AEROSPACE REPORT NO.
ATR-92(2778)-3

Some Examples of Verifying Stage 1 Hardware Descriptions Using the State Delta Verification System (SDVS)

30 September 1992

Prepared by

I. V. FILIPPENKO, J. M. BOULER, and B. H. LEVY
Computer Systems Division

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

Engineering and Technology Group

19941214 005

DTIC QUALITY INSPECTED 1

PUBLIC RELEASE IS AUTHORIZED

THE AEROSPACE CORPORATION
El Segundo, California

AEROSPACE REPORT NO.
ATR-92(2778)-3

Some Examples of Verifying Stage 1 Hardware Descriptions Using the State Delta Verification System (SDVS)

Prepared by

I. V. FILIPPENKO, J. M. BOULER, and B. H. LEVY
Computer Systems Division

Accesion For	
NTIS CRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

30 September 1992

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

Engineering and Technology Group

PUBLIC RELEASE IS AUTHORIZED

Report No.
ATR-92(2778)-3

SOME EXAMPLES OF VERIFYING STAGE 1 VHDL HARDWARE DESCRIPTIONS USING THE STATE DELTA VERIFICATION SYSTEM (SDVS)

Prepared

I. V. Filippenko
I. V. Filippenko

[illegible]
J. M. Bouler

Beth Levy
B. H. Levy

Approved

Beth Levy
B. H. Levy, Manager
Computer Assurance Section

DB Baker
D. B. Baker, Director
Trusted Computer Systems Department

DB Baker for CA Sunshine
C. A. Sunshine, Principal Director
Computer Science and Technology
Subdivision

Abstract

We illustrate, by a sequence of examples, how the State Delta Verification System (SDVS) can be used to create formal specifications and correctness proofs for hardware descriptions in Stage 1 VHDL, a subset of the VHSIC Hardware Description Language (VHDL).

The examples include the following:

- a handshake protocol for interprocess communication
- a counter
- a description involving `TRANSPORT` delay
- a description involving a `WAIT` statement embedded in a conditional
- a description involving a `WAIT` statement embedded in a loop
- a description involving an `EXIT` from a nested loop
- a shift-and-add multiplier

Of these, the first two and the last are realistic hardware descriptions, while the remainder are intended to demonstrate the additional functionality of Stage 1 VHDL compared to Core VHDL, the original SDVS VHDL language subset.

Contents

1 Introduction

The State Delta Verification System (SDVS), under development over the course of several years at The Aerospace Corporation, is an automated verification system that aids in writing and checking proofs that a computer program or (design of a) digital device satisfies a formal specification. The system takes its name from the distinguishing formulas, called *state deltas*, of its underlying temporal logic.

The purpose of this report is to show, by a sequence of examples, how SDVS can be used to create formal specifications for and correctness proofs of hardware descriptions in the Stage 1 VHDL subset of the VHSIC Hardware Description Language (VHDL). Familiarity on the reader's part with SDVS, as described in [1], is assumed. We also assume knowledge of the basics of behavioral VHDL (simulation cycle, processes, signals, drivers); references [2] and [3] are useful supplements to the *VHDL Language Reference Manual* [4]. Access to [5] and [6] is helpful, but not strictly necessary.

As explained in Section 2, SDVS is being adapted to VHDL in stages. To date, three VHDL subsets have been defined and incorporated into SDVS: Core VHDL [7], Stage 1 VHDL [6], and Stage 2 VHDL [8], each being a language superset of the one preceding. The seven examples presented in this report are all written in Stage 1 VHDL; listed in order of presentation, they are:

- a handshake protocol for interprocess communication
- a counter
- a description involving `TRANSPORT` delay
- a description involving a `WAIT` statement embedded in a conditional
- a description involving a `WAIT` statement embedded in a loop
- a description involving an `EXIT` from a nested loop
- a shift-and-add multiplier

The "handshake protocol" example, borrowed from lecture notes prepared by Vantage Analysis Systems, Inc. [9], illustrates a small but interesting hardware function described in Stage 1 VHDL.

The "counter" example is an extension of one presented in [5], with a somewhat more general specification.

The "shift-and-add multiplier" example is by far the most substantial: its discussion addresses the sorts of issues that can arise in the course of a rather complex correctness proof of a relatively sophisticated Stage 1 VHDL hardware description.

