Axiomatic Semantics of a Hardware Specification Language*

Xin Hua and **Hantao Zhang**
Dept. of Computer Science, The University of Iowa, Iowa City, IA 52242
{xin,hzhang}@cs.uiowa.edu

Abstract

Formal hardware design verification is to examine whether a structural specification of a circuit meets its behavioral specification. Despite of the recent progress on formal verification, there is a big gap between hardware designers and verifiers, partially because there are no common specification languages for them to use. Our study shows that formal semantics could bridge such a gap. By providing an axiomatic semantics to an existing hardware design language called the Iowa Logic Specification Language (ILSL), we show that a circuit description in ILSL can be automatically converted into a set of first-order formulas which is the semantic description of the circuit and is acceptable by an existing theorem prover called the Rewrite Rule Laboratory (RRL). In particular, we show how iterative statements of ILSL are converted into recursive functions of RRL. Our work thus results in a uniform specification language in which both hardware design and automatic verification can be done.

Key words: reliability, design language, formal verification, formal semantics

1 Introduction

The motivation of the study on formal semantics of hardware design languages is to apply formal methods to hardware design verification. Formal verification has gained a lot of attention recently as a viable alternative to simulation. Several notable achievements have been reported: Cullyer *et al* designed a microprocessor chip called VIPER which was (partially) formally verified and implemented in silicon [2]; Hunt formally verified FM8502, a PDP11-like microprocessor [3].

A hardware design in general consists of two basic parts: circuit description and function description. Circuit description specifies the physical construction of the circuit, called *structural specification*. Function description specifies what the circuit does, called *behavioral specification*. Formal hardware design verification is to check whether the structural specification meets its behavioral specification. Since a design is usually done hierarchically (i.e., bottom-up), in this case, formal verification is to check whether the behaviors of low level components, together the assembled structural specification, imply the behaviors of the higher level components. Because conducting proofs needs expertise of theorem provers, the later work is usually done by experts in theorem proving, whom we call verifiers in the paper.

The design languages used by today's designers like VHDL [6] lack the expressiveness for behavioral descriptions, and usually focus on physical properties of circuits, such as wire connection, driven power, fan in/out, signal synchronization and so on. These languages do not explicitly provide the logical basis for formal verification. On the other hand, specification languages (where only the abstract functionality is described and the physical details are omitted) used by the verifiers are limited by constraints of the theorem provers used for automatic verification, and are hardly understandable to hardware designers. For instance, VIPER was written in Gordon's high order logic (HOL), and Hunt's FM8502 microprocessor was written in Boyer-Moore's computational logic. The lack of a common design language seriously prevents formal verification from being widely applied to critical devices. The experiment of the VIPER project also reveals this problem [2]. For example, the errors found in VIPER's specification by the verifiers of VIPER project were not present in the actual chip. The manufacturers used a different specification because they could not use the same one used by the verifiers [1].

For verification to be effective, we must first move to a situation in which the same sources are used by designers, verifiers and manufacturers. Uniting these various communities is the motive of this research. This paper reports our first attempt to bridge the big gap introduced by the differences of specification languages between designers and verifiers. In general, it is difficult for a language to meet the demands of both sides. We propose a formal semantics for (a subset of) an existing hardware design language. By providing a set of semantic rules which transforms the structural specification of a circuit into a set of logical formulas, we obtain automatically the axioms needed to prove the correctness of the behavioral specification of the circuit for an automated theorem prover. That is, we use predicates to interpret the meaning of statements in a circuit design language so that the formal verification problem is converted into a theorem proving problem. A set of predefined predicates and a set of

*Partially supported by the National Science Foundation Grants no. CCR-9009414 and INT-9016100.

0-8186-2610-0/92 $3.00 © 1992 IEEE

translation rules provide the basis to perform an automatic translation. The formulas transformed from the behavioral specification correspond the theorems to be proved. Automated hardware verification then consists of proving these theorems from the axioms derived from the structural specifications.

2 Formal Semantics as Bridge Between Designers and Verifiers

2.1 *ILSL* and *RRL*

After studying several hardware design languages, we chose the Iowa Logic Specification Language (*ILSL*) as our first candidate. *ILSL*, developed by Jones [4] as a teaching tool since 1983, is a textual language for describing the structure of digital systems in pure dataflow terms. It can be considered as a subset of VHDL, and has a powerful set of abstraction facilities that allow compact and well-structured descriptions of circuits. Several digital systems consisting thousands of gates are designed in *ILSL*. One of the major reasons to choose *ILSL* is that its parser (written in Pascal) is available to us.

