Formal Verification of the Island Tunnel Controller using Multiway Decision Graphs

Z. Zhou† X. Song† S. Tahar† E. Cerny† F. Corella‡ M. Langevin§

† D'IRO, Université de Montréal, Canada, {zhouz,song,tahar,cerny}@iro.umontreal.ca
‡ Hewlett-Packard Company, USA, fcorella@hprpcd.rose.hp.com
§ GMD-SET, Germany, langevin@gmd.de

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 perform various verification experiments on the ITC example including combinational verification, invariant checking and behavioral equivalence checking. We show how several errors that were present in the original designs have been identified using the counterexample facility of the MDG tools. 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 several solutions.

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, 16]. 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], HDDs [7] and K*BMDs [12] have been developed to represent arithmetic functions more compactly than ROBDDs. A number of methods [8, 5, 14], which verify the overall functionality of designs at an abstract level, have also emerged. Those methods use *abstract* variables to denote data signals and *uninterpreted* function symbols to denote data operations, and hence are well suited for the verification of RTL designs.

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*. This 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. We also developed a procedure for combinational circuit verification based on the canonicity of MDGs [8]. The MDG operators and the verification algorithms are implemented in the Prolog-based MDG tools [22].

Burch and Dill [5, 15] 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) [15]. Galter [14] presented a similar approach with which he verified the Tamarack-3 microprocessor. These methods, however, are not applicable to state exploration-based verification, such as model checking or behavioral equivalence checking, while MDGs provide a tool for both validity checking and verification based on state-space exploration.

Cyrluk and Narendran [**?**] defined a first order temporal logic – Ground Temporal Logic (GTL) which also uses uninterpreted function symbols. Using a decidable fragment of GTL, they may automate part of the verification at a higher level of abstraction in the PVS theorem prover.

Zhou and Burleson [20] proposed another kind of diagrams: *ConTrast* (Conditional Term Rewriting on Attribute Syntax Trees). ConTrast uses attribute grammars (for storing semantic information) to represent arithmetic expressions at an abstract level, and conditional term rewriting to translate an expression into its canonical form. Thus, proving the equivalence of two algebraic expressions corresponds to checking isomorphism of the two ConTrast diagrams. The method is interesting and we conjecture that it could be incorporated into other more general methods such as those based on MDGs.

In this paper we present a case study of MDG-based formal hardware verification using the Island Tunnel Controller (ITC) [13] as an example. This example was originally used to illustrate the notation of a heterogeneous logic system supporting diagrams as logic entities. 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. In [17], we presented a case study on a real ATM fabric with emphasis on demontrating the capability of MDGs.

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 show how we identify several errors in the original designs of [13] using the counterexample facility in the MDG tools. We also present experimental results, including a comparison with results obtained using SMV and VIS. Section 6studies in detail the non-termination problem of the abstract state enumeration and present several solutions. Conclusions and future research directions are given in Section 7.

2 MDGs and MDG-based verification approach

The formal logic underlying MDGs is a many-sorted first-order logic, augmented with the distinction between *abstract sorts* and *concrete sorts*. Concrete sorts have *enumerations*, while abstract sorts do not. An enumeration is a set of *individual constants*. Beside individual constants, the vocabulary consists of *generic constants*, *variables*, and *function symbols* (also called *operators*). Generic constants and variables each have a sort. An individual constant is treated as having multiple sorts, one for each enumeration of which it is a member. An n-ary function symbol ($n > 0$) has a type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$, where $\alpha_1 \ldots \alpha_{n+1}$ are sorts. α_{n+1} is also called the *target sort*. The (well-typed) terms and their sorts are defined as usual. Generic constants can be viewed as 0-ary function symbols.

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 from the data path to the control circuitry.

A term that has no concrete subterms other than individual constants is said to be *concretely reduced*. A term of the form "$f(A_1, \ldots, A_n)$" where f is a cross-operator and $A_1 \ldots A_n$ are concretely reduced terms, is a *cross-term*. Note that no concrete variable can occur in a concretely-reduced term or in a cross-term. An *equation* is an expression "$A_1 = A_2$" where A_1 and A_2 are terms of the same sort α. *Atomic formulas* are the equations, plus **T** (truth) and **F** (falsity). *Formulas* are built from the atomic formulas in the usual way using logical connectives and quantifiers. In the following, we describe a graph representation of a subset of these formulas.

A *multiway decision graph* (MDG) is a finite, directed acyclic graph (DAG) where the leaf nodes are labeled by formulas, the internal nodes are labeled by terms, and the edges issuing from an internal node N are labeled by terms of the same sort as the label of N. Such a graph G represents a formula defined inductively as follows: (i) if G consists of a single leaf node labeled by a formula P, then G represents P; (ii) if G has a root node labeled A with edges labeled $B_1 \ldots B_n$ leading to subgraphs $G'_1 \ldots G'_n$, and if each G'_i represents a formula P_i, then G represents the formula $\bigvee_{1 \leq i \leq n}((A = B_i) \wedge P_i)$.

Just as Bryant's OBDDs [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]. Among other things, these conditions specify the kinds of nodes that may appear in an MDG. An internal node may be labeled by a variable of concrete sort, with edges issuing from the node labeled by individual constants in the enumeration of the sort; or by a variable of abstract sort, with edges labeled by concretely reduced terms of that sort; or by a cross-term, of concrete sort α, with edges labeled by individual constants in the enumeration of α. All leaf nodes are labeled **T**, except when the graph has a single node, which may be labeled **T** or **F**.

