TRANSLATING LTL SPECIFICATION TO MDG-HDL

Fang Wang, Ali Habibi and Sofiène Tahar
Concordia University, ECE Dept., Montreal, Quebec, H3G 1M8 Canada
{f_wang,habibi,tahar}@ece.concordia.ca

Abstract

MDG-HDL is the hardware description language accepted by the Multiway Decision Graphs (MDGs) package. Linear Temporal Logic (LTL) is a widely used formal language for specifying design properties. To develop an LTL model checking algorithm on MDGs, we first need to translate the LTL specification into an automaton described in MDG-HDL. In this paper, we present and compare two methods for translating LTL specifications to MDG-HDL using two existing tools Wring and LTL2AUT. Experimental results based on a set of benchmark LTL formulas show that the Wring based approach is better than the LTL2AUT implementation with respect to the size of the generated automata but is worse in terms of CPU time used.

1. INTRODUCTION

Multiway Decision Graphs (MDGs) [1] are a relatively new class of decision diagrams which subsume traditional Reduced Ordered Binary Decision Diagrams (ROBDDs) [1] and extend them with abstract data sorts and uninterpreted functions. In an MDG, a data signal can be described as an abstract variable instead of a boolean vector in ROBDDs [3], a data operation can be described by an un-interpreted function and cross-operators are useful to model feedback signals from the datapath to the control circuitry. Thus, MDGs are more compact than ROBDDs for describing circuits with datapaths. The MDG tools provide several formal verification applications including equivalence checking, invariant checking, sequential equivalence checking and branching time temporal logic model checking.

This paper is part of a project to extend the MDG tools with a Linear Temporal Logic (LTL) model checking algorithm [9]. LTL is a widely used formal language for specifying design specification. LTL model checking uses ω-automata as the unifying models for a system and its specification, and checks whether the language of the system is contained in the language of its specification [7]. There are two main steps in these methods: translating the LTL specification into ω-automaton and checking the language containment. The size of the automaton generated is critical to the capability of this method due to the possible state space explosion problem [9]. The hardware description language that the MDG tools accept is MDL-HDL. Thus, for the realization of an MDG-based LTL model checking tool, we first need to generate a translator from LTL specification to MDG-HGL.

Constructing an ω-automata from LTL formulas is well developed and many excellent tools are freely available in the public domain areas of formal methods [10]. In this paper, we make use of two such tools, namely Wring [8] and LTL2AUT [4]. We implemented for each a transformation approach to generate HDG-HDL code from LTL formulas. We also tested our implementation with the MDG tools to make sure the syntax is correct and also compared these two approaches with respect to the translation performance.

This paper is organized as follow. In Section 2, we overview the notion of MDGs and the MDG package. In Section 3, we review the basics of LTL transformation techniques, and introduce the tools Wring and LTL2AUT. We discuss our LTL to MDG transformation methods and experimental results in Section 4. Finally, in Section 5, we conclude the paper and point to some future work.

2. MULTIWAY DECISION GRAPHS

Multiway Decision Graphs (MDG) are decision diagrams based on a subset of many-sorted first-order logic, augmented with a distinction between concrete and abstract sorts [2]. Concrete sorts have an enumeration while abstract sorts do not. The distinction is motivated by the natural division between the control circuitry and datapath in digital hardware designs. The distinction between concrete and abstract sort leads to three kinds of function symbols: abstract functions, concrete functions, and cross-operators. Abstract functions are used to describe data operations, while cross-operators are useful for modeling feedbacks from the datapath to the control circuits. Both cross-operators and abstract functions are uninterpreted. Concrete functions on the other hand are fully interpreted.

An MDG is a finite directed acyclic graph [3]. An internal node of an MDG can be a variable of concrete sort

CCECE 2003 - CCGEI 2003, Montréal, May/mai 2003
0-7803-7781-8/03/$17.00 © 2003 IEEE

with its edge labels being the individual constants in the enumeration of the sort; or it can be a variable of abstract sort and its edges are labeled by abstract terms of the same sort; or it can be a cross-term. An MDG may have only one leaf node denoted as **T**, which means all paths in an MDG are true formula. MDGs represent relations as well as sets of states [3].

