

Computing Binary Decision Diagrams for VHDL Data Types

Ronald Herrmann ^{*†}

Hergen Pargmann ^{‡§}

OFFIS
Westerstraße 10-12
26121 Oldenburg, Germany

University of Oldenburg
Ammerländer Heerstraße 114-118
26111 Oldenburg, Germany

Abstract

In this paper we present the translation of VHDL data types into a BDD representation. We describe a computation method which has been applied to the complete set of finite VHDL data types. This includes the encoding of VHDL variables in a set of binary variables and the construction of a total variable order. We are using this technique to build symbolic transition systems and during symbolic model checking for VHDL to translate propositions into BDDs.

1 Introduction

The ESPRIT III project FORMAT (Formal Methods in Hardware Verification) will create a framework for the verification of VHDL implementations for industrial use. VHDL provides a rich set of data types; the price to be paid for this is a higher complexity which has to be handled in the verification of a design. One way to verify that a VHDL implementation meets its specification is to use symbolic model checking [3, 9]. We developed a method to compile VHDL implementations into symbolic transition systems [10] which are represented efficiently using *reduced ordered binary decision diagrams* (RoBDDs, BDDs) [2, 1].

The paper describes the translation of *VHDL data types* into BDDs. This technique is used during the construction of the symbolic transition system and during model checking to compute the BDD representation of the propositions. That provides a link between the transition system and the temporal logic formulae.

The generation of symbolic transition systems for VHDL consists of two steps [10]. In the first phase we

compile behavioral VHDL into a Petri net representation as described in the [5] (EURO-VHDL'94). In the second phase we compile the control flow and the Petri net annotation given as VHDL expressions into BDDs and compute the resulting transition system [10].

The restrictions in our work are only that data types have to be finite and operators must act on finite domains. Most of the available verification tools e.g. the V-Formal tool¹ are restricted to a smaller subset of VHDL so that they are restricted to combinatorial circuits or synchronous state machines with in general the same state-encoding.

Another reason to look for a very efficient treatment of the data types is that data variables contribute much to the so called *state explosion problem*. The verification approach of the FORMAT project has shown that we have to handle only a few hundred control states for some typical case studies with more than 2000 lines of VHDL code but that the data state space for these examples consists of some hundred binary variables which yields to a data state space of more than 2^{100} potential states.

The main idea of our method is the use of a BDD based structure to handle VHDL data types, which includes the representation of variables, constants and intermediate results during operations on this structure. To this end we give the formal semantics of this structure which complies with the semantics of VHDL data types.

This paper is organized as follows. In the next section we give a small example. Section 3 recapitulates the supported VHDL data types and section 4 shows the computation of the binary state space and the construction of an initial global ordering on the BDD variables. In section 5 we define a BDD-based structure which is the basis of our approach and we describe some symbolic operations on this structure. Finally section 6 comprises the conclusion and an outlook.

^{*}The work of this author was supported by ESPRIT project 6128 FORMAT

[†]E-Mail:ronald.herrmann@offis.uni-oldenburg.de

[‡]The work of this author was supported by the "Volkswagenstiftung" project "Informatiksysteme"

[§]E-Mail:hergen.pargmann@informatik.uni-oldenburg.de

¹developed by Bull, distributed by COMPASS Design Automation, Columbia MD 21045

2 Example

The following example shows that the basic structure consists mainly of *type information* and a *vector of BDDs* which encodes a constant value of a VHDL data type, a set of values defined by a VHDL variable or intermediate results.

Example 1 Consider the assignment

$$index' = index + 1$$

where the new value of the VHDL variable *index* is the old value of the variable add up with the constant value 1. Consider the VHDL variable *index* defined as variable *index* : integer range 0 to 7;.