Figure 1 shows examples of MDGs featuring the three kinds of internal nodes in the root position: Figure 1(a) shows an MDG whose root is labeled by a variable x of concrete sort α with enumeration $0, 1, 2, 3$. There are three edges issuing from the root, labeled by the individual constants 0, 1, 3, and leading to three subgraphs G, G', G''. If these subgraphs represent the formulas P, P', P'' respectively, then the graph represents the formula:

$$(x = 0 \wedge P) \vee (x = 1 \wedge P') \vee (x = 3 \wedge P''). \tag{1}$$

Note that there is no root edge labeled 2. If $x = 2$ holds, formula (1) is simply a false sentence.

Figure 1(b) shows an MDG whose root is labeled by a variable x of abstract sort β, with root edges labeled a, u and $f(u)$. Here a and u are, say, a generic constant and a variable, both of sort β, while f is a function symbol of type $\beta \to \beta$. If, as before, G, G', and G'' represent the formulas P, P' and P'' respectively, then the graph represents the formula:

$$(x = a \wedge P) \vee (x = u \wedge P') \vee (x = f(u) \wedge P''). \qquad (2)$$

The labels of edges issuing from an abstract node need not denote pairwise distinct values.

Figure 1(c) shows an MDG whose root is labeled by a cross-term $g(x)$, x being a variable of abstract sort β and g a cross-operator of type $\beta \to \alpha$. The MDG represents the formula:

$$(g(x) = 0 \wedge P) \vee (g(x) = 1 \wedge P') \vee (g(x) = 3 \wedge P''). \qquad (3)$$

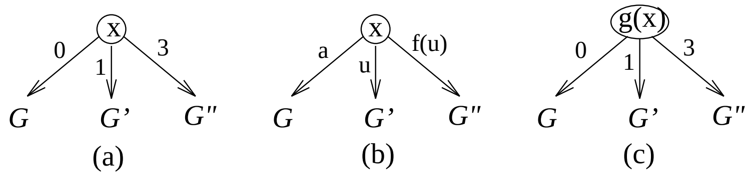


Figure 1: MDG examples

We developed a set of MDG algorithms for computing *disjunction*, *relational product* (*Conjunction* followed by *existential quantification* [4]) and what we call *pruning-by-subsumption*. A detailed description of the algorithms can be found in [8]. Note that there is no *negation* operation for MDGs.

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 and 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 set of states, transition and output relations for finite state machines, MDGs are used to compactly encode set of (abstract) states and transition and output relations for abstract state machines. Thus, it is possible to lift the implicit enumeration [4, 10] technique from the Boolean level to the abstract level, which is referred to as *implicit abstract enumeration* [8].

Based on the implicit abstract enumeration technique, we developed a reachability analysis algorithm that verifies whether an invariant holds in all reachable states of an ASM. One of its variations is the verification of observational equivalence of two ASMs. When an invariant is violated at some stage of the reachability analysis, a counterexample facility gives a sequence of input-state pairs leading from the initial state to the faulty behavior. This is very helpful for identifying design errors. In addition to sequential verification, we also have an equivalence checking procedure for combinational circuits that takes advantage of the canonicity of MDGs [8]. The above MDG operators and verification applications are packaged as MDG tools [22].

3 The Island Tunnel Controller (ITC)

The Island Tunnel Controller (ITC) example was originally introduced by Fisler and Johnson [13]. There is a one lane tunnel connecting the mainland to a small island, as shown in Figure 2(a). 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 [13], 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". For ROBDD-based verification methods, their performance depends on the particular instance of n, as we shall see in Section 5.3.

