Language Emptiness Checking using MDGs

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

ABSTRACT

Multiway Decision Graphs (MDGs) are efficient diagrams suitable for the modeling and automatic verification of register transfer level designs. The MDG tools provide a first-order branching time model checking, sequential equivalence checking, and combinational verification. In this paper, we present a new model checking algorithm based on language emptiness checking using MDGs. The proposed procedure makes use of the Wring tool from UC Berkeley to generate the property automaton. Language emptiness is checked on the product of this latter and the design automaton represented in terms of MDGs. Compared with the existing MDG model checking, our algorithm shows superior performance. We also conducted experimental comparison between our tool, VIS from UC Berkeley and Cadence FormalCheck.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Verification*

General Terms

Verification

Keywords

Formal hardware verification, Model checking, Büchi automata, Language emptiness checking, MDGs (Multiway Decision Graphs), ROBDDs (Reduced Ordered Binary Decision Diagrams)

1. INTRODUCTION

Formal verification [5] is being considered by the industry community as a potential complement to traditional simulation. One of formal verification methods is Language Emptiness Checking (LEC) [1]. LEC uses ω-automata as the unified models for both the system design and the property, and checks whether the language of the system automaton is contained in the language of the property automaton [1]. While the system design automaton recognizes all the system execution sequences, the property automaton accepts those models that violate the property. Verification amounts to checking whether the language recognized by the synchronous product of the above two automata is empty.

Popular LEC algorithms use Propositional Linear Time Temporal Logic (PLTL) formula to describe properties, and finite state machines to represent the system design [8]. PLTL formula can be transformed into Büchi Automata (BA) [8] using various tools. Known language emptiness model checking tools are implemented using ROBDDs (Reduced Ordered Binary Decision Diagrams) [6]. Since ROBDDs require a Boolean representation of the circuit and the size of a ROBDD grows, sometimes exponentially, with the number of Boolean variables. Therefore, ROBDD based verification cannot be directly applied to circuits with complex and large datapaths.

Multiway Decision Graphs (MDGs) [2] have been proposed as a new class of decision graphs that comprises, but is much broader than, the class of ROBDDs. The underlying logic of MDGs is a subset of many-sorted first-order logic with a distinction between concrete and abstract sorts. A concrete sort has an enumeration while an abstract sort does not. Hence, a data signal can be represented by a single variable of abstract sort, rather than by Boolean variables, and a data operation can be viewed as a black-box and represented by an uninterpreted function symbol. MDGs are thus much more compact than ROBDDs for designs containing a complex datapath [2]. At present, the following MDG based applications are available: Combinational Circuits Verification, Sequential Equivalence Checking, Invariant Checking and Model Checking (MC). In MDG MC, the property to be verified is described by a first-order CTL* temporal logic formula, called $\mathcal{L}_{\mathcal{MDG}}$ [9].

Motivated by a desire to combine the automation feature of language emptiness checking and MDG's abstract representation capacity, in this paper, we propose a new Language Emptiness Checking approach based on MDGs (MDG LEC). MDG LEC uses $\mathcal{L}_{\mathcal{MDG}}{}^*$, a subset of $\mathcal{L}_{\mathcal{MDG}}$, to describe property, and ASM to model the system design. In the proposed checking procedure, we first transform $\mathcal{L}_{\mathcal{MDG}}{}^*$ formulas into PLTL formulas. Second, we use the Wring [7] tool to generate a BA. Third, we build a synchronous product of the system design and the property. Finally, we check the language emptiness of the product, and report the verification result.

The rest of this paper is organized as follows: Section 2

GLSVLSI'03, April 28–29, 2003, Washington, DC, USA.

reviews some preliminaries on MDG and $\mathcal{L}_{\mathcal{MDG}}^*$. Section 3 presents the methodology of language emptiness checking using MDGs with a simple example. Section 4 proposes the language emptiness checking algorithm. Section 5 presents some experimental results. Finally, Section 6 concludes the paper.