The MDG package contains a set of basic operators including disjunction, relation product (image operation), pruning-by-subsumption (for test of set inclusion); and a reachability analysis procedure based on implicit abstract enumeration [3]. Thereafter, a set of MDG tools have been developed providing applications for hardware verification such as combinational circuits verification, equivalence checking for two state machines, invariant property checking and model checking. The MDG hardware description language (MDG-HDL) [10] that the MDG tools accept allows the use of abstract variables and uninterpreted function symbols. MDG-HDL supports structural descriptions, behavioral descriptions, or the 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 a truth table [10].

3. LTL AND BUCHI AUTOAMTA

In this section, we first review the basic concepts of LTL and ω-automata, then describe the algorithms of construction ω-automata from LTL formulas.

3.1 Linear Temporal Logic

Linear time temporal logic formulas are composed of a set of atomic propositions, the standard Boolean connectives, and the temporal operators *X (next time), U(until)*. In following, we describe the general syntax and semantics of LTL.

Syntax. Given a set of propositions *P*, LTL formulas are defined inductively as follows.

1. Every member of *P* is a formula,
2. If *p* and *q* are formulas, then so are $\neg p$, $p \wedge q$, $p \vee q$, $X\, p$, $p\ U\ q$.

Semantics. The semantics is defined on a linear structure $M = (S, x, L)$, where *S* is the set of states, $x: N \to S$ is an infinite sequence of state, and $L: S \to P$ is a function that labels each state with the set of atomic propositions in *P*. We use x^i to denote the infinite sequence of state $(s_i, s_{i+1}, \ldots)$ and $M, x \models p$ to mean that in structure *M* formula *p* in true on the sequence *x* starting at s_0.

1. $M, x \models p$ iff for $p \in P$
2. $M, x \models \neg p$ iff not the case $M, x \models p$
3. $M, x \models p \wedge q$ iff $M, x \models p$ and $M, x \models q$
4. $M, x \models p \vee q$ iff $M, x \models p$ or $M, x \models q$
5. $M, x \models Xp$ iff $M, x^1 \models p$
6. $M, x \models pUq$ iff there is an $i \geq 0$, $M, x^i \models p$ and $M, x^j \models q$

Other temporal operators, such as *G (always)* and *F (eventually)*, as well as the constants *True* and *False* are defined abbreviations as follows respectively: $Fp = TrueUp$, $Gp = \neg F(\neg p)$, $True = \neg p \vee p$ and $False = \neg p \wedge p$.

3.2 Büchi Automaton

An automaton is a finite state transition with acceptance condition. An ω-automaton is an automaton on infinite words and Büchi Automaton (BA) [2] is a type of ω-automaton with the acceptance condition consists of several sets of accepting states. An BA is defined as a tuple $A = (S, S_0, \delta, F)$, where *S* is a finite set of states, S_0 is the set of initial states, δ is the transition relation and F is the set of accepting states. A run is accepted if it contains at least one state in every fair set infinitely often.

The language of an BA is defined by the labels on the states. A labeled BA is a triple (*A, D, L*), where *A* is the BA, *D* is some finite domain and *L* is a labeling function mapping from the states of BA to the subset of the domain *D*. A labeled BA accepts a word from the *D* regular language corresponding to the acceptance run.

3.3 From LTL to Büchi Automaton

A construction algorithm from LTL to Büchi automaton generates an automaton recognizing all infinite sequences that satisfy a given LTL formula. The central part of the algorithm is a tableau-like procedure for building a graph [7], which decomposes a formula according to its Boolean structure. Temporal operators of the formula are expanded by the expansion rule $pUq \equiv q \vee (p \wedge X(pUq))$. The procedure continues until the original formula becomes a propositional in terms of constants, atomic propositions, and sub-formulas starting with *X*. States are obtained from the nodes of the graph returned by the algorithm and labeled by sets of subformulas. Initial states are the states directly connecting to the

node labeled by the original formula. Translation relations are derived from connecting what has to be true immediately to what has to be from the next state on. Acceptance conditions are added to each state with the formula $X(pUq)$ in its label.