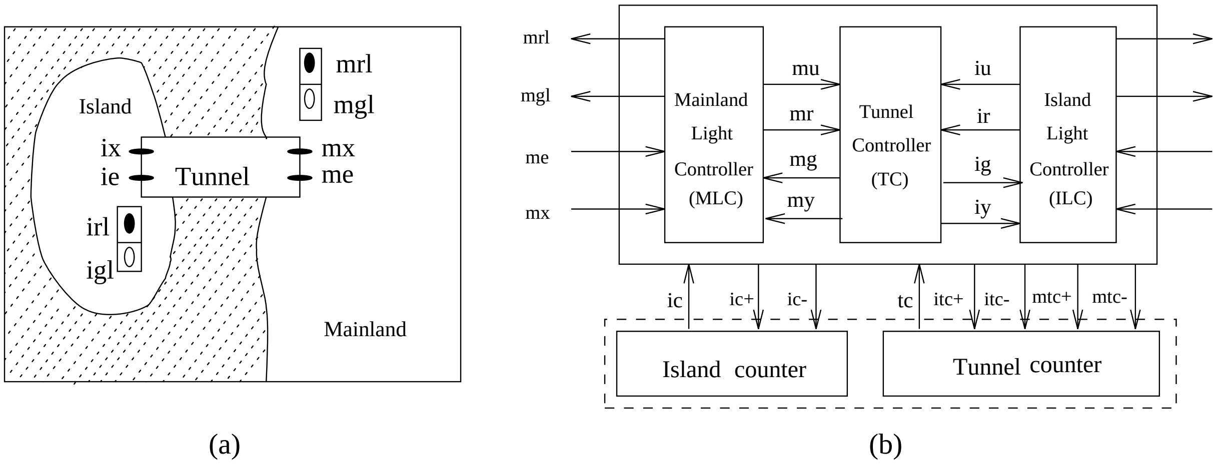
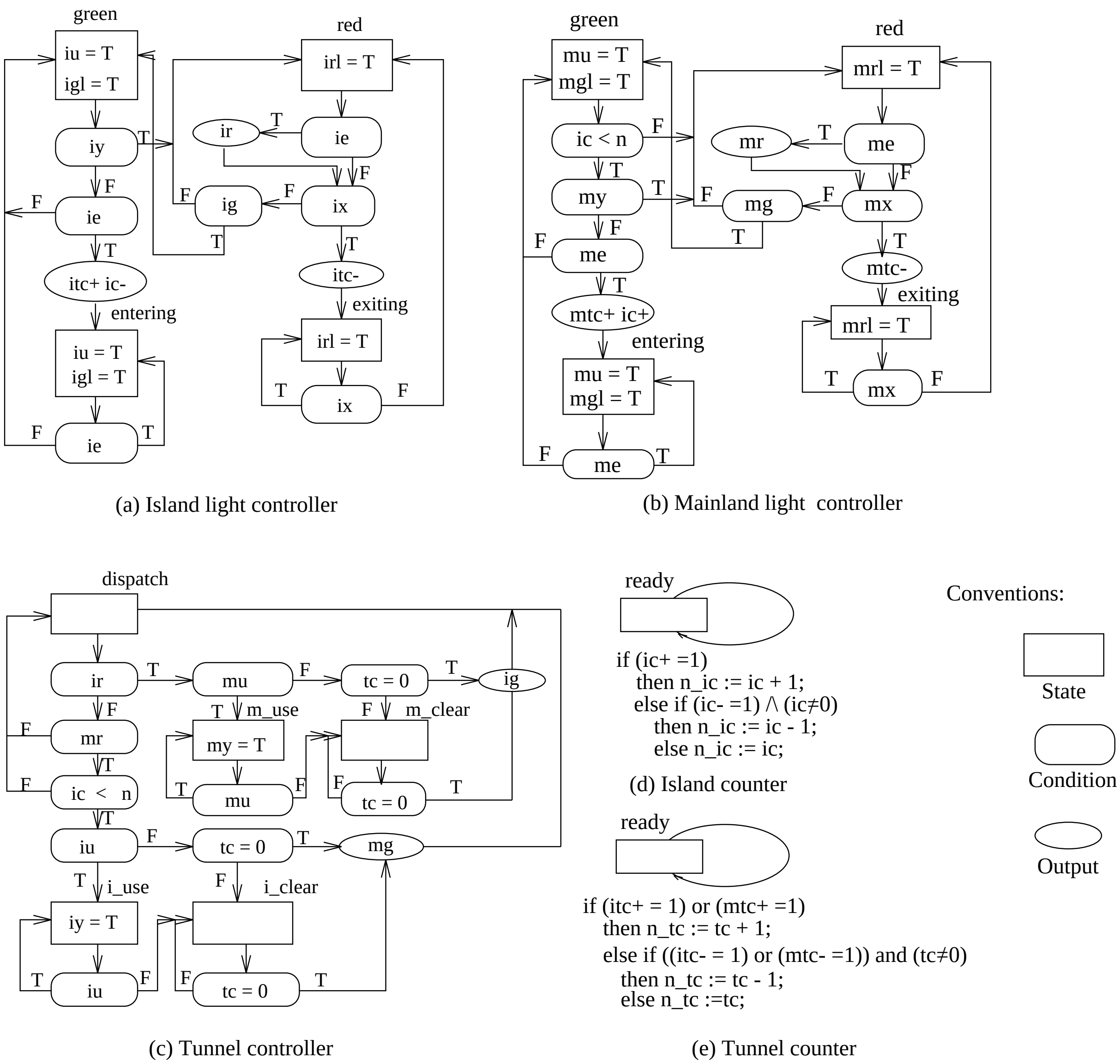


Figure 2: The Island Tunnel Controller

Fisler and Johnson [13] proposed a specification of ITC using three communicating controllers and two counters shown in Figure 2(b). Their state transition diagrams are shown in Figure 3[1]. The island light controller (ILC) (Figure 3(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 counter and the tunnel counter 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 $mtc+$[2], 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.

[1] Figure 3(a), (b) and (c) are taken from [13] with the exception of replacing 16 by n in the original diagrams.

[2] We use $itc+$ and $mtc+$ instead of $tc+$ in [13]. Similarly for $tc-$.



(a) Island light controller (b) Mainland light controller (c) Tunnel controller (d) Island counter (e) Tunnel counter

Figure 3: The state transition diagrams of the Island Tunnel Controller

4 Hardware description

Due to the generic constraint in the ITC, we model the state transition diagrams in Figure 3 as Abstract State Machines (ASMs) [9] rather than Finite State Machines (FSMs). The ASM model represented by MDGs (referred to as MDG–ASM) is obtained by compiling its MDG–HDL description (a Prolog-style HDL). MDG–HDL supports structural descriptions, behavioral ASM descriptions, or a mixture of structural and behavioral descriptions. A structural description is usually a netlist of components connected by signals, and a behavioral description is given by a tabular representation of the transition relation or truth table. Currently, we are implementing a translator for a subset of VHDL to MDG–HDL.

Like ROBDDs, MDGs require a fixed order of node labels along all paths. This ordering is supplied by the user. Unlike ROBDDs where all variables are Boolean, in MDGs every variable/signal must be assigned an appropriate sort, and a type definition must be provided for all functions. If needed, rewrite rules may be provided to partially interpret the otherwise uninterpreted function symbols.

