issuing from the root and *Q_id* is the ID of the result graph.

- **Result table for term substitution:** term_subst_rt(*Key, TermId, S, SubstId*).

  TermId is the ID of a term to be substituted. *S* is the substitution set. *SubstId* is the ID of the result term.

5. Pretty-Printing Module

The printing module provides utilities for pretty printing MDGs and terms.

5.1 Printing options: pp_flag(*Flag*).

pp_flag(*Flag*) should be set before using the printing utilities. It is used to control the printing options.

- No printing: If pp_flag(!no_pp), then nothing will be printed.
- Printing to screen: If pp_flag(!pp) or no pp_flag/1 is defined, then statistics of graphs at each printing point will be printed on the screen.
- Interactive printing: If pp_falg(!interactive), then it will ask for printing instruction (no printing, printing graph statistics only, printing to screen or printing to a file) and print accordingly.
- Printing to file: If pp_flag(FileName), then statistics of graphs at each printing point will be appended to file FileName. The *FileName* cannot be '!pp', '!no_pp' or '!interactive'.

5.2 Printing predicates

5.2.1 Printing a graph: pp(*Name, P*).

Mode: pp(+, +).

Arguments: *Name* will be printed as the reference to the MDGt. *P* is an MDG.

Function: It prints a graph in a format as explained in [5].

5.2.2 Printing a list of graphs: pps(*Name, Ps*).

Mode: pps(+, +).

Arguments: *Name* will be printed as the reference to the MDGs. *Ps* is a list of MDGs.

Function: It prints a list of MDGs.

5.2.3 Printing a linear graph: ppl(*P*).

Mode: ppl(+).

Arguments: *P* is a linear MDG.

Function: It prints a linear MDG.

5.2.4 Printing a term to screen: print_term(*Term*).

Mode: print_term(+).

Arguments: *Term* is a term.

Function: It prints a term.

5.2.5 Printing a term to a stream: print_term(*S, Term*).

Mode: print_term(+, +).

Arguments: *Term* is a term, *S* is the name of a stream.

Function: It prints a term to stream *S*.

5.2.6 Printing MDGs: pp_mdgs(*Ids, G*).

Mode: pp_mdgs(+, +).

Arguments: *Ids* is a list of graph IDs. *G* is the graph array.

Function: It prints a list of MDGs according to *Ids*.

5.2.7 Printing relations: pp_mus(*N, Parts, MUs*).

Mode: pp_mus(+, +, +).

Arguments: *N* is the name of the relation. *N*=nxst/outp/drv/mux. *Parts* is a list of variables which identifies partition blocks. *MUs* is a list of pairs (*Ms, Us*), where *Us* is a list of symbol orders and *Ms* is a list of MDGs. Each pair represents a partition group. *Ms* is a list of individual relations without recombining. The variables in *Us* can be abstracted (existentially quantified) after *Ms* are applied.

Function: It prints partitioned relations.

6. Reachability Analysis Module

Reachability analysis module exports three predicates. *time_gctns/1* and *time_ns/1* provide the CPU time used for image computation and garbage collection during image computation. *rean/16* is the main reachability analysis procedure which also incorporates a counterexample facility.

rean(P,Q,J,Rt_MUs,Ro_MUs,Drv_MUs,Mux_MUs,C,Ps,Ns,Module,Epsilon,G1,E1,G,E).

Mode: rean(+,+,+,+,+,+,+,+,+,+,+,-,+,+,-,-).

Arguments:

- *P* is the MDG representing the frontier set of states.
- *Q* is the MDG representing the visited set of states.
- *J* indicates current transition number.
- *Rt_MUs and Ro_MUs* are the partitioned transition and output relations. *Drv_MUs* and *Mux_MUs* are the partitioned relations for bus drivers and multiplexers. Their formats will be explained below.
- *C* is the MDG representing an invariant.
- *Ps* and *Ns* are the two lists of MDGs representing the frontier set of states and inputs in each iteration starting from the most recent one.
- *Module* is the module name where necessary data can be found.
- *Epsilon* is a flag which may be 0/1 indicating the success or failure of the reachability analysis.

A partitioned relation [3] (*Rt_MUs, Ro_MUs, Drv_MUs* or *Mux_MUs*) is a list of pairs (*Ms, Us*), where *Us* is a list of symbol orders and *Ms* is a list of MDGs. Each pair represents a partition group. *Ms* is a list of individual relations without recombining. The variables corresponding to the symbol orders in *Us* are abstracted (existentially quantified) after *Ms* are applied.

rean/16 is a recursive procedure. When calling it from application modules, *Ps* and *Ns* are usually set to empty ('[]') and *J* set to 1.

Function: Given a state machine, check in every reachable state if an invariant *C* holds.

Data required: the following data should be declared *dynamic* in module *Module*.