The remaining examples are intended primarily to demonstrate the additional functionality of Stage 1 VHDL compared to Core VHDL.

Before embarking on the examples, we present an overview of SDVS and its application to hardware verification in Section 2, a definition of the Stage 1 VHDL subset in Section 3, and in Section 4 an itemization of the function and predicate symbols in the SDVS Simplifier for reasoning about VHDL time and waveforms during the symbolic execution of VHDL descriptions.

In general, each example is organized according to the following topics:

- General Description
- VHDL Code
- State Delta Specification
- Batch Proof
- Discussion

A few examples are simple enough that their proofs require no further comment, in which case the Discussion section is omitted. On the other hand, the proof of the "shift-and-add multiplier" requires static reasoning using integer and bitstring lemmas, which are exhibited in separate sections. Owing to its complexity, we have chosen to present this particular example last.

Finally, it is to be understood that the actual pathnames to files referenced in this report are irrelevant to the report's content. All the reader needs to know is that a path to a `.vhdl` file *must* be supplied as the argument to the `vhdltr` ("translate VHDL file") command, and that we follow the convention of supplying a path to a `.spec` file as the argument to the `read` ("read file") command.

2 SDVS and Hardware Verification

The long-term goal of the SDVS project is to create a prototype of a production-quality verification system that is useful at all levels of the hierarchy of digital computer systems; our aim is to verify hardware from gate-level designs to high-level architecture, and to verify software from the microcode level to application programs written in high-level programming languages. We are currently extending the applicability of SDVS to both lower levels of hardware design and higher levels of computer programs. Detailed information on the system may be found in [1].

Several features distinguish SDVS from other verification systems (refer to [10] for a detailed discussion). The underlying temporal logic of SDVS, called the *state delta logic*, has a formal model-theoretic semantics. SDVS is equipped with a theorem prover that runs in interactive or batch modes; the user supplies high-level proof commands, while many low-level proof steps are executed automatically. One of the more distinctive features of SDVS is its flexibility — there is a well-defined and relatively straightforward method of adapting the system to arbitrary application languages (to date, we have tackled ISPS, Ada, and VHDL). Furthermore, descriptions in the application languages potentially serve as either specifications or implementations in the verification paradigm. A given application language is incorporated into SDVS by being translated to the state delta logic via a Common Lisp *translator* program, which is (generally) automatically derived from a formal denotational semantics for the application language.

Following the approach taken successfully with Ada [11], we have envisioned an incremental adaptation of SDVS to a sequence of VHDL subsets of increasing semantic complexity [12]. For a simple but nontrivial initial language subset, dubbed Core VHDL, in fiscal year 1989 we formally specified and implemented a translator and made attendant enhancements to the SDVS inference machinery. In fiscal year 1990, the Core VHDL translator was substantially revised and applied to a variety of examples [5]. The VHDL features comprising Core VHDL were carefully selected to circumscribe a manageable set of essential language constructs, e.g. entity declarations, architecture bodies, concurrent processes, and sequential signal assignment statements [13]. A translator for a somewhat larger VHDL subset, called Stage 1 VHDL, was implemented in fiscal year 1991 [6].

We did not intend that either Core VHDL or Stage 1 VHDL be used to express real hardware applications. Rather, our goal was the rapid prototyping of an SDVS interface to VHDL — the semantic specification of some key VHDL features and experimentation with small, though interesting, examples. This work formed the basis for Stage 2 VHDL, a much more capable and realistic language subset incorporated into SDVS in fiscal year 1992.

In some aspects of its behavior, the VHDL translator functions like a simulator kernel [7, 14]. Unlike simulation, however, translation into state delta logic enables correctness proofs by *symbolic execution*. An SDVS *correctness proof* demonstrates that one state delta implies another, or that two state deltas are equivalent. Thus, a formal relationship between VHDL descriptions and abstract specifications must be created, on which the SDVS proof strategies may be brought to bear.