4.1 Behavioral specification

We take as *specification* the ITC state transition diagrams in Figure 3. 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-term $equz(tc)$ represents the condition "$tc = 0$" and models the feedback from counter to the control circuitry; $equz$ is a cross-function symbol of type $[wordn \rightarrow bool]$.

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, respectively, and let n_is, n_ms and n_ts be the corresponding next state variables. We define a concrete sort mi_sort having the finite enumeration $\{green, red, entering, exiting\}$. The variables is and ms (and also their next state variables n_is and n_ms) are then assigned to be of this sort mi_sort. Similarly, we let variables ts and n_ts to 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-term $lessN(ic)$, where the uninterpreted cross-function $lessN$ of type $[wordn \rightarrow bool]$ represents the operation "$< n$".

Once the above algebraic specifications are defined, the state transition diagrams can be easily transformed into a tabular form which are then translated directly into MDG–HDL, and compiled into the internal MDG representation of the state transition relations. The transition relation for the tunnel counter in MDG–HDL looks as follows:

```
component(tunnel_counter, table([
                  [equz(tc), itc+, mtc+, itc-, mtc-, n_tc],
                  [0, 0, 0, 1, 0, dec(tc)],
                  [0, 0, 0, 0, 1, dec(tc)],
                  [*, 1, 0, 0, 0, inc(tc)],
                  [*, 0, 1, 0, 0, inc(tc)]|tc])).    % 'tc' is the default value.
```

Figure 4 shows the MDG of the transition relation of the tunnel counter obtained after compilation.

4.2 Implementation

In [13], the authors gave two implementations for ITC. Observing that the MLC (Figure 3(b)) is quite similar to ILC (Figure 3(c)), the implementations of MLC can be derived from that of ILC by small modifications. Figure 5(a) shows an implementation I_1 of MLC using a state-encoding of *green*=00, *entering*=01, *exiting*=10 and *red*=11, where the two bits are the values of register $r1$ (the left bit) and $r2$ (the right bit). Figure 5(b) is a simplified implementation I_2 derived

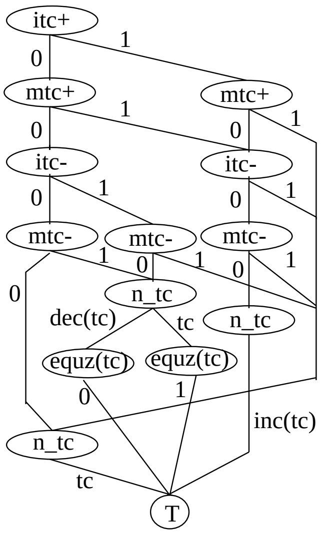


Figure 4: Transition relation MDG of the tunnel counter

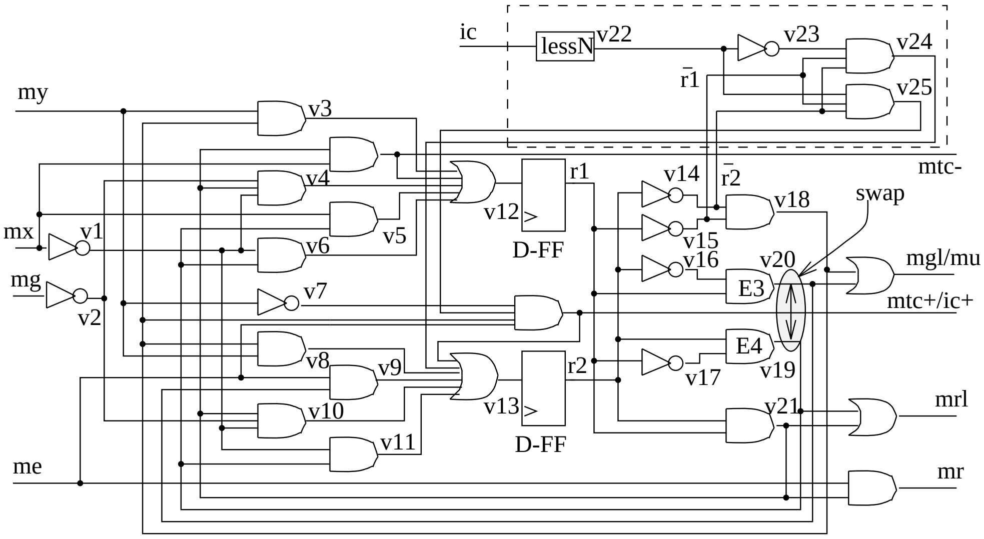
from that in Figure 5(a). The circuit diagrams excluding the portion in the dashed frame and its related connections are exactly the same as the ILC implementations in [13]. The additional circuitry in the dashed frame implements the next-state function associated with the condition "$ic < n$" (Figure 3(b)), i.e., in state *green*, if the condition is not satisfied, the MLC transits to *red*. The bold circles in the circuit diagrams identify design errors that we found using our MDG verification tools. Details will be explained in Section 5.1.

To produce an MDG representation of the implementations, we create a netlist of component instances using the predefined component definitions in MDG–HDL. However, if we wish to compare two different hardware descriptions, we must make sure that the same function symbol is used in both descriptions for the same data operation, since the function symbols are uninterpreted. For example, if we use $lessN$ in the specification, we should also use it in the implementation.

