Formal Verification of the Island Tunnel Controller using Multiway Decision Graphs

Z. Zhou[4], X. Song[1], S. Tahar[2], E. Cerny[1], F. Corella[3] and M. Langevin[4]

[1] D'IRO, Université de Montréal, Canada, {song,cerny}@iro.umontreal.ca
[2] Dept. of ECE, Concordia University, Canada, tahar@ece.concordia.ca
[3] Hewlett-Packard Company, USA, fcorella@hprpcd.rose.hp.com
[4] Nortel Technology, Canada, {zzhou,mlangevin}@nortel.ca

Abstract. *Multiway Decision Graphs* (MDGs) have recently been proposed as an efficient representation tool for RTL designs. In this paper we demonstrate the MDG-based formal verification technique on the example of the Island Tunnel Controller. We also provide comparative experimental results for the verification of a number of properties using two well-known ROBDD-based verification tools SMV (Symbolic Model verifier) and VIS (Verification Interacting with Synthesis). Finally, we study in detail the non-termination problem of the abstract state enumeration and present an solution.

1 Introduction

For automated hardware verification, ROBDDs [1] have proved to be a powerful tool [4, 10]. However, since they require a Boolean representation of the circuit, the size of an ROBDD grows, sometimes exponentially, with the number of Boolean variables. Therefore, ROBDD-based verification cannot be directly applied to circuits with complex datapaths.

There are two different ways to deal with this problem. One is to reduce the complexity of hardware designs by means of data abstraction [6, 14]. Another way is to represent hardware designs at different levels of abstraction, and verify the designs hierarchically. Recently, a number of ROBDD extensions such as BMDs [2] and HDDs [7] have been developed to represent arithmetic functions more compactly than ROBDDs. There also emerged a number of methods [8, 5, 11] which verify the overall functionality of Register-Transfer Level designs at an abstract level, using *abstract* variables to denote data signals and *uninterpreted* function symbols to denote data operations.

We have proposed a new class of decision graphs, called *Multiway Decision Graphs* (MDGs) [8], 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 a vector of 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 datapath.

In [8] we described a basic set of MDG operators and a reachability analysis algorithm based on *abstract implicit enumeration*. The reachability algorithm verifies whether an invariant holds in all reachable states of an *abstract state machine* [9]. One application of the algorithm is the verification of observational equivalence of synchronous circuits.

Burch and Dill [5, 13] proposed a validity checking algorithm for processor verification which is also based on the use of abstract sorts and uninterpreted function symbols. A logic expression representing the correctness statement is generated using symbolic simulation. The algorithm is then used to check its validity. With carefully chosen heuristics for avoiding exponential case splitting, the authors verified a subset of a RISC pipeline processor DLX [5] and a protocol processor (PP) [13]. Cyrluk and Narendran [11] defined a first-order temporal logic – Ground Temporal Logic (GTL) which also uses uninterpreted function symbols. Using a decidable fragment of GTL, they can automate the verification in the PVS theorem prover. These methods, however, are not applicable to verification problems that require state-space exploration, because they can only deal with paths of fixed length. They cannot represent sets of states and compute fixpoints on sets of states. MDGs, on the other hand, provide a tool for both validity checking and verification based on state-space exploration.

In this paper we present a case study of MDG-based hardware verification using the Island Tunnel Controller (ITC) [12] as an example. This example was originally used to illustrate the notation of a heterogeneous logic system supporting diagrams as logic entities [12]. However, no verification experiments were performed. Although the example is small and does not represent the scale of designs we can verify, it is ideal for illustration purposes since it covers most current MDG applications. Moreover, the example is appropriate in size to be used to illustrate in detail the state generalization techniques for dealing with the non-termination problem of our reachability analysis procedure.

The rest of the paper is organized as follows: We briefly review the Multiway Decision Graphs in Section 2 and introduce the Island Tunnel Controller example in Section 3. In Section 4, we describe the hardware modeling of the ITC at an abstract level. In Section 5, we explain the verification of the ITC example including combinational verification, invariant checking and behavioral equivalence checking. We also present experimental results, including a comparison with results obtained using SMV and VIS. Section 6 studies in detail the non-termination problem of the abstract state enumeration and presents several solutions. Finally, Section 7concludes the paper.

2 MDGs and MDG-based Verification Approaches

The formal logic underlying MDGs is a many-sorted first-order logic, augmented with the distinction between *abstract sorts* and *concrete sorts*. This is motivated by the natural division of datapath and control circuitry in RTL designs. Concrete sorts have *enumerations* which are sets of *individual constants*, while abstract sorts do not. Variables of concrete sorts are used for representing control signals, and variables of abstract sorts are used for representing datapath signals. Data operations are represented by uninterpreted function symbols. An n-ary function symbol has a type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$, where $\alpha_1 \ldots \alpha_{n+1}$ are sorts. The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. Let f be a function symbol of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$. If α_{n+1} is an abstract sort then f is an *abstract function symbol*. If all the $\alpha_1 \ldots \alpha_{n+1}$ are concrete, f is a *concrete* function symbol. If α_{n+1} is concrete while at least one of $\alpha_1 \ldots \alpha_n$ is abstract, then we refer to f as a *cross-operator*; cross-operators are useful for modeling feedback signals from the datapath to the control circuitry.

