Comparing Generic State Machines*

M. Langevin E. Cerny

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal, C.P. 6128, Succ. A
Montréal, Québec, CANADA, H3C 3J7

e-mail: {langevin, cerny}@iro.umontreal.ca, FAX: (514) 343-5834

Abstract

This paper presents a technique for comparing generic state machines (i.e., machines where the size of manipulated data is not yet specified). The machine behavior is modeled using transfer formulas, a special kind of first order logic formulas. The technique is a mix of theorem proving methods with an automata comparison algorithm, in which first order logic terms are used to represent the values of the inputs, states and outputs of the machines.

1 Introduction

Attractive algorithms [5,7,8,9] have been proposed to compare finite state machines (synchronous circuit). In the case of circuits composed of a control part and data part, verification can be reduced to comparing only the control parts if a mapping is established between the data parts. If this mapping is not known, a complete comparison has to be performed, but this can be intractable due to the state explosion resulting from the data width. This is unfortunate especially when the data width is meaningless as far as the circuit behavior is concerned. Moreover, when the data width is not yet specified, i.e. generic circuits, this kind of comparison is impossible because such circuits are not finite state machines anymore.

This paper is concerned with the verification of generic state machines. The comparison of this kind of circuits can be tractable if the data nodes are considered as symbols representing bit vectors (terms), instead of sets of Boolean variables. Unfortunately, propositional logic is no more sufficient for modeling and comparing such synchronous circuits, and predicate calculus has to be used. Systems such as HOL [11] or Boyer-Moore [2] are not ideal for comparing generic state machines, however. HOL is too powerful and this implies that no decision procedure is available for proofs, while Boyer-Moore which is efficient for inductive proofs is not well adapted for substitution [18] that is required in state exploration of the product machine.

In this paper we present a formalism sufficiently powerful for modeling the behavior of generic state machines; it is a subset of predicate calculus. The language HOP is a different formalism specialized to model generic state machines [10]; even if an efficient algorithm is proposed to compose two such descriptions, nothing has been proposed to compare them. We then give an algorithm for comparing such

* Partially supported by NSERC Canada Grant No MEF0040113, and by the equipment loan of the CMC

machines; it is a mix of theorem proving methods with an automata comparison algorithm, in which first order logic terms are used to represent the values of the inputs, states, and outputs of the machines. In Section 2, we introduce the formalism and give an example. The comparison algorithm is presented in Section 3, including a partial solution to the difficult problem of detecting already visited states of generic machines. Finally, possible improvements of our approach are discussed in Section 4.

2 Modeling Generic State Machines

A generic state machine (or a generic synchronous circuit) is a machine where the width of some data paths is not yet numerically specified. This section introduces a formalism for modeling generic state machines, and presents how this model can be extracted from the register transfer level (RTL) description of a synchronous circuit.

2.1 Formalism

The objects **variable** and **operator** are used to represent the bit vector value of the input, output, and memory nodes. All objects possess an attribute which is the width of the bit vector represented by the object. A variable x representing a bit vector of width w is noted x:w. An operator is a function transforming a set (possibly empty) of bit vectors (the parameters) into a bit vector of a fixed width, e.g., predefined operators used in some HDL [1]. An operator OP representing a bit vector of width w, and taking p formal parameters $x_1,\ldots,x_p$ such that the i^{th} parameter represents a bit vector of width w_i, is noted $OP(x_1{:}w_1,\ldots,x_p{:}w_p){:}w$. The operators of arity zero, i.e., taking no parameters, are **operator-constants**.

A **bit-constant** of width w is a bit vector $(b_{w-1},\ldots,b_1,b_0)$ where $b_i = 0$ or 1 is the i^{th} bit of the constant. The truth values TRUE and FALSE are represented respectively with the one-bit constants (1) and (0). Also, a special symbol DC is used to represent don't care values; its width corresponds to the width of the node to which it is affected. Each DC symbol is considered a distinct variable symbol, and it is used to represent the initial value of certain memory nodes. Moreover, the DC symbols are used for behavioral modeling of incompletely specified machines.