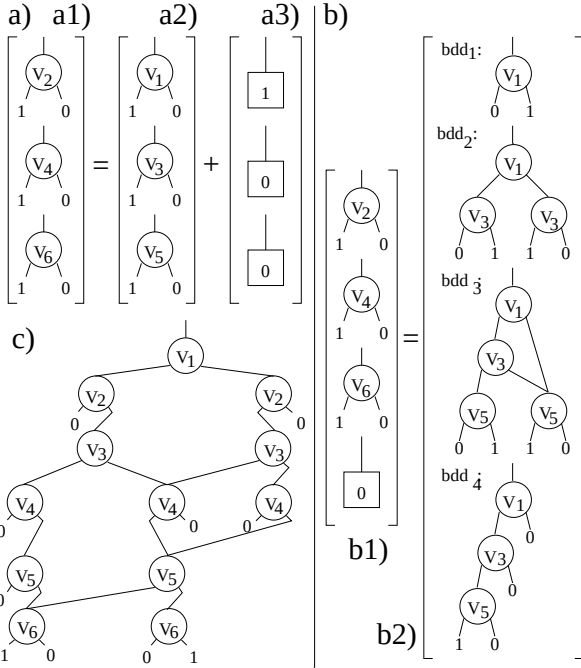


Figure 1: BDDs for example $index' = index + 1$. The left resp. right arc of a BDD variable node represents the *true* resp. *false* case. 1 represents the terminal node *true* and 0 *false*.

Three binary variables are necessary to represent variable *index*. In order to compute the BDD for a data transformation, we use two sets of three binary variables each, one set to represent the current value *v* and one to represent the new value *v'*. According to heuristics for computing a variable order we interleave these binary variables (cf. section 4). We represent *index* through the variables *v5*, *v3*, *v1* and *index'* through the variables *v6*, *v4*, *v2*. The BDD variables *v1* and *v2* represent the least significant bit of each representation and the BDD variable ordering δ is given as follows: $v_1 <_{\delta} v_2 <_{\delta} v_3 <_{\delta} v_4 <_{\delta} v_5 <_{\delta} v_6$.

The algorithm for the computation of the BDDs works bottom up. We represent the referenced objects *index'*, *index* and 1 in terms of vectors of BDDs. The representations of *index'* and *index* both are vectors with three BDDs where each of the BDDs consists only of one nonterminal node referencing the initial BDD variable of the binary encoding of the VHDL object. Figure 1 a1) resp. a2) shows the representation of *index'* resp. *index*. The constant 1 can be represented through a vector with one BDD (*true*), but to equal the types we enlarge this representation with adding additional BDDs constant *false* to the vector (cf. figure 1 a3)).

Now we perform the addition using a function *ADD*, which is defined on vectors of BDDs. For simplification we write BDD^n to denote the set of BDD vectors with length *n* and $x[j]$ to select the *j*-th BDD of a BDD vector *x*. *xor_{BDD}* (exclusive disjunction), *ite_{BDD}* (if-then-else), *or_{BDD}* (disjunction), *and_{BDD}* (conjunction) and *not_{BDD}* (negation) are standard operations defined on BDDs. 1_{BDD} denotes the BDD constant *true* and 0_{BDD} the BDD constant *false*.

```

 $BDD^{n+1}$  ADD( $BDD^n$  x,  $BDD^n$  y) {
  local  $BDD^{n+1}$  result;
  local  $BDD$  help, carry =  $0_{BDD}$ ;
  for i=1 to n do
    help = xorBDD(y[i], carry);
    carry = iteBDD(x[i], orBDD(y[i], carry),
                  andBDD(y[i], carry));
    result[i] = iteBDD(x[i], notBDD(help), help);
  od;
  result[n + 1] = carry;
  return(result)}

```

The BDD vector shown in b2) is the result of *ADD*(a2, a3) resp. *ADD*(*index*, 1) resp. *index* + 1. The important point is that we use a BDD vector to represent the intermediate result which consists of four BDDs (b2). This is caused by the possible overflow. To equal the types we enlarge the representation of *index'* by adding a leading BDD 0_{BDD} (b1). The relational BDD for example 1 is shown in figure 1 c). In the last step we have to handle the assignment operator. If we need a functional representation of the data transformation the result can be derived from figure 1 b). The new value of the BDD variable *v2* is represented through BDD *bdd1*, for *v4* and *v6* the new values are represented through *bdd2* resp. *bdd3*. If we would like to compute a relational BDD the result BDD is built as the conjunction of the equivalence of the elements of the vectors according to the vector position (cf. section 5.2 definition 8).