2. PRELIMINARIES

2.1 Multiway Decision Graphs

An MDG is a finite directed acyclic graph G where the leaf nodes are labeled by formulas, the internal nodes are labeled by terms, and edges issuing from an internal node N are labeled by the terms of the same sort as the label of N [2]. Such a graph represents a formula defined inductively as follows. (i) If G consists of a single leaf node by a formula P, G represents P. (ii) If G has a root labeled X with edges labeled $A_1, \ldots, A_n$ leading to subgraphs $G_1, \ldots, G_n$, and each G_i represents P_i, then G represents the formula $\bigvee_{1\leq i\leq n}((x = A_i) \wedge P_i)$. MDGs efficiently represent relations as well as sets of states.

The MDG tools accept a Prolog-style hardware description language called MDG-HDL. MDG-HDL allows the use of abstract variables for representing data signals, and supports structural descriptions, behavioral descriptions, or the mixture of structural descriptions 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 of state machines [10].

2.2 Abstract Büchi Automaton

MDGs are intended to represent ASMs [2]. An abstract state machine is obtained by letting some data input, state or output variables be of an abstract sort, and the datapath operations be uninterpreted function symbols. An ASM is described by a tuple $D = (X, Y, Z, F_I, F_T, F_O)$, where X, Y, and Z are pairwise disjoint sets of the input, state, and output variables, respectively. While F_I is an MDG representing the set of initial states, F_T is an MDG representing the transition relation, and F_O is an MDG representing the output relation.

We define an *Abstract Büchi Automaton* (ABA) as an ASM with the acceptance condition consisting of several sets of acceptance states which are subsets of states set. Thereafter, an ABA is a tuple $\Lambda = (X, Y, F_I, F_T, C)$, where X, Y, F_I, F_T denote the same as in the definition of an ASM, and $C = \{c_1, c_2, \ldots, c_n\}$ is a set of MDGs representing acceptance condition.

2.3 A First-Order LTL: $\mathcal{L}_{\mathcal{MDG}}^*$

$\mathcal{L}_{\mathcal{MDG}}^*$ is a subset of the first order temporal logic of formula $\mathcal{L}_{\mathcal{MDG}}$ [9]. $\mathcal{L}_{\mathcal{MDG}}^*$ atomic formulas are Boolean constants True and False, or equations of the form $t_1 = t_2$, where t_1 is an ASM system variable (input, output or state variable) and t_2 is either an ASM system variable, an individual constant, an ordinary variable or a function of ordinary variables. Ordinary variables are defined to remember the values of the system variables in the current state. The basic formulas (called $Next\_let\_formulas$) in which only the temporal operator X (next time) is allowed are defined as follows.

- Each atomic formula is a $Next\_let\_formula$,

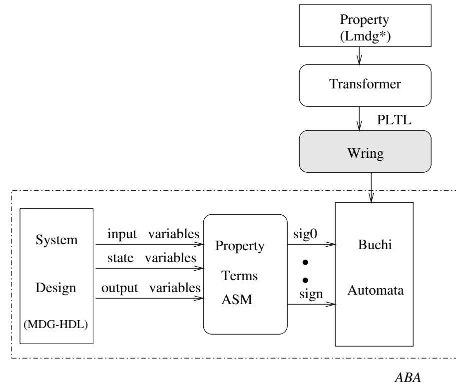

Figure 1: Language Emptiness Checking using MDGs.

- If p, q are $Next\_let\_formulas$, then so are $\neg p$ (*not p*), $p \wedge q$ (*p and q*), $p \vee q$ (*p or q*), $p \rightarrow q$ (*p implies q*), Xp (*next p*), and $\mathsf{LET}\ (v = t)\ \mathsf{IN}\ p$, where t is a system variable and v an ordinary variable.