The values that a circuit node can hold are represented with a **term** (a symbolic value). A term is defined recursively as:

i) A constant is a term,
ii) A variable is a term, and
iii) If $OP(x_1{:}w_1,\ldots,x_p{:}w_p){:}w$ is an operator and $T_1,\ldots,T_p$ are terms such that the width of T_i is w_i then $OP(T_1,\ldots,T_p)$ is a term.

We consider that the operators have well defined semantics, modeled with a set of rewriting rules which forms a complete reduction system (noetherian and confluent) [4]. This means that two terms are equivalent iff their reduced forms are syntactically equal. One such set of operators and rewriting rules is

presented in [16]. Moreover, known techniques can be used to assure that the set of rewriting rules is complete [4].

The 1-bit wide terms are **predicates**. An **assignment** is a homomorphic mapping ν from the set of terms into the set of constants [4]. The value of a term T at ν, noted $\nu(T)$, is the constant resulting of the evaluation of T where each of its variable symbols is replaced by a constant specified by ν. For example, let ADD(x:2,y:2):2 be an operator; if T = ADD(DC,DC) is a term and ν = {first DC = (0,1), second DC = (1,0)} is an assignment then $\nu(T)$ = (1,1). As mentioned earlier, the DC symbols are considered as distinct symbols.

The **alphabet** (image) of a term T of width w, noted $\alpha(T)$, is the set of constants represented by T: $\alpha(T) = \{y = (b_{w-1},\ldots,b_0) \mid \exists$ an assignment $\nu \ni y = \nu(T)\}$. For example, using the same operator as above and the variable symbols a:2 and b:2, $\alpha(ADD(a,b))$ = {(0,0), (0,1), (1,0), (1,1)} and $\alpha(ADD(a,a))$ = {(0,0), (1,0)}. The alphabet of a set $X = \{x_1:w_1,\ldots,x_n:w_n\}$ of n distinct variables, denoted $\alpha(X)$, is $\{0,1\}^w$ where $w = \sum w_i$. A **generic state machine** is a 6-tuple (X,Y,Z,I,δ,λ) where

$X = \{x_1:u_1,\ldots,x_n:u_n\}$ is the set of input variables,
$Y = \{y_1:v_1,\ldots,y_r:v_r\}$ is the set of state variables,
$Z = \{z_1:w_1,\ldots,z_m:w_m\}$ is the set of output variables,
I is the initial symbolic state,
$\delta:\alpha(X) \times \alpha(Y) \rightarrow \alpha(Y)$ is the next state function, and
$\lambda:\alpha(X) \times \alpha(Y) \rightarrow \alpha(Z)$ is the output function.

In the output and the next state functions, the input, state, and output variables are used to represent the value of the input, memory, and output nodes of the circuit. Generic nodes of the machine are those which have their width unspecified. A symbolic state of the machine is a tuple $(T_1,\ldots,T_r)$ where T_i, a term of width v_i, represents the value of y_i. Since the values of the input, output, and state variables are modeled with terms, the next state and output functions can be described using formulas of predicate calculus [17]. In our case, however, only a special kind of formulas is used, called **transfer formula** (TF). A TF for a non-constant term T of width w describes the possible values which correspond to T. The general form of a TF is: (similar to the VHDL case statement [21])

$cond_1 \wedge \mathbf{EQU}(T,T_1) \vee cond_2 \wedge \mathbf{EQU}(T,T_2) \vee \ldots \vee cond_k \wedge \mathbf{EQU}(T,T_k)$ where

i) **EQU** is a **data transfer predicate** (DTP) representing the transferred values,
ii) T_i's are **transferred terms**, the width of T_i is also w and T does not appear in T_i,
iii) the conditions $cond_i$ are formulas generated from a set $P_1,\ldots,P_h$ of predicates (the **control terms**), using only the connectives $\wedge$, $\vee$, and — (logical not),
iv) for all $i \neq j$, $cond_i \wedge cond_j$ = False, and
v) $\bigvee_{i=1}^{n} cond_i$ = True.