The theorem prover we chose for automated verification is *RRL* [5], with which we have extensive experiences. *RRL* is a general theorem prover based on rewrite techniques and has been used to prove many theorems which present a challenge to other theorem provers. For hardware verification, we used *RRL* to verify a dozen of small digital systems. More notable work was done by Narendran and Stillman to verify a design of a Sobel image processing chip (consisting more than 10,000 transistors) in VHDL [7]. Two bugs were found by *RRL* in that design and the verification was completed when the bugs were fixed. Note that the design of the Sobel image processing chip was transformed by hand into first-order formulas accepted by *RRL*.

A circuit description in *ILSL* consists of five sections: (a) the circuit name, (b) the inputs, (c) the outputs, (d) the parts used to make the circuit, and (e) the wires which connect the inputs, the outputs and the parts. In order to support formal specification, we decided to add one more section which describes the behavior of the circuit, or the relation between the inputs and the outputs.[1] As an example, the following is a description of a full-adder in *ILSL* (*b2n* is a predefined function which transforms a boolean value into a binary number):

```
circuit   fadder;
 inputs   cin, in1, in0;
 outputs  cout, sout;
 parts    o1, o2: or(2);
          x1, x2: xor;
          a1, a2, a3: and(2);
 wires    x1.out to x2.in0;
          o2.out to cout;
          a2.out to o1.in(0);
          a3.out to o1.in(1);
          a1.out to o2.in(0);
          cin to x1.in0, a1.in(0), a2.in(0);
          in1 to x1.in1, a1.in(1), a3.in(1);
          in0 to x2.in1, a2.in(1), a3.in(0);
          o1.out to o2.in(1); x2.out to sout;
 behaviors
          2 * b2n(cout) + b2n(sout)
           = b2n(cin) + b2n(in1) + b2n(in0);
end
```

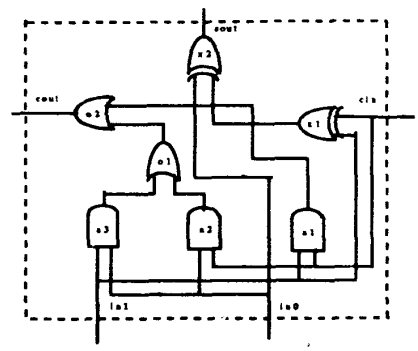


Figure 1: Circuit of the full adder

In the following, we show how circuit designs written in *ILSL* are converted into logic formulas accepted by *RRL*. We define axiomatic semantics of *ILSL* in terms of first-order logic formulas and provide a set of semantic rules which translate *ILSL* statements into the formulas. Especially, we provide a semantic rule for iterative statements (`for` loop) in *ILSL* which involve the bounded universal quantifiers. The verification of an N-bit adder in *RRL* is given as an example to illustrate our approach.

2.2 Semantic Rules for Predefined Parts

Any hardware design language must have some predefined components that are "bricks" for the designer to build his circuits. We have to predefine a set of functions as "semantic bricks" for the semantics of the design language. For example, the and-gate, which is one of the predefined parts in *ILSL*, may have three different representation as shown in Figure 2.

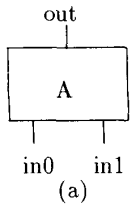
In Figure 2, (a) is a graphical representation of the and-gate, (b) is the declaration statement of *ILSL* and (c) is its semantics written in the first-order logic.

The not-gate is another predefined part. We define $not\_p : bool \times bool \rightarrow bool$ as

$$not\_p(in, out) := (in \equiv \neg out).$$

For the *ILSL* statement "`parts x: not`" (which means part `x` is a negate-gate), the corresponding se-

[1] Because behavioral specification needs auxiliary functions such as arithmetic operations, and we have not decided what set of operations are sufficient, the work on this section is still in progress and, currently, the parser of *ILSL* cannot accept this section.



(a)

`parts A: and(2)`

(b)

$A.in0 \wedge A.in1 \equiv A.out$

(c)

Figure 2: Three representations of an and-gate

mantic rule is straightforward:

$$\texttt{parts x:not} \xmapsto{T} not\_p(x.in, x.out),$$

where $\xmapsto{T}$ denotes this is a transformation rule. Similarly, semantic rules for other noniterative statements can be obtained (we will discuss them later).

