MDG Model Checker User's Manual

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

1. Introduction

RT (Register-Transfer) level hardware designs can be modelled as Abstract State Machines (ASMs) [3][4]. ASMs 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.

The MDG tools include an MDG package, a reachability analysis procedure and the applications for combinational and sequential hardware verification at RT level[5]. Recently, a model checker for a first-order linear-time temporal logic was added into the MDG package. In this document, we describe how to use the MDG model checker**.**

Please not that the MDG tools are intended for Abstract State Machine (ASM) [3][4][5] 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.

Contents of this document. This document is about how to use the model checker tool within the MDG package. It contains:

- the syntax and the semantics of the property specification language L_{MDG};
- the fairness constraints;
- user commands for the applications
- practical issues in using the applications

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

Further references. This manual is not self-contained. Readers should refer to the references, especially [3] for terminology and techniques about Multiway Decision Graphs and reachability analysis of abstract state machines, and [1] for detailed property checking procedures. To use the applications, though it is not necessary, some preliminary knowledge of Prolog is preferred.

2. A First-Order Linear-time Temporal Logic: $\mathcal{L}_{\mathrm{MDG}}$

Given a description of an ASM, and a set of *ordinary variables* which are available for use in the specification of the property, the *atomic formulas* of $\mathcal{L}_{\mathrm{MDG}}$ are Boolean constant *True*, *False*, or equations $t_1 = t_2$, where t_1 is an ASM_variable, t_2 is an ASM_variable or a constant or an ordinary variable or a function of ordinary variables. The *Next_let_formulas* are defined as follows:

(1) each atomic formula is a Next_let_formula,

(2) if p, q are Next_let_formulas, then so are:

$!p$ (not p), $p\&q$ (p and q), $p|q$ (p or q), p $->q$ (p implies q), **X**p, and

LET ($v = t$) **IN** p, where t is an ASM_variable and v an ordinary variable.

We allow the formula **LET** ($v_1 = t_1$) & ... & ($v_n = t_n$) **IN** p as a shorthand for **LET** ($v_1 = t_1$) **IN** *((* **LET** ($v_1 = t_1$) **IN** *(......* **LET** ($v_n = t_n$) **IN** *p)));* and we call ($v_1 = t_1$) & ...& ($v_n = t_n$) as a *Let_equation*.

The properties allowed in $\mathcal{L}_{\mathrm{MDG}}$ can have the following templates:

A(Next_let_formula**)**

AG (Next_let_formula **)**

AF (Next_let_formula **)**

A(Next_let_formula **) U (** Next_let_formula **)**

AG ((Next_Let_formula **) => (F (** Next_let_formula)))

AG ((Next_let_formula **) => ((** Next_let_formula) **U** (Next_let_formula)))

Syntax of $\mathcal{L}_{\mathrm{MDG}}$**:** We give the syntax of L_{MDG} in BNF. A *terminal symbol* is written in bold style, a *nonterminal symbol* is written in regular style starting with an upper case letter. Square brackets denote options. The start symbol is Property_file.

Property_file ::=

Property**;**

Property ::=

A (Next_let_formula **)**

| **AG (** Next_let_formula)

| **AF (** Next_let_formula)

| **A (** Next_let_formula **) U (** Next_let_formula)

| **AG ((** Next_Let_formula **) => (F (** Next_let_formula)))

| **AG ((** Next_let_formula **) => ((** Next_let_formula) **U (** Next_let_formula)))

Next_let_formula ::=

X Next_let_formula

| **LET (** Let_equation **) IN (** Next_let_formula **)**

| Next_let_formula **->** Next_let_formula

(Note: the first Next_let_formula can only contain concrete variables)

| Next_let_formula **&** Next_let_formula

| Next_let_formula **|** Next_let_formula

| **!** Next_let_formula

(Note: the Next_let_formula can only contain concrete variables)

| (Next_let_formula)

| Basic_formula

Basic_formula ::= Lterm = Rterm | *True* | *False*

(Note: *True, False* are Boolean constants)

Lterm ::= ASM_variable_Name

Rterm ::=

ASM_variable_Name

| OrdVar_Name

| IntegerConstant

| SymbolicConstant

| Function (only applies in a Next_let_formula prefixed by Let_equation**)**

Let_equation ::=

Let_equation **&** Let_equation

| (Let_equation)

| OrdVar_Name **=** ASM_variable_Name

Function ::= Function_Name **(** Parameter_List**)**

Parameter_List ::= Parameter **|** Parameter_List **,** Parameter

Parameter ::= OrdVar_Name **|** Function

ASM_variable_Name ::= [a-bd-eg-uw-z][A-Za-z0-9_]*

OrdVar_Name ::= [v][A-Za-z0-9_]*

Function_Name ::= [f][A-Za-z0-9_]*

IntegerConstant ::= [0-9]*

SymbolicConstant ::= [c][A-Za-z0-9_]*

Semantics of $\mathcal{L}_{MDG}$: A *path* is a sequence of total_states (a *total_state* is an assignment to the set of state, input and output variables). We use $\pi_0 = (s_0, s_1, s_2, \ldots)$ to denote a full path and $\pi_i = (s_i, s_{i+1}, s_{i+2}, \ldots)$ a suffix path starting from s_i. All the formulas in $\mathcal{L}_{MDG}$ are *path formulas*. Further, if a path formula contains no temporal operators, then it is also a *state formula*.