For any conditions, the TF represents one and only one transferred term T_i. Let Var(f) be the set of non-DC variables appearing in the control and transferred terms of the TF f. For describing the output function λ, a TF f is defined for each output variable of the machine, such that each variable in Var(f) is an input or a state variable. Similarly, for the next state function δ, a TF f is defined for each next state variable (i.e., primed state variable), such that each variable of Var(f) is an input or state variable. It was shown in [14] that TFs can be efficiently represented and manipulated using directed acyclic graph similar to BDDs [3], in which the internal nodes are labeled by the control terms and the leaf nodes contain the transferred terms.

Example: Consider a 3-word deep stack where each word consists of n bits. The inputs of the machine are the controls *nop* and *pop*, and the n-bit vector *in*, while the outputs are the error flag *err* and the n-bit vector *out*. The state variable of the machine are the stack pointer *sp* and the stack memory *sm*. If the control input *nop* is true then nothing happens, otherwise a pop (if the input *pop* is true) or a push (if the input *pop* is false) is performed. The output *err* is True if a pop is attempted when the stack is empty (*sp* = (1,1)), or a push when the stack is full (*sp* = (0,0)). The output *out* is the value of the top of the stack at any time (when the stack is empty *out* is zero, otherwise, it is the word of *sm* pointed by *sp*). The stack memory *sm* contains 3 words. The operator Mem(v_0:n,v_1:n,v_2:n):3*n symbolizes a memory containing 3 words of n bits, indexed from 0 to 2 inclusively. In fact, this operator represents an array where the operators Read and Write (defined below) are used to consult and update the array. The other operators used for describing the behavior of the stack are:

Zero():n	represents the n-bit vector zero,
EquZ(x:2):1	return true iff x is zero,
Not(x:2):2	complements all bits of x,
Inc(x:2):2	increments x modulo 4,
Dec(x:2):2	decrements x modulo 4,
Read(m:3*n,a:2):n	read memory m at address a, and
Write(m:3*n,a:2,d:n):3*n	write data d in memory m at address a.

The machine description is Stack-Spec = (X_S,Y_S,Z_S,I_S,δ_S,λ_S) where

X_S = {in:n,nop:1,pop:1},
Y_S = {sp:2,sm:3*n},
Z_S = {out:n,err:1},
I_S = ((1,1), Mem(DC,DC,DC)),
δ_S is represented using the following TFs, one for each of the next state variables,

$$(nop \vee pop \wedge EquZ(Not(sp)) \vee \overline{pop} \wedge EquZ(sp)) \wedge \mathbf{EQU}(sp',sp) \vee$$
$$\overline{nop} \wedge pop \wedge \overline{EquZ(Not(sp))} \wedge \mathbf{EQU}(sp',Inc(sp)) \vee \overline{nop} \wedge \overline{pop} \wedge \overline{EquZ(sp)} \wedge \mathbf{EQU}(sp',Dec(sp))$$

$(nop \vee pop \vee \overline{pop} \wedge EquZ(sp)) \wedge \mathbf{EQU}(sm',sm) \vee$
$\overline{nop} \wedge \overline{pop} \wedge \overline{EquZ(sp)} \wedge \mathbf{EQU}(sm',Write(sm,Dec(sp),in))$

λ_S is described using the following TFs, one for each of the output variables.

$EquZ(Not(sp)) \wedge \mathbf{EQU}(out,Zero) \vee \overline{EquZ(Not(sp))} \wedge \mathbf{EQU}(out,Read(sm,sp))$

$(nop \vee pop \wedge \overline{EquZ(Not(sp))} \vee \overline{pop} \wedge \overline{EquZ(sp)}) \wedge \mathbf{EQU}(err,(0)) \vee$
$\overline{nop} \wedge (pop \wedge EquZ(Not(sp)) \vee \overline{pop} \wedge EquZ(sp)) \wedge \mathbf{EQU}(err,(1))$

The first TF of the next state function of this machine describes the next value of the node *sp*; it is incremented if a pop has occurred, decremented if a push has occurred, or unchanged if no operation has occurred. The other TFs can be interpreted in a similar way.