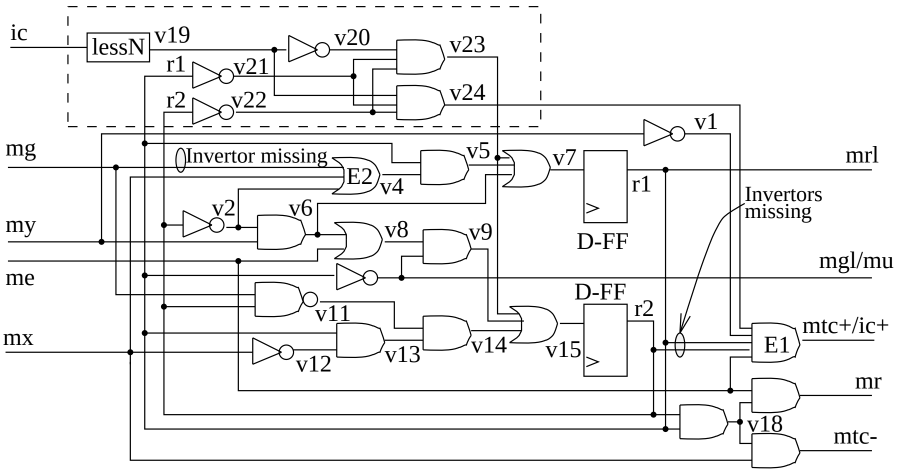
5 Verification of the Island Tunnel Controller

In this section, we explain the various verification experiments performed on the ITC example using our MDG tools. This includes behavioral equivalence checking, combinational verification and invariant checking. We also demonstrate how we identify and correct several errors that were present in the original implementations. For comparison reasons, we also experimented with the invariant checking using two ROBDD-based tools: SMV and VIS.

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.



(a) Implementation I_1



(b) Implementation I_2

Figure 5: Two implementations of the Mainland Light Controller

5.1 Equivalence checking of MLC implementations against their specification

We first compared the two ILC implementations given in [13] against their behavioral specification. We found and corrected the design errors before further experiments. All these errors were confirmed by the original designer. In the following, however, we verify, for illustration purposes, the MLC implementations including the errors. As our first experiment, we compare the simplified implementation I_2 of the MLC against the specification by forming a product machine and performing reachability analysis. The initial state of the product machine is arbitrary, but constrained under the state encoding relation between the implementation and the specification. The invariant asserts the equivalence of the corresponding observable outputs of the two descriptions. In the first run, the equivalence invariant was violated and the following counterexample was generated:

Counterexample 1:

Assumptions:	$lessN(ic\#1) = 1$.				
Initial state:	$ms\_S = green$;	$r1\_I_2 = 0$;	$r2\_I_2 = 0$.		
Clock cycle 1					
Symbolic inputs:	$my = 0$;	$me = 1$;	$ic = ic\#1$		
Symbolic Outputs:	$mr\_S = 0$;	$mrl\_S = 0$;	$mgl\_S = 1$;	$mtc{+}\_S = \mathbf{1}$;	$mtc{-}\_S = 0$;
	$mr\_I_2 = 0$;	$mrl\_I_2 = 0$;	$mgl\_I_2 = 1$;	$mtc{+}\_I_2 = \mathbf{0}$;	$mtc{-}\_I_2 = 0$.

The variable $ic\#1$ is the symbolic value of the input ic during the first cycle. $lessN(ic\#1) = 1$ is an assumption imposed on it. We can observe that the $mtc+$ has different values in the specification and the implementation. By checking Figure 3, we found that two invertors were missing on the inputs of gate E1 (Figure 5(b)). After correcting these errors, the equivalence check produced yet another counterexample:

Counterexample 2:

Assumptions:					
Initial state:	$ms\_S = red$;	$r1\_I_2 = 1$;	$r2\_I_2 = 1$		
Clock cycle 1:					
Symbolic inputs:	$mg = 1$;	$mx = 0$;	$ic = ic\#1$		
Symbolic state:	$ms\_S = green$;	$r1\_I_2 = 1$;	$r2\_I_2 = 0$		
Clock cycle 2:					
Symbolic input:	$my = 0$;	$me = 0$;	$ic = ic\#2$		
Symbolic Output:	$mr\_S = 0$;	$mrl\_S = \mathbf{0}$;	$mgl\_S = \mathbf{1}$;	$mtc{+}\_S = 0$;	$mtc{-}\_S = 0$;
	$mr\_I_2 = 0$;	$mrl\_I_2 = \mathbf{1}$;	$mgl\_I_2 = \mathbf{0}$;	$mtc{+}\_I_2 = 0$;	$mtc{-}\_I_2 = 0$.

This time we identified the error as a missing invertor on the input line of gate E2 (Figure 5(b)). Once corrected, the behavioral equivalence check succeeded. Similarly, we verified the MLC implementation I_1 (Figure 5(a)) against the specification and identified another design error: the output lines $v20$ and $v19$ of gates E3 and E4, respectively, should be swapped.

All above errors were corrected in 1/2 hour with the help of the counterexample facility. The execution time and resource requirements of the experiments are given in Table 1. Rows *Faulty impl.* I_1 and *Corrected impl.* I_1 are the verification of the implementation I_1 with and without the original errors against the behavioral specification. Similarly for rows *Faulty impl.* I_2 and *Corrected impl.* I_2.

	Island Light Controller			Mainland Light Controller		
	Time (Sec)	Mem (MB)	#Nodes	Time (Sec)	Mem (MB)	#Nodes
Faulty impl. I_1	3.4	1.5	1015	5.0	1.7	1361
Corrected impl. I_1	3.4	1.5	988	5.0	1.6	1292
Faulty impl. I_2	2.8	1.2	833	3.6	1.5	992
Corrected impl. I_2	2.7	1.2	806	3.6	1.6	988

Table 1: Behavioral verification for the implementations against their specifications

5.2 Combinational verification of MLC implementations

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 MDGs 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 are given in Table 2. The execution time (column *CPU*) is essentially the compilation time for constructing transition and output relation MDGs.

