Computing Reachable Control States of Systems Modeled with Uninterpreted Functions and Infinite Memory

Adrian J. Isles, Ramin Hojati and Robert K. Brayton
{aji, hojati, brayton}@eecs.berkeley.edu

Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley, CA 94720

Abstract. We present an approach for automatically computing the set of control states reachable in systems modeled with uninterpreted functions, predicates and infinite memory. In general, the abstract state spaces of systems modeled in this fashion are infinite and exact state enumeration based procedures may not terminate. Using the Integer Combinational Sequential (ICS) concurrency model *[HB95]* as our underlying formalism, we show how 'on-the-fly' state reduction techniques, which preserve control invariance properties, can be used to significantly speed-up reachability computations on such abstract hardware representations, collapsing infinite state spaces to finite ones in some cases. The approach presented in this paper is automatic and if it terminates, will produce the exact set of reachable control states of abstract hardware models. Our techniques have been implemented in an ICS state reachability tool and experimental results are given on several examples.

1 Introduction

The use of interpreted and uninterpreted integer functions, interpreted equality, and interpreted memory operations has been shown to be very useful for abstracting away the complexity of datapath found in hardware. In general, the abstract state space of a system modeled in this fashion is infinite and exact state enumeration based procedures may not terminate. However, many important properties that one would like to verify of such systems can be expressed in terms of determining if a set of control states of the system is reachable. In some cases, the infinite state spaces of such systems can be reduced to a finite set which preserve all behaviors with respect to such properties. In this paper, we address the problem of automatically computing the set of reachable control states of systems described with abstract datapath representations utilizing state reduction techniques to reduce the number of reached states.

The Integer Combinational/Sequential (ICS) concurrency model, presented in *[HB95]*, can be used to model abstract hardware representations. Here, control is represented using finite relations and latches ranging over finite domains. Control can move data around in the datapath and can get information about their values by applying uninterpreted or interpreted predicates to them. The datapath component consists of variables and latches that are assigned (ICS) terms, which are abstract representations of integer values of unbounded width. Uninterpreted functions are provided in the datapath for performing computations. Finally, interpreted memory operations model memory with an unbounded number of locations. An ICS state consists of assignments of finite values to the finite latches, terms to the integer latches, and predicates on terms. In addition, a memory is represented as a table of address/data-value pairs, both of which are also ICS terms.

1.1 Our Approach

We show how automatic state reductions can be used during state enumeration when computing the set of reachable control states of a model. Note that the set of reachable control states is obtained by projecting the state space on to its finite latch assignments. We also propose a new state data structure, called *ICS State Pairs (ISPs),* that can be used to efficiently compute the set of reachable control states of systems modeled with ICS. Intuitively, an ISP is a pair $(c(x), d)$ representing a set of ICS states all of which have the same terms assigned to integer latches, memory, and predicates, but different assignments to the finite state variables. $c(x)$ is an ROBDD *[Bry86]* representing a set of assignments to the finite latch variables x appearing in the model, and d a directed acyclic graph (DAG) that is a syntactical representation of terms assigned to integer latches, memory, and predicates. During state enumeration, the set of reachable states is stored as a set of ISPs. Transitions between states are performed using standard image computation for the finite part, and a special form of symbolic simulation for the integer part. State reductions *[HIB97]* are also performed during symbolic simulation to minimize the number of states that need to be explored. Using the distinction given in *[CZXLC94],* our approach can be considered a combination of *abstract explicit* and *abstract implicit state enumeration.* It is an abstract explicit technique in the sense that we do not build formulas for alternate datapath assignments that may occur due to case splitting. Thus, an assignment of $z := mux(b, g(a,b), f(x))$ could result in two ISPs $(b = 0, z = g(a,b))$ and $(b = 1, z = f(x))$. An alternative would be to represent this behavior implicitly using a single formula, $(z = g(a,b) \wedge b = 0) \vee (z = f(x) \wedge b = 1)$, as could be done using the approach in *[CZXLC94].* Of course, our approach is implicit in the sense that we represent the finite part using BDDs and our algorithm reduces to implicit state enumeration in the absence of datapath. By treating datapath and control separately, we can exploit well-known techniques for manipulating each. During state enumeration, the number of ISPs grows only in the number of different assignments to integer assignments that have been seen.