2.2 Composition

Since TFs can be used for describing RTL components of synchronous circuits, the generic state machine description can be easily extracted from an implementation defined in terms of interconnected components. The designer specifies its interface, i.e., its set of input and output variables. Also, the initial state is specified or can be computed (e.g., reset signal). The state variables of the circuit are the state variables appearing in the component models. The interconnections are the internal nodes of the circuit, they carry no state information. Every loop in the circuit must contain clocked memory elements. In order to obtain the generic state machine description, the next state and output functions are extracted from the circuit implementation by a composition operation.

The extraction algorithm is similar to [14] for comparing two synchronous circuits having the same set of registers; the idea is to transform all TFs into an **observable** form. A TF f is observable iff each variable of Var(f) is an input or a state variable. In order to compute the output function, the TF for a particular output variable is found in one component model of the circuit implementation, and it suffices to transform this TF into its observable form. This is achieved by substituting all internal variables appearing in the TF by their corresponding observable TF, and by applying rewriting rules to the operators appearing in the TF [14,15]. The extraction of the next state function proceeds in a similar fashion. In fact, the machine description is the conjunction of all component models in which predicate calculus rules are applied to abstract internal nodes.

Figure 1 shows a possible implementation of Stack-Spec, named Stack-Impl (the clock signal is implicit). The stack is constructed using shift registers, instead of a memory bank. We assume that the counter is reset at power on (this initial state should be verified, e.g. [19]). The extracted generic state machine is Stack-Impl = $(X_I,Y_I,Z_I,I_I,\delta_I,\lambda_I)$ where:

X_I = {in:n,nop:1,pop:1},

Y_I = {c:2,r_1:n,r_2:n,r_3:n},
Z_I = {out:n,err:1},
I_I = ((0,0),DC,DC,DC),

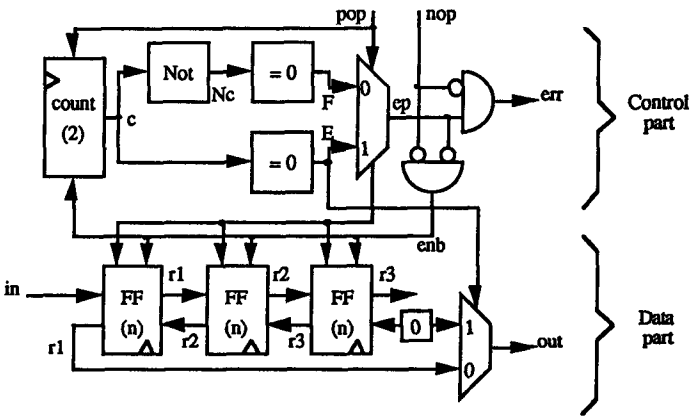

Figure 1: Implementation of the stack

δ_I is described with the following TFs:

Derived observable TF for c

$(nop \vee pop \wedge EquZ(c) \vee \overline{pop} \wedge EquZ(Not(c))) \wedge \mathbf{EQU}(c',c) \vee$

$\overline{nop} \wedge pop \wedge \overline{EquZ(c)} \wedge \mathbf{EQU}(c',Dec(c)) \vee \overline{nop} \wedge \overline{pop} \wedge \overline{EquZ(Not(c))} \wedge \mathbf{EQU}(c',Inc(c))$

Derived observable TF for r_1

$(nop \vee pop \wedge EquZ(c) \vee \overline{pop} \wedge EquZ(Not(c))) \wedge \mathbf{EQU}(r_1',r_1) \vee$

$\overline{nop} \wedge pop \wedge \overline{EquZ(c)} \wedge \mathbf{EQU}(r_1',r_2) \vee \overline{nop} \wedge \overline{pop} \wedge \overline{EquZ(Not(c))} \wedge \mathbf{EQU}(r_1',in)$

Derived observable TF for r_2

$(nop \vee pop \wedge EquZ(c) \vee \overline{pop} \wedge EquZ(Not(c))) \wedge \mathbf{EQU}(r_2',r_2) \vee$

