ACL2VHDL Translator: A Simple Approach to Fill the Semantic Gap

Jun Sawada

IBM Austin Research Laboratory
sawada@us.ibm.com

Abstract. We wrote an ACL2 to VHDL translator for our verification purpose. One major problem of translation between programming languages and the ACL2 language is the semantic gap caused by the translation; it is not easy to translate one language to another while precisely preserving its semantics. Our approach is to write a translator for a small subset of the ACL2 language for which there is no loss of semantic correctness. This seemingly restricted translator turned out to be a promising approach for combining ACL2 and VHDL testing/verification tools. This paper discusses the details of the translator and the bit-vector libraries used in the translatable ACL2 functions.

1 Introduction

The ACL2 theorem prover has been used for formally verifying many pieces of hardware and software. Typically, verification is performed on an abstract model which is manually written in the ACL2 language. This allows interesting analysis on various aspects of computation, including the hardware verification of pipelined machine controls[8], floating point algorithms[6], and cache coherence protocols. However, there is always a doubt whether a manually-written abstract model is a correct representation of an actual implementation. It is desirable to analyze actual programming and hardware description languages, rather than its manual translation.

However, formally verifying actual programs or RTL description of hardware using a theorem prover is not very popular. This is partially because real-world software and hardware are far more complicated than what can be easily handled by a theorem prover. But there is also a language barrier that prevents or at least discourages researchers and engineers from analyzing, for instance, C and VHDL using a theorem prover.

Typically, when ACL2 is used to verify software rewritten in some programming language, it must be translated to or interpreted in the ACL2 language. There are two types of approaches: deep embedding and shallow embedding. In a deep embedding approach, one writes an interpreter of the target programming language in ACL2, and proves theorems about the results of interpretation. In a shallow embedding approach, one writes a translator from the programming language to the ACL2 language, and proves properties about translated programs in the form of ACL2 functions.

There are pros and cons in both approaches. In a deep embedding approach, it is often easy to define an interpreter with a precise semantic correctness. Data in the target programing language are typically represented by special data-types in ACL2. Liu and Moore[5] use this approach to precisely model the semantics of Java bytecode. On the downside of the deep embedding, the interpreter is slow when executed. Proof on the interpreter is sometimes cumbersome, as one has to reason in two steps about the interpreter that handles the target language.

The shallow embedding is opposite. Since we do not have to deal with an interpreter acting like a middleman, it is often straightforward to deal with a translated program. However, translation is often imprecise. One of the reasons is that data-types in programming languages are not isomorphic to ACL2 data-types. For example, `nil` in ACL2 means both Boolean false and empty list. Another example is that ACL2 integers are big numbers without limit, while most programming languages use 32-bit or 64-bit integers. There are also some concepts that are difficult to translate to ACL2, such as pointers in programming languages.

Both approaches have been used in hardware verification as well. In ACL2-based verification projects, Hunt's DUAL-EVAL[4] is an example of deep embedding, while Russinoff's floating-point verification uses shallow embedding[7]. When translating hardware description language into the ACL2 language, there are additional problems deriving from the differences of computation models. For example, the concurrent signal assignment statement in VHDL is a collection of assignment statements which are not ordered. When evaluating, concurrent assignments must be repetitively applied until no more values are updated. In a sense, it is performing a fix-point calculation. Thus, translation of concurrent assignments in VHDL to ACL2 is not straightforward. Even if the translation can done, the resulting program may become too complex to analyze by a theorem prover.

Our goal in this paper is to define a translation that allows the analysis of VHDL code in ACL2. Our approach is rather simple but effective. First, we define a translator from ACL2 to VHDL, not the other way around. We rely on the VHDL-level verification tools in comparing translated ACL2 functions and VHDL implementations of hardware. We can also use ACL2 to prove theorems about the ACL2 functions before the translation. In this way, we never have to work on complicated VHDL source code at the ACL2 level. When the VHDL implementation of hardware is changed, as it happens so often, the VHDL-level verification tools have to simply be rerun. We use ACL2 in the analysis of the high-level concepts such as algorithms that do not change very often.