3 VHDL Data Types

As mentioned in the introduction the restrictions to our work are only that data types have to be finite and operators must act on finite domains. Finite VHDL data types can be divided into two groups: *scalar types* and *composite types*. We support *enumeration types* and *integer types* of limited range as *scalar types* and *composite types*. This includes both arrays and records of values which may have components of any supported type [8]. It is allowed to declare subtypes as defined in [8].

Now we give an abstract syntax of the supported generalized VHDL type τ .

Definition 1 Type τ . *The set $\mathbf{T}$ of VHDL types τ is inductively defined as follows:*

$\langle \tau \rangle ::= \langle \text{simple } \tau \rangle \mid \langle \text{composite } \tau \rangle$
 $\langle \text{simple } \tau \rangle ::= \text{enum}(e_1, \dots, e_n) \mid \text{sr}(o, r)$
 $\langle \text{composite } \tau \rangle ::= \text{array}(\langle \text{simple } \tau \rangle, \langle \tau \rangle) \mid \text{record}(\text{list of } (\langle \text{identifier } \rangle, \langle \tau \rangle))$
where $e_1, \dots, e_n$ are identifier denoting elements of an enumeration type. Offset $o \in \mathbf{Z}$ denotes the lower bound of a subrange type (sr) with range $r \in \mathbf{N}$ elements so that $o + r - 1$ defines the upper bound. An array is a structure with n elements of the same type $\langle \tau \rangle$. n is defined by the number of elements of $\langle \text{simple } \tau \rangle$ and each element of $\langle \text{simple } \tau \rangle$ defines the index of an array element. An element of a record can be denoted by the corresponding identifier and has an associated type $\langle \tau \rangle$.

The function *Size* determines the number of bits which are necessary to represent an object of type τ in a binary representation.

Definition 2 Size. *The function $\text{size} : \mathbf{T} \rightarrow \mathbf{N}$ is inductively defined as follows:*

- $\text{size}(\text{enum}(e_1, \dots, e_n)) = \lceil \log_2(n) \rceil$,
- $\text{size}(\text{sr}(o, r)) = \lceil \log_2(r) \rceil$,
- $\text{size}(\text{array}(\text{enum}(e_1, \dots, e_n), et)) = n * \text{size}(et)$
- $\text{size}(\text{array}(\text{sr}(o, r), et)) = r * \text{size}(et)$
- $\text{size}(\text{record}((id_1, et_1), \dots, (id_n, et_n))) = \sum_{i=1}^n \text{size}(et_i)$

The following function assigns to each constant value a binary representation.

Definition 3 type2bin. *The function*

$\text{type2bin} : \mathbf{T} \rightarrow \bigcup_{\tau \in \mathbf{T}} (\mathbb{D}_\tau \rightarrow \mathbb{B}^{\text{size}(\tau)})$ *assigns a binary coding to each data domain $\mathbb{D}_\tau$ of a VHDL type $\tau \in \mathbf{T}$. Instead of $\text{type2bin}(\tau)$ we write type2bin_τ .*

If v is a vector with n elements representing the binary coding then $v[j]$ is the value of the j -th element of the vector ($0 \leq j < n$).

Consider the type *Severity_Level* = **enum**(note, warning, error, failure). Each element can be represented using an integer value which represents the position of the element implicit defined in the type definition. Because we need a binary representation we can use the binary encoding of the integer position. The following table shows the result of the function *type2bin* applied to type *Severity_Level*(SL).

SL	note	warning	error	failure
position	0	1	2	3
type2bin	(0 , 0)	(1 , 0)	(0 , 1)	(1 , 1)

A VHDL expression can be built using the described types and standard operations defined on these types [8]. Let *EXPR* be the set of all VHDL expressions. The type concept is strong except the use of the type *sr*. This means that for operations on objects with type *sr* the offsets and/or the ranges of the type need not be equal. We adapt the offsets and the ranges of two arguments of type *sr* automatically (cf. section 5.2).