$\overline{nop} \wedge pop \wedge \overline{EquZ(c)} \wedge \mathbf{EQU}(r_2',r_3) \vee \overline{nop} \wedge \overline{pop} \wedge \overline{EquZ(Not(c))} \wedge \mathbf{EQU}(r_2',r_1)$

Derived observable TF for r_3

$(nop \vee pop \wedge EquZ(c) \vee \overline{pop} \wedge EquZ(Not(c))) \wedge \mathbf{EQU}(r_3',r_3) \vee$

$\overline{nop} \wedge pop \wedge \overline{EquZ(c)} \wedge \mathbf{EQU}(r_3',Zero) \vee \overline{nop} \wedge \overline{pop} \wedge \overline{EquZ(Not(c))} \wedge \mathbf{EQU}(r_3',r_2)$

λ_I is described with the following TFs:

Derived observable TF for out

$EquZ(c) \wedge \mathbf{EQU}(out,Zero) \vee \overline{EquZ(c)} \wedge \mathbf{EQU}(out,r_1)$

Derived observable TF for err

$(nop \vee pop \wedge \overline{EquZ(c)} \vee \overline{pop} \wedge \overline{EquZ(Not(c))}) \wedge \mathbf{EQU}(err,(0)) \vee$

$\overline{nop} \wedge (pop \wedge EquZ(c) \vee \overline{pop} \wedge EquZ(Not(c))) \wedge \mathbf{EQU}(err,(1))$

3 Comparison Algorithm

This section presents a comparison algorithm for verifying the equivalence of two generic state machines. The skeleton of the algorithm is presented in the Section 3.1, while some critical parts of the algorithm are detailed in the subsequent sections: Comparison of the output values, and computation of the new reached states of the product automaton.

3.1 Algorithm Skeleton

The goal of the comparison is to verify if two generic state machines have the same observable behavior. First, the two machines must have the same interface. Let $M_1 = (X,Y_1,Z,I_1,\delta_1,\lambda_1)$ and $M_2 = (X,Y_2,Z,I_2,\delta_2,\lambda_2)$ be the generic state machines to be compared, where $Y_1 = \{y_{11},\ldots,y_{1q}\}$ and $Y_2 = \{y_{21},\ldots,y_{2r}\}$. As in [8], the idea is to explore the reachable state space of the product machine $M_1 \times M_2$, without constructing it explicitly. A total symbolic state of the product machine is the concatenation of symbolic states of M_1 and M_2. In particular, the total initial state I_T is the concatenation of I_1 and I_2. The next state and output functions of the product machine are defined as usual [13]. Our algorithm is a compromise between the breadth first traversal [8] and the depth first traversal [9]. Symbolic input values are fed to the machine during the traversal, but symbolic states are visited one at the time. The algorithm consists in enumerating the symbolic states of the product machine reachable from the total initial state. Since a symbolic state in fact represents a set of numeric states of the machine, the algorithm does not degenerate into state enumeration. The algorithm is presented at Figure 2.

```
PROCEDURE Compare-GSM(M1, M2);
    Var     Reach, From, New: Set-Of-Total-States;
            S: Total-State;                          # Concatenation of Y1 and Y2
            X: Input;
    BEGIN
    Reach := From := {IT};
    WHILE (From ≠ Ø) DO
        BEGIN
        S := An-Element-Of(From);
        From := From - {S};
        X := New-Input();
        Compare-Ouput-Values(M1, M2, X, S);
        New := Generate-New-Reached-States(M1, M2, X, S, Reach);
        From := From ∪ New;
        Reach := Reach ∪ New;
        END;
    END;
```

Figure 2: Comparison algorithm

The set *Reach* is used to keep track of the symbolic states reached during the exploration, while the set *From*, a subset of *Reach*, contains the reached states not already visited. The set *New* computed from a particular state S using the procedure *Generate-New-Reached-States* (to be described in Section 3.3) contains all the states reachable from Y but not included in the *Reach* set. The procedure *New-Input* generates new symbolic variable(s) representing the value of the input(s). The traversal of the state space proceeds in a depth first or breadth first manner depending on the state sequence returned by the procedure *An-Element-Of*.