We write $\pi_i, \sigma \models p$ (resp. $s, \sigma \models p$) to mean that a path formula p (resp. a state formula p) is true at path π_i (resp. at a total_state s). We use $Val_{s\cup\sigma}(t)$ to denote the value of term t under a ψ-compatible assignment s to state, input and output variables, and a ψ-compatible assignment σ to the ordinary variables. We then define $\models$ inductively as follows:

- $s, \sigma \models t_1 = t_2$ iff $Val_{s\cup\sigma}(t_1) = Val_{s\cup\sigma}(t_2)$.
- $s, \sigma \models\ !p$ iff it is not the case that $s, \sigma \models p$.
- $s, \sigma \models p\ \&\ q$ iff $s, \sigma \models p$ and $s, \sigma \models q$.
- $s, \sigma \models p \mid q$ iff $s, \sigma \models p$ or $s, \sigma \models q$.
- $s, \sigma \models p\text{->}q$ iff $s, \sigma \models\ !p$ or $s, \sigma \models q$.
- $s, \sigma \models$ **LET** $(v = t)$ **IN** p iff $s, \sigma' \models p$ where $\sigma' = (\sigma \setminus \{(v, \sigma(v))\}) \cup \{(v, Val_{s\cup\sigma}(t))\}$.

- $\pi_i, \sigma \models p$ where p is a state formula, iff $s_i, \sigma \models p$.
- $\pi_i, \sigma \models\ !p$ iff it is not the case that $\pi_i, \sigma \models p$.
- $\pi_i, \sigma \models p\&q$ iff $\pi_i, \sigma \models p$ and $\pi_i, \sigma \models q$.

• $\pi_i, \sigma \models p|q$ iff $\pi_i, \sigma \models p$ or $\pi_i, \sigma \models q$.

• $\pi_i, \sigma \models p\text{->}q$ iff $\pi_i, \sigma \models\ !p$ or $\pi_i, \sigma \models q$.

• $\pi_i, \sigma \models \mathbf{X}p$ iff $\pi_{i+1}, \sigma \models p$.

• $\pi_i, \sigma \models \mathbf{G}p$ iff $\pi_j, \sigma \models p$ for all $j \geq i$.

• $\pi_i, \sigma \models \mathbf{F}p$ iff $\pi_j, \sigma \models p$ for some $j \geq i$.

• $\pi_i, \sigma \models p\mathbf{U}q$ iff for some $k \geq i$, $\pi_j, \sigma \models q$, and $\pi_k, \sigma \models p$ for all j $(i \leq j < k)$.

• $\pi_i, \sigma \models$ **LET** $(v = t)$ **IN** p iff $\pi_i, \sigma' \models p$, where $\sigma' = (\sigma \setminus \{(v, \sigma(v))\}) \cup \{(v, Val_{si \cup \sigma}(t))\}$.

Given a property in $\mathcal{L}_{\mathrm{MDG}}$ regarding an ASM under a given interpretation ψ, the property holds on the ASM iff the property is true for all paths starting from each of the initial states.

3. Fairness constraints

When verifying liveness properties, one is usually interested only in the so-called *fair* infinite computation paths. A fair computation path is a path along which the states satisfy each fairness condition infinitely often.

In the literature, various methods for specifying fairness constraints have been developed for CTL model checking and language containment using L-automata. In our method, we impose fairness constraints in a similar way as in the method based on language containment, i.e., by specifying *cycle set*s. Each cycle set consists of states that satisfy an exception condition H. If a set of states repeat forever along a path and the set of states is covered by any given cycle sets, then that path is not considered. I.e., only the computation paths along which the states satisfy every $!(H_i)$ ($1 \leq i \leq n$) infinitely often are considered. Therefore, $!(H_i)$ can be viewed as the fairness constraints. We call the formula representing the exception condition H_i an *H_formula*.

The syntax of an H_formula is defined as follows:

(1) the equation $t_1 = t_2$ is an H_formula, where t_1 is an ASM_variable and t_2 is an ASM_variable or a constant.

(2) if p, q are H_formulas, then so are $!p$, $p\&q$, $p|q$, $p\ ->q$, $\mathbf{X}p$.