Second, the translator can handle a small subset of ACL2 language. We define an ACL2 book about bits, bit-vectors, and 32-bit integers. Our translator accepts ACL2 functions that use only the functions from this ACL2 book. This way, we can translate ACL2 functions to VHDL without loss of semantic correctness.

In this paper, we first introduce the ACL2 bit-vector library used in the translated functions, then go on to describe the details of the translation pro-

gram. This paper assumes the minimum knowledge of VHDL. For a tutorial of VHDL, we recommend Bhasker's book[1].

2 Bit-Vector Library

In the standard ACL2 distribution, there is a library called the IHS (Integer Hardware Specification) library. It has been developed for specifying hardware in the verification project on a DSP processor[2]. In the IHS library, bits are represented by ACL2 integer 0 and 1. Bit-vectors are represented by integers. This simple representation allows quick execution of the model specified in the IHS library. Also the IHS library defines many operations over bits and bit-vectors, and provides related lemmas.

However, the IHS library is not adequate as a basis of ACL2-to-VHDL translation. First, the integer representation of bits and bit-vectors prevents us from defining a one-to-one mapping between the ACL2 and VHDL data-types. When you say 1, it could mean integer 1, bit 1, or a bit-vector whose value happens to be 1. And even if it is known to be a bit-vector of value 1, it could be a bit-vector of length 1, a bit-vector of length 10, or a bit-vector of any length.

Also, this ambiguity in the length of bit-vectors disallows the definition of certain functions. For example, let us consider a function `msb` that returns the most significant bit of a bit-vector. Should we define `(msb 1)` to be 0 or 1? It depends on the length of the bit-vector represented by 1.

We defined a new bit-vector library called *BV library*. In this library a bit is represented as list `(BIT 0)` or `(BIT 1)`. A bit-vector is represented by a list `(BV` v n`)`, where positive integer v is the value of the bit-vector, and n is its length. In other words, the corresponding bit-vector is the least significant n bits of the binary representations of v.

By building it on top of the IHS library, we simplified the definition of the BV library. For example, let us look at the definition of function `bv&`, which returns a concatenation of two bit-vectors:

```
(defun bv& (a b)
  (bv (logapp (bv-size b) (bv-val b) (bv-val a))
      (+ (bv-size a) (bv-size b))))
```

Function `bv` is a constructor for bit-vectors; `(bv` *val size*`)` returns a list `(BV` *val size*`)`. Function `bv-val` and `bv-size` returns the value and the length of a bit-vector, respectively. Function `(logapp` n $bv1$ $bv2$`)` is an IHS function that concatenates bit-vectors $bv2$ and $bv1$ assuming that the length of $bv1$ is n. In this definition, `bv&` calculates the value of the concatenated vectors by calling the IHS function. The length of the concatenated vector is the sum of the lengths of two vectors. In this way, we reused definitions and lemmas in the IHS library as much as possible in the BV library.

Bit constant `*b0*` and `*b1*` are defined to be `(BIT 0)` and `(BIT 1)`, respectively. Type predicate `(bitp x)` is true if `x` is a bit. We define operators over bits such as `b-not`, `b-ior`, `b-and`, `b-xor`, and `b-eqv`.[1]

Following are basic functions on bit-vectors defined in the BV library. A complete list is provided in Table 1 and 2 in Section 3. Function `(lsb v)` returns the least significant bit of bit-vector `v`, and `(msb v)` returns the most significant bit. Function `(msbits v)` returns `v` after removing the least significant bit, while `(lsbits v)` returns `v` except the most significant bit. Function `(b&bv b v)` and `(bv&b v b)` add bit `b` to the bit-vector `v` as the most significant bit and the least significant bit, respectively. Hence:

```
(msb (b&bv b v)) = b
(lsbits (b&bv b v)) = v
(lsb (bv&b v b)) = b
(msbits (bv&b v b)) = v
```

In a sense, `msb` and `lsb` act like `car` for bit-vectors, `lsbits` and `msbits` like `cdr`, and `b&bv` and `bv&b` like `cons`.