The semantics of VHDL expressions as defined in [8] will be denoted by the function $\xi[\cdot]$:

$\xi[\cdot] : \text{EXPR} \rightarrow \text{VALUATIONS} \rightarrow \text{DOMAIN}$

We denote the value of a VHDL expression $e \in \text{EXPR}$ under a given valuation ρ_{VHDL} of the free VHDL variables by $\xi[e]_{\rho_{\text{VHDL}}} \in \text{DOMAIN}$.

Let $\text{type} : \text{EXPR} \rightarrow \mathbf{T}$ be a function returning the result type of a VHDL expression.

4 Binary Domain and Initial Variable Ordering

Let *V* be the set of all VHDL objects like variables, ports and signals referring to a VHDL design. *V*, type information and the relationship between ports, signals and local variables are represented in the global symbol table *ST* [5]. Now we can compute the number of binary variables n to $n = \sum_{o \in V} \text{size}(\text{type}(o))$. Because we want to describe state transformations we need two sets $\dot{V}$ and $\dot{V}'$ with n binary variables each. The dot should indicate that e.g. $\dot{v}$ is a binary variable and $\dot{V}$ is a set of binary variables. Let $\text{BD} = \dot{V} \cup \dot{V}'$ be the binary domain representing the binary state space.

To make symbolic model checking feasible for real applications it is important to find a *global* total order δ on the elements of *BD* so that the BDDs defining the transition system and all intermediate BDDs during model checking have a reasonable size. Dynamic variable reordering may help in some cases to keep BDD sizes manageable, but the price is often paid with an enormous increase in run time [6, 11]. We determine

the global variable order δ by performing data flow analysis and by computing dependency relations from the VHDL expressions. During the computation of the variable order we assign *priorities* to VHDL objects and to binary variables.

Definition 4 Priorities. *If a binary variable $\dot{v}_1$ has a higher (lower) priority than $\dot{v}_2$ this implies $\dot{v}_1 <_\delta \dot{v}_2$ ($\dot{v}_2 <_\delta \dot{v}_1$). If a typed VHDL variable v has a higher (lower) priority than w this implies $\forall_{i \in \{1 \dots m\}, j \in \{1 \dots n\}} \dot{v}_i <_\delta \dot{w}_j$ (resp. $\forall_{i \in \{1 \dots m\}, j \in \{1 \dots n\}} \dot{w}_j <_\delta \dot{v}_i$ for lower priority) where $\dot{v}_1, \dots, \dot{v}_m$ are the variables used for the binary representation of the variable v and $\dot{w}_1, \dots, \dot{w}_n$ are the variables used for the binary representation of the variable w .*

We give four examples of used heuristics:

1. We extract variables which are read only (VHDL input ports, constants, generics) (resp. write only) and give them a high priority; write only variables are assigned a low priority.
2. We are extracting array access e.g. $A(i)$ so that i obtains a higher priority than A .
3. If we perform arithmetic on integer types, we try to interleave the BDD variables used by the binary representations.
4. If $\dot{v}'$ is the binary variable representing the new value corresponding to $\dot{v}$ then $(\dot{v} <_\delta \dot{v}')$ should be true.

We use heuristics 1-3 to assign priorities to VHDL objects and compute a partial order δ_{VHDL} on these objects. Then we transfer each object into its binary representation with respect to heuristic 4 defined on the binary variable level. The result is a structure referred to as *BDD-Domain*, which maps two sets of totally ordered binary variables to each VHDL object.

5 Symbolic Representation of VHDL Data Types

To define a translation scheme we have to define a structure which represents for an arbitrary VHDL data object symbolically the set of all possible values and during computation all intermediate results, respectively. Based on this structure (referred to as BDD-Object) we have to define algorithms handling the operations defined on VHDL data types.