A *multiway decision graph* (MDG) is a finite, directed acyclic graph (DAG). An internal node of an MDG can be a variable of a concrete sort with its edge labels be 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* (whose function symbol is a cross-operator). An MDG may only have one leaf node denoted as **T**, which means all paths in the MDG are true formulas. Thus, MDGs essentially represent relations rather than functions. Just as Bryant's ROBDDs [1] must be *reduced* and *ordered*, MDGs must also be reduced and ordered, and obey a set of other well-formedness conditions given in [8]. We developed a set of MDG algorithms for computing *disjunction*, *relational product* (*Conjunction* followed by *existential quantification* [4]) and *pruning-by-subsumption*. A detailed description of the algorithms can be found in [8].

A state machine is described using finite sets of input, state and output variables, which are pairwise disjoint. The behavior of a state machine is defined by its transition/output relations, together with a set of initial states. An *abstract description* of the state machine, called *abstract state machine* (ASM) [9], is obtained by letting some data input, state or output variables be of an abstract sort, and the datapath operations be uninterpreted function symbols. Just as ROBDDs are used to represent sets of states, and transition/output relations for finite state machines, MDGs are used to compactly encode sets of (abstract) states and transition/output relations for abstract state machines. We thus lift the implicit enumeration [4, 10] technique from the Boolean level to the abstract level, and refer to it as *implicit abstract enumeration* [8]. Starting from the initial set of states, the set of states reached in one transition is computed by the relational product operation. The frontier-set of states is obtained by *pruning* (removing) the already visited states from the set of newly reached states using pruning-by-subsumption. If the frontier-set of states is empty, then a least fixed point is reached and the reachability analysis procedure terminates. Otherwise, the newly reached states are merged (using disjunction) with the already visited states and the procedure continues the next iteration with the states in the frontier-set as the initial set of states.

One variation of the reachability analysis is invariant checking and the verification of observational equivalence of two ASMs verified as an invariant of the product ASM. When an invariant is violated at some stage of the reachability analysis, a counterexample is given as a sequence of input-state pairs leading from the initial state to the faulty behavior. We also have an equivalence checking procedure for combinational circuits that takes advantage of the canonicity of MDGs [8]. The MDG operators and verification procedures are packaged as MDG tools [18].

3 The Island Tunnel Controller (ITC)

The Island Tunnel Controller (ITC) example was originally introduced by Fisler and Johnson [12]. There is a one lane tunnel connecting the mainland to a small island, as shown in Figure 1. At each end of the tunnel, there is a traffic light. There are four sensors for detecting the presence of vehicles: one at tunnel entrance (*ie*) and one at tunnel exit on the island side (*ix*), and one at tunnel entrance (*me*) and one at tunnel exit on the mainland side (*mx*). It is assumed that all cars are finite in length, that no car gets stuck in the tunnel, that cars do not exit the tunnel before entering the tunnel, that cars do not leave the tunnel entrance without traveling through the tunnel, and that there is sufficient distance between two cars such that the sensors can distinguish the cars.

In [12], an additional constraint is imposed: "at most sixteen cars may be on the island at any time". The number "sixteen" can be taken as a parameter and it can be any natural number. Thus the constraint may be read: "at most n ($n \geq 0$) cars may be on the island at any time". With ROBDD-based verification, the performance depends on the particular instance of n, as we shall see in Section 5.3 .