Our approach is automatic and if it terminates will produce the exact set of reachable control states of the system. Note that it was shown in *[HIKB67]* that, in general, computing this set is undecidable. Hence, no algorithm can guarantee that it can always compute the set of control states of every ICS model. If our procedure does not terminate, it can still be used for partial verification, by performing reachability for a finite number of steps (no false negatives are produced). In addition, not only can our approach be used as a standalone verification technique, but maybe useful in conjunction with other verification procedures that require an approximation to the set of reachable states as input, such as the ones reported in *[BD94, JDB95].*

1.2 Previous Work

The modeling of hardware using uninterpreted functions has been a traditional approach used in the theorem proving community for abstracting away datapath found in hardware such as microprocessors *[Hunt85, SB90].* Many of these techniques are not automatic and require a great deal of user guidance. *[BD94]* presented an

automatic approach for verifying that a pipeline microprocessor, modeled with uninterpreted functions, implements its instruction set architecture (ISA). Their technique requires an approximation to the set of reachable states as input. One of the primary advantages of ICS is that this ISA verification can be performed without requiring any such state invariant from the user. However, it may be possible for our technique to be used to compute state invariants needed as input to their approach. As will be shown in Section 5, this is a computationally simpler task in our framework than performing full ISA verification. Another technique that is similar in flavor to ICS is a data structure called Multiway Decision Graphs (MDGs) *[CZXLC94]*. An MDG is a decision diagram that is similar to a BDD except that it allows for uninterpreted function symbols to be represented in the graph. Their technique is implicit, but they do not fully model interpreted equality or interpreted memory operations. The approaches in *[Cor94, LC91]* also seem similar to ours, except they represent control states explicitly. *[IHB96]* implemented an algorithm for ICS reachability as proposed in *[HB95]* and presented experimental results on verifying systems modeled using ICS. The results where poor, however, due to an explosion in the number of states on even simple designs. *[HIB97]* presented a set of state reduction techniques, some of which are used in this paper, but no algorithm was given for using them when performing reachability computations.

The flow of the rest of this paper is as follows. In Section 2, an overview of ICS is given. The reader is encouraged to read *[HB95]* for a more detailed presentation. In Section 3, we review state reduction techniques that can be used with perform state enumeration. In Section 4, we present ISPs and show how they can be used for reachability. In Section 5, experimental results are given.

2 An Overview of ICS Models

In this section, an overview of a subset of the ICS syntax and semantics is given. ICS models hardware are similar to conventional models representing hardware, with additional constructs that can be used to model non-deterministic gates, integer functions, integer predicates and infinite memory.

2.1 Syntax

The primitives consist of variables and *generalized gates* that can include tables, interpreted functions and predicates, uninterpreted functions and predicates, constant creators, latches, and memory functions.

Variables. Variables are of two types: finite and integer. Finite variables take values from some finite domain and integer variables are assigned symbolic expressions called *ICS terms*. These terms are built recursively from numerals, constants, and interpreted and uninterpreted functions. Therefore, numerals and constants are ICS terms, and if f is an n-ary function and $t_1,\ldots,t_n$ are ICS terms, then $f(t_1,\ldots,t_n)$ is also an ICS term.

Tables. A table is a relation defined over a set of finite variables, divided into inputs and outputs.

Interpreted Functions and Predicates. A predefined set of functions and relations over integers are built in. The interpreted functions are $x := y$ and $z := mux(b, x, y)$,

where x, y are integer variables and b is binary. The interpreted predicates $x = y$ and $x = c$ are also allowed where c is a non-negative integer.

Uninterpreted Functions and Predicates. These are a set of function and predicate symbols where only their arities and domain variables are given. Predicates of the form $x = term$, where x is an integer variable, and *term* is an ICS term are also allowed.

Constant Creators. A constant creator is a special element with no inputs which creates a new *fresh constant* (i.e. a function with no argument) each time called. A constant creator can be used to model unconstrained integer input.

Latches. A latch is defined on two variables over the same domain: input (or next state) and output (or present state). Latches can either be finite or integer-valued.

Memory Functions. Two functions *read* and *write* are provided with their usual interpretation; *read* is a binary function of a memory element and a location; *write* is a ternary function, whose arguments are a memory element, a location, and a value.

Definition 2.1.1 A *state* is a triple (*latches*, *memories*, *predicates*), where,

a. *latches* is a set of assignments to the latches.

b. *memories* is a set of memory elements, where a memory element is a set of pairs of ICS terms, where the first denotes a location and the second a value.

c. *predicates* is a set of atomic formulas, where an atomic formula is any interpreted or uninterpreted predicate applied to ICS terms.

Definition 2.1.2 The set *Terms(s)* of a state s denotes the set of all ICS terms, closed under subterms, assigned to the integer latches, memories and predicates of s.