Function `(bitn n v)` returns the `n`'th bit of bit-vector `v`. The most significant bit is always indexed by 0, and the index increases toward the less significant bits. In VHDL, an IEEE standard ulogic vector defined to be `std_ulogic_vector(0 to len-1)` has the same indexing. Different vendors prefer different styles of indexing, i.e. indexing from the most significant bit or indexing from the least significant bit. Additionally, VHDL allows indexing starting from any integer. To keep the BV library simple, we decided to allow only the indexing from the most significant bit to the least significant bit starting from 0.

Other operations defined over bit-vectors include sub range of bit-vector `(bits v i j)`, bit-wise negation `(bv-not v)`, bit-wise AND `(bv-and v1 v2)`, bit-wise OR `(bv-ior v1 v2)`, logical shift to left `(bv-<< v a)`, logical shift to right `(bv->> v a)`, arithmetic-shift to right `(bv-*>> v a)`, binary addition `(bv+ v1 v2)`, binary increment `(bv++ v1)`, equality operation `(bv-eq? v1 v2)`, and binary comparisons such as greater-than relation `(bv-gt? v1 v2)`. We also defined a 0-length bit-vector `*bv-nil*` and a conversion function from bit to bit-vector of length 1, `(b2bv b)`.

There are also a few functions for control. Function `(bv-if b v1 v2)` returns bit-vector `v1` if `b` is 1, and returns `v2` otherwise. We also define a macro `bv-cond` which works like the `cond` control structure in ACL2.

Using the functions described above, any functions over bit and bit-vectors can be defined. We found that the BV library can easily express most of the functions defined in hardware, such as when we described multipliers and floating point instructions.

[1] The IHS library has bit operators with the same name. However, we define our bit-vector library in a separate ACL2 package, thus they are different functions.

3 Conversion of ACL2 to VHDL

We wrote a translation program from the ACL2 language to VHDL. We call it *ACL2VHDL translator*. This translation program only handles bit and bit-vectors and 32-bit integers. Bits are mapped to VHDL standard ulogic, bit-vectors are mapped to standard ulogic vectors, and 32-bit integers are mapped to VHDL integers. Translatable ACL2 functions are those defined only in terms of `let`, `let*` and the functions in the BV library. Any other functions are not allowed in the translated function, including basic ACL2 functions `if` and `cons`. This limitation simplifies the implementation of the translation program and also helps to preserve the semantics during the function translation.

Some of the bit-vector functions in the BV library such as `(bv&` $v0$ $v1$`)` and `(bv-and` $v0$ $v1$`)` have corresponding VHDL operators, namely `(`$v0$ `&` $v1$`)` and `(`$v0$ `AND` $v1$`)`. We defined additional VHDL functions so that every bit and bit-vector functions from the BV library has a corresponding VHDL function. Such additional functions are defined in a VHDL file named acl2_support.vhdl. Some of them are `msb(v)`, `lsb(v)`, `lsbits(v)`, `msbits(v)`, `bitn(n,v)`, `bits(v,i,j)`, and `bv_gt(v0,v1)`. The names of the BV functions and the corresponding VHDL functions are shown in Table 1 and 2. Due to the restriction of the characters that can be used in VHDL function names, the function names in the BV library and acl2_support.vhdl may not be identical, but their semantics should be.

Integers are also problems in translation. In ACL2, integers are big numbers without limit, but VHDL's integer is often implemented as 32-bit integers. In order to fill the gap, we define the 32-bit integer type `(int32p i)`, and 32-bit integer operations such as `(int32+ i j)`, `(int32- i j)` and `(int32* i j)`. We allow only 32-bit integer and the associated operations in the translatable ACL2 functions, but not ordinary ACL2 arithmetic operations such as `+`, `-` and `*`.