Figure 1 diagrams our current paradigm for hardware verification, showing the relationships

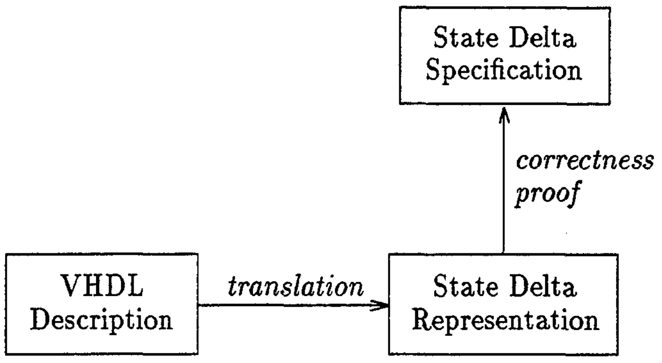

Figure 1: Hardware Verification Paradigm (Short-Term)

between a VHDL description and its state delta representation and specification. The box at the left is what a hardware designer normally sees; it contains a VHDL data flow or behavioral architecture (the initial VHDL subsets do not encompass structural architectures). The VHDL translator automatically maps the VHDL description to its state delta representation, according to the semantics that the translator implements. The specification of intent for the VHDL description is hand-coded in the state delta language by the person performing the verification. A correctness proof verifies that the lower-level state delta representation of the description implies the upper-level state delta specification.

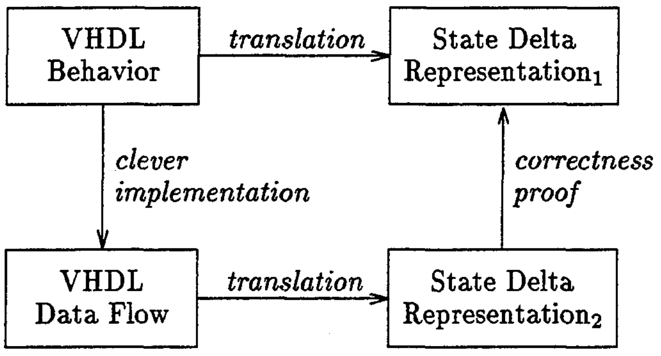

Figure 2: Hardware Verification Paradigm (Long-Term)

Figure 2 diagrams our long-term paradigm for hardware verification. Unlike Figure 1, Figure 2 involves two different VHDL descriptions. The hardware designer implements the behavioral description by a data flow description. The VHDL translator automatically maps both descriptions to their separate state delta representations, and a correctness proof demonstrates that the one description indeed implements the other. This second paradigm has not yet been exercised by SDVS in the context of VHDL, although it is employed extensively for ISPS verification (see [15, 16]).

3 Stage 1 VHDL

Stage 1 VHDL is a relatively small, but powerful, subset of VHDL. As in developing Ada for SDVS, our approach with VHDL has been to implement increasingly complex language subsets; this enables a smooth, structured approach to hardware verification. The first such subset was Core VHDL. The second subset, Stage 1 VHDL, is described here. Other stages are described in [12].

Stage 1 VHDL encompasses a small collection of language constructs that yield insight into the state delta semantics and translation of concurrency and timing. To the extent that they have been incorporated in SDVS, the previous two application languages, ISPS and Ada, provide no experience in this arena.

Differences between VHDL and most other programming languages stem principally from the presence of `SIGNAL` objects, `PROCESS` statements, and the event-driven simulation semantics. Consequently, we focus on these features in Stage 1 VHDL. Stage 1 VHDL descriptions are limited to the specification of hardware behavior or data flow, rather than structure. More comprehensive VHDL subsets for SDVS will include constructs for the structural description of hardware in terms of its hierarchical decomposition into connected subcomponents.

The primary VHDL abstraction for modeling a digital device is the *design entity*. A design entity has two parts: an *entity declaration*, giving an external view of the component by declaring the input and output *ports*, and an *architecture body*, giving an internal view of the system in terms of its structure or behavior.

In Stage 1 VHDL, each architecture body is constrained to be *behavioral*, consisting of a set of declarations and concurrent `PROCESS` statements. As described below, processes define the functional interpretation of the device being modeled. The data types of Stage 1 VHDL are *integers*, *booleans*, *bits*, and *bitvectors* (arrays of bits).[1]

A `PROCESS` statement is a block of sequential *zero-time statements* that execute sequentially but "instantaneously" in *zero time* [14], and some (possibly none) special sequential `WAIT` statements, whose execution suspends the process and allows time to elapse. A process may schedule future values to appear on data holders called *signals*, by means of *sequential signal assignment* statements. The execution of a signal assignment statement puts (at least one) new *transaction*, or time-value pair, on a *driver* of the signal assigned to, which is known as the *target signal*.