For the `fadder` example given in section 2.1, we assume that the following semantic rules:

$$\begin{aligned}
&\texttt{parts y:and(n)} \xmapsto{T} and_p(n, y.in, y.out)\\
&\texttt{parts z:or(n)} \xmapsto{T} or_p(n, z.in, z.out)\\
&\texttt{parts w:xor} \xmapsto{T} xor_p(w.in0, w.in1, x.out)\\
&\texttt{wires x to y} \xmapsto{T} x = y.
\end{aligned}$$

By applying these semantic rules on each statement in `parts` section and `wires` section, we get $or\_p(2, o1.in, o1.out)$ and $xor\_p(x1.in0, x1.in1, x1.out)$ from `o1:or(2)` and `x1:xor` in the `parts` section; we get $cin = x1.in0$ from "`cin to x1.in0`" in the `wires` section, and so on. Finally, we get a conjunction of formulas, which, denoted by $fadder\_p(cin, in1, in0, cout, sout)$, is a semantic function (predicate) with the same name and input/output variables.

Definition 1

$$\begin{aligned}
&fadder_p(cin, in1, in0, cout, sout)\\
&:= \; or_p(2, o1.in, o1.out) \wedge or_p(2, o2.in, o2.out) \wedge\\
&\quad xor_p(x1.in0, x1.in1, x1.out) \wedge\\
&\quad xor_p(x2.in0, x2.in1, x2.out) \wedge\\
&\quad and_p(2, a1.in, a1.out) \wedge\\
&\quad and_p(2, a2.in, a2.out) \wedge\\
&\quad and_p(2, a3.in, a3.out) \wedge\\
&\quad cin = x1.in0 \wedge cin = a1.in(0) \wedge\\
&\quad cin = a2.in(0) \wedge\\
&\quad in1 = x1.in1 \wedge in1 = a1.in(1) \wedge\\
&\quad in1 = a3.in(1) \wedge\\
&\quad in0 = x2.in1 \wedge in0 = a2.in(1) \wedge\\
&\quad in0 = a3.in(0) \wedge x1.out = x2.in0 \wedge\\
&\quad a2.out = o1.in(0) \wedge a3.out = o1.in(1) \wedge\\
&\quad o1.out = o2.in(1) \wedge a1.out = o2.in(0) \wedge\\
&\quad x2.out = sout \wedge o2.out = cout
\end{aligned}$$

The correctness of the design is equivalent to the proof of

$$\begin{aligned}
&b2n(cin) + b2n(in1) + b2n(in0)\\
&\quad = \; 2 * b2n(cout) + b2n(sout)\\
&\quad \textbf{if} \quad fadder_p(cin, in1, in0, cout, sout).
\end{aligned}$$

2.3 Semantic Rules for Iterative Statements

Iterative statements need special care. For example, the *ILSL* statement

```
wires for i in 0..n do x(i) to y(i) endfor
```

indicates the connection between `x(i)` and `y(i)`. For the sake of its automatic reasoning power, a theorem prover prefers recursive functions over iterative statements, because recursive functions facilitate automated induction. Note that induction is indispensable for large hardware verification (this is specially true to *RRL*).

The first-order formula directly derived from the above iterative statement is a conjunction of equalities: $\bigwedge_{i=0}^{n}(x(i) = y(i))$. Because it involves quantified variables, the induction prover in *RRL* cannot accept this formula. Fortunately, in [8], we suggested a technique which uses recursively defined functions to remove those kinds of quantifiers. At first, we introduce an auxiliary function for_wire as:

Definition 2 for_wire : $number \times number \times number \times array \times array \Rightarrow bool$

$$\begin{aligned}
&for_wire(0, d1, d2, x, y)\\
&\quad := \; x(d1) = y(d2)\\
&for_wire(n+1, d1, d2, x, y)\\
&\quad := \; for_wire(n, d1, d2, x, y) \wedge\\
&\qquad x(n+1+d1) = y(n+1+d2)
\end{aligned}$$

It is easy to see that $for\_wire(n, 0, 0, x, y)$ is equivalent to $\bigwedge_{i=0}^{n}(x(i) = y(i))$. The semantic rule for the iterative statement is given below:

```
wires
  for i in 0..n do
  x(i) to y(i)        ⟼T for_wire(n, 0, 0, x, y)
  endfor
```

In the `parts` section of *ILSL*, the designer may use an array to specify a group of components that are instances of the same subcircuit. For example, `parts x(0..n): fadder` is logically equivalent to

```
{ x(0):  fadder;
  x(1):  fadder;
  ... ...
  x(n):  fadder;
```

Its formal semantics is

$$\bigwedge_{i=0}^{n} \quad fadder\_p(x.cin(i), x.in1(i), x.in0(i), x.cout(i), x.sout(i)).$$

We introduce the recursive function arr_fadder to handle the existential quantifier in the above formula.