Definition 2.1.3 Given two ICS terms t_1 and t_2, and two sets of atomic formulas $P = \{p_1, \ldots p_n\}$ and $Q = \{q_1, \ldots q_n\}$, t_1 is equal to t_2 subject to P and Q, iff the formula $p_1 \wedge \ldots \wedge p_n \wedge q_1 \wedge \ldots \wedge q_m \rightarrow t_1 = t_2$ is valid (i.e. true under any interpretation, given to all constants, uninterpreted function symbols, and uninterpreted predicates appearing in P, Q, t_1, and t_2). The equality of two ICS terms can be decided using the algorithm given in *[Sho79]*.

2.2 Operational Semantics

Given a state $s = (L, M, P)$, a transition to a state $s' = (L', M', P')$ of s is obtained by starting from inputs and present state latch variables and assigning a value to each variable $o = g(i_1, \ldots, i_n)$ that is consistent with its inputs $i_1, \ldots, i_n$ and generalized gate g. We denote the *partial state* $s(g)$ as the values assigned to all variables, predicates and memory before the gate g has been processed. Once all gates are processed, we assign to L' the values given to the next state latch variables of the partial state. Similarly, we give M' and P' the new assignments to M and P that are assigned in the partial state obtained at the end of the procedure. In the following, if i_k is an integer variable, then it is assigned a term t_k by $s(g)$. Otherwise, it is assigned a finite value z_k by $s(g)$.

1. If g represents a constant creator, then introduce a new fresh constant c_1 and let $o = c_1$.
2. If g represents an uninterpreted function f_i, then assign o the term $f_i(t_1,\ldots,t_n)$.
3. If g represents the function $mux(i_b, i_1, i_2)$, where i_b is a binary value, and i_1, i_2 are integer variables, assign o the term t_1 if $z_b = 0$ and t_2 if $z_b = 1$.
4. If g represents a finite relation R_g, assign o a value z_o such that $(z_1,\ldots,z_n,z_o) \in R_g$
5. If g represents an integer predicate $p(i_1,\ldots,i_n)$ and if $P \rightarrow p(t_1,\ldots,t_n)$ is valid, then assign $o = 1$. If $P \rightarrow \neg p(t_1,\ldots,t_n)$ is valid, then let $o = 0$. Otherwise, create two partial states, one with $o = 1$ and $P = P \cup \{p(t_1,\ldots,t_n)\}$, and the other with $o = 0$ and $P = P \cup \{\neg p(t_1,\ldots,t_n)\}$.
6. If g represents $read(M_k, i_1)$ where t_1 is a term representing an address, if there exists an address/value pair $(a,d) \in M_k$ where $P \rightarrow t_1 = a$ is valid, the assign $o = d$. Otherwise, perform all of the following, which may result in multiple partial states. 1) For each memory addresses $(a_1,d_1) \in M_k$ where $P \rightarrow t_1 = a_1$ is satisfiable, create a new partial state with $P = P \cup \{(t_1 = a_1)\}$ and $o = d_1$. 2) Create a new partial state with $P = P \cup \{(t_1 \neq a_1)\}$ for each $(a_1,d_1) \in M_k$, where $P \rightarrow t_1 \neq a_1$ is satisfiable. Introduce a new fresh constant d' and let $M_k = M_k \cup \{(t_1,d')\}$ and $o = d'$. Case (1) corresponds to reading an address that has previously been written. Case (2) corresponds to reading a memory location that has never been written to before, in which case, a new constant d' is returned.
7. If g represents $write(M_k, i_1, i_2)$, then proceed similarly to a read.

Note that case splitting can result in (4), (5), (6), and (7). However, the resulting state graph is finite branching, i.e. for every state, there are a finite number of successor states.

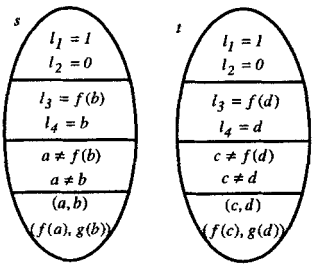
State enumeration can be performed by computing a fixed point starting from the initial state of the model using the operational semantics described above. Note that two states of a model are the same if and only if they are assigned the same values to all latches, predicates, and memory for all interpretations given to the uninterpreted functions, predicates and constants appearing in both states.

2.3 State Reductions for Property Verification

Verification of systems modeled with ICS can be performed via language containment. Here properties and fairness constraints are placed only on the finite latches in the model. Thus, any state reduction technique that preserves the sequential behavior of all the finite variables will not cause any loss in verification accuracy. This is notion is formally defined below.