Signals act as data pathways between processes. Each process applies operations to values being passed through the design entity. We may regard a process as a program implementing an algorithm, and a Stage 1 VHDL description as a collection of independent programs running in parallel.

In full VHDL, a target signal appearing in multiple processes has correspondingly many drivers; this is called a *resolved signal*. As did Core VHDL, Stage 1 VHDL disallows resolved signals: a signal is not permitted to appear as the target of a signal assignment

[1]An additional *physical* type `TIME`, used for time expressions in `WAIT` statements, is modeled by type `INTEGER`.

statement in more than one process body; equivalently, every signal has a unique driver.

Stage 1 VHDL removes three restrictions previously imposed by Core VHDL:

1. In Stage 1 VHDL, iterative constructs may occur inside a process body; these comprise the `LOOP` and `WHILE` statements (but not `FOR` loops).
2. In Stage 1 VHDL, a `WAIT` statement can occur wherever any other sequential statement can occur, e.g. nested inside conditionals or loops.
3. In Stage 1 VHDL, *transport delays* in signal assignments can be explicitly indicated, overriding the default of *inertial delay*.

In lieu of explicit `WAIT`s, a process may have a *sensitivity list* of signals that activate process resumption upon receiving a new value. The sensitivity list implicitly inserts a `WAIT` statement as the last statement of the process body.

The VHDL *concurrent signal assignment* statement is not included in Stage 1 VHDL, as it is always equivalent to a `PROCESS` statement (with a sensitivity list) comprised of just that signal assignment. Thus *data flow* architectures, which employ a register-transfer style of description using concurrent signal assignment statements, are seen to be special cases of behavioral architectures.

Both a concrete and an abstract syntax for Stage 1 VHDL have been defined [6], as required, of course, for the implementation of the Stage 1 VHDL translator. Perhaps the following summary provides the best way of seeing the language subset at a glance:

- design entities: entity declarations + architecture bodies
- object declarations: `CONSTANT`, `VARIABLE`, `SIGNAL` & `PORT`
- sequential statements
  - `NULL` statement
  - variable assignment (scalar & bitvector)
  - signal assignment (scalar & bitvector)
  - inertial or `TRANSPORT` delay
  - conditional constructs: `IF`, `CASE`
  - iteration constructs: `LOOP`, `WHILE`
  - `WAIT` statement, possibly nested in other constructs
- concurrent statements
  - `PROCESS` statement

4 Simplifier Theories for VHDL

SDVS correctness proofs of VHDL descriptions necessitate the extension of the SDVS Simplifier with two new theories, which provide knowledge about the domains *VHDL Time* and *VHDL Waveforms*.

This section describes briefly the function and predicate symbols of these theories, as well as their interpretations. For information on other Simplifier theories, the reader is referred to the SDVS 11 Users' Manual [1].

4.1 VHDL Time

The character "`t`" is used to denote the theory of VHDL Time. The command "`activate t`" activates the solver for the theory of VHDL Time, making available the Simplifier's automatic deductive capabilities for the VHDL Time domain; "`deactivate t`" deactivates this solver.

The language of the theory of VHDL Time contains the function and predicate symbols described by the following table, in which T denotes the domain of VHDL time objects, N the domain of nonnegative integers, and P the domain of propositional (boolean) values.

VHDL Time Symbols

function symbol	simp symbol	description	type
time	`TIME`	time constructor	$N \times N \to T$
timeglobal	`TIMEGLOBAL`	global time selector	$T \to N$
timedelta	`TIMEDELTA`	delta time selector	$T \to N$
timeplus	`TIMEPLUS`	time addition	$T \times T \to T$

predicate symbol	simp symbol	description	type
timelt	`TIMELT`	time less than	$T \times T \to P$
timele	`TIMELE`	time less than or equal	$T \times T \to P$
timegt	`TIMEGT`	time greater than	$T \times T \to P$
timege	`TIMEGE`	time greater than or equal	$T \times T \to P$

The interpretations of the VHDL Time symbols are as follows.

Function *time* takes two nonnegative integers, m and n, and constructs $time(m, n)$, a VHDL time object.