Definition 3 $arr\_fadder : number \times array \times array \times array \times array \times array \Rightarrow bool$

$$\begin{aligned} &arr\_fadder(0, cin, in1, in0, cout, sout) \\ &\quad := \quad fadder\_p(cin(0), in1(0), in0(0), cout(0), sout(0)) \\ &arr\_fadder(n+1, cin, in1, in0, cout, sout) \\ &\quad := \quad arr\_fadder(n, cin, in1, in0, cout, sout) \wedge \\ &\qquad fadder\_p(cin(n+1), in1(n+1), in0(n+1), cout(n+1), sout(n+1)) \end{aligned}$$

Now we can use arr_fadder to specify the array expression on the full adder subcircuit. The semantic rule is the following:

```
parts
 x(0..n):
  fadder;
```
$\stackrel{T}{\longmapsto}$ $arr\_fadder(n, x.cin, x.in1, x.in0, x.cout, x.sout)$.

Alternatively and more generally, we can define a high-order function which converts an array of any subcircuits into a recursive formula. That is, for an arbitrary subcircuit $subc$, its semantic function $subc\_p$ can be derived hierarchically from its specification. Two auxiliary functions, $subc\_o$ and $subc\_v$, can be defined based on the definition of $subc\_p$, where $subc\_o(obj)$ is true if and only if obj is an instance of the subcircuit $subc$, and $subc\_v(obj)$ is a shorthand for the list of instantiated parameters of $subc\_o(obj)$.

For example, let $subc$ be an arbitrary circuit and X, Y, Z be its input/output signals and $object$ be an instance of $subc$, then $subc\_o(object)$ is the same as:

$$subc\_p(object.X, object.Y, object.Z)$$

and $subc\_v(object)$ is a shorthand for

$$(object.X, object.Y, object.Z).$$

For the N-bit adder example, after the semantic function $fadder\_p(cin, in1, in0, cout, sout)$ is derived, the instantiation function $fadder\_o(X)$ denotes

$$fadder\_p(X.cin, X.in1, X.in0, X.cout, X.sout),$$

which corresponds to a full-adder whose name is "X" and $fadder\_v(X)$ denotes

$$(X.cin, X.in1, X.in0, X.cout, X.sout).$$

Finally, we define the function $part\_arr$ to generate the recursive definitions of the semantic functions for the array components in `part` specifications.

Definition 4 $part\_arr : arr\_obj \times predicate \rightarrow pair\_of\_eqn$

$$\begin{aligned} &part\_arr(obj, subc\_p) \\ &\quad := \quad \{arr\_subc(0, subc\_v(obj)) = subc\_o(obj(0)), \\ &\qquad arr\_subc(n+1, subc\_v(obj)) \\ &\qquad\quad = arr\_subc(n, subc\_v(obj)) \wedge \\ &\qquad\quad subc\_o(obj(n+1))\} \end{aligned}$$

The recursive definition of arr_fadder can be generated by $part\_arr(fa, fadder\_p)$.

3 Definitions and Semantic Rules

We give in this section the complete list of the transformation rules for the N-bit adder. Recall that a circuit description in *ILSL* (Version 9) consists of five sections, **`circuit, inputs, outputs, parts,`** and **`wires`**. These sections describe the circuit name, the input signals, the output signals, the parts used to make the circuit, and the wires which connect the inputs, the outputs and the parts.

There are seven kinds of the predefined gates in *ILSL*:

`nand(n)`	negate-and-gate with n inputs,
`not`	negate-gate,
`and(n)`	and-gate with n inputs,
`or(n)`	or-gate with n inputs,
`nor(n)`	negate-or-gate with n inputs,
`xor`	exclusive-or for two inputs,
`equ`	equal between two inputs.