	Time (Sec)	Mem (MB)	#Nodes
ILC	2.7	1.4	783
MLC	3.7	1.5	1108

Table 2: Combinational verification of the implementations

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[3]. We also provide their CTL (Computational Tree Logic) formulas which are used for invariant checking by ROBDD-based verification tools SMV and VIS.

P1: Cars never travel both 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 (! (($ic- = 1$) & ($ic+ = 1$))).

The invariant checking is based on reachability analysis of state machines. However, the reachability analysis procedure may not terminate in general [8], which is the case for the ITC example. To deal with the problem, we developed a heuristic state generalization method which is to be explained in detail in Section 6. The method has false negatives, i.e., if an invariant is violated, we have to examine, e.g., by simulation, whether the counterexample is caused by a *real* error in the design.

Table 3 shows the invariant checking results of the conjunction of P1, P2 and P3 on the ITC specifications.

5.4 Comparative results using FSM model

Besides the ASM-based verification experiments using MDGs, we also verified the above example invariants (P1, P2 and P3) using FSM-based methods. For MDG tools, the counts tc and ic

[3] Fisler and Johnson [13] proposed a set of properties that the ITC design should satisfy. Currently, we only consider the verification of invariants.

Properties	Time (Sec)	Mem (MB)	#Nodes
P1, P2, P3	55	2.7	4329

Table 3: Invariant checking using the ASM model

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) and VIS. SMV [18] is a symbolic model checker for CTL formulas developed by McMillan at Carnegie-Mellon University. We only used the option '-f' (forward search) and the node order was generated automatically by the system.

VIS [3], developed at University of California at Berkeley, is an integrated tool for verification, simulation and synthesis of finite state systems. It contains Fair CTL model checker and behavioral equivalence checker for sequential circuits, language emptiness check for Büchi automata, and combinational verification. We use the special command for invariant checking which is more efficient than the general model checking command according to the VIS manual. The node order was also generated automatically by the system.

Table 4 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. 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:

1. Time increases exponentially with the counter width for concrete representations of the problem.

2. 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 then those of ROBDDs. Also the MDG package is an early prototype written in Prolog, while the other tools are more mature implementations written in C/C++. In particular, the Prolog implementation uses a lot of memory, which impacts the runtime. There is no reason why a careful implementation of an MDG package in C should be much slower than an ROBDD package on concrete problems.

3. On the other hand, only the MDG tools can verify a parameterized implementation having n bits, and it does so quite efficiently in time independent of the datapath width.

It may be argued that the data abstraction method [6, 16] 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 to provide an appropriate data abstract function. Moreover, such data abstraction is not always possible. One example is the 4×4 Fairisle ATM switch fabric recently verified using MDGs [17, 19]. The switch fabric routes incoming cells – a finite stream of words, from input ports to output ports. The first word of the stream contains control information (4 bits) which makes routing decisions. In general, if

	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

Table 4: Invariant checking of ITC specifications. ('–' means that the verification did not terminate, and '*' means that the verification is not possible.)

the control information need n bits, then it is impossible to reduce the word width to less than n. Thus for datapaths containing mixed control/data information, ROBDD-based datapath reduction technique is no more applicable. 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.

The false negatives introduced for solving non-termination problem of MDG-based method may seem a drawback with respect to ROBDD-based verification methods. In practice, however, finite-state verification algorithms often take too long (as in Table 4), or exhaust memory; the fact that they are theoretically terminating is then of no practical consequence.

6 Termination of abstract state enumeration

In this section, we focus on the termination problem for abstract state enumeration and present several different solutions.

The reachability analysis algorithm is based on abstract implicit enumeration [8] where sets of states as well as transition and output relations are represented using MDGs. 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. 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 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.

Pruning is implemented by a special MDG operation *pruning-by-subsumption* (PbyS) which deserves a special mention here. It takes as input two MDGs G and H and produces the result MDG G' by removing zero or more paths from G and then reducing the graph if needed. The paths removed are those subsumed (or covered) by H. PbyS is used for multiple purposes. It provides a form of frontier-set simplification [4, 10]. It is also used for detecting termination and for checking that an invariant is satisfied during reachability analysis [8].

6.1 Non-termination problem and termination with state generalization

6.1.1 Review

An abstract state machine may have an infinite member 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). 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 3(e)), which is a processor-like circuit, as an example.

Consider an ASM description of the tunnel counter with the transition relation as given in Section 4.1, and let the initial state be a generic constant zero of sort *wordn*. Figure 6(a) shows the set of initial states N_0, the set of states reached in one transition S_1, and the frontier-set of states N_1. 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* [4] *abstract* variable a. Then, as shown in Figure 6(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 due to the state generalization. For instance, the state $(tc = dec(a)) \wedge (equz(a) = 0)$ becomes a reachable state which covers the previously unreachable state $(tc = dec(\mathsf{zero})) \wedge (equz(\mathsf{zero}) = 0)$.