Note that only concrete ASM_variables may appear in the H_formulas. All the fairness constraints imposed are stored in a file, which is interpreted in before the model checking procedure is invoked.

4. User Commands for Applications

In general, it takes two steps to check a property expressed in L_{MDG}. The first step is to automatically build additional ASMs that represent the Next_let_formulas appearing in the property and the exception conditions if fairness constraints are imposed, and then connect these additional ASMs to the original ASM. The property parser developed in C with Yacc&Lex will be needed for this step. The second step is to check a simpler property on the composite machine. The MDG model checker developed in Prolog within the MDG package will be used for this step.

4.1 The commands to run the property parser

The command “next” is used to build additional ASMs that represent the Next_let_formulas appearing in a property. Four arguments are needed:

1. file path name: the complete path where the circuit description file, order file and the property file are stored.
2. circuit description file name
3. order file name
4. the property file name

The command will be as follows:

`next` filePath circuitDescriptionFile orderFile propertyFile

If a property contains a function, then the executable file "func" will be needed. When the parser sees a function, it uses the “func” to process it. So the “func” file should be copied to the same directory as "next".

The command “fair” is used to build additional ASMs that represent the exception conditions in case fairness constraints are imposed. The same four arguments as for the “next” command are needed. **Note that if the “next” command has been applied, then the output files from the “next” command should be used as arguments for the “fair”.** The “fair” command looks as follows:

`fair` filePath circuitDescriptionFile orderFile exceptionConditionsFile

The executable code for “next” , “func” and “fair” can be found at ~xu/MC_PARSER.

4.2 The commands to run the MDG model checker

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 model checking related files:

- **`seq.`** For sequential verification.
- **`mc.`** For model checking.

Then choose a correct property type according to the property to be verified:

Property type options:

1: A	2: AG	3: AF	4: AU
5: E	6: EG	7: EF	8: EU
9: AGAF	10: AGAU		
11: AF with fairness	12: AGAF with fairness		
13: AU with fairness	14: AGAU with fairness		

For example, if a property with template A(Next_let_formula) is to be verified, then the user should choose type 1; if a property with template AF(Next_let_formula) is to be verified, then the user should choose type 3.

Note that the types 5 - 8 are for properties with existential path quantifiers. Generally speaking, the user should not use any of these property templates. They can only be used when the ASM to be verified is purely concrete. i.e., no single abstract variable appears in the ASM description.

When you have chosen the property type, 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. **Please note that the output files of the “next” command or “fair” command should be used as the input files.**

When the program finishes, it will give out the property checking result and provide detailed resource statistics. However, at this moment, the tool will not provide a counter-example when a property fails.

5. Practical Issues in using the model checker

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

In property file:

1. Variable names in a ACTL property cannot begin with c (symbolic constants have to be declared starting with c);

2. Variable names in a ACTL property cannot begin with f (function names have to start with f);

3. Functions are used in a limited fashion: functions are used in the Rterm (right hand term of a Basic_formula) and only in a Next_let_formula prefixed by Let_equation(s).

In circuit description and order file:

1. No extra space chars after each signal declaration. For example,

   signal(xcnt_B, wordn).   -----> no extra space char after ")."

   [Reason: the parser needs to create a signal list. It checks that after ")." there should be a "\n" and then go to the next line. It may be too restricted, but this could detect other unwanted characters after ")."]

2. It is needed that in "next_state_partition" declaration, no more than one [[signal-name]] is allowed in one line, and should be in the following format:

   next_state_partition([----> no extra space char after "["

   [[n_pc]],

   [[n_flgb_B]],

   [[n_xcnt_B]]

   ]).   ---->   no extra space char after "])."

The same restriction applies to “order-main” declaration in the order file.

In algebraic file:

1. Add in the algebraic file one function declaration:

   function(absComp,[wordn,wordn],bool).

   and the rewrite rule: xtrr([],absComp(X,X),1).

2. Add in the algebraic file all the additional functions used in the property.

6. Bugs

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

7. References

[1] Y. Xu. *Model Checking for a First-order Temporal Logic Using Multiway Decision Graphs*.PhD thesis, D'IRO, University of Montreal, 1999.

[2] Y. Xu, E. Cerny, X. Song, F. Corella, O. Ait Mohamed. Model Checking for a First-Order Temporal Logic using Multiway Decision Graphs. In *Proceedings of Conference on Computer-Aided Verification* (CAV'98). Vancouver, BC, Canada, July 1998.

[3] Z. Zhou. *Multiway Decision Graphs and Their Applications in Automatic Formal Verification of RTL Designs*. PhD thesis, D'IRO, University of Montreal, 1997.

[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) User's Manual*. December, 1995.