

Automatic Verification of Extensions of Hardware Descriptions

Hans Eveking
 Institut für Datentechnik
 Technische Hochschule Darmstadt
 D-6100 Darmstadt, Fed. Rep. of Germany

The extension of a hardware description is a description where all properties of the original one are maintained. The concept applies to a variety of design and verification problems including logic-verification and the verification of behavioral vs. structural descriptions. For a systematic discussion, several classes of temporal behavior and HDL-constructs for their representation are introduced. The verification tool LOVERT is surveyed which allows for the automatic verification of several types of extensions.

1 Extensions of Descriptions

In the following, the correctness of finite state systems is discussed in terms of an HDL-based hardware specification technique [3]. One hardware description, the *specification*, defines the meaning of correctness for another one, the *implementation* (Fig. 1).

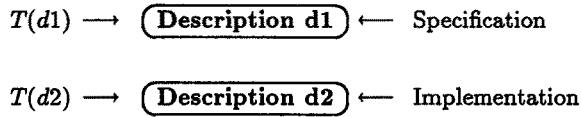


Fig. 1: Basic situation of a hardware specification technique

Hardware specification techniques are based on the concept of the *axiomatization* of HDL descriptions [4]. The axioms associated with a hardware description d have two sources (i) the *model-specific axioms* which are due to the hardware-model involved, e.g., the axioms of boolean algebra, (ii) the *description-specific axioms* reflecting the properties of the specific description d . The model- and description-specific axioms associated with a hardware description characterize a *theory*, i.e., a formal system of the predicate calculus.

Definition 1: A formula A is a correct statement about a hardware description d iff it is a theorem of the associated theory $T(d)$, i.e.,

$$\vdash_{T(d)} A. \quad (1-1)$$

The relationship between the specification and the implementation can be discussed in terms of the relationship between the associated theories. We study classes of a particularly simple relationship between two descriptions $d1$ and $d2$:

Definition 2: A description $d2$ is an *extension* of $d1$ iff for all axioms of $T(d1)$, i.e., for $A(T(d1))$ holds

$$\vdash_{T(d2)} A(T(d1)). \quad (1-2)$$

Note that on the basis of 1-2 *all* correct statements about $d1$ are correct about $d2$, too. The limitations of the concept of extension are due to the fact that the underlying modelling concepts of $d1$ and $d2$ have to be the same since the model-specific axioms of $T(d1)$ have also to be theorems of $T(d2)$ as required by 1-2. Problems involving temporal abstraction or value homomorphisms for which an interpretation of the theory $T(d1)$ is necessary [3,5] are not covered by the concept of extension.

In Section 2, the semantics of finite state systems is defined in terms of some concepts of mathematical systems theory [9]. HDL-representations of several types of temporal behavior are proposed. The HDL-constructs are taken from the CONLAN family of HDL's [8]. An axiomatization of all HDL-constructs will be given.

In Sections 3-5, several types of extensions will be introduced, and proof-procedures will be discussed.

2 Classes of Temporal Behavior

We study systems that can be characterized by means of *time-functions*. A time-function represents the values that can be observed at a *carrier*, i.e., a point of observation. We consider time-functions to be functions from the set of natural numbers representing the time into some range, e.g., the set of boolean values $\{L, H\}$. Let P be a set of n time-functions f_1, \dots, f_n . $P(t)$ denotes the

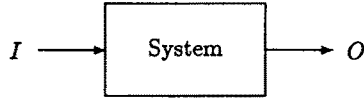


Fig. 2: A system with inputs and outputs

n -tuple $(f_1(t), \dots, f_n(t))$, i.e., the tuple of all n values at point t of time. $P(< t_1, t_2 >)$ denotes the $(t_2 - t_1 + 1)$ -tuple of all $P(t)$ in the interval $t_1 \leq t \leq t_2$.

A set P of time-functions associated with a system can be partitioned into a set I of input-functions and a set O of output-functions (Fig. 2). The classification of the temporal behavior of a system is based on the question:

Which information about $I(< 0, t >)$ and $O(< 0, t-1 >)$ determines $O(t)$ uniquely ?


In the rest of this Section, HDL-constructs for the representation of three classes of behavior will be presented.