6.1.2 State generalization for general cases

The complete ITC specification is composed of 5 communicating state machines (Figure 3). 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 to it directly. 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 Figure 3(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* and is 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 7,

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

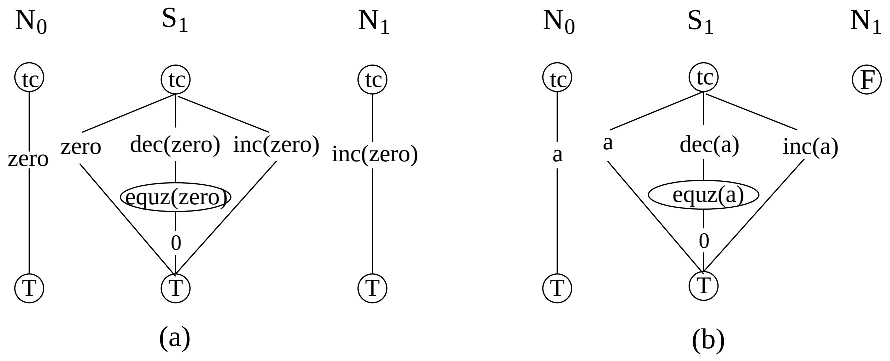


Figure 6: Non-termination problem and initial state generalization method for the tunnel counter.

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*. MLC and TC are supposed to keep their initial state values.

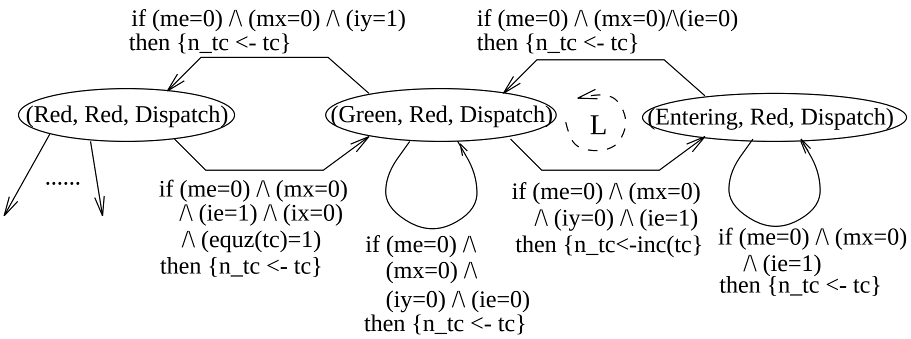


Figure 7: A fraction of state transition diagram for the composed machine.

To illustrate the failure of the initial state generalization method on this ITC 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 8 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 7.

Starting from N_0, after three transitions, the set of newly reached states S_3 contains two states: $(is = green) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = inc(a)) \wedge (equz(a) = 1)$ and $(is = entering) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = inc(a)) \wedge (equz(a) = 1)$, where the latter is subsumed by N_2. However, the former cannot be subsumed by N_1 since $inc(a)/a$ is not a valid substitution in this case. 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 7) with a data operation inc on the abstract state variable tc. This loop, in fact, resembles the ASM of a processor-like circuit. Therefore, we denote such a control loop with data operations as a *processor-like* loop. 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.

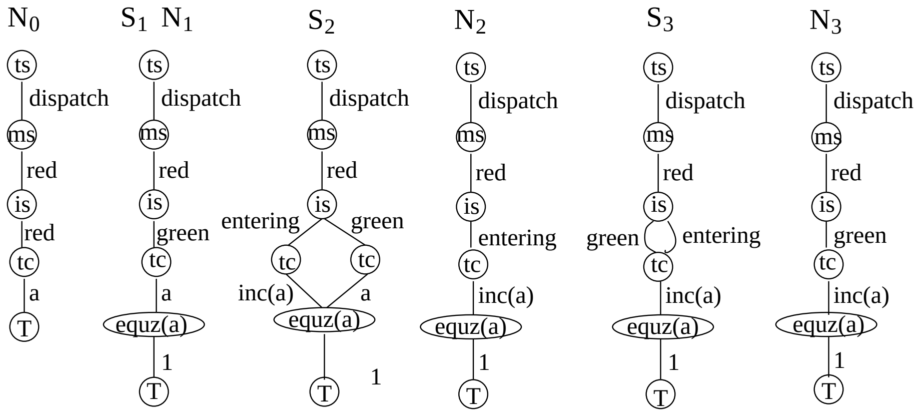


Figure 8: State enumeration for the composed machine using initial state generalization technique.

The above analysis 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 9 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 $equz(zero) \to 1$. As this state is the entry state of 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.

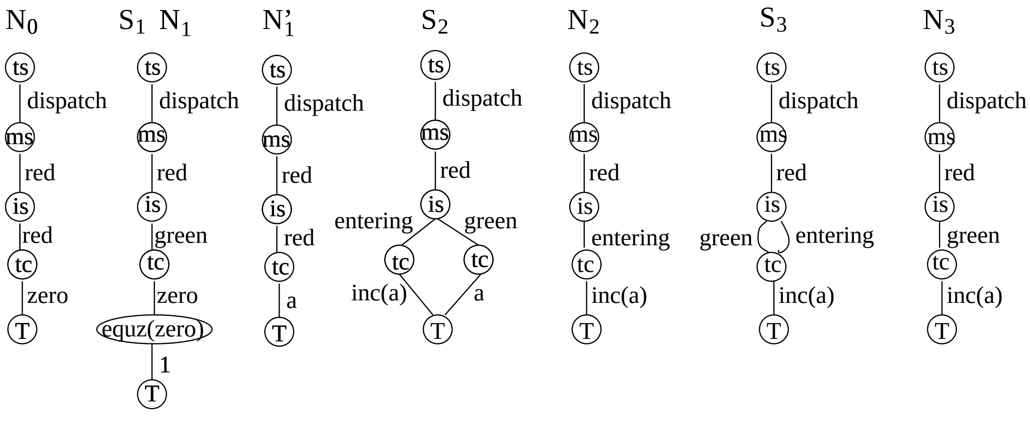


Figure 9: State enumeration for the composed machine using the extended state generalization technique.