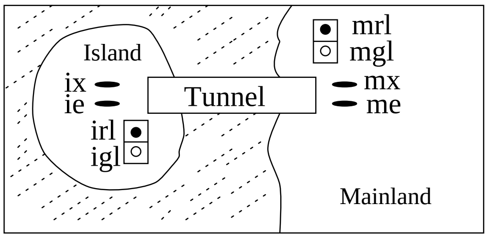
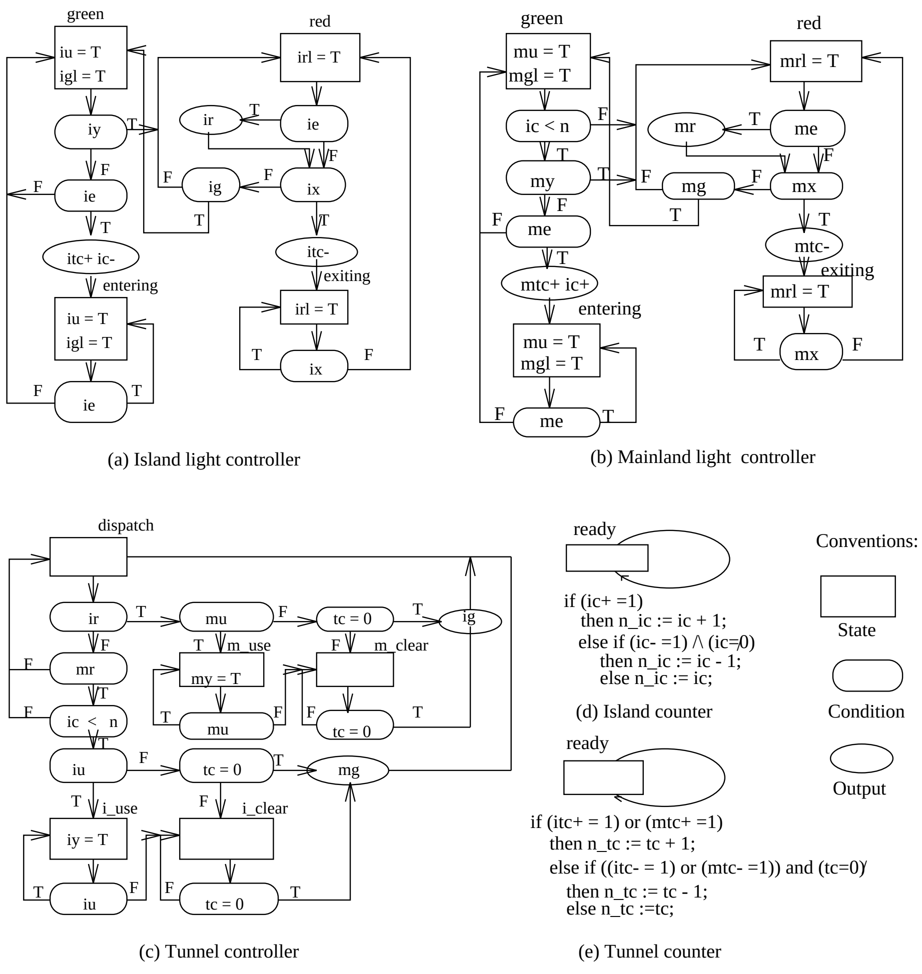


Fig. 1. The Island Tunnel Controller

Fisler and Johnson [12] proposed a specification of ITC using three communicating controllers. Their state transition diagrams are shown in Figure 2 (Figures 2(a), (b) and (c) are taken from [12] except that 16 is replaced by n). The island light controller (ILC) (Figure 2(a)) has four states: *green, entering, red* and *exiting.* The outputs *igl* and *irl* control the green and red lights on the island side, respectively; *iu* indicates that the cars from the island side are currently occupying the tunnel, and *ir* indicates that ILC is requesting the tunnel. The input *iy* requests ILC to release control of the tunnel, and *ig* grants control of the tunnel. A similar set of signals is defined for the mainland light controller (MLC). The tunnel controller (TC) processes the requests for access issued by the ILC and MLC. The island and the tunnel counters keep track of the numbers of cars currently on the island and in the tunnel, respectively. For the tunnel counter, at each clock cycle, the count tc is increased by 1 depending on signals $itc+$ and $mtc+$ or decremented by 1 depending on $itc-$ and $mtc-$ unless it is already 0. The island counter operates in a similar way, except that the increment and decrement signals are $ic+$ and $ic-$, respectively.



(a) Island light controller

(b) Mainland light controller

(c) Tunnel controller

(d) Island counter

(e) Tunnel counter

Fig. 2. The state transition diagrams of the Island Tunnel Controller

4 Hardware Description of the Island Tunnel Controller

We take as *specification* the ITC state transition diagrams in Figure 2. In this section, we show how they are modeled as abstract state machines.

Both the island and the tunnel counters have each only one control state, *ready*, hence no control state variable is needed. An abstract state variable ic (tc) represents the current count number. At each clock cycle, the count is updated according to the control signals. In this abstract description of a counter, the count ic (tc) is of abstract sort, say *wordn* for n-bit words. The control signals ($ic+, ic-$, etc.) are of *bool* sort with the enumeration $\{0, 1\}$. The uninterpreted function *inc* of type $[wordn \rightarrow wordn]$ denotes the operation of increment by 1, and *dec* of the same type denotes decrement by 1. The cross-function $equz(tc)$ of type $[wordn \rightarrow bool]$ represents the condition "$tc = 0$" and models the feedback from counter to the control circuitry. Figure 3 shows the MDG of the transition relation of the tunnel counter for a specific variable order.