Predicate definitions for predefined parts in *ILSL* are:

Definition 5 The predicate not_p catches the meaning of *not* operation.
$not\_p : bool \times bool \rightarrow bool$
$not\_p(in, out) := (in = \neg out)$

Definition 6 The and_op is an auxiliary function to generate a conjunction list of elements in a boolean array.
$and\_op : number \times array\_of\_bool \rightarrow bool$
$and\_op(2, in) := in(0) \wedge in(1)$
$and\_op(n+1, in) := and\_op(n, in) \wedge in(n)$

Definition 7 With the help of and_op, the predicate and_p defines the meaning of *and* operation on a boolean array.
$and\_p : number \times array\_of\_bool \times bool \rightarrow bool$
$and\_p(n, in, out) := (and\_op(n, in) = out)$

Definition 8 The or_op is an auxiliary function to generate a disjunction list of elements in a boolean array.
$or\_op : number \times array\_of\_bool \rightarrow bool$
$or\_op(2, in) := in(0) \vee in(1)$
$or\_op(n+1, in) := or\_op(n, in) \vee in(n)$

Definition 9 With the help of *or_op*, the predicate *or_p* defines the meaning of *or* operation on a boolean array.
$or\_p : number \times array\_of\_bool \times bool \rightarrow bool$
$or\_p(n, in, out) := or\_op(n, in) = out$

Definition 10 The predicate *nand_p* defines the meaning of *negate_and* operation on a boolean array.
$nand\_p : number \times array\_of\_bool \times bool \rightarrow bool$

$$nand\_p(n, in, out) := \neg and\_op(n, in) = out$$

Definition 11 The *nor_p* predicate is for *negate_or* operation.
$nor\_p : number \times array\_of\_bool \times bool \rightarrow bool$
$nor\_p(n, in, out) := \neg or\_op(n, in) = out$

Definition 12 The predicate *xor_p* is for *exclusive_or* operation.
$xor\_p : bool \times bool \times bool \rightarrow bool$
$xor\_p(in0, in1, out) := in0 \wedge \neg in1 \vee \neg in0 \vee in1 = out$

Definition 13 Th predicate *equ_p* checks whether two inputs are equal to each other.
$equ\_p : bool \times bool \rightarrow bool$
$equ\_p(in1, in2, out) := (in1 = in2) = out$

Here are the semantic rules for the predefined parts and iterative statements:

1. `parts u:nand(n)` $\xrightarrow{T}$ $nand\_p(n, u.in, u.out)$
2. `parts x:not` $\xrightarrow{T}$ $not\_p(x.in, x.out)$
3. `parts y:and(n)` $\xrightarrow{T}$ $and\_p(n, y.in, y.out)$
4. `parts z:or(n)` $\xrightarrow{T}$ $or\_p(n, z.in, z.out)$
5. `parts v:nor(n)` $\xrightarrow{T}$ $nor\_p(n, v.in, v.out)$
6. `parts w:xor` $\xrightarrow{T}$ $xor\_p(w.in0, w.in1, x.out)$
7. `parts t:equ` $\xrightarrow{T}$ $equ\_p(t.in1, t.in2, t.out)$
8. `wires x to y` $\xrightarrow{T}$ $x = y$.
9. `wires for i in 0..n do`
   `x(i+d1) to y(i+d2)`
   `endfor`
   $\xrightarrow{T}$ $for\_wire(n, d1, d2, x, y)$
10. `parts x(0..n): fadder`
    $\xrightarrow{T}$ $arr\_fadder(n, x.cin, x.in1, x.in0, x.cout, x.sout)$.

4 Verification of an N-bit Adder

In this section we give an example of the verification of an N-bit adder, where $N \geq 1$ and the N-bit adder is built up by $n+1$ full-adders ($n \geq 0$) and a full-adder is built up by `or`, `xor` and `and` gates. We use the semantic rules to get its semantic functions, which are actually the abstract specification of the circuit. Based on those semantic functions, *RRL* can automatically prove its correctness.

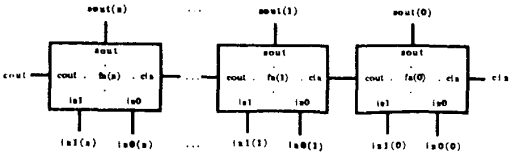


Figure 3: Circuit of the N-bit adder

This example illustrates how to apply semantic rules on the design of an N-bit adder and verify its correctness. In this design, an N-bit adder consists of N full-adders and each full-ader is built up by six predefined gates. From bottom up, we verify the design of the full-adder first, and then the N-bit adder. The structural specification of the full-adder and its semantic description *fadder_p* can be found in section 2. We need to prove the following theorem to validate the design of the full-adder.