It remains now to know, in practice, 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. In the next section, three solutions are discussed.

6.2 Practical state generalization methods

We propose three solutions to the non-termination problem: *extended initial state generalization*, *circuit generalization* and *heuristic state generalization*.

6.2.1 Extended initial state generalization

With this method, we first identify all processor-like loops by manually analyzing the design. For each entry state of the processor-like loop, we let it be one of the initial states and generalize the specific state variables by assigning them fresh variables as the initial values. For the ITC example (Figure 7), the set of initial states is declared as : $\{(is = green) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = a)\}$.

For simple or regular designs, manual analysis is often possible. The processor-like circuit is an example where the initial state itself is the entry state of the processor-like loop. However, for complex designs, manual analysis pose difficulties.

6.2.2 Circuit generalization

Circuit generalization is also based on the manual identification of processor-like loops. Instead of declaring their entry states as initial states, we generalize the design by adding non-determinism in the circuit or, equivalently, the transition relations. Here again for the ITC example, the transition from control state $(is = red) \wedge (ms = red) \wedge (ts = dispatch)$ to $(is = green) \wedge (ms = red) \wedge (ts = dispatch)$ can be generalized to

$$\textsf{if } (me = 0) \wedge (mx = 0) \wedge (ie = 1) \wedge (ix = 0) \wedge (equz(tc) = 1) \textsf{ then } n\_tc \leftarrow x$$

where x represents a primary input signal of sort *wordn*. Suppose that x takes a fresh variable a as its input at the transition, then the newly reached state becomes $(is = green) \wedge (ms = red) \wedge (ts = dispatch) \wedge (tc = a)$ which is exactly a generalized state. However, the limitation of this method is similar to that of the first one.

6.2.3 Heuristic state generalization

We also 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 guaranting the 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 the states that are not within processor-like loops are also generalized.

As a trade-off, we propose a heuristic solution for 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.

6.2.4 Discussions

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 an invariant is violated, we have to examine, e.g., by simulation, whether the counterexample thus produced corresponds to a *real* design error.

The above three methods all require human interferences, however, at different stages of the verification process. The circuit generalization method introduces minimum false negatives since the state generalization is performed precisely for those essential state variables during state enumeration. For the extended initial state generalization, some of the additional initial states could be unreachable, which results in a larger reachable state space. But the effect is still predictable since it is only related to those initial states. Therefore, it could be easier to analyze counterexamples for both methods. On the contrary, 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 3 are obtained using the heuristic state generalization technique on the complete ITC specification composed of the 5 ASMs.

7 Conclusions and future work

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 ITC 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 obtained results for invariant checking with the ROBDD-based verification tools SMV and VIS, where we showed the MDG approach can easily handle arbitrary data widths. Finally, we studied in detail the non-termination problem of the abstract state enumeration and presented several different solutions. There still exist open problems related to the handling of liveness/fairness properties with the ASM approach. We are currently in the course of developing a model checking algorithm for a restricted first-order temporal logic.

Acknowledgments

We wish to thank N. Boulerice for her assistance in the experiments.

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. *Technical Report UCB/ERL M95*, Electronics Research Laboratory, University of California, Berkeley, December 1995.

[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 International Conference on Computer-Aided Design (ICCAD'95).* San Jose, California, USA, November 1995.

[8] F. Corella, Z. Zhou, X. Song, M. Langevin and E. Cerny. Multiway decision graphs for automated hardware verification. To appear in the journal *Formal Methods in System Design.* Preliminary version published as IBM technical report RC19676, July 1994.

[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] R. Drechsler, B. Becker and S. Ruppertz. K*BMDs: a new data structure for verification. In *Proc. IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (Charme'95).* Frankfurt, Germany, October 1995.

[13] 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, August 1995.

[14] D. Galter. Symbolic verification of instruction-set processors. Master of Mathematics thesis, Dept. of Computer Science, University of Waterloo, 1994.

[15] 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.

[16] D. E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, Carnegie Mellon University, 1993.

[17] M. Langevin, S. Tahar, Z. Zhou, X. Song and E. Cerny. Behavioral V verification of an ATM switch fabric using implicit abstract state enumeration.

In *Proc. IEEE International Conference on Computer Design (ICCD'95).* 1996.

[18] K. L. McMillan. *Symbolic model checking.* Kluwer Academic Publishers, Boston, Massachusetts, 1993.

[19] 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.

[20] Z. Zhou and W. Burleson. A canonical representation of algebraic expressions in high-level synthesis. *Technical report TR-94-CSE-9,* University of Massachusetts at Amherst, October 1994.

[21] Z. Zhou, X. Song, F. Corella, E. Cerny, and M. Langevin. Description and verification of RTL designs using Multiway Decision Graphs. In *Proc. IFIP Conference on Hardware Description Languages and their Applications (CHDL'95).* Chiba, Japan, August 1995.

[22] Z. Zhou, X. Song, F. Corella, M. Langevin, E. Cerny and S. Tahar. MDG tools for the verification of RTL designs. To appear in *Proc. 8th International Conference on Computer-Aided Verification (CAV'96).* New Brunswick, New Jersey, USA, July 1996.