The ACL2VHDL translator converts functions using 32-bit integers. For example

```
(defun plus (a b)
  (declare (xargs :guard (and (int32p a) (int32p b))))
  (int32+ a b))
```

is translated into

```
function plus(a : integer;b : integer)
  return integer is
  variable result : integer;
begin
  result := (a + b);
  return result;
end plus;
```

Since every BV function has a corresponding VHDL function, we can directly translate an ACL2 function whose definition only includes BV functions. However, in addition to the BV functions, we allow `let` and `let*` statements, which

Table 1. BV library functions that can be used in the translated functions and the corresponding VHDL functions. VHDL functions are defined in the standard IEEE 1164 library or defined in a new VHDL package for the conversion purpose. This table shows bit logical operations, bit-vector logical operations, and bit manipulation functions. Function `bv-cond` has no corresponding function, and it is translated to an expression using `ite`

ACL2 Function	VHDL Function	Descriptions
`*b0*`	`'0'`	Bit one (constant).
`*b1*`	`'1'`	Bit zero(constant).
`b-not`	`not`	Bit negation.
`b-ior`	`or`	Bit OR.
`b-nor`	`nor`	Bit NOR.
`b-and`	`and`	Bit AND.
`b-nand`	`nand`	Bit NAND.
`b-xor`	`xor`	Bit exclusive OR.
`b-andc1`	`andc1`	Bit AND with the first argument complemented.
`b-andc2`	`andc2`	Bit AND with the second argument complemented.
`b-orc1`	`orc1`	Bit OR with the first argument complemented.
`b-orc2`	`orc2`	Bit OR with the first argument complemented.
`b-maj`	`maj`	Majority function.
`b-if`	`ite`	IF-statement returning a bit.
`*bv-nil*`	`""`	Bit-vector of length zero(constant).
`bv-not`	`not`	Bit-wise negation.
`bv-and`	`and`	Bit-wise AND.
`bv-ior`	`or`	Bit-wise OR.
`bv-xor`	`xor`	Bit-wise XOR.
`bv-neg`	`bv_neg`	2's complement.
`bv-if`	`ite`	IF-statement returning a bit-vector.
`bv-cond`		Cond-statement returning a bit-vector.
`b2bv`	`b2bv`	Conversion from bit to bit-vector of length 1.
`bitn`	`bitn`	N'th bit from the most significant bit.
`msb`	`msb`	The most significant bit.
`lsb`	`lsb`	The least significant bit.
`msbits`	`msbits`	Removing the least significant bit.
`lsbits`	`lsbits`	Removing the most significant bit.
`bv&b`	`&`	Concatenating a bit after a bit-vector.
`b&bv`	`&`	Concatenating a bit before a bit-vector.
`bv&`	`&`	Concatenating two bit-vectors.
`bits`	`bits`	Subrange of a bit-vector.
`bv-right`	`bv_right`	The rightmost n bits of a bit-vector.
`bv-left`	`bv_left`	The leftmost n bits of a bit-vector.
`zeros`	`pad0`	A bit-vector with every bit set to 0.
`ones`	`pad1`	A bit-vector with every bit set to 1.
`pad0`	`pad0`	Identical to `zeros`.
`pad1`	`pad1`	Identical to `ones`.
`fill-0`	`fill_0`	Bit zeros filling a subrange.
`fill-1`	`fill_1`	Bit ones filling a subrange.

Table 2. A continued list of BV library functions and the corresponding VHDL functions. Here we define shifting operations, arithmetic operations, arithmetic relations, and logical operations over bits in a vector.