Theorem 1 *Verification of fadder:*

$$b2n(cin) + b2n(in0) + b2n(in1) = 2 * b2n(cout) + b2n(sout) \quad \textbf{if} \quad fadder\_p(cin, in1, in0, cout, sout).$$

RRL (written in common lisp) takes 4 seconds on a Sparc station 1 to complete the proof. Once it is proved, *RRL* saved it as a rewrite rule so that in the later proof *RRL* can use the behavioral property of the full-adder without touching its structural details again.

The structural specification of the N-bit adder is:

```
circuit nadder(integer n);   /* n=N-1 */
  inputs   cin, in0(0..n), in1(0..n);
  outputs  sout(0..n), cout;
  parts    fa(0..n): fadder;
  wires    cin to fa(0).cin;
           in1(0) to fa(0).in1;
           in0(0) to fa(0).in0;
           fa(0).sout to sout(0);
           fa(n).cout to cout;

           for i in 0..n-1 do
               fa(i).cout to fa(i+1).cin;
               in1(i+1) to fa(i+1).in1;
               in0(i+1) to fa(i+1).in0;
               fa(i+1).sout to sout(i+1);
           endfor;
  behaviors
    e2(n+1)*cout + busv(sout,n) =
     busv(in1, n) + busv(in0, n) + b2n(cin);
end.
```

In the above specification, `in0` and `in1` are the two input vectors; `cin` (resp. `cout`) is the carry in (resp. out) bit; and `sout` is the output vector. The definitions of *e2* and *busv* are given below. In fact, once we

define a function in *RRL*, we can use that function in the **behaviors** section of *ILSL*.

Definition 14 $e2 : number \rightarrow number$
$e2(0) := 1;$
$e2(n+1) := 2 * e2(n).$
This function computes 2^n.

Definition 15 $busv : number \times array \rightarrow number$
$busv(0, x) := x(0);$
$busv(n+1, x) := e2(n+1) * x(n+1) + busv(n, x).$
This function computes $\sum_{i=0}^{n} 2^i * x(i)$.

The desired behavior of the N-bit adder is:

$$2^{n+1} * cout + \sum_{i=0}^{n} 2^i * sout(i)$$
$$= \sum_{i=0}^{n} 2^i * in1(i) + \sum_{i=0}^{n} 2^i * in0(i) + cin.$$

Here we assume *true* is represented by 1 and *false* by 0.

Firstly, we translate the above circuit specification into a predicate *nadder_p*.

```
nadder_p(n, cin, in0, in1, sout, cout) :=
     arr_fadder(n,fa.cin,fa.in1,fa.in0,
                fa.cout,fa.sout)      /*r10*/
 and (cin = fa(1).cin)                /*r8*/
 and (in1(1) = fa(1).in1)             /*r8*/
 and (in0(1) = fa(1).in0)             /*r8*/
 and (fa(1).sout = sout(1))           /*r8*/
 and for_wire(n-1,0,1,fa.cout,fa.cin) /*r9*/
 and for_wire(n-1,1,1,in1,fa.in1)     /*r9*/
 and for_wire(n-1,1,1,in0,fa.in0)     /*r9*/
 and for_wire(n-1,1,1,fa.sout,sout)   /*r9*/
 and (fa(n).cout = cout).             /*r8*/
```

Note that fa.cout(i) is the same as fa(i).cout in our notations and `/*r10*/` indicates that the formula was derived by applying rule 10.

The formula *nadder_p* contains a set of equalities. The following rewriting rules are generated from these equalities:

$cin \xrightarrow{T}$ fa(1).cin
$in0 \xrightarrow{T}$ fa.in0
$in1 \xrightarrow{T}$ fa.in1
$sout \xrightarrow{T}$ fa.sout
$cout \xrightarrow{T}$ fa(n).cout

Note that

$$\left.\begin{array}{l} in1(0) = fa(0).in1 \\ for\_wire(n-1, 1, 1, in1, fa.in1) \end{array}\right] \Rightarrow in1 = fa.in1.$$

By the above rewriting rules, and after removing trivially true predicates like $fa.sout = fa.sout$ and $for\_wire(n-1, fa.in1, fa.in1)$, we obtain a simplified predicate of *nadder_p* as:

Definition 16

$$\begin{array}{rl} & nadder\_p(n, fa.cin(0), fa.in0, fa.in1, \\ & \quad fa.sout, fa.cout(n)) \\ := & arr\_fadder(n, fa.cin, fa.in1, \\ & \quad fa.in0, fa.cout, fa.sout) \wedge \\ & for\_wire(n-1, 0, 1, fa.cout, fa.cin) \end{array}$$

After the simplification, it is easy to see that a 1-bit adder is actually a full-adder and the N-bit adder is built by an $(N-1)$-bit adder and a full-adder.