3.2 Comparison of Output Values

The comparison of the output values at a particular symbolic total state S under a symbolic input vector X is performed by the *Compare-Output-Value* procedure. For each output z two TFs TF_1 and TF_2 are computed using λ_1 and λ_2, respectively, based on the values of X and S. The sets $Var(TF_1)$ and $Var(TF_2)$ contain only the symbolic inputs of the product machine. If these TFs are equivalent for each output then the two machines have the same observable behavior in state S.

Since the system of rewriting rules is complete and the TFs are represented as BDDs, two TFs are equivalent iff their reduced forms are syntactically equal [14,15]. For example, during the comparison of Stack-Spec and Stack-Impl, the total initial state I_T of the state node (sp, sm, c, r_1, r_2, r_3) is ((1,1), Mem(DC,DC,DC), (0,0), DC, DC, DC); in this state the following TFs are computed for *out* and *err*, for both the Stack-Spec and Stack-Impl, given the symbolic input (in_1,nop_1,pop_1):

EQU(out,Zero)

$(nop_1 \vee \overline{pop_1}) \wedge$ **EQU**(err,(0)) $\vee$ $\overline{nop_1} \wedge pop_1 \wedge$ **EQU**(err,(1))

Consequently, the two machines are equivalent in the initial state. As will be seen in the next section, the total state S_1 = ((1,0), Mem(DC,DC,in_1), (0,1), in_1, DC, DC) is reachable from I_T. Again, the machines are equivalent in the state S_1 since the following TFs are computed for *out* and *err*, for both the Stack-Spec and Stack-Impl, given the symbolic input (in_2,nop_2,pop_2):

EQU(out,in1)

EQU(err,(0))

3.3 Computation of New Reached States

This section is concerned with the computation of the states reachable from a symbolic state S under the symbolic input X, and the detection of states reached previously. This is performed by the *Generate-New-Reached-States* procedure. First, the computation of the reachable states is explained. The TF for the next state tuple (y_{11}',...,y_{1q}',y_{21}',...,y_{2r}') can be computed from the next state function δ_1 and δ_2, based on the value of X and S. Let f be the TF resulting from the conjunction of the TFs for the next state variables.

Here again, the only non-DC variables that can appear in f, i.e., Var(f), are the symbolic inputs of the product machine. The reachable states are the data transferred by f, i.e., the transferred terms in the leaves of the BDD graph representation of f. For example, the possible next state values of the product machine Stack-Spec×Stack-Impl reachable directly from the initial state are described by the following TFs, one for each next state variable:

$(nop_1 \vee pop_1) \wedge \mathbf{EQU}(sp',(1,1)) \vee \overline{nop_1} \wedge \overline{pop_1} \wedge \mathbf{EQU}(sp',(1,0))$

$(nop_1 \vee pop_1) \wedge \mathbf{EQU}(sm',Mem(DC,DC,DC)) \vee \overline{nop_1} \wedge \overline{pop_1} \wedge \mathbf{EQU}(sm',Mem(DC,DC,in_1))$

$(nop_1 \vee pop_1) \wedge \mathbf{EQU}(c',(0,0)) \vee \overline{nop_1} \wedge \overline{pop_1} \wedge \mathbf{EQU}(c',(0,1))$

$(nop_1 \vee pop_1) \wedge \mathbf{EQU}(r_1',DC) \vee \overline{nop_1} \wedge \overline{pop_1} \wedge \mathbf{EQU}(r_1',in_1)$

$\mathbf{EQU}(r_2',DC)$

$\mathbf{EQU}(r_3',DC)$

The transferred terms of the conjunction of the above TFs represent the two following states:

((1,1), Mem(DC,DC,DC), (0,0), DC, DC, DC),
((1,0), Mem(DC,DC,in_1), (0,1), in_1, DC, DC).