ACL2 Function	VHDL Function	Descriptions
bv-uextend	bv_uexnted	Unsigned extension of a bit-vector.
bv-sextend	bv_sexnted	Signed extension of a bit-vector.
bv-lsh	sll	Shift left logical with an integer shift amount.
bv-ash	sla	Shift left arithmetic with an integer shift amount.
BV-<<	bv_sll	Shift left with a bit-vector shift amount.
BV->>	bv_srl	Shift right logical with a bit-vector shift amount.
BV-*>>	bv_sra	Shift right arithmetic with a bit-vector shift amount.
bv+	+	Binary addition.
bv+carry	bv_carry	Carry out from a binary addition.
bv-	-	Binary subtraction.
bv*	*	Binary multiplication.
bv++	increment	Binary increment.
bv--	decrement	Binary decrement.
bv-inc	add_bit	Binary increment if a bit input is set.
bv-eq?	bv_eq	bit-vector equality.
bv-neq?	bv_neq	bit-vector inequality
bv-gt?	bv_gt	Binary greater than.
bv-ge?	bv_ge	Binary greater than or equal to.
bv-lt?	bv_lt	Binary less than.
bv-le?	bv_le	Binary less than or equal to.
bv-all-ones?	and_reduce	AND'ing all bits in a bit-vector.
bv-all-zeros?	nor_reduce	NOR'ing all bits in a bit-vector.
bv-some-zeros?	nand_reduce	NAND'ing all bits in a bit-vector.
bv-some-ones?	or_reduce	OR'ing all bits in a bit-vector.
bv-and-all	and_reduce	Identical to bv-all-ones?.
bv-ior-all	or_reduce	Identical to bv-all-zeros?.
bv-right-ior	bv_right_or	OR'ing the rightmost n bits of a bit-vector.
bv-left-ior	bv_left_or	OR'ing the leftmost n bits of a bit-vector.
bv-lz	leadz	Number of zero bits in a bit-vector.

requires a special treatment. Basically, variable bindings in a `let` expression are translated into VHDL sequential assignments.

When the ACL2VHDL translator first parses an ACL2 function definition, it extracts a list of bindings introduced by `let`. In order to translate the ACL2 `let` semantics correctly, this extraction algorithm renames a variable every time it is rebound by a `let` expression. For example, let us look at the translation of function `foo`:

```
(defun foo (a)
  (declare (xargs :guard (and (bvp a) (equal (bv-size a) 1))))
  (let ((x (b&bv *b1* a)))
    (let ((x (b&bv *b0* x)))
      (let ((x (b&bv *b1* x)))
        (msb x)))))
```

The ACL2VHDL translator converts it into a VHDL function:

```
  function foo(a : std_ulogic_vector)
    return std_ulogic is
    variable x : std_ulogic_vector(0 to 1);
    variable x_1 : std_ulogic_vector(0 to 2);
    variable x_2 : std_ulogic_vector(0 to 3);
    variable result : std_ulogic;
  begin
    x := (b1 & a);
    x_1 := (b0 & x);
    x_2 := (b1 & x_1);
    result := msb(x_2);
    return result;
  end foo;
```

In this translation,[2] bit `*b0*` and `*b1*` are translated to VHDL constant `b0` and `b1`. ACL2 function `b&bv` and `msb` are translated into corresponding VHDL functions `&` and `msb`, respectively. The scope analysis introduced new variables `x_1` and `x_2` to the VHDL code to store the value of `x` bound in the second and third `let` expressions. (Note we convert each ACL2 function to an individual VHDL function, and we do not have to worry about name collisions with existing VHDL variables or signals.) The scope analysis was performed to follow the LISP scope rules, thus we allow `let` expressions inside the binding of another `let` expression. For example,

```
(defun bar (a)
  (declare (xargs :guard (and (bvp a) (equal (bv-size a) 1))))
  (let ((x (let ((x (b&bv *b0* a))) (b&bv *b1* x))))
    (bv& x x)))
```

[2] For readability of the paper, the output from the ACL2VHDL translator is indented properly, although the raw output is not pretty-printed.

is translated into:

```
function bar(a : std_ulogic_vector)
  return std_ulogic_vector is
  variable x_4 : std_ulogic_vector(0 to 1);
  variable x : std_ulogic_vector(0 to 2);
  variable result : std_ulogic_vector(0 to 5);
begin
  x_4 := (b0 & a);
  x := (b1 & x_4);
  result := (x & x);
  return result;
end bar;
```

In this translation, `x_4` is used to store the value of `x` in the inside `let` expression, while `x` represents the `x` of the outside `let`.