We have the following theorem to prove:

Theorem 2 *Verification of n-bit adder:*

$$\begin{array}{rl} & busv(n, fa.in1) + busv(n, fa.in0) + b2n(fa.cin(0)) \\ = & e2(n+1) * fa(n).cout + busv(n, fa.sout) \\ \textbf{if} & nadder\_p(n, fa.cin(0), fa.in0, fa.in1, \\ & \quad fa.sout, fa.cout(n)) \end{array}$$

Again, this theorem can be proved by *RRL* automatically.

5 The Proof of Theorem 2 in *RRL*

In order to prove Theorem 2 in *RRL*, we have to specify the data type of *num* (natural numbers) and define a couple of arithematic functions that are not defined in the previous sections.

We used 0 and *suc* (successor) as the constructors of *num* and defined the following arithematic axioms for + (plus), * (multiply), *b2n* (bool to num) and *sub1* (minus 1) operations.

```
x + 0 := x
x + suc(y) := suc(x + y)
x * 0 := 0
x * suc(y) := x + x * y
x * (y + z) == (x * y) + (x * z)
b2n(false) := 0
b2n(true)  := 1
sub1(0) := 0
sub1(suc(x)) := x
```

Then we input definition 1, 2, 3, 14, 15 and 16 into *RRL*. Theorem 1 was proved first by *RRL* and then used as a lemma in the proof of Theorem 2. The semantic functions defined before can be directly accepted by *RRL*. The minor modifications were that `aref(x, i)` was used to replace the array reference $x(i)$ and a couple of variables were renamed.

RRL converted the given definitions, axioms and lemmas into a set rewriting rules, then started to prove Theorem 2 by induction on the width parameter n.

The following are quoted from the output of *RRL*.

```
RRL-> prove
 ((e2(suc(xi)) * b2n(aref(cout, xi))) +
    busv(bout, xi))
 == busv(in0, xi) + busv(in1, xi) +
    b2n(aref(cin, 0))
 if nadder-p(xi, aref(cin, 0), in1, in0,
             bout, aref(cout, xi))
  ... ...
Let P(xi) be [main]
 (b2n(aref(cout, xi)) * e2(xi)) +
    (b2n(aref(cout, xi)) * e2(xi)) +
    busv(bout, xi)
 == b2n(aref(cin, 0)) + busv(in0, xi) +
    busv(in1, xi))
 if arr-fadder(xi, cin, in1,
               in0, cout, bout) and
    for-wire(sub1(xi), 0, suc(0), cout, cin)

The induction will be done on xi in
  arr-fadder(xi, cin, in1, in0, cout, bout),
  and will follow the scheme:
    [#1] P(0)
    [#2] P(suc(x)) if  { P(x) }
    ... ...
```

The basic case $P(0)$ is straightforward. The inductive case $P(suc(x))$ if $P(x)$ was split into three subgoals by *RRL*:

```
Conjecture [#2] is split into:
[#2.1] (b2n(aref(cout, suc(x)))*e2(suc(x)))
     + (b2n(aref(cout, suc(x)))*e2(suc(x)))
     + busv(bout, suc(x))
   == b2n(aref(cin, 0)) + busv(in0, suc(x))
      + busv(in1, suc(x))
   if arr-fadder(suc(x), cin, in1,
                 in0, cout, bout) and
      for-wire(sub1(suc(x)), 0,
               suc(0), cout, cin) and
      not(arr-fadder(x, cin, in1, in0,
                     cout, bout))
[#2.2] (b2n(aref(cout,suc(x)))*e2(suc(x)))
      + (b2n(aref(cout,suc(x)))*e2(suc(x)))
      + busv(bout, suc(x))
   == b2n(aref(cin, 0)) + busv(in0,suc(x))
      + busv(in1,suc(x))
   if arr-fadder(suc(x), cin, in1,
                 in0, cout, bout) and
      for-wire(sub1(suc(x)), 0, suc(0),
               cout, cin) and
      not(for-wire(sub1(x), 0, suc(0),
                   cout, cin))
[#2.3] (b2n(aref(cout,suc(x)))*e2(suc(x)))
    + (b2n(aref(cout, suc(x)))*e2(suc(x)))
    + busv(bout, suc(x))
   == b2n(aref(cin, 0)) + busv(in0, suc(x))
      + busv(in1, suc(x))
   if arr-fadder(suc(x), cin, in1, in0,
                 cout, bout) and
      for-wire(sub1(suc(x)), 0, suc(0),
               cout, cin) and
      (((b2n(aref(cout, x)) * e2(x))
       + (b2n(aref(cout, x)) * e2(x))
       + busv(bout, x)) =
      (b2n(aref(cin, 0)) + busv(in0, x)
       + busv(in1, x)))
```