2.1 Static behavior

A system has *static* behavior iff $O(t)$ is determined uniquely by $I(t)$ for all t . A typical example is the behavior of an AND-gate (Fig. 3).

L (low) and H (high) are the boolean constants. Static behavior is described by means of connections to carriers of type `btm1` (boolean terminal)

$$g \text{ .} = e$$



t	0	1	2	3	4	...
$a(t)$	L	L	H	L	L	...
$b(t)$	H	L	H	H	L	...
$g(t)$	L	L	H	L	L	...

Fig. 3: Static behavior of an AND-gate

with the meaning $\forall t: g(t) = e(t)$.

For the boolean functions & (and), | (or), || (exor) and ~ (not), corresponding boolean functions are defined in the predicate calculus, e.g.

$$(and(a, b) = r) \iff (a = H) \wedge (r = b) \vee (a = L) \wedge (r = L).$$

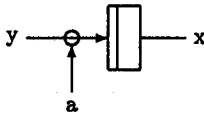
The meaning of an HDL-expression is defined by means of a time-function, too. For instance, the time-function $\lambda(t)(and(a(t), b(t)))$ is associated with the boolean expression $a \& b$ of Fig. 3. As a result, the meaning of the statement $g := a \& b$ of Fig. 3 is

$$\forall t: g(t) = (\lambda(t)(and(a(t), b(t))))(t) = and(a(t), b(t)).$$

2.2 Transitional behavior

A system has *transitional* behavior iff $O(t)$ is uniquely determined by $I(t-1)$ and $O(t-1)$ for $0 < t$. The behavior of a Moore-machine [6] is an example of transitional behavior. This type of behavior is described by conditional transfers (Fig. 4) into carriers of type budv1 (boolean unit delay variable). If the transfer condition a is H at point $t-1$ then the value of x at point t becomes

IF a THEN $x \leftarrow y$ ENDIF



t	0	1	2	3	4	5	...
$a(t)$	L	L	H	L	L	H	...
$y(t)$	H	H	H	L	L	L	...
$x(t)$	L	L	L	H	H	H	...

Fig. 4: Transitional behavior of a transfer

the value of $y(t-1)$; otherwise, the old value of x is maintained. A default value L is assumed for point 0 of time.

The semantics of a conditional transfer is thus

$$\begin{aligned} x(0) &= L, \\ \forall t: (0 < t) \wedge (a(t-1) = H) &\Rightarrow (x(t) = y(t-1)), \\ \forall t: (0 < t) \wedge (a(t-1) = L) &\Rightarrow (x(t) = x(t-1)). \end{aligned} \quad (2-1)$$

There may be several conditional transfers into one carrier. Assume n conditional transfers into the carrier x :

IF a_1 THEN $x \leftarrow y_1$ ENDIF,
 ...,
 IF a_n THEN $x \leftarrow y_n$ ENDIF

If transfer collisions are excluded then the meaning is:

$$\begin{aligned}
 & x(0) = L, \\
 & \forall t: (0 < t) \wedge (a1(t-1) = H) \Rightarrow (x(t) = y1(t-1)), \\
 & \dots \\
 & \forall t: (0 < t) \wedge (an(t-1) = H) \Rightarrow (x(t) = yn(t-1)), \\
 & \forall t: (0 < t) \wedge (a1(t-1) = L) \wedge \dots \wedge (an(t-1) = L) \Rightarrow (x(t) = x(t-1)). \quad (2-2)
 \end{aligned}$$

If a transfer condition or source expression is a boolean expression, e.g.,

IF a & b THEN x <- y ENDIF

then an anonymous time-function is associated with the boolean expression (Section 2.1). The time-functions of all carriers are thus bound to point $t-1$ of time. In the example, we obtain

$$\begin{aligned}
 & x(0) = L, \\
 & \forall t: (0 < t) \wedge (and(a(t-1), b(t-1)) = H) \Rightarrow (x(t) = y(t-1)), \\
 & \forall t: (0 < t) \wedge (and(a(t-1), b(t-1)) = L) \Rightarrow (x(t) = x(t-1)).
 \end{aligned}$$