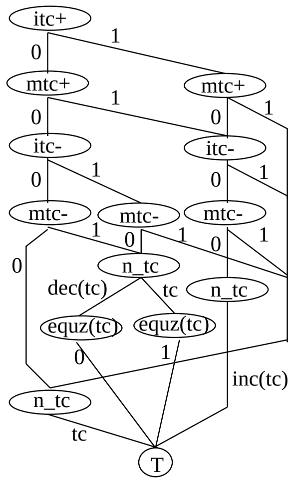


Fig. 3. Transition relation MDG of the tunnel counter

Each of the controllers can have a single control state variable which takes all the possible states as its values. Thus the enumeration of those states constitutes the (concrete) sort of the variable. Let is, ms and ts be the control state variables of the three controllers ILC, MLC and TC. We assign the variables is and ms (and also their next state variables n_is and n_ms) the sort mi_sort having the enumeration $\{green, red, entering, exiting\}$. Similarly, we let variables ts and n_ts be of sort ts_sort which has the enumeration $\{dispatch$, i_use, m_use, i_clear, $m\_clear\}$. All other control signals (ie, ix, me, mx, etc) are of sort *bool*. The condition "$ic < n$" is represented by the cross-function $lessN(ic)$ of type $[wordn \rightarrow bool]$.

We consider an *implementation* as a netlist of components connected by signals. It is described using predefined component definitions in MDG–HDL which is the input language for our tools. The specification is also modeled as an ASM and represented using MDGs, which is compiled from the MDG-HDL descriptions.

5 Verification of the Island Tunnel Controller

In this section, we explain the various verification experiments performed on the ITC example using our MDG tools. All experiments (including those using SMV and VIS) were done on a Sun SPARCstation 10. In the result tables, column *Time* is the CPU time in seconds used for compiling the circuit descriptions and for the reachability analysis, including counterexample generation if necessary. Column *Mem* is the memory allocated in megabytes. Column *#Nodes* is the number of total MDG (or ROBDD) nodes generated.

5.1 Equivalence Checking

In [12], two ILC implementations are given, denoted as I_1 and I_2 where the later is a simplified version of I_1. We first compared I_1 and I_2 of the ILC against the behavioral specification (Figure 2(a)). We found one design error in I_1 and two errors in I_2. For more details, refer to [19]. All the errors were corrected in half an hour with the help of the counterexample facility. These errors were confirmed by the designer. Since MLC is similar to ILC, we also derived the implementations for MLC from the ILC implementations, and compared them with the MLC behavioral specification (Figure 2(b)).

The execution time and resource requirements of the experiments are given in Table 1. Rows *Faulty* I_1 *vs. Spec.* and *Corrected* I_1 *vs. Spec.* are the verifications of the implementation I_1 with and without the original errors against the behavioral specification. Similarly for rows *Faulty* I_2 *vs. Spec.* and *Corrected* I_2 *vs. Spec.*.

5.2 Combinational Verification

For both MLC and ILC, it is possible to check the equivalence of implementation I_1 with the simplified version I_2 using combinational verification, since the registers in the two circuits have one-to-one correspondence. For combinational verification, we compare two circuits having the same input, output and state variables (if any). For each circuit, we derive an MDG relating each output and state variable to the inputs and state variables (These MDGs collectively represent the output and transition relations). Then, using the canonicity property of MDGs, we simply check that corresponding MDGs for the two circuits have the same MDG identifier. The experimental results for comparing the corrected I_1 and I_2 are given in the last row in Table 1. The execution time is essentially the compilation time for constructing the transition and the output relation MDGs.

Table 1. Equivalence checking and combinational verification

	Island Light Controller			Mainland Light Controller		
	Time(sec)	Mem(MB)	#Nodes	Time(sec)	Mem(MB)	#Nodes
Faulty I_1 vs. Spec.	3.4	1.5	1015	5.0	1.7	1361
Corrected I_1 vs. Spec.	3.4	1.5	988	5.0	1.6	1292
Faulty I_2 vs. Spec.	2.8	1.2	833	3.6	1.5	992
Corrected I_2 vs. Spec.	2.7	1.2	806	3.6	1.6	988
Corrected I_1 vs. I_2	2.7	1.4	783	3.7	1.5	1108

5.3 Checking Invariants on the ITC Specification

Property checking is useful for verifying that a specification satisfies certain requirements. We list below three example properties (invariants) that we verified[5]. We also provide their CTL (Computational Tree Logic) formulas which are used for invariant checking by the ROBDD-based verification tools SMV and VIS.

P1: Cars never travel both in directions in the tunnel at the same time. AG (! ((igl=1) & (mgl=1))).

P2: The tunnel counter is never signaled to increment simultaneously by ILC and MLC. AG (!(($itc+ = 1$) & ($mtc+ = 1$))).

P3: The island counter is never signaled to increment and decrement simultaneously. AG (! (($ic- = 1$) & ($ic+ = 1$))).

For the purpose of comparison, we first show the experimental results for the verification of the above example invariants (P1, P2 and P3) using FSM-based methods. For MDG tools, the counts tc and ic are now assigned a concrete sort according to the counter width which is determined by the instantiation of the constraint, i.e., the maximum number of cars that are allowed on the island. In addition, we also used two well-known ROBDD-based tools: SMV (V2.4.4) [16] and VIS [3].