Definition 2.3.1 Given a state s of an ICS model M, we denote by $F(s)$ the set of traces (strings in the language) starting from s projected onto the set of finite variables in M. s is said to be *trace equivalent* to a state t iff $F(s) = F(t)$. s is said to be *trace contained* in another state t iff $F(s) \subseteq F(t)$.

In *[HIB97]*, a set of sufficient conditions was given for detecting trace equivalent ICS states. These techniques are able to detect trace equivalent states in certain situations where the same values are assigned to the finite variables and assignments of terms in the integer part are different. An example of such a condition is *isomorphic states*. Here we say that two states are isomorphic if renaming the set of symbolic constants in the first state results in the second. In *[HIB97]*, it was shown that if two states are isomorphic, then they are trace equivalent. An example of two isomorphic states is given in Figure 2.1.



State s is isomorphic to state t since they are assigned the same values to the finite latches and renaming the constants a and b appearing in s to c and d results in a state that is syntactically equivalent to t.

Figure 2.1

3 Detecting Trace Containment

It is easy to show that if a state t is trace contained in a state s, then the set of control state assignments that are reachable from t are also reachable from s. Thus, if computing the set of reachable control states is the objective, it suffices to explore transitions from new states that are not trace contained in previously seen states. This optimization can not only reduce the time and space requirements during state enumeration, but also greatly increases the chances of reaching a fixed point. In this section, we introduce a sufficient condition, called *sub-isomorphic ICS states,* for [-pdetecting trace containment between ICS states. Intuitively, a state s is sub-isomorphic to a state t if s can be obtained from t by replacing terms with constants and deleting extra memory locations and predicates. The formal definition is given below.

Definition 3.2 A state s is *sub-isomorphic* to a state t if they both assign the same values to the finite state variables in the model and there exists a function $\pi : Terms(s) \rightarrow Terms(t)$ where any constant in $Terms(s)$ can map to any term in $Terms(t)$ and the following condition holds. 1) For each numeral $i \in Terms(s)$, there exist $i \in Terms(t)$, and $\pi(i) = i$. 2) For each term $g(u_1,\ldots,u_n) \in Terms(s)$ with an outermost function symbol g, $\pi(g(u_1,\ldots,u_n)) = g(\pi(u_1),\ldots,\pi(u_n))$. 3) π agrees with all assignments of latches, memory and predicates in the state. Thus, if a term u has been assigned to the i-th latch in s, $\pi(u)$ is assigned to the i-th latch in t. For memory, each address/value pair $(a,d) \in M_k$ in s maps to a location $(\pi(a),\pi(d)) \in M_k$ appearing in t. Similarly, each predicate $p(u_1,\ldots,u_n)$ appearing in s, maps to a predicate $p(\pi(u_1),\ldots,\pi(u_n))$ appearing in t.

In *[HIB97]* it was shown that deleting predicates, memory locations or replacing terms with fresh constants results in a state with more behaviors. Theorem 3.2 uses this to prove that sub-isomorphic checking is a sufficient condition for detecting trace containment.

Theorem 3.2 If s is a ***sub-isomorphic*** state of t, then $F(t) \subseteq F(s)$.
Proof. Let $\pi : Terms(s) \rightarrow Terms(t)$ be a function inducing a sub-isomorphism between s and t. Construct a state t' from t by first deleting all memory locations and predicates which according to π does not correspond to any memory location or predicate in s. In addition, for each constant $c \in Terms(s)$, replace occurrences of $\pi(c) \in Terms(t)$ with a constant $\hat{c}$. Then $F(t) \subseteq F(t')$, by *[HIB97]*. Note t' and s are isomorphic, i.e. one can be obtained from the other by renaming the constants of one with the other, and thus $F(t') \subseteq F(s)$.