An efficient algorithm based on the above tableau construction was proposed in [7] for the construction of BA on the fly. An improvement of this algorithm, based on syntactic simplification, is discussed in [4]. An extended algorithm of [7] [4], based on rewriting of LTL formulas and simplifying the resulting BA, was presented in [8]. Based on the above algorithms, there are many construction tools, such as LTL2AUT [4], EQLTL and Wring [8]. In next section, we will discuss the translation of BA to MDG-HDL using Wring and LTL2AUT.

4. FROM LTL TO MDG-HDL

In this section, we present both tools LTL2AUT and Wring. LTL2AUT is developed in C while Wring is developed in Perl; LTL2AUT outputs a text description for the generated automata while Wring is in a Verilog.

4.1 The Wring method

Wring[1] was developed by Somenzi and Bloem [8] at UC Berkeley using the Perl language. The implementation is based on a medular-block structure and outputs an BA as Verilog module (monitor). Figure 1 (a) shows the Verilog code for the LTL formula `p1 U p2`. The corresponding MDG-HDL code is given in Figure 1 (b).

The BA Verilog description can be transformed into an MDG-HDL program using concrete variables. To translate the generated automaton into MDG-HDL, we modify the Verilog monitor as follows. We declare a new concrete sort *bstateSort* enumerating all states of the automaton. Next, we define a variable *bstate* of this sort and a set of Boolean variables a_{ccn} to represent the acceptance conditions. The number of the set is equivalent to the number of the acceptance sets. If there is only one set, we simply declare it as *acc*. Then we generate an MDG table with *bstate,* and the formula presentations input variables, and *n_bstate* to represent the transition relation. The value is assigned according to the transition relation of the automaton. Similarly, we get the table for the acceptance condition with the *bstate* and a_{ccn}.

```
typedef enum {n1,n2,n3,Trap} states;
module Buechi(clock,p1,p2,fair);
input clock,p1,p2;
output fair;
states reg state;
states wire ND_n1_n2;
assign ND_n1_n2 = $ND(n1,n2);
assign fair = (state == n3);
initial state = n1;
always @ (posedge clock) begin
 case (state)
  n1:
    case ({p1,p2})
      2'b00: state = Trap;
      2'b01: state = n2;
      2'b10: state = n1;
      2'b11: state = ND_n1_n2;
    endcase
  Trap:
      state = Trap;
  n2,n3:
      state = n3;
 endcase
end
endmodule
```

(a)

```
conc_sort(bstateSort, [n1, n2,
n3, trap]).
 signal (p1, bool).
 signal (p2, bool).
 signal(acc, bool).
 signal(bstate, bstateSort).
 component(bcomp, table([
   [bstate, p1,p2, n_bstate],
    [n2,*, *, n3],
    [n3,*, *, n3],
    [n1,*, 1, n2],
    [n1,1, *, n1],
   [n1, 0, 0, trap],
   [trap, *,*, trap] ])).
component(bc_comp0, table([
     [bstate, acc],
      [n3,1]|0])).

st_nxst(bstate,n_bstate).
init_val(bstate, n3).
next_state_partition([
    [[n_bstate]] ]).
```

(b)

Fig. 1. Verilog and MDG-HDL Models for p1 U p2

4.2 The LTL2AUT method

LTL2AUT [2] is a software package for generating Büchi automata from LTL formulas, developed by Daniele et. al. at Rice univerisity using the C language[4]. LTL2AUT generates an automaton graph model in text form with a set of nodes labeling five items: *State, Old, Nxt, Succ*, and *Acc*. Initial states are marked separately.

In the transformation from automaton to MDG-HDL description, we first get the respective states from the nodes. Initial states are as listed in the text, and transition relation are derived from *Succ*. Labels of states are derived from the atomic propositions with or without negation appearing in *Old.* Finally, the acceptance conditions are acquired from the *Acc*. Different accepting sets are labeled by different numbers such as the states in condition 0 are labeled by *Acc*: 0. Similarly to the Wring based approach, we describe the automaton transition using MDG tables.

Figure 2 shows an BA model for the same LTL formula of Figure 1. Figure 2 (a) is the LTL2AUT model, while Figure 2 (b) gives the corresponding MDG-HDL model.