Function *timeglobal* takes a VHDL time object $time(m, n)$ and returns m, the global time component.

Function *timedelta* takes a VHDL time object $time(m, n)$ and returns n, the delta time component.

Function *timeplus* takes two VHDL time objects, $time(m_1, n_1)$ and $time(m_2, n_2)$, and returns a time object that is their sum, according to the following (idiosyncratic) definition:

- if $m_2 = 0$, then
$$timeplus(time(m_1, n_1), time(m_2, n_2)) = time(m_1, n_1 + n_2)$$
- if $m_2 \neq 0$, then
$$timeplus(time(m_1, n_1), time(m_2, n_2)) = time(m_1 + m_2, 0)$$

Predicates *timelt*, *timele*, *timegt*, and *timege* compare two VHDL time objects according to the lexicographic order in their components.

4.2 VHDL Waveforms

The character "`w`" is used to denote the theory of VHDL Waveforms. The command "`activate w`" activates the solver for the theory of VHDL Waveforms, making available the Simplifier's automatic deductive capabilities for the VHDL Waveforms domain; "`deactivate w`" deactivates this solver.

The language of the theory of VHDL Waveforms contains the function and predicate symbols described by the following table, in which W denotes the domain of VHDL waveform objects, TR the domain of transaction objects, T the domain of VHDL time objects, N the domain of nonnegative integers, P the domain of propositional (boolean) values, and U the universal domain (any arbitrary object is in U).

VHDL Waveforms Symbols

function symbol	simp symbol	description	type
waveform	`WAVEFORM`	waveform constructor	$TR^+ \rightarrow W$
transaction	`TRANSACTION`	transaction constructor	$T \times U \rightarrow TR$
inertial_update	`INERTIAL_UPDATE`	waveform update	$W \times TR^+ \rightarrow W$
transport_update	`TRANSPORT_UPDATE`	waveform update	$W \times TR^+ \rightarrow W$
val	`VAL`	driver value	$W \times T \rightarrow U$

predicate symbol	simp symbol	description	type
preemption	`PREEMPTION`	preemption test for update	$W \times TR \rightarrow P$

The interpretations of the VHDL Waveforms symbols are as follows.

Function *waveform* takes a sequence of one or more transaction objects, $transaction_1$, $transaction_2$, ..., and constructs $waveform(transaction_1, transaction_2, \ldots)$, a waveform object.

Function *transaction* takes a VHDL time object $time(m,n)$ and a value v, and constructs a transaction object $transaction(time(m,n),v)$.

Functions *inertial_update* and *transport_update* each take a waveform object and a sequence of transaction objects, and return the updated waveform according to the VHDL Language Reference Manual's algorithms for updating projected output waveforms using inertial and transport delays, respectively (see [4], Section 8.3.1).

Function *val* takes a waveform object and a VHDL time object, and returns the value of that transaction in the waveform whose time is nearest to, but not greater than, the time object.

Predicate *preemption* takes a waveform and a transaction, and determines whether that transaction will preempt (replace) prior transactions on the waveform as a result of the VHDL algorithm for inertial driver update.

5 Handshake Protocol

5.1 General Description

The Stage 1 VHDL description for this example, displayed in Section 5.2, is assumed to reside in a file `handshake.vhdl`.

Adapted from a set of lecture notes produced by Vantage Analysis Systems, Inc. [9], the description defines a *handshake protocol* for communicating data between two processes, in which:

- Process A sends data to Process B; and
- Process A suspends until the "acknowledge" signal ack is received from Process B.