Let $\dot{V}$ be a set of binary variables. Then $BDD(\dot{V})$ represents the set of BDDs built over variables on $\dot{V}$. We denote the semantics of a BDD bdd by $\xi_{BDD} \llbracket bdd \rrbracket$ and its value under a given valuation ρ of its binary variables by $\xi_{BDD} \llbracket bdd \rrbracket_\rho \in \{0, 1\}$.

Definition 5 BDD-Object. *A BDD-Object o over a set of binary variables V is a structure $\mathbf{o} = (\mathbf{type}, \mathbf{size}, \mathbf{value}, \mathbf{condition})$ where*

- **type** $\in \mathbf{T}$ codes the VHDL type information,
- **size** $\in \mathbf{N}$ gives the number of bits used to code an object of type **type**,
- **value** $\in BDD(V)^{size}$ is a vector of BDDs where **value** $[j]$ codes the value of the j -th bit of the represented VHDL expression,
- **condition** $\in BDD(V)$ encodes the allowed representation for an object of type **type**, i.e. for any valuation ρ of V $\xi_{BDD} \llbracket \mathbf{condition} \rrbracket_\rho = 1$ iff the vector $(\xi_{BDD} \llbracket \mathbf{value}[1] \rrbracket_\rho, \dots, \xi_{BDD} \llbracket \mathbf{value}[\mathbf{size}] \rrbracket_\rho)$ represents a member of the domain $\mathbf{D}_{\mathbf{type}}$ of type **type**.

To explain the use of the BDD *condition* consider the type $\tau = \mathbf{sr}(10, 6)$. For a binary representation of a variable v with type τ we need three variables ($size(\tau) = 3$), but with three binary variables we represent a range with 8 elements. Note that e.g. $type2bin_\tau(10) = (0, 0, 0)$. The BDD *condition* for variable v restricts the possible values of v to the specified range $10 \leq v \leq 15$ (domain $\mathbf{D}_\tau$ of type τ) and codes that $(0, 1, 1)$ and $(1, 1, 1)$ are no valid values for the binary representation of the variable v .

Let $BO(V)$ denote the set of all BDD-Objects. In the sequel we will define a mapping β which assigns to every VHDL expression e a BDD-Object $\beta(e)$. The semantics of this encoding $\beta(e)$ is expressed by the following lemmata: the encoding β will be given in such a way that it is independent of the concrete syntactical description e.g. the expressions $a + a$ and $2 * a$ will have the same encoding.

Lemma 1 *Let e and e' be two arbitrary VHDL expressions: $\xi \llbracket e \rrbracket = \xi \llbracket e' \rrbracket \implies \beta(e) = \beta(e')$*

To explain the semantics of the BDDs coding the value of a BDD-Object let e be an arbitrary VHDL expression which depends on the VHDL variables $x_1, \dots, x_n$. $\beta(e(x_1, \dots, x_n)) = (t, s, v, c)$ is the BDD-Object representation of e and v is a vector of BDDs ($v = (bdd_1, \dots, bdd_s)$). For simplification we write $\beta(e)[j]$ to select the j -th BDD bdd_j of the BDD vector.

Each VHDL variable x_j is associated a set of binary variables $\{\dot{v}_{m_{j-1}+1}, \dots, \dot{v}_{m_j}\}$. For any given valuation ρ_{VHDL} of the VHDL variables $x_1, \dots, x_n$ exists an equivalent valuation $\sigma(x_j)$ of the binary variables which assigns to $(\dot{v}_{m_{j-1}+1}, \dots, \dot{v}_{m_j})$ the value $type2bin_{type(x_j)}(\rho_{VHDL}(x_j))$.

The relation between the semantics of the BDDs from the BDD vector v in the BDD-Object representation and the binary representation of the values of a VHDL expression $e(x_1, \dots, x_n)$ coincides according to the following lemma:

Lemma 2 $\forall_{1 \leq j \leq s} (\xi_{BDD} [\beta(e)[j]]_{\rho[\sigma(x_1), \dots, \sigma(x_n)]}$
 $= type2bin_{type(e)}(\xi[e(x_1, \dots, x_n)]_{\rho_{VHDL}})[j])$

β can be defined inductively over the structure of the VHDL operators used to build the expressions. To this end we have to define for each VHDL operation op_v a corresponding operation op_β on BDD-Objects and we have to define how β assigns a BDD-Object to a VHDL variable or constant.