Using the temporal operators G (*always*), F (*eventually*) and U (*until*), the properties allowed in $\mathcal{L}_{\mathcal{MDG}}^*$ can be of the following form:

$Property ::=$ $(Next\_let\_formula)$; or $(\mathsf{G}\ Next\_let\_formula)$; or $\mathsf{F}\ (Next\_let\_formula)$; or $(Next\_let\_formula\ \mathsf{U}\ Next\_let\_formula)$; or $\mathsf{G}\ (Next\_let\_formula \rightarrow \mathsf{F}\ Next\_let\_formula)$; or $\mathsf{G}\ Next\_let\_formula \rightarrow (Next\_let\_formula\ \mathsf{U}\ Next\_let\_formula)$;

3. THE PROPOSED METHODOLOGY

MDG LEC accepts an ASM modeling the system design and a $\mathcal{L}_{\mathcal{MDG}}^*$ describing the property. To conduct the language emptiness checking using MDGs, we propose a procedure as shown in Figure 1. There are four steps in the procedure: (1) use a Transformer of $\mathcal{L}_{\mathcal{MDG}}^*$ formulas into PLTL formulas, (2) use the Wring tool [1] to generate a BA corresponding to the transformed PLTL formula, (3) compose the system design, property terms ASM, and BA into a composed machine. The *property terms ASM* is a device to realize the transformation from the $\mathcal{L}_{\mathcal{MDG}}^*$ to PLTL. Finally, (4) use a dedicated algorithm to check the language emptiness of the composed ABA.

For narrative simplicity, we illustrate the procedure following a simple example that deals with an abstract counter [9]. One property for this specification is: In state *fetch_st*, if the instruction input is *no_op*, the counter *pc* should keep its value in the next state. The $\mathcal{L}_{\mathcal{MDG}}^*$ description is: **Prop1:** $G((state = fetch\_st \wedge input = no\_op) \rightarrow LET(v = pc)\ IN\ X\ (pc = v))$.

We first use the Transformer to transform the $\mathcal{L}_{\mathcal{MDG}}^*$ formula into the PLTL formula and negate it. This function is implemented using two rewriting rules: (i) each atomic formula $t_1 = t_2$ is rewritten as a proposition predicate, say, $p = 1$, and (ii) each $Next\_let\_formula$ with an abstract variable described as $\mathsf{LET}\ (v = t)\ \mathsf{IN}\ p$ is rewritten as a proposition predicate, say $q = 1$. With these rules, an $\mathcal{L}_{\mathcal{MDG}}^*$ for-

[1] Wring constructs Büchi automata for *PLTL* formulas that are composed of a set of atomic propositions, the standard Boolean connectives, and the temporal operators X, U, F, G and R (*release*) [7].

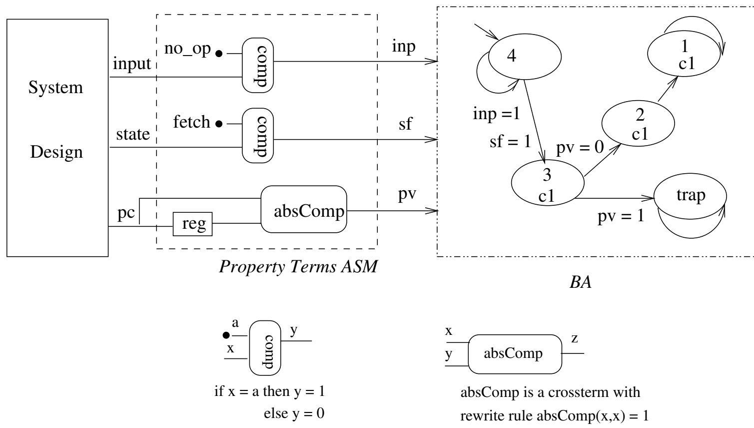

Figure 2: An Example of Composed State Machine.