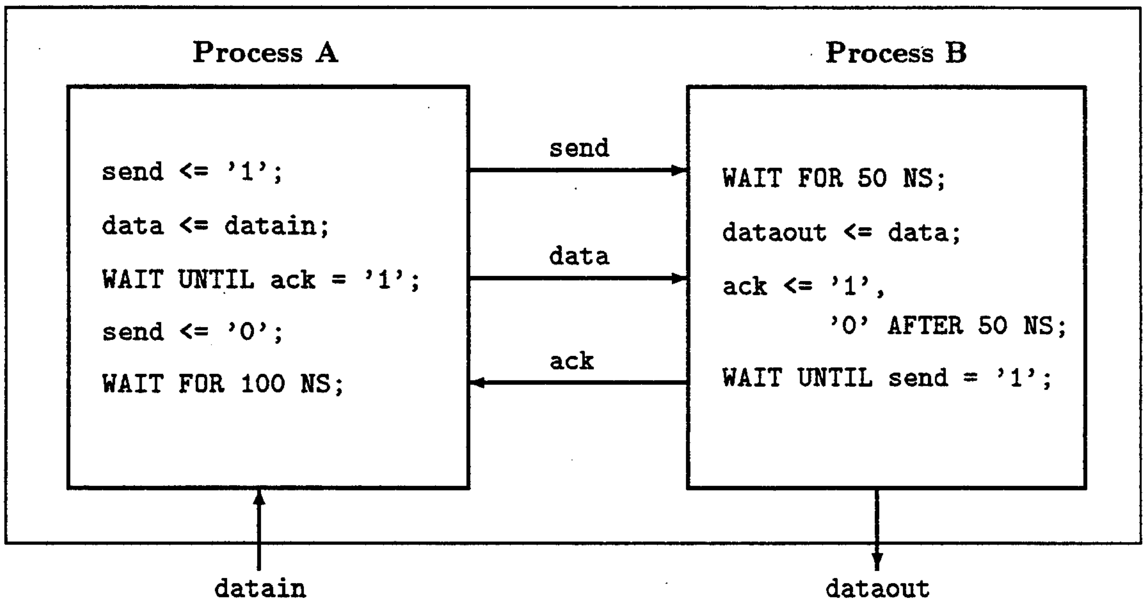

Figure 3: Handshake Protocol

5.2 VHDL Code

```
PROGRAM handshake IS


ENTITY handshake IS

  PORT ( SIGNAL datain  : IN  INTEGER;
         SIGNAL dataout : OUT INTEGER );

END handshake;


ARCHITECTURE register_transfer OF handshake IS

  SIGNAL send, ack : BIT;
  SIGNAL data : INTEGER;

BEGIN

  a : PROCESS

  BEGIN

    send <= '1';
    data <= datain;
    WAIT UNTIL ack = '1';
    send <= '0';
    WAIT FOR 100 NS;

  END PROCESS a;


  b : PROCESS

  BEGIN

    WAIT FOR 50 NS;
    dataout <= data;
    ack <= '1', '0' AFTER 50 NS;
    WAIT UNTIL send = '1';

  END PROCESS b;

END register_transfer;
```

5.3 State Delta Specification

The state delta handshake.sd below, our specification for this example, is assumed to reside in a file handshake.spec.

It expresses the claim:

> At any point at which the translation of handshake.vhdl holds, there will be a later point at which the model will have been elaborated and such that at some later point, the datain will have been transmitted to dataout, its receipt acknowledged, and the send flag for that data transmission reset. Furthermore, there will be a still later point, corresponding to increments in vhdltime of first 100 nanoseconds (= 100,000,000 femtoseconds) and then 1 delta cycle, at which the send flag will be set for the next transmission.

```
handshake.sd =

  [sd pre: (vhdl(handshake.vhdl))
      mod: (all)
     post: (vhdl_model_elaboration_complete(handshake),
            [sd pre: (true)
              comod: (all)
                mod: (all)
               post: (#dataout = .datain,
                      #ack = 1(1),
                      #send = 0(1),
                      [sd pre: (true)
                        comod: (all)
                          mod: (all)
                         post: (#vhdltime =
                                  timeplus(timeplus(.vhdltime,
                                                    time(100000000,0)),
                                           time(0,1)),
                                #send = 1(1))])])]
```

5.4 Batch Proof

A straightline symbolic execution proof is all that is needed for this example. Notice that we need to open proofs of the nested state deltas in handshake.sd at the appropriate points.

```
(defproof handshake.proof
   "(setflag autoclose off,
     vhdltr \"testproofs/vhdl/handshake.vhdl\",
     read \"testproofs/vhdl/handshake.spec\",
     prove handshake.sd
       proof:
         (go vhdl_model_elaboration_complete(handshake),
          prove g(2)
            proof:
              (go #dataout = .datain & #ack = 1(1) & #send = 0(1),
               prove g(4)
                 proof:
                   (go #send = 1(1),
                    close),
               close),
          close))")
```

6 Two-Bit Counter

6.1 General Description

The Stage 1 VHDL description for this example, displayed in Section 6.2, is assumed to reside in a file `counter.vhdl`.