The ACL2VHDL translator also performs type inference on the ACL2 functions. Note in the examples above, function `foo` and `bar` return a bit and a bit-vector of length 6, respectively. In VHDL, such type information of results and intermediate values are needed to define variables. The ACL2VHDL translator infers such required type information from the type of the input arguments. The user must provide a guard expression to each function in order to annotate the type of all the input arguments.

The translated ACL2 function can take only bits, bit-vectors, and 32-bit integers as arguments, and the types of all input variables must be determined from the guard expression. The guard expression must be a conjunction of form `(and` $cond_1$ $cond_2$ ... $cond_n$`)`. For a bit type argument `b`, `(bitp b)` should appear as a conjunct. 32-bit integer type argument `i` should have a conjunct `(int32p i)` in the guard expression. bit-vector argument `v` of length n has to have conjunct expressions `(bvp v)` and `(equal (bv-size v)` n`)` in the guard expression. The length of n can be either a verbatim positive number or an expression of type `int32p`, which might use 32-bit integer type arguments. For example,

```
(defun test1 (x m n)
  (declare (xargs :guard (and (int32p m)
                              (int32p n)
                              (bvp x)
                              (equal (bv-size x) (int32+ n m)))))
  (b&bv *b1* x))
```

is translated into:

```
function test1(x : std_ulogic_vector;m : integer;n : integer)
  return std_ulogic_vector is
  variable result : std_ulogic_vector(0 to (((n + m) + 1) - 1));
begin
  result := (b1 & x);
```

```
    return result;
  end test1;
```

The ACL2VHDL translator ignores any conjuncts in the guard expression that are none of the forms discussed above.

One exception to the restriction of input argument types are big integers as a first argument of `bv`. This is because we sometimes need to specify a constant bit-vector which is longer than 32 bits. The ACL2VHDL translator directly converts such a `bv` expression into a bit-vector constant in VHDL.

Type inference algorithm knows the type of basic functions defined in the BV library. It also stores the type of a newly defined function. From the type information of functions and arguments, the type inference algorithm deduces result types. Since most functions in the BV library can take bit-vectors of variable lengths, our type inference system is implemented to handle parametrized types. For example, the type of function `b&bv` is represented by a list `((bit (bv a)) (bv (+ a 1)))`, which means `b&bv` takes a bit and a bit-vector of length `a` as arguments and returns a bit-vector of length `a`+1.

After variable renaming for let expressions and type inference, the process is rather simple to generate the VHDL functions. The ACL2VHDL translator converts the body expression of an ACL2 function into a sequence of assignments. For each let binding, it creates a separate variable assignment in VHDL. The translator does not generate VHDL control constructs such as IF-THEN-ELSE statements. Even ACL2 function `bv-if` is translated into the corresponding VHDL if-then-else function `ite`. For example

```
(defun test3 (a x y)
  (declare (xargs :guard (and (bitp a)
      (bvp x) (equal (bv-size x) 32)
      (bvp y) (equal (bv-size y) 32))))
  (bv-if a (bv& x y) (bv& y x)))
```

is translated into

```
  function test3(a : std_ulogic;
                 x   : std_ulogic_vector;
                 y   : std_ulogic_vector)
    return std_ulogic_vector is
    variable result : std_ulogic_vector(0 to 63);
  begin
    result := ite(a, (x & y), (y & x));
    return result;
  end test3;
```

Since the ACL2VHDL translator does not accept `if`, translated ACL2 functions cannot be recursive. Generally speaking, not all recursive functions can be translated into VHDL. Recursion is allowed for VHDL functions only if the compiler can statically unroll a recursive function to a flat definition. In other words,

functions can recurse only on statically known values, such as the length of bit-vectors or constant integers. It is possible to translate such a limited recursive function. However, the recursive VHDL function must use VHDL IF-THEN-ELSE statements, and thus requires a significant modification of our translator which currently outputs only sequential assignment statements in the function body.

4 Discussion

We implemented a small translator from the ACL2 language to VHDL using the BV library. Because the translated language is limited, we can translate ACL2 functions to VHDL functions while minimizing the semantic gap between two completely different languages.