Table 2 shows the results for checking the conjunction of P1, P2 and P3 on various counter widths. For the SMV columns, the $Time$ is the *user* time, while for VIS and MDG columns, it is the *elapsed* time including loading the Verilog or MDG-HDL description file, compilation and invariant checking. For SMV and VIS, we used the node ordering generated by the systems, while for MDG tools, we used manual ordering since no heuristic ordering algorithm is available for the moment. Many different factors influence the experimental results shown in the table. The three tools use different integer encodings, different variable ordering, and different partitioning of the transition relation. But notwithstanding these differences, the table clearly shows the following: (i) Time increases exponentially with the counter width for concrete representations of the problem, and

[5] Fisler and Johnson [12] proposed a set of properties that the ITC design should satisfy. Currently, we only consider the verification of invariants.

(ii) the MDG figures are substantially greater than the others for concrete representations. This is because the MDG data structure and the MDG algorithms are far more complicated than those of ROBDDs.

However, MDG tools are able to verify a parameterized implementation having n bits for the counter width, which is not the case for SMV and VIS. The last row in Table 2 gives the results for verifying the properties when we model the design as an ASM rather than an FSM using MDGs. The verification is performed quite efficiently in time independent of the datapath width.

Table 2. ITC Invariant checking. ('–' means that the verification did not terminate in certain amount of time, and '*' means that the verification is not possible.)

	SMV			VIS			MDG		
Counter Width	Time (sec)	Mem (MB)	#Nodes	Time (sec)	Mem (MB)	#Nodes	Time (sec)	Mem (MB)	#Nodes
4 bits	1.2	1.2	10043	15.4	0.5	6492	430	8	19670
5 bits	4.1	1.2	10463	18.9	0.5	3887	810	10	27668
6 bits	16.7	1.2	11240	44.5	0.6	8902	1719	15	41751
7 bits	79.7	1.2	15047	429.9	1.2	33447	5486	26	69911
8 bits	360	1.6	29474	1686	2.4	43428	–	–	–
9 bits	1564	2.1	59292	7584	5.1	128426	–	–	–
10 bits	6263	3.2	117890	31255	9.9	327090	–	–	–
11 bits	–	–	–	–	–	–	–	–	–
n bits	*	*	*	*	*	*	55	2.7	4329

It may be argued that the data abstraction method [6, 14] is sufficient to imply the correctness of this ITC example, i.e., we reduce n to a small number encoded by a few bits, e.g., 2 bits (4), 4 bits (16), etc. Yet in general, the equivalence of the reduced circuit against the original one is not verified mechanically. Also, it is not always obvious how to construct an appropriate data abstract function, or such data abstraction may not even be possible. One such example is the 4×4 Fairisle ATM switch fabric recently verified using MDGs [15, 17], where the datapath contains mixed data/control information. In general, if the control information needs n bits, then it is impossible to reduce the word width to less than n. Hence, in this case ROBDD-based datapath reduction technique is no more practical. On the other hand, using the MDG-based approach, we naturally allow the abstract representation of data while the control information is extracted from the datapath using cross-functions.

Invariant checking is based on reachability analysis of state machines. However, the reachability analysis of ASMs may not terminate in general [8] which is the case for the ITC example. To deal with this problem, we developed a heuristic state generalization method to be explained next.

6 Termination of Abstract State Enumeration

6.1 Review

An abstract state machine may have an infinite number of states due to the abstract variables and the uninterpreted nature of function symbols. Thus a least fixed point may not be reached in state enumeration. In [8], the non-termination problem is explained and a solution is given to a class of interesting problems known as *processor-like* circuits by generalizing the initial state (i.e., by replacing abstract constants with abstract variables as initial values of some of the registers). Informally, a processor-like circuit usually starts from a *ready* state, performs data operations in one or more cycles and then returns to the *ready* state. We first briefly review the initial state generalization method using the tunnel counter (Figure 2(e)), which is a processor-like circuit, as an example.

Let the initial state of the tunnel counter be a generic constant zero of the abstract sort *wordn*. Figure 4(a) shows the MDGs N_0, S_1 and N_1 representing the set of initial states, the set of states reached in one transition, and the frontier-set of states, respectively. There are three paths in S_1, i.e., there are three abstract states. The state $tc = \mathsf{zero}$ is subsumed by N_0 and is thus pruned. The state $(tc = dec(\mathsf{zero})) \wedge (equz(\mathsf{zero}) = 0)$ is, in fact, an unreachable state. If we rewrite $equz(\mathsf{zero})$ to an individual constant 1 using the rewrite rule $equz(\mathsf{zero}) \rightarrow 1$, it yields a contradiction (1=0) and the path can be eliminated. The frontier-set of states N_1 thus contains the only state $tc = inc(\mathsf{zero})$. If we continue the reachability analysis with N_1, at the k-th iteration we will have the state $tc = inc^k(\mathsf{zero})$ (where $inc^k(\mathsf{zero})$ is a short hand for $inc(....(inc(\mathsf{zero}))...))$. This illustrates the non-termination problem due to the fact that the terms that label the edges can be arbitrarily large and hence arbitrarily many.