[#2.1] and [#2.2] were proven because of the contradictions between arr_fadder and $\neg arr\_fadder$, for_wire and $\neg for\_wire$ in their conditions. By applying the lemma

```
b2n(aref(cout, x)) * e2(x)
  == b2n(aref(cin, suc(x))) * e2(x)
  if for-wire(x, 0, suc(0), cout, cin)
```

and the induction hypothesis of Theorem 2, [#2.3] was reduced to true. The proof of Theorem 2 was thus completed.

6 Summary

We have supplied a set of semantic rules which specify the axiomatic semantics of *ILSL* and translate structural specifications of *ILSL* into first-order logic formulas accepted by *RRL*. In order to facilitate the proof process in *RRL*, three preprocessing steps are used: (1) simplification; (2) getting rid of input/out variables; and (3) derivation of recursive definitions from iterative statements.

Our approach can be generalized in the following sense: For any hardware design language, we may define its formal semantics (which captures its functionality) by a set of semantic rules. This set of semantic rules translates a hardware specification into an abstract specification written in an intermediate language, then the abstract specification can be easily tailored to fit various theorem provers. Although there is no single, comprehensive intermediate language for formal semantics, we think that the first-order language is a good one to define the axiomatic semantics of a hardware design language, because it is well understood by both designers and verifiers, and allows efficient automated proofs.

Our future work includes a parallel study between *ILSL* and VHDL (*ILSL* can be considered as a subset of VHDL by some trivial modifications), and the completement of the parser which takes *ILSL* as input and outputs the formulas from both the structural specification and the behavioral specification. Including the section on behavioral specifications into a hardware design language is especially important for formal verification, because for verifiers to undertake hardware proofs, documentation is vital; the verifiers will not be knowledgeable in engineering matters, and a behavioral specification of a circuit will save a lot of time spent working out connections which are obvious to engineers. At the first stage of this research, we will use the improved *ILSL* in the class for students to learn the concept of formal verification. While the work on *ILSL* evolves, we hope in the near future to develop a CAD workstation for hardware designers to do automatic verification.

References

[1] Brock, B., Hunt, W.A.: (1990) Report on the formal specification and partial verification of the VIPER microprocessor. Tech. Report 46-90, Computational Logic, Inc. Austin, TX.

[2] Cohn A.: (1988) A proof of correctness of the VIPER microprocessor: the first level. In: VLSL Specification, Verification and Synthesis, Kluwer Academic Publishers.

[3] Hunt, W.A., (1985) FM8501: A verified microprocessor. Ph.D Thesis, University of Texas at Austin, TX.

[4] Jones, D.W., (1988) Iowa logic simulator user's manual. Dept. of Computer Science, The University of Iowa, Iowa City, IA.

[5] Kapur, D., Zhang, H.: (1989) "An overview of RRL: Rewrite Rule Laboratory", *Proc. of the third International Conference on Rewriting Techniques and its Applications* (RTA-89), April 1989, Chapel Hill, NC, Lecture Notes in Computer Science, Vol. 355, Springer-Verlag, Berlin, 513-529.

[6] Lipsett, R., Schaefer, F.C., Ussery, C.: (1989) VHDL: Hardware Description and Design. Kluwer Academic Publishers.

[7] Narendran, P., and Stillman, J. (1988) Formal verification of the Sobel image processing chip. In the Proc. of Design Automation Conference.

[8] Zhang, H., Guha, A., Hua, X.: (1991) Using algebraic specifications in Floyd-Hoare assertions. In: Rus, T. (ed.): Proc. of Second International Conference on Algebraic Methodology and Software Technology, Iowa City, Iowa.