In the next subsections we describe the bottom-up translation of a VHDL expression into a BDD-Object.

5.1 Symbolic Coding of Variables and Constants

The function $type2bdd$ is used to assign a vector of BDDs to a constant element of a type τ .

Definition 6 $type2bdd$. *The function*

$type2bdd : \mathbf{T} \rightarrow \bigcup_{\tau \in \mathbf{T}} (\mathbb{D}_\tau \rightarrow BDD(\dot{V})^{size(\tau)})$ *assigns to*

each data domain $\mathbb{D}_\tau$ of a VHDL type $\tau \in \mathbf{T}$ a vector representation of BDDs constant true resp. constant false. Instead of $type2bdd(\tau)$ we write $type2bdd_\tau$.

Let $\tau \in \mathbf{T}$, $x \in \mathbb{D}_\tau$ and $0 < j \leq size(t)$. Then

$$type2bdd_\tau(x)[j] = \begin{cases} 1_{BDD} & \text{if } (type2bin_\tau(x)[j] = 1), \\ 0_{BDD} & \text{otherwise} \end{cases}$$

We can determine a BDD-Object representation for a constant element c with $type(c) \in \mathbf{T}$ to

$$\beta(c) = (type(c), size(c), type2bdd_{type(c)}(c), 1_{BDD}).$$

To determine the BDD-Object representation for a variable v of type $\tau \in \mathbf{T}$ we assign a set of $n = size(\tau)$ binary variables to the variable v . Let $\dot{v}_1, \dots, \dot{v}_n$ be the set of binary variables representing the variable v . Then the BDD-Object representation for v is

$$\beta(v) = (\tau, n, (BDD(\{\dot{v}_1\}), \dots, BDD(\{\dot{v}_n\})), c).$$

$BDD(\{\dot{v}_i\})$ is a BDD with only one nonterminal node referencing the variable $\dot{v}_i$ and representing the function $f_i(\dot{v}_i) = \dot{v}_i$. In this case the BDD c restricts the values represented by the BDD vector $(BDD(\{\dot{v}_1\}), \dots, BDD(\{\dot{v}_n\}))$ to all members of $\mathbb{D}_\tau$.

$$c = \bigvee_{x \in \mathbb{D}_\tau} \left(\bigwedge_{0 < j \leq n} type2bdd_\tau(x)[j] =_{BDD} BDD(\{\dot{v}_j\}) \right)$$

$=_{BDD}$ denotes the equivalence operator on BDDs.

Example 2 *Let o be a VHDL object of type $colour = enum(red, green, blue)$.*

- $size(colour) = 2$,
- $type2bdd_{colour}(green) = (1_{BDD}, 0_{BDD})$,
- $\beta(\xi[green]) = (colour, 2, (1_{BDD}, 0_{BDD}), 1_{BDD})$,

- $\beta(\xi[o]) = (colour, 2, (bdd_1, bdd_2), c)$ where bdd_1 is a BDD referencing only the first binary variable which is assigned in the BDD-Domain to o . bdd_2 is a BDD referencing the second variable assigned in the BDD-Domain. c is a BDD which restricts the value of o to the values red, green or blue.

5.2 Symbolic Operations on VHDL Data Types

Each operation on VHDL data types has to be defined on BDD-Objects. Before we can execute some operations on BDD-Objects we have to adjust the types if both arguments of an operation must have the same type (strong type concept). Two different representations of type sr can be adjusted through shifting the representations to the same *offset* and using the same *size* for the BDD-Object representation. We leave out the definition of functions which shorten or prolong BDD vectors removing resp. adding leading BDDs constant 0_{BDD} if necessary. Cf. section 2 for definition of function ADD .