The non-termination problem can be avoided by *generalizing* the initial value zero of the state variable tc to a *fresh* [6] *abstract* variable a. Then, as shown in Figure 4(b), the states $tc = inc(a)$ and $(tc = dec(a)) \wedge (equz(a) = 0)$ in S_1 become instances of $tc = a$ under substitutions $inc(a)/a$ or $dec(a)/a$, respectively. They thus can be pruned and the state exploration procedure terminates right away (the MDG $N_1 = \mathbf{F}$ represents an empty set). Note that the method may have false negatives since the reachable state space is enlarged by the state generalization. For instance, the state $(tc = dec(a)) \wedge (equz(a) = 0)$ becomes a reachable state which covers the previously unreachable state $(tc = dec(\mathsf{zero})) \wedge (equz(\mathsf{zero}) = 0)$.

6.2 State Generalization for General Cases

The complete ITC specification is composed of 5 communicating state machines (Figure 2). Among them, the tunnel and the island counters are typical processor-like circuits. However, the composed ASM is no more a processor-like circuit, and the initial state generalization technique is not applicable directly.

[6] A *fresh* variable is disjoint with all other variables.

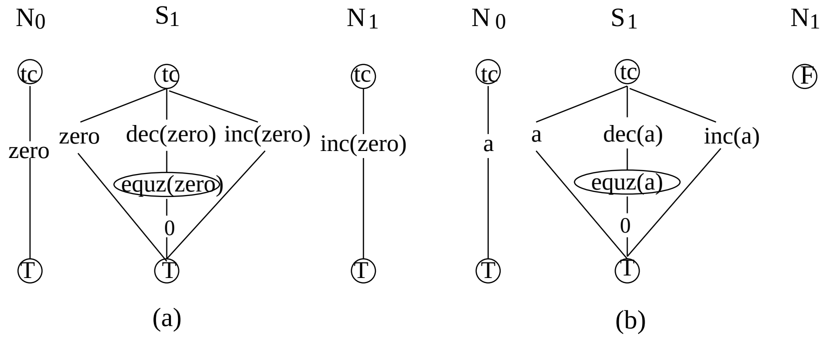


Fig. 4. Non-termination problem and initial state generalization for the tunnel counter.

In the following, we present a new state generalization technique which solves the non-termination problem for a larger class of circuits. To simplify the presentation, we only consider a specification consisting of the ASMs in Figures 2(a), (b), (c) and (e).

Initially, we assume there are no cars in the tunnel. Therefore, the tunnel counter is reset to zero. To be safe, the lights on both the island and the mainland sides are reset to *red*, and the tunnel controller is in the state *dispatch*, ready to take requests. The set of initial states thus includes only one state $(is = red) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = \mathsf{zero})$. In Figure 5, we show a small fraction of the state transition diagram of the composed machine related to the transitions of the ILC between the states *red* and *green*, and the cycles between *green* and *entering*. The Mainland Light Controller and the Tunnel Controller keep their initial state values.

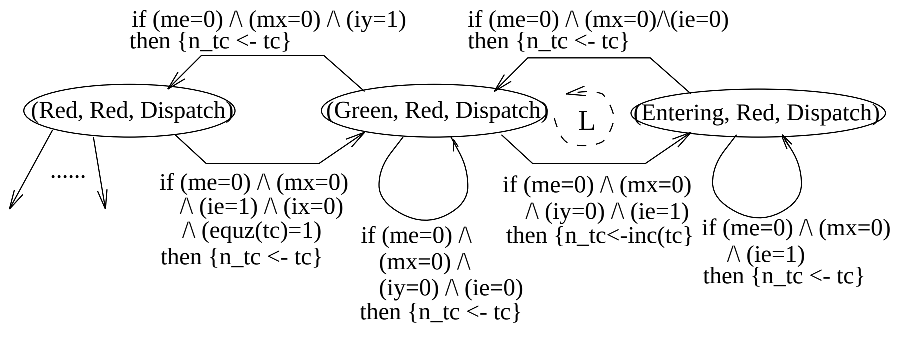


Fig. 5. A fraction of the state transition diagram for the composed machine.

To illustrate the failure of the initial state generalization method on this example, we again try the above method by setting the initial state to $(is =$

$red) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = a)$ where a is an abstract variable of sort *wordn*. Figure 6 shows the MDGs representing the set of newly reached states (S_i, $i = 1, 2, ...$) and the frontier-set of states (N_i, $i = 0, 1, 2, ...$) in three iterations following the transitions depicted in Figure 5.

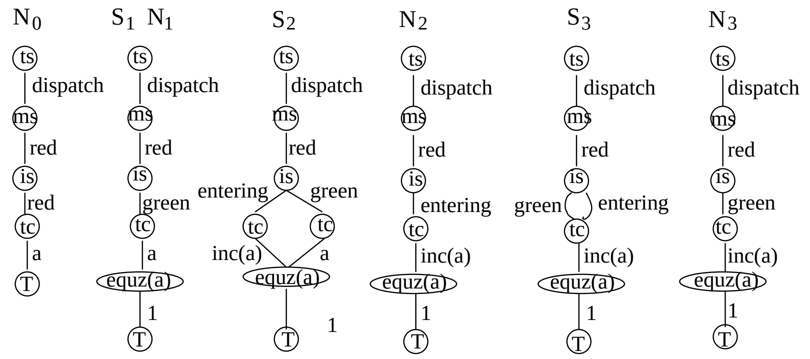


Fig. 6. State enumeration using initial state generalization technique.