2.3 Quasi-transitional behavior

A system has *quasi-transitional* behavior iff $O(t)$ is uniquely determined by $I(t)$ und $O(t-1)$ for $0 < t$. An example is the behavior of a latch (Fig. 5) described by a conditional assignment to a

IF a THEN r := y ENDIF

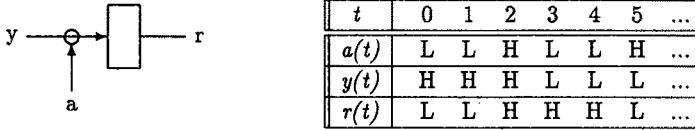


Fig. 5: Quasi-transitional behavior of a latch

carrier of type `bvar1` (boolean variable). Note that in the example of Fig. 5, the output r follows the input y directly at points 2 and 5 of time.

The meaning of the conditional assignment of Fig. 5 is defined by

$$\begin{aligned}
 & x(0) = L, \\
 & \forall t: (0 < t) \wedge (a(t) = H) \Rightarrow (x(t) = y(t)), \\
 & \forall t: (0 < t) \wedge (a(t) = L) \Rightarrow (x(t) = x(t-1)). \quad (2-3)
 \end{aligned}$$

The input/output behavior of an automaton of Mealy-Type [6] is also quasi-transitional.

3 Static Descriptions vs. Static Descriptions

The first type of extensions applies to situations where the specification as well as the implementation are given by static descriptions.

Definition 3: A description is called *static* iff all carriers have static behavior.

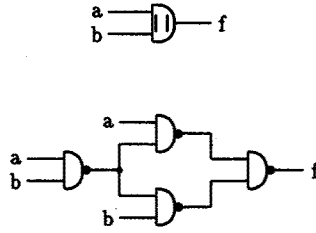


Fig. 6: An example of logic-verification

EXOR-function is implemented by means of a network of NAND-gates. To prove the correct implementation, the boolean terms of the output f are derived for the specification as well for the implementation; then the equivalence \simeq of both terms is shown:

$$a \mid b \quad \simeq \quad \sim(\sim(a \& \sim(a \& b)) \& \sim(b \& \sim(a \& b)))$$

Efficient procedures for the equivalence-proof of complex boolean expressions are available due to the work of Bryant [2] and Madre/Billon [7]. Since most HDL's provide bit-vectors as a basic type for the convenient description of complex circuits, the problem of logic-verification involves also the equivalence-proof of *vector-expressions*.

Definition 4: Two vector-expressions are equivalent, $a \cong b$, iff for all elements i holds $a[i] \simeq b[i]$.

The semantics of vectors and vector-operations can be defined by means of lists and list-operations [4]. Three examples of increasing complexity are shown in Fig. 7. The first problem is easily solved if the commutativity of the boolean and-function is extended for vector-functions. In the

$$\begin{aligned} a[1:8] \& b[1:8] & \cong & b[1:8] \& a[1:8] \\ \text{adc}(a[1:8], a[1:8], 0) & \cong & a[1:8] \# 0 \\ \text{gt}(a, b) & \cong & \sim(\text{adc}(0\#a, 1\#b, 0)[2]) \end{aligned}$$

Fig. 7: Three examples of equivalent vector-expressions

second problem, the function `adc` is used which adds two n -bit vectors and a carry-input returning a normalized $n + 1$ -bit vector $1 : n + 1$. The left-most bit is the most-significant bit. For instance,

$$\text{adc}(a[1:8], b[1:8], c)$$

is a 9-bit vector; the carry-output is `adc(a[1:8], b[1:8], c)[1]`. The second equivalence of Fig. 7 is based on the fact that a plus a is equivalent to a multiplication of a by 2, i.e., a left-shift of

a or the concatenation # of a and 0.¹ The third problem is even more difficult: in order to compare two vectors a and b, b is subtracted from a adding the complement of b to a; the most-significant bit of the sum has to be inverted.