We can notice that the first state has been reached earlier (it is the initial state), while the second state is new. The detection of already reached states is performed by comparing the reachable states with the set of reached states: If the set of numeric states represented by a reachable symbolic state S is included in the set represented by a reached symbolic state then we can conclude that the state S has been reached previously. As will be seen below, this is just a sufficient condition, not a necessary one. In the following, we suppose that each DC symbol is renamed using an unique variable symbol. As for terms, the set of numeric states represented by a symbolic state S is the alphabet $\alpha(S)$ of S. Let $S = (T_1{:}w_1,\ldots,T_k{:}w_k)$ be a symbolic state, then $\alpha(S) = \{(c_1,\ldots,c_k)$ where c_i is a w_i-bit constant $\mid \exists$ an assignment $v \ni c_i = v(T_i)\}$.

A **substitution** is a homomorphic mapping σ from terms into terms (states into states), associating terms to some variables appearing in a term (state) [12]. A term (state) S is an **instance** of a term (state) T iff there exist a substitution σ such that $S = \sigma(T)$. For example, the state ((1,1), Mem(DC_6,DC_7,in_1), (0,0), DC_8, DC_9, Zero), reached after a push and a pop from I_T, is an instance of the initial state ((1,1), Mem(DC_0,DC_1,DC_2), (0,0), DC_3, DC_4, DC_5) since the following substitution can be used: $\sigma = \{DC_0 \leftarrow DC_6, DC_1 \leftarrow DC_7, DC_2 \leftarrow in_1, DC_3 \leftarrow DC_8, DC_4 \leftarrow DC_9, DC_5 \leftarrow$ Zero$\}$. This is an important result:

Theorem: For all symbolic terms (states) T and substitution σ: $\alpha(T) \supseteq \alpha(\sigma(T))$.

The proof is based on the fact that all occurrences of each variable of T are replaced by the same term. This can only reduce the possible numeric values represented by the occurrences of the variable in T, and thus also the numeric values represented by T, i.e., $\alpha(T)$. Using this theorem, we can conclude that a state S has been reached previously if there exists a state S_i in *Reach* such that S is an instance of S_i. Of course, this is a sufficient condition, but not a necessary one. There could be two state S_i and S_j such that $\alpha(S_i) \cup \alpha(S_j) \supseteq \alpha(S)$ where S is neither an instance of S_i nor of S_j, but this is a harder condition to detect. To determine if a term is an instance of another one, a simple adaptation of the unification algorithm can be used [12].

In order to improve the detection process, the set *Reach* should be in a reduced form, i.e., no state in the set is an instance of another state of the set. Using this algorithm, the comparison of the two stack machines is performed by exploring the following four symbolic states:

((1,1), Mem(DC,DC,DC), (0,0), DC, DC, DC),
((1,0), Mem(DC,DC,in_1), (0,1), in_1, DC, DC),
((0,1), Mem(DC,in_2,in_1), (1,0), in_2, in_1, DC),
((0,0), Mem(in_3,in_2,in_1), (1,1), in_3, in_2, in_1).

However, what can we say about the finite termination of the exploration? The sequence of new total states S_1,S_2,... reachable from the initial state must be bounded. We know that the generic nodes of the machine can take an unbounded number of values since their sizes are unspecified. However, we remark that in the traversal of the product machine of Stack-Spec and Stack-Impl, the generic constant value (i.e., Zero) never appears as a parameter of an operator in a reduced term. In a case like this, an infinite sequence of symbolic states where each one is not the instance of another one is impossible, hence finite exploration is assured. This result can be generalized: We know that a term can be represented as a tree where the leaf nodes are variable or constant symbols, and the internal nodes are operator symbols [4]. The **depth** of a node (subterm) in a term is its distance from the root. If the depth of generic constants is always finite in all reduced terms computed during the comparison of machines then a finite exploration is assured. This result holds because an infinite number of terms where each term is not an instance of another one cannot be computed. In the case where the finite depth condition on the generic constants is not verified, finite exploration is not assured. It could be interesting to determine conditions that imply infinite exploration, but this may be undecidable in general.