Starting from N_0, after three transitions, the set S_3 of newly reached states contains two states: $p: \ (is = green) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = inc(a)) \wedge (equz(a) = 1)$ and $q: \ (is = entering) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = inc(a)) \wedge (equz(a) = 1)$. It can be easily checked that q is an instance of the only state in N_2. Thus, it can be pruned. On the other hand, p is not an instance of any visited states. In the visited states, only the state in N_1 matches p with the same values for control state variables. However, for the abstract state variable and the cross-term, we have a pair of contradictry substitutions $inc(a)/a$ and a/a, which means no valid substitution exists. Therefore, p is considered as a new state in N_3. If we continue the reachability analysis with N_3, the value of tc will become $inc(...inc(a)...)$ with an unbounded number of inc.

A closer examination of this non-termination problem shows that the iterations from N_1 to N_3 represents the loop L (Figure 5) with a data operation inc on the abstract state variable tc. This loop, in fact, resembles the ASM of a processor-like circuit. The state $(is = green) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = a) \wedge (equz(a) = 1)$ in N_1 can thus be considered as the initial (entry) state of loop L. However, N_1 is not a generalized state because of the assumption $equz(a) = 1$, which is the reason for non-termination.

The above analysis leads to the observation of a *processor-like loop*. A processor-like loop starts from a control state and returns to this control state after one or more transitions, with data registers updated according to the data operations. Note that a processor-like circuit is represented by an ASM having one and only one processor-like loop. This suggests that it is the entry state of a processor-like loop that should be generalized rather than the initial state of the state machine. In the above example, we have to generalize the state

$(is = green) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = a) \wedge (equz(a) = 1)$ in N_1 instead of the states in N_0. Figure 7 shows the state enumeration procedure using the extended method. Starting from N_0, S_1 is reached in one transition: $\{(is = green) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = \mathsf{zero}) \wedge (equz(\mathsf{zero}) = 1)\}$, which can be simplified to $\{(is = green) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = \mathsf{zero})\}$ using the rewrite rule $equz(zero) \rightarrow 1$. As this state is the entry state of the processor-like loop L, we generalize the constant value of tc to a *fresh abstract* variable a and remove the constraint $equz(a) = 1$. Thus the frontier set of states becomes $N_1'= \{(is = green) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = a)\}$. After two transitions, the frontier-set N_3 becomes $\{(is = green) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = inc(a))\}$, where the only state in this set is subsumed by N_1' under substitution $inc(a)/a$. The procedure thus terminates.

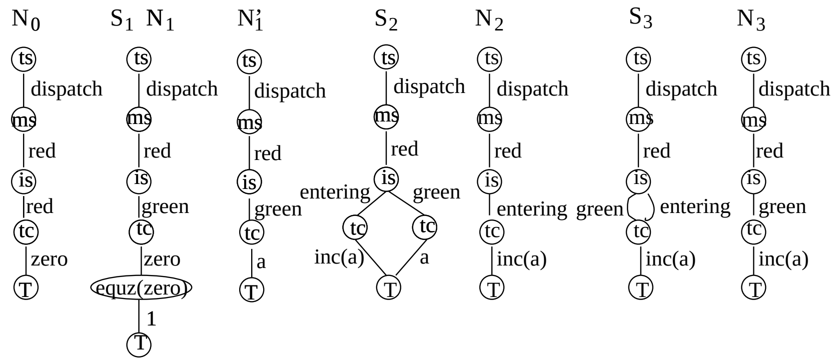


Fig. 7. State enumeration using the extended state generalization technique.

It remains to determine now when and on which state variables the generalization is to be performed, i.e., how to identify processor-like loops and perform the generalization operation. For some circuits, e.g., simple microprocessors, it is possible to identify all processor-like loops by inspection and to manually perform state generalization on the entry states of the loops. However, in general, to find the entry states of all processor-like loops could be very difficult. In the next subsection, we propose a heuristic method.

6.3 Heuristic State Generalization

We developed a heuristic method for state generalization based on the following observation: reachability analysis terminates if we generalize any state within a processor-like loop. Once we generalize a state in a loop, it covers all the abstract states having the same control state values, thus guaranteeing termination.

One method is to perform generalization on every abstract state variable at each clock cycle, i.e., to replace every term that labels an edge issuing from an abstract node with a fresh abstract variable. However, the reachable state space is unnecessarily enlarged since states that are not within processor-like loops are also generalized. As a trade-off, we propose a heuristic solution to this problem: After a certain number of state transitions (specified by the user), if the MDG size of the frontier-set keeps increasing, the value of each state variable in the MDG is generalized. With this heuristic, the state to be generalized is more likely to be within a processor-like loop.