4 ICS State Enumeration: Data Structures and Algorithms

Function *ISP_Reach*
$R = (c_0(x), d_0)$; $N = R$
while $N \neq 0$ {
  foreach $(c(x), d) \in N$ {
    foreach $(c'(x), d') \in Next((c(x), d))$ {
      $(\tilde{c}(x), \tilde{d}) = Membership(R, (c'(x), d'))$
      if $\tilde{c}(x) \neq 0$ $N' = N' \cup (\tilde{c}(x), d')$
    }
  }
  $N = N'$; $N' = 0$
}

Function *Membership*
$\tilde{c}(x) = c'(x)$, $\tilde{d} = d'$
foreach $(c_i(x), d_i) \in R$ {
  if *isSubIsomorphic*(d_i, d') {
    $\tilde{c}(x) = \tilde{c}(x) \wedge \bar{c}_i(x)$
    if $\tilde{c}(x) = 0$ **return**
    if *found* = *FALSE* **and**
      isIsomorphic(d_i, d')
      $\tilde{d} = d_i$; *found* = *TRUE*
  }
}
if *found* = *FALSE* $R = R \cup \{(\tilde{c}(x), \tilde{d})\}$

Figure 4.1 *State Enumeration Using ISPs*

In this section, we present new data structure, called *ICS State Pairs* (ISPs), for performing state enumeration using ICS models. ISPs allow for sets of states that have the same assignment to datapath values to be represented efficiently. Intuitively, an ISP is a pair $(c(x), d)$, where $c(x)$ is an ROBDD representing a set of assignments to the finite latch variables x, and d is a directed acyclic graph (DAG) representing terms assigned to integer latches, memory, and predicates. Thus, $(c(x), d)$ represents a set of ICS states that all have the same set of ICS terms assigned to the integer variables but may have different assignments to the finite state variables. The state enumeration algorithm using ISPs, *ISP_Reach,* is presented in Figure 4.1. Here, R is a set of ISPs representing the set of reachable states of the system and N the frontier set. Starting from the initial state(s) of the model, represented by $(c_0(x), d_0)$, the algorithm computes transitions to states for which there are no states in R that are sub-isomorphic to it. $Next((c(x), d))$ returns a set of ISPs representing the set of states that have one step transitions from the states in $(c(x), d)$. Next states of the finite part

can be computed using BDDs. A special form of symbolic simulation is be used for computing the next states of the integer part. The function *Membership* searches the set of ISPs in R and removes finite states, which have sub-isomorphic integer assignments. Therefore, if $(c_1(x),d_1)$ is in R and a new state $(c_2(x),d_2)$ is found, then the states $c_2(x) \wedge c_1(x)$ can be removed from $c_2(x)$ if d_1 is sub-isomorphic to d_2. The rest of this section presents the details of the algorithm. In Section 4.1, we formally define ICS State Pairs and isomorphic checking using ISPs. In Section 4.2, we discuses how transitions between states represented as ISPs are computed.

4.1 ICS State Pairs

Consider a set of ICS states S of a model M with different assignments to the finite latch variables $x=(x_1,\ldots,x_l)$ but with isomorphic assignments to the datapath state variables (consisting of integer latches, predicates, and memory). We can represent S by an *ISP* $(c(x),d)$. Here $c(x)$ is an ROBDD representing the characteristic function $X_c : 2^l \rightarrow \{0,1\}$ of the set of all assignments to x by the states in S. $d=(V_d,E_d)$ is a labeled directed acyclic graph that is a syntactical representation of the set of assignments to integer latches, memory, and predicates, and is isomorphic to all datapath assignments of states in S. This representation is an extension of a congruence closure graph presented in *[NO80]* and we use their notation here. For each vertex $v \in V_d$, $\lambda(v)$ represents its label, $\delta(v)$ its out degree. The edges leaving each vertex are ordered, such that $v[i]$ is the i-th successor of v. Each term t is represented by a vertex v_t. For a constant term $t=c_i$ (or numeral), $\lambda(v_t)=c_i$ and $\delta(v_t)=0$. If $t=f_i(t_1,\ldots t_n)$, then $\lambda(v_t)=f_i$, $\delta(v_t)=n$ and $v_t[1],\ldots,v_t[n]$ represent the terms $t_1,\ldots,t_n$. The root vertices in d represent assignments of terms to integer latches, memory and predicates. Thus, if v represents an assignment of t to the integer latch l_i, then $\lambda(v)=l_i$, $\delta(v)=1$ and $v[1]$ represents t. If v represents an assignment of (a_i,d_i) to the memory element M_i, then $\lambda(v)=M_i$, $\delta(v)=2$ and $v[1]$ and $v[2]$ respectively represent terms a_i and d_i. If v represents an n-ary predicate $p_i(t_1,\ldots,t_n)$, then it is defined similarly, except that if the predicate is false in a state, then $\lambda(v)=\neg p_i$.