To address such a variety of proof-complexity, the LOVERT approach [1] follows a two-step procedure:

- Step 1: two expressions are rewritten using a number of rewrite-rules. If the rewriting results in textually identical expressions then the equivalence is proven (Fig. 7). An example of a



Fig. 8: Transformation of an equivalence-proof into an identity-proof

rewrite-rule is the concatenation of the adc-function: two concatenated adders

```
adc( x, y, adc( v[1:n], w[1:n], cin)[1]) #
adc( v[1:n], w[1:n], cin)[2:n]
```

are equivalent to one adder with catenated inputs:

```
adc( x # v[1:n], y # w[1:n], cin).
```

- Step 2: if the rewrite-technique fails, the vector-expressions are compiled into the basic boolean functions and, or and not. Vector expressions are sliced into single bits. The Madre/Billon tautology checking technique [7] is then applied (Fig. 9). Since the rewrite-rules used in Step 1 are not confluent, the second step ensures the completeness of the approach.

The following table shows the cpu-time of a SPARC-station needed to solve the third problem of Fig. 7 depending on wordlength:

Wordlength	8	16	32	64	128
CPU-time	0.8 sec.	0.9 sec.	1.0 sec.	1.8 sec.	3.4 sec.

The two-step procedure has a significant advantage in a situation where a design error is detected. The behavior of a verification tool in an error situation is an important aspect for its acceptance by a designer. In the example of Fig. 10, a 16-bit adder is implemented by means of four 4-bit adders; however, the carry chain is broken at the carry input of the instance a2 since the carry input is erroneously set to 0 rather than to the carry output a1.co of the first adder. If the implementation of Fig. 10 is compared with the specification of an 16-bit adder

```
adc( a[1:16], b[1:16], 0)
```

¹For convenience, the HDL constants 0 and 1 are overloaded and represent the boolean constants L and H as well as the integers 0 and 1

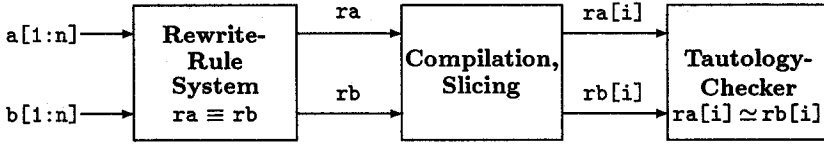


Fig. 9: Equivalence-proof of vector expressions

```

DESCRIPTION rt7483(IN  x, y: btml[1:4]; ci: btml;
                   OUT s: btml[1:4]; co: btml) BODY
  s  . =  adc( x, y, ci)[2:5],
  co . =  adc( x, y, ci)[1]
ENDrt7483
USE  a1( a[13:16], b[13:16], 0,      sum[13:16]),
     a2( a[9:12],  b[9:12], 0,      sum[9:12]),
     a3( a[5:8],   b[5:8],  a2.co, sum[5:8]),
     a4( a[1:4],   b[1:4],  a3.co, sum[1:4]): rt7483 ENDUSE

```

Fig. 10: Incorrect implementation of a 16-bit adder

then the response of LOVERT is the simplified expression of the implementation:

```
adc( a[1:12], b[1:12], 0) # adc( a[13:16], b[13:16], 0)[2:5]
```

The rewrite-rule system is able to simplify the expression for the three correctly chained 4-bit adders applying the simplification-rule for adders shown above; this results in an 12-bit adder which is concatenated with the last (and erroneously uncoupled) 4-bit adder. The expression gives thus a hint to the place where the problem is located.

This example shows also that LOVERT is able to cope with the problems involved in the aggregation of bit-sliced circuits.

4 Transitional vs. Structure-Oriented Static/ Transitional Descriptions

The second type of extensions applies to systems with sequential behavior.

Definition 5: A description is called *transitional* iff all carriers have transitional behavior.

Clearly, a description consisting of a collection of conditional transfers is transitional.

The main purpose of a transitional description is to display which transfers take place under which mutually exclusive conditions. The class of transitional descriptions comprises representations of simple state-diagrams as well as specifications of complex microprograms.

A further class of descriptions is the classical implementation of finite state machines as a composition of a transitional and a static subsystem (Fig. 11).

Definition 6: A *static/transitional description* is a combination of a static and of a transitional description.

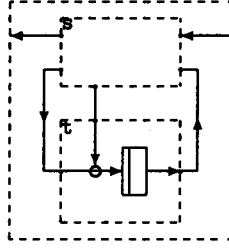


Fig. 11: Mixed transitional/static description

Definition 7: A *structure-oriented description* is a static/transitional description where each carrier of the transitional part occurs exactly once as destination.