4.3 Experimental Results

To test the two implementations, we used a set of benchmark formulas [8], which we translated to automaton modeled in the MDG-HDL and verified them using

1. http://www-cad.eecs.berkeley.edu/Respep/Research/vis/index.html

2. http://www.cs.rice.edu/CS/Verification/Software/software.html

MDG tools. We tested the obtained MDG-HDL descriptions with two MDG applications: State exploration to test the generated automaton and ensure syntax correctness, and sequential equivalence checking to test that the automata generated by both methods are indeed equivalent for every formula.

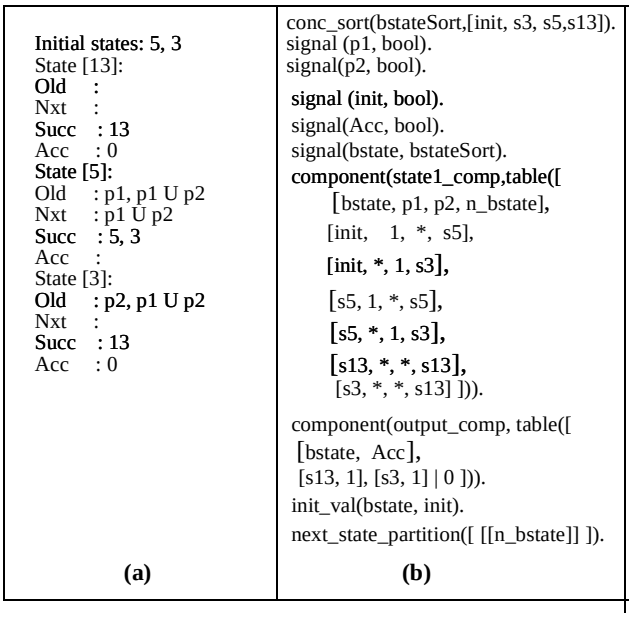

Fig. 2. LTL2AUT and MDG-HDL Models for p1U p2

Table 1 summarizes obtained experimental results on a 296 MHz Sun station with of 768MB memory including the CPU time used in sec., the size of the automaton in terms of number of states, the number of transactions , and the number of accepting conditions. From Table 1, we can see that the implementation with Wring is worse in terms of CPU time, This is due to the fact that first, Wring is implemented in Perl while LTL2AUT is in C; second, Wring itself includes more algorithms in order to simplify the output automaton than LTL2AUT. Thus, the Wring implementation produces smaller automaton size then the LTL2AUT one.

5. CONCLUSION AND FUTURE WORK

In this paper, we described two methods to translate LTL specifications into MDG-HDL. The first one is an adaption of the Wring tool, while the second one is based on the LTL2AUT tool. Both implementations were tested on a set of benchmarks formulas. Experimental results show that the implementation with Wring is better than LTL2AUT with respect to the size of the automaton but is worse in terms of the CPU time used.

Since the generation of automata from LTL is the immediate form needed for the intend MDG LTL model checking, the size is more important than the CPU time. We hence conclude to use the Wring based method in our next work step. As we mentioned in the introduction, this translation is the first part of a large project to develop LTL model checking with MDGs. The MDG abstract description capability accepts more abstract description of specification. The alternative goal is to build a first-order LTL model checking tool which checks abstract design models described in MDG-HDL with respect to first-order specifications that use abstract variables and uninterpreted functions.

6. REFERENCES

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computer, Vol. 35, No. 8, 1996, pp 677-691.

[2] E. M. Clarke, O. Grumberg, D. Peled. Model Checking, MIT Press, c1999.

[3] F. Corella, Z. Zhou, X. Song, M. Langevin and E. Cerny. Multiway decision graphs for automated hardware verification. Formal Methods in System Design, Vol. 10, No. 1, 1997, pp 7-34.

[4] M. Daniele, F. Giunchiglia and M. Y. Vardi. Improved Automata Generation for Linear Temporal Logic, Computer Aided Verification, LNCS 1633, Springer Verlag, 1999, pp. 249-260.

[5] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly Automatic Verification of Linear Temporal Logic, Protocol Specification Testing and Verification, 1995, pp. 3-18.