Given two ISPs $(c_1(x),d_1)$, $(c_2(x),d_2)$, we say that d_1 is *sub-isomorphic* to d_2 if there exists a function $\sigma : V_1 \rightarrow V_2$ such that if $\sigma(v_1)=v_2$ and $\lambda(v_1) \neq c_i$ then $\lambda(v_1)=\lambda(v_2)$ and $\sigma(v_1[i])=v_2[i]$. If d_1 is isomorphic to d_2, then σ is one-to-one and such that if $\sigma(v_1)=v_2$, and $\lambda(v_1)=c_i$ then $\lambda(v_2)=c_j$, otherwise $\lambda(v_1)=\lambda(v_2)$, and $\sigma(v_1[i])=v_2[i]$. Note that for the root vertex corresponding to a latch assignment, there is only one vertex that is can map to. However, for root vertices corresponding to memories, there may be multiple cases, since all address/value pairs assigned in the same memory element will have the same label. This is also the case for root vertices corresponding to predicates, since each such vertex is only labeled by the operation that it represents. It is straightforward to prove that two states are sub-isomorphic if and only if their ISP representations are sub-isomorphic.

Currently, we check for isomorphism (and sub-isomorphism) by choosing a mapping between root vertices in d_1 and d_2, and then recursively mapping each successor vertex down to the leaves. In the worst case, however, all possible cases may have to be explored.

4.2 Computing ISP State Transitions

Given $(c(x), d)$ corresponding to a set of states of M, $Next(c(x), d)$ computes the set $\{(c'_1(x), d'_1), (c'_2(x), d'_2), \ldots\}$ representing successor states in $(c(x), d)$. This next state computation is performed using a combination of an implicit technique to represent transitions between state variables in the finite part and explicit techniques for the integer datapath and memory. In the following, we denote by $b = (b_1, \ldots, b_m)$ a set of variables corresponding to predicate gates, which are 'inputs' to the finite part from the integer part. $w = (w_1, \ldots, w_n)$ denotes the set of variables corresponding to 'outputs' of the finite part going to the integer part, and $y = (y_1, \ldots, y_l)$ denotes the set of next state variables of the finite latches. To compute transitions between finite states, we first create a BDD representing the transition relation, $T_F(x, w, b, y)$, of the finite part. This can be obtained by taking the intersection of all the finite relations in M. Computing next states is then performed in two steps. First, we compute the function $U(w, b, y) = \exists x (T_F(x, w, b, y) \wedge c(x))$, which represents the set of possible transitions of the finite part. $T_F(x, w, b, y)$ assumes that b and w are free inputs and outputs of the finite part, therefore, some transitions in $U(w, b, y)$ may not be possible. Next, for each possible set of assignments of b, w, and y given in $U(w, b, y)$, we compute the set of next states $\{d'_i\}$ of the integer part by starting with w and d and performing a special form of symbolic simulation. If the outputs of the integer part are not consistent with b, then the transition is invalid and thrown out. Otherwise, we create a set of pairs $\{(c'(x), d'_i)\}$, where $c'(x)$ corresponds to assignments given by y. In general, since we attempt to compute d'_i for each possible assignment of b, w, and y, the number of cases that need to be considered is equivalent to the number of minterms in $U(w, b, y)$. In the next section, we show that we can consider assignments of b, w, y corresponding to a cube in $U(w, b, y)$ simultaneously. Since the number of cubes representing $U(w, b, y)$ can be exponentially smaller than the number of minterms, this optimization allows us to significantly speed up next state computation. In section 4.2.2, we discuss performing symbolic simulation of the integer part.

4.2.1 Cube Enumeration

In the following, we denote by 2 an absent literal in a cube, i.e. a *don't care*. Intuitively, the absence of a literal in a cube of $U(w, b, y)$ represents a set of assignments that are independent of case splitting (or case removal) that may occur in the integer part. Thus, $b_i = 2$ implies that the set of all other assignments to b, w, and y in the cube are the same for both $b_i = 0$ and $b_i = 1$. Even if $b_i = 0$ is inconsistent with the datapath, the choice of $b_i = 1$ does not require the other assignments to b, w, and y, to be reconsidered. Below we give the procedure for performing symbolic simulation with cubes, depending on whether the absent literal corresponds to a mux input (w_i), predicate output (b_i), or next state assignment (y_i) during symbolic execution of gates in M.

Mux Input. Assume $g = mux(w_i, i_1, i_2)$ and let t_1 and t_2 be the terms assigned to i_1 and i_2 by the partial state $s(g)$. If $P \rightarrow t_0 = t_1$ is valid, then assign the output t_1.