Structure-oriented descriptions are amenable to an implementation by hardware since each transfer and each connection refers to one substructure.

We will now discuss a situation where the specification is a transitional description and where the implementation is a structure-oriented static/transitional description.

Assume several transfers into one carrier x in the transitional description d1:

```
DESCRIPTION d1 ...
  IF a1 THEN x <- y1 ENDIF,
  ...,
  IF an THEN x <- yn ENDIF
ENDd1
```

According to Definition 7, there is only one transfer into x in the description d2:

```
DESCRIPTION d2 ...
  IF ta THEN x <- ty ENDIF,
  ta .= ...,
  ty .= ...,
  ...
  ASSERT c1, ..., cm END
ENDd2
```

The assertions $c1, \dots, cm$ are used to specify don't care conditions, e.g., to exclude unreachable states or input combinations which must not occur.

To demonstrate 1-2, it must be proven for each transfer

```
IF ai THEN x <- yi ENDIF
```

of d1 that

1. the condition ai of d1 implies the condition ta of d2:

$$\vdash_{T(d2)} \forall t: (c1(t) = H) \wedge \dots \wedge (cm(t) = H) \wedge (ai(t) = H) \Rightarrow (ta(t) = H) \quad (4-1)$$

2. the condition a_i implies the equality of the sources y_i and ty , respectively:

$$\vdash_{T(d_2)} \forall t : (c1(t) = H) \wedge \dots \wedge (cm(t) = H) \wedge (ai(t) = H) \Rightarrow (ty(t) = yi(t)). \quad (4-2)$$

3. in the *storage situation* (see 2-2) the content of x remains unchanged if there is no a_i active:

$$\vdash_{T(d_2)} \forall t : (c1(t) = H) \wedge \dots \wedge (cm(t) = H) \wedge (a1(t) = L) \wedge \dots \wedge (an(t) = L) \Rightarrow (ta(t) = L) \vee (ty(t) = x(t)). \quad (4-3)$$

Note that 4-1, 4-2 and 4-3 refer to the point t of time only and amounts thus to an equivalence proof of vector-expressions. The LOVERT system is also able to cope with this type of extensions. Typical examples of application are the proof of the correct implementation of microprograms by means of a register-bus structure and a microprogram-sequencer including a ROM. An average verification time of 1 - 2 sec. per transfer on a SPARC-station was observed. The correct implementation of a complete microprogram is thus proven in a few minutes.

5 Transitional vs. Quasi-Transitional Descriptions

In the third type of extensions, transitional ("buffered") behavior of transfers is implemented by means of quasi-transitional ("unbuffered") assignments.

An example-specification of transitional-behavior is given in Fig. 12.

```
DESCRIPTION d1 ...
  DECLARE r1, r2: budv1 END
  IF p2 & b1(r2) THEN r1 <- v1(r2) ENDIF,
  IF p1 & b2(r1) THEN r2 <- v2(r1) ENDIF
ENDD1
```

Fig. 12: Specification of transitional behavior

The implementation by means of a quasi-transitional description using conditional assignments is shown in Fig. 13.

Definition 8: A description is called *quasi-transitional* iff all carriers have quasi-transitional behavior.

As an example of the proof of 1-2, we study one of the axioms associated with $r1$ of the specification (see 2-1):

$$\forall t : (0 < t) \wedge (p2(t-1) = H) \wedge (b1(r2(t-1)) = H) \Rightarrow (r1(t) = v1(r2(t-1))). \quad (5-1)$$

The corresponding axiom of the implementation of $r1$ is according to 2-3

$$\forall t : (0 < t) \wedge (p1(t) = H) \wedge (b1(r2(t)) = H) \Rightarrow (r1(t) = v1(r2(t))). \quad (5-2)$$

In order to accommodate 5-1 and 5-2, we have to require:

$$\forall t : (0 < t) \wedge (p1(t) = H) \Rightarrow (p2(t-1) = H) \quad (5-3)$$

and

$$\forall t : (0 < t) \wedge (p1(t) = H) \Rightarrow (r2(t) = r2(t-1)). \quad (5-4)$$

```

DESCRIPTION d2 ...
  DECLARE r1, r2: bvar1 END
  IF p1 & b1(r2) THEN r1 := v1(r2) ENDIF,
  IF p2 & b2(r1) THEN r2 := v2(r1) ENDIF
ENDd2