[6] C. Kern and M. R. Greenstreet. Formal Verification in Hardware Design: a Survey, Transactions on Design Automation of Electronic Systems, vol. 4, 1999, pp. 123-193.

[7] R. P. Kurshan. Automata-Theoretic Verification of Coordinating Processes, Princeton University Press, 1994.

[8] F. Somenzi and R. Bloem. Efficient Buchi Automata from LTL Formulae, Computer Aided Verification, LNCS 1877, Springer-Verlag, 2000, pp. 247-263

[9] M. Y. Vardi. An Automata-theoretic Approach to Linear Temporal Logic, Logics for Concurrency: Structure versus Automata, LNCS 1043, Springer Verlag, 1996, pp. 238-266.

[10] Z. Zhou and N. Boulerice. MDGs Tools (V1.0) User's Manual, D'IRO, University of Montreal, June 1996.

Table 1. Experimental Results and Comparison of LTL to MDG-HDL Translations using LTL2AUT and Wring.

Formulae	Using LTL2AUT				Using Wring			
	CPU Time	# States	# Trans.	# Acc.	CPU Time	# States	# Trans.	# Acc.
p U *q*	0.02	4	6	1	0.08	4	6	1
p U (*q* U *r*)	0.01	5	10	2	0.15	5	10	1
~(*p* U (*q* U *r*))	0.01	8	19	0	0.28	7	13	0
GF *p* ->GF *q*	0.01	5	11	2	0.27	5	11	1
F*p* U G*q*	0.01	6	14	2	0.22	4	7	1
G*p* U *q*	0.01	6	8	1	0.17	5	6	1
~(GF *p* ->GF *q*)	0.02	8	16	2	0.43	4	8	1
~(GF *p* <->GF *q*)	0.01	9	32	2	0.80	6	14	1
p R (*p* ∨ *q*)	0.01	5	11	2	0.12	4	6	0
X*p* U X*q* ∨ ~ X (*p* U *q*)	0.01	10	16	1	0.16	5	8	1
(X*p* U *q*) ∨ ~X(*p* U (*p* ∨ *q*))	0.01	10	21	1	0.29	8	14	1
G(*p* ->F*q*) ∧ (X*p* U *q*) ∨ ~X(*p* U (*p* ∧ *q*))	0.03	17	61	2	1.79	11	30	1
G(*p* ->F*q*) ∧ ((X *p* U X *q*) ∨ ~X(*p* U *q*))	0.01	21	63	2	0.15	4	11	1
G(*p* ->F*q*)	0.01	4	11	1	0.16	4	11	1
~G(*p* ->X (*q R r*))	0.01	6	10	2	0.17	6	10	1
~(GF*p* ∨ FG *q*)	0.04	5	16	2	0.36	4	8	1
G(F*p* ∧ F*q*)	0.01	5	20	2	0.26	4	12	2
F*p* ∧ F ~*p*	0.02	9	17	2	0.37	9	17	1
(X~*q* ∧ ~*r*) *R* (X(~s U ~*p*) *R* (~*r*) U (~s *R* ~*r*))	0.02	69	391	2	91.84	9	15	1
(G(*q* ∨ GF *p*) ∧ G (*r* ∨ GF~*p*)) ∨ G*q* ∨ G*r*	0.02	11	39	1	0.92	9	22	2
(G*q* ∨ FG *p*) ∧ G(*r* ∨ FG ~*p*)) ∨ G*q* ∨ G *r*	0.01	11	39	1	0.70	5	7	1
~((G(*q* +GF*p*) ∧ G(*r* +GF~*p*)+G(*q*)+G*r*)	0.01	35	130	2	2.89	12	30	1
~((G(*q* ∨ FG*p*) ∧ G(*r* ∨ FG~*p*)) ∨ G*q* ∨ G*r*)	0.01	35	142	2	2.78	13	36	1
G(*q* ∨ X G*p*) ∧ G(*r* ∨ XG~*p*)	0.01	7	21	0	0.33	6	10	0
G(*p* ∨ (X*q* ∧ X~*q*))	0.01	5	10	0	0.05	1	1	0
(*p* U*q*)∨ (*q* U *p*)	0.02	6	11	2	0.22	4	5	1