- abs_inputs(*Xs, Os*): *Xs* is a list of abstract input variables. *Os* is a list of their corresponding custom order.
- ren(*Name, Largest_VarOrd, RenSs*), where *Name* identifies a particular renaming substitution, *RenSs* is the renaming substitution given as a list of triples (*Label, NewLabel, NewOrder*), and *Largest_VarOrd* is the number of the largest *NewOrder* in the list. The triple defines a renaming substitution from *Label* to *NewLabel,* at the same time provides the order of the new label. If RenSs=[], then *Name, Largest_VarOrd* are don't-care.
- busdrivers(*Bus, BusDrivers*) : *BusDrivers* is a list of signals which are the drivers for *Bus*.
- mux(*Signal, Cs*) : *Signal* is a select signal of a muxtiplexer. *Cs* is the list of valid values for *Signal*.

7. Application Utilities*

This section explains the data and main procedures for compiling hardware descriptions used in the combinational and sequential verification modules.

7.1 Data

7.1.1 Data for statistics

5. CPU time used for building relations: time_tr(*Task, T*).

   Task=nxst/outp/drv/mux. *T* is the CPU time (same for the following).

6. CPU time used for building MDGs of individual components: time_cc(*T*).

7. CPU time used for garbage collection during image computation: time_gctns(*T*).

8. CPU time used for image computation: time_ns(*T*).

7.1.2 Data for hardware description

9. Abstract input variables: abs_inputs(*Xs, Os*).

   Xs is a list of abstract input variables and *Os* is a list of corresponding custom order of the variables in *Xs*.

10. Bus drivers: busdrivers(*Bus, BusDrivers*).

    Bus is the name of the bus and *BusDrivers* is a list containing the bus drivers.

11. Component indexed by its output: comp(*Z, Cn ,Inputs*).

    Z is the output of the component, *Cn* is the component's name and *Inputs* is a list of inputs of the component.

12. Component indexed by its name: comp_mdg(*Cn, Id*).

    Cn is the component name and *Id* is the ID of the MDG representing the component.

13. Primary input: input(*X*).

    X is a primary input.

14. Multiplexer: mux(*W, Cs*).

    W is the sort of the output of the multiplexer, *Cs* is a list of individual constants which the select of the mux should take.

15. Primary output: output(*Z*).

    Z is a primary output

16. Symbol: symbol(*S, Kind*).

S is the name of a symbol and *Kind=abs_sort/conc_sort/function/gen_const,* indicating whether *S* is of concrete or abstract sort, a function symbol or a generic constant. For example, if **zero** is a generic constant, then we will have symbol(**zero**, *gen_const*).

7.1.3 Miscellaneous

17. Parsing error indicator: error.

    It is asserted when there is an error during compiling the circuit descriptions.

18. Graph printing: shown(*Id*).

    Id is the ID of an MDG. shown(*Id*) indicates that the graph (identified by *Id*) is printed.

7.2 Predicates for Parsing

7.2.1 Term format: reformat(*Term, Sort, NewTerm, SecVars, E1, E*).

Mode: reformat(+, +, +, +, -, -).

Function: formats a term (rewritten in Prolog style HDL) into its internal representation. It is usually used in circuit compilation

Arguments: *Term* is the term to be formatted. *NewTerm* is the result after formatting. If *Term* is an individual constant, *Sort* is undefined. If *Term* is a generic constant or a variable, the *Sort* is the sort of *Term*. If *Term* is a function, then *Sort* is the target sort of the function. *SecVars* is a list of signals appearing in *Term*. It provides the secondary variables.

File: homeMDG/programs/applications/utilities/util_parser1.pl.

7.2.2 Component compilation: compile_circuit(*G1, E1, G, E*).

Mode: compile_circuit(+, +, -, -).

Function: Build MDGs for components of a circuit. It checks if a component module is predefined in the MDG-HDL. If not, an error message is given and the procedure stops.

Arguments: Refer to section 4.1 for the definition of these arguments.

File: homeMDG/programs/applications/utilities/util_parser1.pl.

7.2.3 Invariant compilation: compile_cond(*Cond_mdg, G1, E1, G, E*).

Mode: compile_cond(+, +, +, -, -).

Function: Build an MDG for an invariant. The invariant is expressed as a combinational circuit (see [5]) in the invariant specification file.

Arguments: *Cond_mdg* is the result MDG.

7.2.4 Relations compilation: compile_main(*Nxst_MUs ,Outp_MUs ,Drv_MUs ,Mux_MUs ,G1, E1,G,E*).

Function: Build the transition and output relation MDGs for the circuit. At the same time, build relations for bus drivers and multiplexer outputs which will be used in run-time checking.

Arguments: Nxst_MUs, Outp_MUs, Drv_MUs and Mux_MUs are the partitioned transition relations, output relations, bus driver relations and multiplexer relations, respectively.

File: homeMDG/programs/applications/verify_seq/compile_seq1pl.

8. Miscellaneous issues

8.1 Programming style

The computation based on decision graphs is deterministic and hence it does not require the power of the Prolog search engine. In fact, the codes are procedure-like. However, the unification mechanism is useful for matching and substitution of terms.

8.2 Files and database

When using the MDG operators, appropriate data, files and the libraries (basics, lists, logarr, ordsets, sets) must be loaded. The databases must be cleared at the beginning of the application.

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*

References

[1] F. Corella, Z. Zhouz, 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 and N. Boulerice. MDG Tools (V1.0) User's Manual. December, 1995.

Appendix

A. 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 various 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.