```

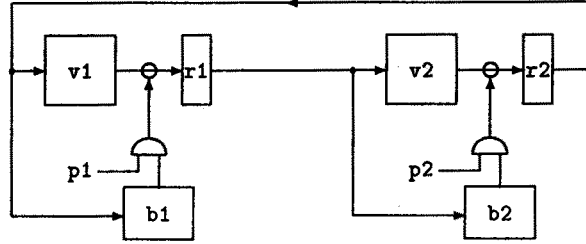


Fig. 13: Quasi-transitional implementation

The axioms associated with $r2$ are (see 2-3)

$$\begin{aligned}
 r2(0) &= L, \\
 \forall t: (0 < t) \wedge (p2(t) = H) \wedge (b2(r1(t)) = H) &\Rightarrow (r2(t) = v2(r1(t))), \\
 \forall t: (0 < t) \wedge ((p2(t) = L) \vee (b2(r1(t)) = L)) &\Rightarrow (r2(t) = r2(t-1)).
 \end{aligned}$$

Hence 5-4 is satisfied by

$$\forall t: (0 < t) \wedge (p1(t) = H) \Rightarrow (p2(t) = L). \quad (5-5)$$

Considering 5-3 and 5-5 we see that the correct implementation is ensured by a two-phase clock with the two phases $p1$ and $p2$. An example behavior is given in Fig. 14.

t	0	1	2	3	4	...
$p1(t)$	H	L	H	L	H	...
$p2(t)$	L	H	L	H	L	...

Fig. 14: Two-phase clock

As a result, it was shown that the quasi-transitional description of Fig. 13 has also transitional behavior. Note that the specification $d2$ of Fig. 12 is an abstraction from real hardware: there is no circuit that corresponds, e.g., to the expression $p2 \ \& \ b1(r2)$.

6 Summary

The relatively simple concept of extension covers a number of relevant design and verification problems including the problem of logic-verification, the problem of the correct implementation of state-diagrams by means of structural resources, and the problem of the implementation of state-diagrams by means of two-phase clocked systems.

The introduction of several classes of temporal behavior represented by appropriate HDL-constructs allows for a systematic discussion of several types of extensions.

HDL-based hardware specification techniques offer a number of advantages: (i) hardware is represented in a convenient and user-friendly way, (ii) relevant classes of verification problems can be discussed in terms of the relationship between different forms of HDL-descriptions, (iii) the designer does not need proof-expertise since specialized verification tools support fully automatic verification.

The proof problems are complicated by the vector-functions provided by most HDL's for the compact representation of hardware. The LOVERT-approach proposes a combination of rewrite and tautology-checking techniques which makes the automatic verification of complex designs feasible.

References

- [1] A. Bartsch, H. Eveking, H.-J. Faerber, M. Kelelatchew, J. Pinder, and U. Schellin. LOVERT - a logic verifier of register-transfer level descriptions. In L. Claesen, editor, *Proc. IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design*, pages 522-531, 1989.
- [2] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE C-85*, 677-691, 1986.
- [3] H. Eveking. The application of CHDL's to the abstract specification of hardware. In Koomen/Moto-Oka, editor, *Proc. CHDL '85 (Tokio)*, pages 167-178, North-Holland, 1985.
- [4] H. Eveking. Axiomatizing hardware description languages. *International Journal of VLSI Design*, 1990.
- [5] H. Eveking. Formal verification of synchronous systems. In G. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 137-152, North-Holland, 1985.
- [6] J. Hartmanis and R.E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice Hall, 1966.
- [7] J.C. Madre and J.P. Billon. Proving circuit correctness by formally comparing their expected and extracted behavior. *Proc. 25th Design Autom. Conf.*, 205-210, 1988.
- [8] R. Piloty, M. Barbacci, D. Borriore, D. Dietmeyer, F. Hill, and P. Skelly. *CONLAN Report*. Springer-Verlag, Berlin Heidelberg New-York Tokio, 1983.
- [9] T. G. Windeknecht. *General Dynamical Processes*. Academic Press, 1971.