Definition 7 $Adjust$. *The function $Adjust_\beta : (BO(\dot{V}) \times BO(\dot{V})) \rightarrow (BO(\dot{V}) \times BO(\dot{V}))$ is defined as follows: $Adjust((t_1, s_1, v_1, c_1), (t_2, s_2, v_2, c_2)) :=$*

$$\begin{cases} ((t_r, s_r, v_{r1}, c_1), (t_r, s_r, v_{r2}, c_2)) & \text{if } (t_1 = sr(o_1, r_1)) \\ & \wedge (t_2 = sr(o_2, r_2)), \\ ((t_1, s_1, v_1, c_1), (t_2, s_2, v_2, c_2)) & \text{otherwise} \end{cases}$$

where

- $o_{min} := \min(o_1, o_2)$,
- $r_{new} := \max(o_1 + r_1, o_2 + r_2) - o_{min}$,
- $t_r := sr(o_{min}, r_{new})$, • $s_r := size(t_r)$,
- $v_{r1} := \begin{cases} v_1 & \text{if } (o_1 = o_{min}), \\ ADD(v_1, type2bdd_{sr(0, o_1 - o_2)}(o_1 - o_2)) & \text{otherwise} \end{cases}$
- $v_{r2} := \begin{cases} v_2 & \text{if } (o_2 = o_{min}), \\ ADD(v_2, type2bdd_{sr(0, o_2 - o_1)}(o_2 - o_1)) & \text{otherwise} \end{cases}$

In the following we give the formal definition of the equal operator $'=_{\beta}'$ and the plus operator $'+_{{\beta}}'$ used in section 2.

Definition 8 $Equal$ Operator $'=_{\beta}'$. *The VHDL operator $equal =_{\beta} : BO(\dot{V}) \times BO(\dot{V}) \rightarrow BO(\dot{V})$ is defined as follows:*

$$((t_1, s_1, v_1, c_1) =_{\beta} (t_2, s_2, v_2, c_2)) := (t_r, s_r, v_r, c_r)$$

where $((t_3, s_3, v_3, c_3), (t_4, s_4, v_4, c_4)) = Adjust((t_1, s_1, v_1, c_1), (t_2, s_2, v_2, c_2))$ in

- $t_r := Boolean$, • $s_r := 1$,
- $v_r := (\bigwedge_{1 \leq j \leq \min(s_3, s_4)} (v_3[j] =_{BDD} v_4[j]))$,
- $c_r := \begin{cases} c_3 \wedge c_4 & \text{if } (t_3 = t_4) \\ 0_{BDD} & \text{otherwise,} \end{cases}$

Definition 9 Plus Operator '+_{BDD}'. The VHDL operator plus $+_{\beta} : BO(\dot{V}) \times BO(\dot{V}) \rightarrow BO(\dot{V})$ is defined as follows:

$((t_1, s_1, v_1, c_1) =_{\beta} (t_2, s_2, v_2, c_2)) := (t_r, s_r, v_r, c_r)$
 where $((t_3, s_3, v_3, c_3), (t_4, s_4, v_4, c_4)) =$
 $Adjust((t_1, s_1, v_1, c_1), (t_2, s_2, v_2, c_2))$ in

- $v_r := \begin{cases} ADD(v_3, v_4) & \text{if } (t_3 = t_4 = sr(o, r)), \\ \text{undefined} & \text{otherwise,} \end{cases}$
- $s_r := \begin{cases} s_3 + 1 & \text{if } ((t_3 = t_4 = sr(o, r)) \text{ and } \\ & \text{not}(v_r[s_3 + 1] =_{BDD} 0_{BDD})), \\ s_3 & \text{otherwise,} \end{cases}$
- $t_r := \begin{cases} sr(o, 2^{s_r}) & \text{if } (t_3 = t_4 = sr(o, r)), \\ \text{undefined} & \text{otherwise,} \end{cases}$
- $c_r := \begin{cases} c_3 \wedge c_4 & \text{if } (t_3 = t_4 = sr(o, r)), \\ 0_{BDD} & \text{otherwise,} \end{cases}$

Finally we sketch the selection of an array element. We have to distinguish between the selection of a component through a constant address (*static selection*, e.g. A[5]) and through a variable address (*dynamic selection*, e.g. A[i + j]). In the last the address can be an arbitrary VHDL expression depending on a set of free VHDL variables. To this end we have to map the dynamic selection to static selection using nested if-then-else structures and to translate them into BDDs. For the static array selection we compute the type of the element out of the array type. Then we map the address into the index type of the array and extract the resulting BDD vector for the BDD-Object representing the array element.