The description models a two-bit counter, that is, a device that counts "ticks" of a "clock" modulo 4. A "tick," for the purposes of this example, is a change of the `clock` signal from bit '0' to bit '1'. The number of such ticks is recorded on output bitvector q of length 2.

Ordinarily, a *design under test* is used as a component in a *test bench* with another entity, the *test generator*, whose purpose is to generate test values. Since Stage 1 VHDL does not encompass such structural descriptions, we incorporate a test-generating process `clock_driver` as part of the architecture body in the description below. This process generates the ticks of signal `clock`, which are counted by process `count_up`.

6.2 VHDL Code

```
PROGRAM counter IS


ENTITY count IS

  PORT ( SIGNAL q : OUT BIT_VECTOR(1 DOWNTO 0) );

END count;


ARCHITECTURE behavior OF count IS

    SIGNAL clock : BIT;

BEGIN

  clock_driver: PROCESS

  BEGIN

    clock <= '0', '1' AFTER 50 NS;
    WAIT FOR 100 NS;

  END PROCESS clock_driver;
```

```
    count_up: PROCESS (clock)

      VARIABLE count_value : INTEGER := 0;

    BEGIN

      IF clock = '1' THEN

        count_value := (count_value + 1) MOD 4;

        IF count_value MOD 2 = 0
          THEN q(0) <= '0' AFTER 10 NS;
          ELSE q(0) <= '1' AFTER 10 NS;
        END IF;

        IF count_value / 2 = 0
          THEN q(1) <= '0' AFTER 10 NS;
          ELSE q(1) <= '1' AFTER 10 NS;
        END IF;

      END IF;

    END PROCESS count_up;

  END behavior;
```

6.3 State Delta Specification

The state delta `counter.sd` below, our specification for this example, is assumed to reside in a file `counter.spec`.

It expresses the claim:

> At any point at which the translation of `counter.vhdl` holds, `60 + 100*.n` nanoseconds later (for `n = 0` or `1`) it will be the case that if `|.q|` is the current count of clock ticks, then in another 100 nanoseconds the count `|#q|` of clock ticks will be `(|.q| + 1) mod 4`.

The full specification and proof of the behavior of descriptions such as this one is a topic of current research in SDVS; see Section 6.5.

```
counter.sd =

  [sd pre: (vhdl(counter.vhdl),
            covering(all,n),
            .n = 0 or .n = 1)
    comod:
      mod: (all)
     post: (#vhdltime = time(60 + 100*.n, 0),
            formula(counter.sdbody))]


counter.sdbody =

  [sd pre: (true)
    comod: (all)
      mod: (all)
     post: (#vhdltime = timeplus(.vhdltime,time(100,0)),
            |#q| = (|.q| + 1) mod 4)]
```

6.4 Batch Proof

The proof simply involves arguing by cases on the number n of clock ticks.

```
(defproof counter.proof
   "(setflag autoclose off,
     vhdltr \"testproofs/vhdl/counter.vhdl\",
     read \"testproofs/vhdl/counter.spec\",
     prove counter.sd
       proof:
         cases .n = 0
           then proof:
             (go #vhdltime = time(60,0),
              prove g(2)
                proof:
                  (go |#q| = (|.q| + 1) mod 4,
                   close),
              close)
           else proof:
             (go #vhdltime = time(160,0),
              prove g(2)
                proof:
                  (go |#q| = (|.q| + 1) mod 4,
                   close),
              close))")
```

6.5 Discussion

The intended behavior of the counter is modeled (1) by the expression `|#q| = (|.q| + 1) mod 4`, to indicate that the new value of the two-bit counter q is one greater (modulo 4) than its previous value, and (2) by the `time` expressions, to indicate the hardware times at which q changes value.

The above verification demonstrates that the counter model behaves correctly for the hardware times `time(60,0)` and `time(160,0)`.