Here both cases do not need to be considered, since they will result in the same output assignment. If $t_1 = c$ (or t_2) is an unconstrained constant (is not assign to any other variable state), then assign $o = c$. Here, the case where $w_i = 1$ is contained in the case where $w_i = 0$. Otherwise, create two new partial states and consider both cases $w_i = 0$ and $w_i = 1$ separately.

Predicate Output. Simply continue value propagation. Note that there is an additional advantage here, since a validity check to determine the gate output value does not need to be performed. The correctness of this procedure can be argued by the operational semantics and the fact that it can be shown $F(s) = F(s_p) \cup F(s_{\overline{p}})$ (s_p denotes a state where a predicate p is true).

Finite Next State. If $y_i = 2$, then the next state integer assignment are the same for both $y_i = 0$ and $y_i = 1$ and thus both cases can be considered simultaneously.

4.2.2 Symbolic Simulation of the Datapath

Symbolic simulation is performed directly from the operational semantics. Terms are propagated through the gates in topological order. As reflected in the operation semantics, processing predicate gates and memory operation requires a validity check to be performed. We use the algorithm given in *[NO80]* to perform this check (it can be performed directly on the DAG). If case splitting results, case splitting results in a new DAG being created. Note that state reductions that can be performed by deleting extra information are also performed on-the-fly during execution of the gates of M.

5 Experimental Results

We have implemented the techniques presented in this paper in our second-generation ICS state reachability tool. In this section, we present experimental results on using our tool for performing both pipeline microprocessor verification and computing state invariants. All the experiments below were performed on a DEC-Alpha server running at 250Mhz with 1GB of main memory. Our ICS reachability tool uses the VIS *[VIS96]* verification system as a front end and the Cudd BDD package *[Som97].* The results of all our experiments are shown in Table 5.1. The columns consist of the number of ICS states reached, the number of ISPs, the number of control states, and the number of CPU seconds. In the following, a detail description is given of the experiments that were run. For the two example that did not terminate (ITC and DLX ISA), the results are reported until memory out occurred.

Architectural Verification. ICS can be used to perform architectural verification for pipeline microprocessors, in which one verifies that a pipelined implementation of a microprocessor satisfies its unpipelined specification. The unpipelined version, called the spec, represents the instruction set architecture, which consists of the programmer visible state and instructions. Programmer visible states include logical registers, the program counter and memory. Intuitively, one would like to verify that for any sequence of instructions given to both machines, when the pipeline completes, its programmer visible state will equal the programmer visible state given in the spec. ICS allows the pipeline processor specification and correctness criteria to be expressed very naturally as a safety property. Starting from the initial state, we run the two machines in parallel under the same set of instructions. A non-deterministic stall

is then asserted after an arbitrary number of instructions have been executed. Once the pipeline becomes empty, a comparison is made between the architectural states of the pipeline and the spec. One may also want to check that if a stall is asserted then the pipeline will eventually become empty. This check, which is naturally a liveness property, can be expressed as a safety property by showing that the number of cycles to flush the pipeline is bounded by a constant as determined by the user. We have applied this correctness criterion for verifying a 3-stage presented in *[BCMD90, BD94]* and our tool was able to reach a fixed point (CMU ISA in Table 5.1). In addition, we introduced two bugs into our design and our tool was able to capture both.

We also attempted to verify the 5-stage DLX pipeline presented in *[HP90]*. Unlike the 3-stage pipeline, the rates at which the DLX pipeline and its spec execute instructions are different. We solved this problem by using an instruction memory to model the program input and an (arbitrary) program location to denote the end of the program (instead of a stall signal). This approach requires no synchronization between the two machines. Our tool ran out of memory on this experiment (DLX ISA in Table 5.1), but we were able to find a bug that occurred in the branching logic of the machine.

It is clear that our method is computationally more expensive that the techniques presented in *[BD94, JDB95]*. The theoretical advantage here is in the generality and simplicity of the specification. The practical advantage is that we do not produce false negatives (or false positives), thus every bug that is found is a true error. Moreover, the user doesn't need to specify a state invariant.

Computing Reachable States for Invariance Properties. We computed the set of reachable control states for both the CMU and DLX pipeline (without the specification machine) and our tool was able to reach a fixed point for both examples (CMU Invariant and DLX Invariant in Table 5.1). We also ran our tool on the Island Tunnel Controller design presented in *[FJ95]* (ITC in Table 5.1). However, our procedure did not terminate for similar reasons as those given in *[ZSTCCL96]*.