mula can be transformed into PLTL. Therefore, Prop1 can be rewritten as: $G(sf = 1 \ \wedge \ inp = 1 \rightarrow \ X \ (pv = 1))$. We then put the transformed formula into the Wring tool to get a BA which composes all the languages violating the property. The BA of our example is shown in Figure 2, in which the acceptance states are labeled by c_1. Next, we build the composed machine by hooking the acquired BA with the system design using the *property terms ASM*. Two basic components *comp* and *absComp* are used as depicted in Figure 2. Component *comp* is used to compare concrete sort variable *state* with *fetch_st*, if $state = fetch\_st$, then $sf = 1$; otherwise $sf = 0$. Similarly, by component *comp*, we get $inp = 1$ if $input = no\_op$. To realize $LET(v = pc)\ IN\ X\ (pc = v)$, we use *reg* and *absComp*. *Reg* is used to store the previous counter *pc* value; and *absComp* is used to compare the present *pc* value with the value of *pc* at previous time. These components can be described in MDG-HDL using a netlist of components and tabular behavior description. Finally, we check the language emptiness of the composed machine using the LEC algorithm illustrated in the next section.

4. CHECKING ALGORITHM

The kernel of the language checking algorithm is to find a revisited state which satisfies every set c_i of the acceptance condition C. Simply, we call the states satisfying the condition as *fair states*; others as *unfair states*. In every cycle, we compute the set of fair states, and record them. The set of fair states includes those reachable from the unfair states. Then, we check if there is a revisited fair state. If it is the case, the checking fails. On the other hand, if there exists no fair states, the checking succeeds.

The proposed language emptiness checking algorithm is shown below, where X, Y, F_I, F_T, and C are the set of input variables, the set of state variables, the set of initial states,the transition relation, and the set of acceptance condition, respectively. In this algorithm, the procedure *FairStates* is used to get the set of fair states from the input states. Similarly, *UnfairStates* is to get the set of unfair states. Procedure *ReUnfairStates* is a reachability analysis computing all the unfair states reachable from the input states. *ReFairStates* computes fair states reachable from the input states in one transition step. *Union* computes the disjunction of two sets. *ReVisited* is used to check if there are common states. If it is the case, return failure. Otherwise, add $\{S\}$ into Σ. *NewInputs* generates new inputs, and *NextState* computes the reachable states in the next transition step.

LEC (X, Y, F_I, F_T, C)
begin
K:= 1; $\Sigma := \emptyset$; $S := F_I$;
loop
if $S =$ False then return **success**;
else
for each $c_i \in C$
{
$S_{i_1} :=$ FairStates(S, c_i);
$S_{i_2} :=$ UnfairStates(S, c_i);
$S_{i_3} :=$ ReunfairStates(S_{i_2}, K, F_T, c_i);
$S_{i_4} :=$ RefairStates(S_{i_3}, K, F_T, c_i);
$S =$ Union(S_{i_1}, S_{i_4});
}
if ReVisited$(S, \Sigma) =$ True return **failure**;
else
$\Sigma := \Sigma \cup \{S\}$;
$I :=$ NewInputs(K);
$S :=$ NextState(I, S, F_T);
$K := K + 1$;
end loop;
end

5. EXPERIMENTAL RESULTS

To illustrate the efficiency of our method, we conducted some experiments on a case study design, Island Tunnel Controller (ITC) [4]. Furthermore, we made a comparison between our MDG LEC, MDG MC, VIS [3] and FormalCheck [6].

To model the ITC with ASMs, we define two concrete sorts: *mi_sort* with individual constants {*green, red, entering, exiting*} and *ts_sort* with individual constants {*dispatch, iuse, muse, iclear, mclear*}, one abstract sort *wordn*, and four function symbols $lessn : wordn \rightarrow bool$, $equz : wordn \rightarrow bool$ with $inc : wordn \rightarrow wordn$ and $dec : wordn \rightarrow wordn$. We use *tc* and *ic* are defined as variables of sort *wordn* to describe the number of cars in the island and tunnel, respectively. Other input and output variables are defined as *bool* sort. Since VIS and FormalCheck do not accept abstract variables, we model the counters *ic* and *tc* as 4-bits,