Although we can use only the BV library functions in the definition of translated functions, we found our BV library expressively enough to describe many algorithms used in hardware. We specified multipliers and the behavior of floating-point instructions using this library. We found that the BV library supplies frequently used functions for such work. The ACL2VHDL translator also allows a kind of parametrized function by permitting variable length bit-vectors as arguments. This enables us to convert generalized functions. For example, we can specify a single function that rounds a floating-point mantissa to a bit vector of any length, such as single-precision and double-precision lengths.

It is true that the ACL2VHDL translator is limited with various features, and the most notable missing feature is recursion. Although recursive functions are not frequently used in hardware descriptions, it is useful in describing algorithms.

However, recursive functions are disallowed only in the function to be translated. We can freely define a recursive function in ACL2 using the BV library function. We can also "translate" such a recursive function to a VHDL function by taking a two-step approach. First, we define a non-recursive version of ACL2 function and prove the equivalence between the two ACL2 functions. Then, we translate the non-recursive function into VHDL.

Although the BV library only defines the bit and bit-vectors, it does not provide any mechanisms to define data-structures like records and lists. We did not include them in the translation because we wanted to keep our translator simple, and also because there are no corresponding data-structures in VHDL. One exception is an array. Arrays are used to define memory and register files, and it is desirable to implement it in the future version of the ACL2VHDL translator.

Also our translator does not handle features used to describe hardware in VHDL, such as the concept of entity and ports, signal delays, and state holding elements such as latches. These features are excluded from our translator to keep it simple. Our approach is remarkably different from some projects that tries to model many details of VHDL, such as the work by Georgelin et. al. [3]. Their translator converts VHDL concepts of entity and process into the ACL2, assuming synchronous behavior and transformation on the concurrent

assignments. We do not handle any of those, instead, we rely on VHDL-level verification tools to handle them.

One might ask, then, whether such a simple translator is actually useful in hardware verification. Our experience shows that it is, especially when the translator is combined with VHDL-level verification tools. In one approach, one can compare the translated ACL2 function and actual VHDL implementation by simulation or formal verification tools. For example, we wrote the specification of floating-point conversion instructions, translated it into VHDL, and compared it against the actual VHDL implementation of a floating-point unit using a SAT solver. For simple instructions like a floating point conversion instruction, we can completely verify that a VHDL implementation of floating point unit is equivalent to the ACL2 specification.

Yet another application is translating properties written as ACL2 functions into VHDL, and check them using a VHDL-level formal verification tool. Once those properties are verified at the VHDL level, we can use verified properties as lemmas and construct a verification proof of a large system. For example, we may use such an approach for compositional verification of a multiplier.

Either way, the ACL2VHDL translator is useful when it is combined with VHDL-level verification/testing tools. For future work, we would like to integrate the ACL2 system and VHDL-level verification tools using the ACL2VHDL translator.

References

1. J. Bhasker. *A VHDL Primer*. Prentice Hall, 1992.
2. B. Brock and W. A. Hunt, Jr. Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP. In *1997 IEEE International Conference on Computer Design*, pages 31–36. IEEE Computer Society, Oct. 1997.
3. P. Georgelin, D. Borrione, and P. Ostier. A framework for vhdl combining theorem proving and symbolic simulation. In *ACL2 Workshop*, 2002.
4. W. A. Hunt, Jr. and B. Brock. A formal HDL and its use in the FM9001 verification. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, Prentice-Hall International Series in Computer Science, pages 35–48. Prentice-Hall, Englewood Cliffs, N.J., 1992.
5. H. Liu and J. S. Moore. Java program verification via a jvm deep embedding in acl2. In *To appear in TPHOLs2004, personal communication*, 2004.
6. J. S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the AMD5$_K$86 Floating-Point Division Program. *IEEE Trans. Comp.*, 47(9):913–926, September 1998.
7. D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-Point Multiplication, Division, and Square Root Instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998.
8. J. Sawada and W. A. Hunt, Jr. Processor verification with precise exceptions and speculative execution. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 135–146. Springer Verlag, 1998.