Example	ICS States	ISPs	Control States	Time (sec)
CMU ISA	9	6	7	0.2
DLX ISA	4810	4753	33	2529.2
CMU Invariant	7	4	3	0.1
DLX Invariant	397	44	118	13.8
ITC	2996	746	18	2953.3

Table 5.1 *Experimental Results*

6 Conclusions

We have presented a new approach for automatically computing the set of reachable control states of systems modeled using ICS which allows datapath to be represented abstractly, hence enabling property verification on larger systems than what is possible with standard BDD-based techniques. Our approach is automatic and if it terminates will produce the exact set of reachable control states of the system. It can also be used for partial verification, by performing reachability for a finite number of

steps. Our experimental results show that our approach maybe useful not only as a standalone verification technique, but in conjunction with other verification procedures that may require an approximation to the set of reachable control states as input.

References

[Bry86] R. E. Bryant, *"Graph Based Algorithms for Boolean Function Manipulation"*, IEEE Trans. on Computers, C-35(8):677-691, August 1986.

[BCMD90] Jerry R. Burch, E. M. Clarke, K. L. McMillan, David L. Dill, *"Sequential Circuit Verification Using Symbolic Model Checking"*, Proc. Of the Design Automation Conf., 1990.

[BD94] Jerry R. Burch, David L. Dill, *"Automatic Verification of Pipelined Microprocessor Control"*, Computer Aided Verification, Stanford, CA, June 1994.

[Cor94] F. Corella, *"Automatic Verification of Behavioral Equivalence for Microprocessors"*, IEEE Transactions on Computers, 43(1):115-117, January 1994.

[CZXLC94] F. Corella, Z. Zhou, X. Song, M. Langevin, E. Cerny, *"Multiway Decision Graphs for Automated Hardware Verification"*, IBM technical report RC19676, July 1994.

[FJ95] K. Fisler and S. Johnson, *"Integrating Design and Verification Environments through a Logic Supporting Hardware Designs"*, Proc. IFIP Conference on Hardware Description Languages and their Applications, Chiba, Japan, Aug. 1995.

[HP90] John L. Hennessy, David A. Patterson, *"Computer Architecture A Quantitative Approach"*, Morgan Kaufmann Publishers, 1990.

[HB95] Ramin Hojati, Robert K. Brayton, *"Automatic Datapath Abstraction of Hardware Systems"*, Conference on Computer-Aided Verification, June 1995.

[HIB97] Ramin Hojati, Adrian J. Isles, and Robert K. Brayton, *"Automatic State Reduction Techniques for Hardware Systems Modeled Using Uninterpreted Functions and Infinite Memory"*, IEEE International High Level Design Validation and Test Workshop, Nov 1997.

[HIKB97] Ramin Hojati, Adrian J. Isles, Desmond Kirkpatrick, and Robert K. Brayton, *"Verification Using Uninterpreted Functions and Finite Instantiations"*, Formal Methods in CAD, November 1996.

[Hunt85] W. A. Hunt, Jr. *"FM8501: A verified microprocessor"*, Technical Report 47, University of Texas at Austin, Institute for Computer Science, Dec. 1985.

[IHB96] Adrian J. Isles, Ramin Hojati, and Robert K. Brayton, *"Reachability Analysis of ICS Models"*, SRC Techcon, September 1996.

[JDB95] R.B Jones, D. L. Dill and J. R. Burch, *"Efficient Validity Checking for Processor Verification"*, IEEE/ACM International Conference of Computer Aided Design, 1995

[LC91] M. Langevin, E. Cerny, *"Comparing Generic State Machines"*, Computer Aided Verification, July, 1991.

[NO80] Greg Nelson, Derek C. Oppen, *"Fast Decision Procedures Based on Congruence Closure"*, Journal of the ACM, 27(2):356-364, April 1980, June 1995.

[Sho79] R. E. Shostak, *"A Practical Decision Procedure for Arithmetic With Function Symbols"*, JACM Volume 26, No. 2, April 1979, pp. 351-360.

[Som97] F. Somenzi, *"CUDD: CU Decision Diagram Package, Release 2.1.1"*, Department of ECE, University of Colorado at Boulder, February 1997.

[SB90] M. Srivas and M. Bickford. *"Formal Verification of a Pipelined Microprocessor"*. IEEE Software, 7(5):52-64, Sept 1990.

[VIS96] The VIS Group, *"VIS: A system for Verification and Synthesis"*, Conference on Computer Aided Verification, July 1996.

[ZSTCCL96] Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, M. Langevin, *"Formal Verification of the Island Tunnel Controller Using Multiway Decision Graphs"*, Formal Methods in Computer-Aided Design, November 1996.