Table 1: Experimental Results on ITC with MDG LEC and MDG MC

	MDG LEC			MDG MC		
	CPU Time (sec)	Memory (MB)	# MDG Nodes	CPU Time (sec)	Memory (MB)	# MDG Nodes
Prop1	3.06	2.2	3579	6.3	3.45	6613
Prop2	3.2	2.1	3592	6.3	3.38	6636
Prop3	3.03	3.4	3592	5.88	4.38	6636
Prop4	**4.25**	**2.8**	**4679**	**12.5**	**7.7**	**12321**
Prop5	4.97	2.9	5637	10.67	6.66	11479

Table 2: Experimental Results of Prop4 on ITC with VIS and FormalCheck

	VIS			FormalCheck		
	CPU Time (sec)	Memory (MB)	# BDD Nodes	CPU Time (sec)	Memory (MB)	# States
4 bits	2.5	6.0	8.7e+04	5	5.2	4.16e+06
6 bits	8.0	15.6	4.27e+05	12	5.6	6.68e+07
8 bits	174.5	52.5	1.30e+07	35	7.7	1.07e+09
10 bits	5838	133.6	2.15e+08	174	16.3	1.72e+10
12 bits	26992	614	1.1e+09	817	50.4	2.75e+11
16 bits	-	-	-	29940	696	1.10e+12
18 bits	-	-	-	-	-	-

6-bits, 8-bits, 10-bits, 12-bits, 16-bits and 18-bits vectors. We verified five properties on a 296 MHZ Sun Ultra-2 workstation with 768MB of memory. The experimental results are shown in Table 1 and Table 2. Table 1 gives the performance for verifying the five properties on the model with counters defined as abstract variables by MDG LEC and MDG MC. Table 2 illustrates the performance of verifying one sample property (Prop4) on the models with different counter widths by VIS and FormalCheck ("-" means did not terminate).

Table 1 shows that our new developed algorithm works well. Compared with the MDG MC, our MDG LEC consumes few resources for the verification of the model with abstract counters. Table 2 indicates that while the vector bits of the counters increase, the resource used increase rapidly. Consequently, the vector bits of the counters cannot increase infinitely, due to the limited resources. The experimental results show that when the counters increase to 18-bits, both FomralCheck and VIS cannot terminate after half a day's run (12 hours).

6. CONCLUSIONS

In this paper, we presented a language emptiness checking approach using multiway decision graphs. The proposed procedure transforms first-order temporal formulas $\mathcal{L}_{\mathcal{MDG}}{}^*$ into PLTL using two rewriting rules, then uses the Wring tool to generate a Büchi automaton for the transformed PLTL. Next, it composes a product automaton using the *property terms ASM.* Finally, it applies the language emptiness checking algorithm. We performed some experiments on a simple abstract counter and a more elaborated ITC benchmark. The experimental results show that the proposed algorithm provides much better performance than the existing MDG MC. It also shows superiority over VIS and FormalCheck. In fact, our MDG LEC accepts abstract variables while ROBDD based tools, such as VIS and FormalCheck, have to specify signals as Boolean vectors, which is a key feature of MDGs.

7. REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[2] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, 10(1):7–46, 1997.

[3] R. K. B. et. al. Vis: A system for verification and synthesis. In *Computer Aided Verfication*, LNCS 1102, pages 428–432, Springer Verlag, 1996.

[4] K. Fisler and S. Johnson. Integrating design and verification environments through a logic supporting hardware diagrams. In *IFIP Conference on Hardware Description Language and their Applications*, Chiba, Japan, 1995.

[5] C. Kern and M. R. Greenstreet. Formal verification in hardware design: a survey. *Transactions on Design Automation of Electronic Systems*, 4:123–193, 1999.

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

[7] F. Somenzi and R. Bloem. Efficient buchi automata from ltl formulae. In *Computer Aided Verification*, LNCS 1877, pages 247–263, Springer Verlag, 2000.

[8] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, LNCS 1043, pages 238–266, Springer Verlag, 1996.

[9] Y. Xu, E. Cerny, X. Song, F. Corella, and O. A. Mohamed. Model checking for a first-order logic using multiway decision graphs. In *Computer Aided Verification*, LNCS 1427, pages 219–231, Springer Verlag, 1998.

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