Termination of the abstract state enumeration can be obtained at the cost of false negatives introduced by the state generalization. If the reachability analysis succeeds, we know that the invariant holds even for the enlarged set of reachable states, but if it does not, then we have to examine, e.g., by simulation, whether the counterexample thus produced corresponds to a *real* design error. The heuristic method performs state generalization quite blindly at first, and postpones any manual analysis to the examination of counterexamples, if any.

The results in Table 2 (the last row) are obtained using the heuristic state generalization technique on the complete ITC specification composed of the five ASMs of Figure 2.

7 Conclusions

In this paper, we demonstrated the feasibility of the MDG-based hardware verification at the RT level on a non trivial example–the Island Tunnel Controller. We performed various verification experiments on the example including combinational verification, behavioral equivalence checking and invariant property checking. Using the counterexample facility of the MDG tools, we also showed our ability to identify design errors that were present in the original implementations. Furthermore, we gave a comparative evaluation of the results from invariant checking with the ROBDD-based tools SMV and VIS, and we showed the strength of MDG approach by handling arbitrary data widths. Finally, we studied in detail the non-termination problem of abstract state enumeration and presented a heuristic solution.

The MDG tools are capable of dealing with complex designs. The interested readers may wish to refer to [15, 17] where a case study is presented about the verification of an ATM switch fabric. We are currently developing a model checking algorithm for a restricted first-order temporal logic.

References

1. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.
2. R. E. Bryant and Y. Chen. Verification of arithmetic circuits with binary moment diagrams. In *32nd ACM/IEEE Design Automation Conference (DAC'95)*. San Francisco, California, June 1995.
3. R. K. Brayton et. al. VIS: A system for verification and synthesis. In *Proc. 8th International Conference on Computer-Aided Verification (CAV'96)*. New Brunswick, New Jersey, USA, July 1996.

4. J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, April 1994.
5. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In: D. L. Dill, editor, *Computer Aided Verification*. Lecture Notes in Computer Science 818, Springer Verlag, 1994.
6. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proc. 19th ACM Symp. on Principles of Programming Languages*. January 1992.
7. E. Clarke, M. Fujita and X. Zhao. Hybrid decision diagrams. In *Proc. IEEE Inter. Conf. on Computer-Aided Design (ICCAD'95)*. San Jose, California, USA, Nov. 1995.
8. F. Corella, Z. Zhou, X. Song, M. Langevin and E. Cerny. Multiway decision graphs for automated hardware verification. IBM technical report RC19676, July 1994. To appear in the journal *Formal Methods in System Design*.
9. F. Corella, M. Langevin, E. Cerny, Z. Zhou and X. Song. State enumeration with abstract descriptions of state machines. In *Proc. IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (Charme'95)*. Frankfurt, Germany, October 1995.
10. O. Coudert, C. Berthet and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*. Lecture Notes in Computer Science 407, Springer Verlag, 1989.
11. D. Cyrluk and P. Narendran. Ground Temporal Logic: A logic for hardware verification. In: D. L. Dill, editor, *Computer Aided Verification*. Lecture Notes in Computer Science 818, Springer Verlag, 1994.
12. K. Fisler and S. Johnson. Integrating design and verification environments through a logic supporting hardware diagrams. In *Proc. IFIP Conference on Hardware Description Languages and their Applications (CHDL'95)*. Chiba, Japan, Aug. 1995.
13. R. B. Jones and D. L. Dill. Efficient validity checking for processor verification. In *Proc. IEEE International Conference on Computer-Aided Design (ICCAD'95)*. San Jose, California, USA, November 1995.
14. D. E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie Mellon University, 1993.
15. M. Langevin, S. Tahar, Z. Zhou, X. Song and E. Cerny. Behavioral Verification of an ATM switch fabric using implicit abstract state enumeration. In *Proc. IEEE Inter. Conf. on Computer Design (ICCD'96)*. Austin, Texas, USA, Oct. 1996.
16. K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Boston, Massachusetts, 1993.
17. S. Tahar, Z. Zhou, X. Song, E. Cerny and M. Langevin. Formal verification of an ATM switch fabric using multiway decision graphs. In *Proc. IEEE Sixth Great Lakes Symposium on VLSI*. Ames, Iowa, USA, March 1996.
18. K.D. Anon, N. Boulerice, E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, Y. Xu, Z. Zhou. MDG tools for the verification of RTL designs. In *Proc. 8th International Conference on Computer-Aided Verification (CAV'96)*. New Brunswick, New Jersey, USA, July 1996.
19. Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella and M. Langevin. Formal verification of the Island Tunnel Controller using Multiway Decision Graphs. Technical Report 1042, D'IRO, Université de Montréal, Montréal, Canada, July 1996.

This article was processed using the LaTeX macro package with LLNCS style