4 Conclusion

This paper has presented a technique for comparing generic state machines described at the RT level of abstraction. The originality of the technique is in the combination of theorem proving methods and an automata equivalence algorithm, in which first order logic terms are used to represent the values of the inputs, states and outputs of the machines.

A prototype of our system is under construction, as an evolution of [14,15], however, a number of problems still must be resolved: The possible state space explorable during the comparison of two machines

must be studied in more depth to assure a finite search. Also, a complete system of rewriting rules must be computed (statically or dynamically), because it is required for comparing formulas.

Further research is required to see if states could be represented symbolically using characteristic functions [8,20] and if the incremental technique based on cross-controllability calculation [6] could be used in the case of generic machines.

References

[1] M. R. Barbacci, "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems", IEEE Trans. on Comp., Vol. C-24, No. 2, February 1975.

[2] R. S. Boyer, J. S. Moore, "A Computational Logic", ACM Monograph Series, Academic Press Inc., 1979.

[3] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Trans. on Comp., Vol. C-35, No. 8, August 1986.

[4] B, Buchberger, R. Loos, "Algebraic Simplification", in Buchberger and al. eds., Computer Algebra: Symbolic and Algebraic Computation, Springer, 1982.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond", in Proc. of the Int. Work. on Formal Methods in VLSI Design, Miami, January 1991.

[6] E. Cerny, C. Mauras, "Tautology Checking using Cross-Controllability and Cross-Observability Relations", ICCAD, Santa Clara, November 1990.

[7] O. Coudert, C. Berthet, J. C. Madre, "Verification of Synchronous Sequential Machines Using Symbolic Execution", in Proc. of the Work. on Automatic Verification Methods for Finite State Systems, Grenoble, June 1989.

[8] O. Coudert, C. Berthet, J. C. Madre, "Verification of Sequential Machines Using Boolean Functional Vectors", in L. Claesen, editor, Proc. of the int. Work. on Applied Formal Methods for Correct VLSI Design, Leuven, November 1989.

[9] S. Devadas, H.-K. T. Ma, A. R. Newton, "On the Verification of Sequential Machines at Different Levels of Abstraction", IEEE Transaction on CAD, Vol. 6, No. 7, June 1988.

[10] G. C. Gopalakrishnan, R. M. Fujimoto, V. Akella, N. S. Mani, "HOP: A Process Model for Synchronous Hardware; Semantics and Experiments in Process Composition", Integration: The VLSI Journal, August 1989.

[11] M. Gordon, "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware", in Formal Aspects of VLSI Design, 1986.

[12] K. Knight, "Unification: A Multidisciplinary Survey", ACM Computing Surveys, Vol. 21, No. 1, 1989.

[13] S. Kohavi, "Switching and Finite Automata Theory", McGraw-Hill, New-York, 1978.

[14] M. Langevin, "Automated RTL Verification Based on Predicate Calculus", in Proc. of the Work. on Computer Aided Verification, Rutgers, June 1990.

[15] M. Langevin, E. Cerny, "Verification of Processor-Like Circuit", in Proc. of the Advanced Research Workshop on Correct Hardware Design Methodologies, Turin, June 1991.

[16] T. Larsson, "Hardware Verification Based on Algebraic Manipulation and Partial Evaluation", in Proc. of the Int. Working Conf. on the Fusion of Hardware Design and Verification, Glasgow, July 1988.

[17] Z. Manna, "Mathematical Theory of Computation", McGraw-Hill, 1974.

[18] L. Pierre, "The Formal Proof of the Min-Max Sequential Benchmark Described in CASCADE Using the Boyer-Moore Theorem Prover", in L. Claesen, editor, Proc. of the int. Work. on Applied Formal Methods for Correct VLSI Design, Leuven, November 1989.

[19] C. Pixley, G. Beihl, "Quotient and Isomorphism Theorems of a Theory of Sequential Hadrware Equivalence", in Proc. of the Int. Work. on Formal Methods in VLSI Design, Miami, January 1991.

[20] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, A. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines Using BDDs", ICCAD, Santa Clara, November 1990.

[21] VHDL Language Reference Manual, Intermetrics Inc., 1987.