The semantics of an operation op_{β} on BDD-Objects has to be equivalent to the semantics of the operation op_v on VHDL objects. Consider two arbitrary VHDL expressions e_1, e_2 and an arbitrary VHDL operation op_v . Now an operation op_{β} defined as equivalent operation to op_v has to comply with the following lemma:

Lemma 3 $\forall_{\rho} \forall_{1 \leq j \leq size(type(e_{1op_v e_2}))}$

$$\xi_{BDD}[\beta(e_1 op_v e_2)[j]]_{\rho} = \xi_{BDD}[(\beta(e_1) op_{\beta} \beta(e_2))[j]]_{\rho}$$

6 Conclusion and Outlook

We have presented a method to model VHDL data types using BDDs. This technique is applied to the complete set of finite VHDL data types and provides a formal foundation to prove the correspondence to the VHDL semantics. We are successfully using the current implementation in the FORMAT project within the translation of behavioural VHDL into symbolic transition systems and during symbolic model checking to translate propositions specified as VHDL expressions into BDDs. In a case study VHDL expressions modeling internal waveform updates consist of

more than 5000 characters. Due to the size of the waveform arrays, the size of the computed BDDs differ from a few to more than 100000 nodes.

The use of BDDs within automatic verification tools is only one step to a powerful verification environment. The presented method provides a lot of optimisations. We are currently implementing a model checker based on adaptive state and data abstraction [4]. To this end we extend the presented technique to depth k-abstraction [4]. Depending on the level of data abstraction we restrict the size of the BDD vector used in BDD-Objects to k elements and adapt each operation on BDD-Objects to the restricted size and the handling of overflow information. Another aspect is to focus on the computation of good BDD variable orderings and to adapt the types used within a VHDL specification to types with the smallest possible size. It seems that it is easy to take care of the size of the used types so that together with our optimisations automatic verification can manage real designs.

References

- [1] K.S. Brace, R.L. Rudell and R.E. Bryant. Efficient implementation of a BDD package. In *Proceedings 27th DAC*, Orlando, Florida, 1990
- [2] R.E. Bryant. Graph Based Algorithms for Boolean Function manipulation. In *Transaction on Computers*, Vol C-35, August 1986.
- [3] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and Jim Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *proceedings of the Fifth Annual IEEE Symposium on Logic in Computer science*, June 1990.
- [4] D. Dams, G. Döhmen, R. Gerth, P. Kelb, R. Herrmann and H. Pargmann. Design of a VHDL/S model checker based on adaptive state and data abstraction. In *Proceedings CAV*, Stanford, CA, 1994.
- [5] G. Döhmen. Petri Nets as Intermediate Representation between VHDL and Symbolic Transition Systems. In *Proceedings EURO-DAC with EURO-VHDL*, 1994.
- [6] E. Felt, G. York, R. Brayton and A. Sangiovanni-Vincentelli. Dynamic Variable Reordering for BDD Minimization. In *Proceedings of EURO-DAC'93*, 1993.
- [7] R. Herrmann. Translation of Hardware oriented Data Types into BDDs. *Master Thesis*, University of Oldenburg, Germany, 1992.
- [8] IEEE standard 1076-1987, VHDL LRM, 1987.
- [9] K. L. McMillan. Symbolic Model Checking. *Kluwer Academic Publishers*, Norwell Massachusetts, 1993.
- [10] H. Pargmann and R. Herrmann. Efficient Symbolic Transition Systems for VHDL. Technical Report, *OFFIS*, 1994.
- [11] R. Rudell. Dynamic Variable Ordering for ordered Binary Decision Diagrams. In *Proc. IEEE/ACM International Conference on CAD*, pages 42-47, 1993