More generally, one would wish to require correct behavior of the counter for all hardware times `time(60 + 100*.n, 0)`, where `n` is a nonnegative integer. This more comprehensive specification is easily formulated by slightly modifying the state deltas in Section 6.3, as follows:

```
counter.sd =

  [sd pre: (vhdl(counter.vhdl),
            covering(all,n),
            0 le n)
    comod:
      mod: (all)
     post: (#vhdltime = time(60 + 100*.n, 0),
            formula(counter.sdbody))]


counter.sdbody =

  [sd pre: (true)
    comod: (all)
      mod: (all)
     post: (#vhdltime = timeplus(.vhdltime,time(100,0)),
            |#q| = (|.q| + 1) mod 4)]
```

At present we cannot prove the general specification because the SDVS Simplifier and Stage 1 VHDL translator are currently limited to reasoning about concrete, as opposed to *symbolic*, `time` values. Furthermore, to achieve such general results concerning all hardware cycles, the capability of *induction over VHDL simulation cycles* would be required.

We are currently investigating ways to define the necessary induction rule. Although several technical problems must be overcome, we remain optimistic about the feasibility of verifying VHDL hardware descriptions by symbolic execution with symbolic VHDL `time` expressions.

7 TRANSPORT Delay

7.1 General Description

The Stage 1 VHDL description for this example, displayed in Section 7.2, is assumed to reside in a file transport_delay.vhdl.

The description illustrates the translation and proof of a Stage 1 VHDL description involving a TRANSPORT delay in a sequential signal assignment statement, a language feature not available in Core VHDL.

7.2 VHDL Code

```
PROGRAM transport_delay IS


ENTITY transport_delay IS

  PORT ( SIGNAL a : INOUT INTEGER := 0;
         SIGNAL b : OUT INTEGER
       );

END transport_delay;


ARCHITECTURE behavior OF transport_delay IS

BEGIN

  PROCESS (a)

  BEGIN

    b <= TRANSPORT a AFTER 10 NS;

    IF a < 3 THEN

      a <= a + 1 AFTER 1 NS;

    END IF;

  END PROCESS;

END behavior;
```

7.3 State Delta Specification

The state delta transport_delay.sd below, our specification for this example, is assumed to reside in a file transport_delay.spec.

It expresses the claim:

> At any point at which the translation of transport_delay.vhdl holds, there will be a later point at which the model will have been elaborated, such that at some later point, the time time(13,0) will have been achieved, the values of both a and b at this time will be 3, and the model will have completed execution.

```
transport_delay.sd =

  [sd pre: (vhdl(transport_delay.vhdl))
    comod:
      mod: (all)
     post: (vhdl_model_elaboration_complete(transport_delay),
            [sd pre: (true)
              comod: (all)
                mod: (all)
               post: (#vhdltime = time(13,0),
                      #a = 3,
                      #b = 3,
                      vhdl_model_execution_complete(transport_delay))])]
```

7.4 Batch Proof

A straightline symbolic execution proof is all that is needed for this example.

```
(defproof transport_delay.proof
   "(setflag autoclose off,
     vhdltr \"testproofs/vhdl/transport_delay.vhdl\",
     read \"testproofs/vhdl/transport_delay.spec\",
     prove transport_delay.sd
       proof:
         (go vhdl_model_elaboration_complete(transport_delay),
          prove g(2)
            proof:
              (go vhdl_model_execution_complete(transport_delay),
               close),
          close))")
```

7.5 Discussion

Two types of time delay can be specified by a sequential signal assignment statement, and Stage 1 VHDL encompasses both. *Inertial delay*, the default, models a target signal's inertia that must be overcome in order for the signal to change value; that is, the scheduled new value must persist for at least the time period specified by the delay in order actually to be attained by the target signal. *Transport delay*, on the other hand, must be explicitly indicated in the signal assignment statement with the reserved word `TRANSPORT`, and models a "wire delay" wherein any pulse of whatever duration is propagated to the target signal after the specified delay.

8 WAIT in IF

8.1 General Description

The Stage 1 VHDL description for this example, displayed in Section 8.2, is assumed to reside in a file wait_in_if.vhdl.

The example illustrates the translation and proof of a Stage 1 VHDL description involving WAIT statements embedded in an IF statement, a possibility disallowed in Core VHDL.

The description itself increments the input ports x and y by 2 and 4, respectively, and is intended simply to exercise the occurrences of the embedded WAITs.

8.2 VHDL Code

```
PROGRAM wait_in_if IS

ENTITY wait_in_if IS
  PORT ( SIGNAL x, y: INOUT INTEGER );
END wait_in_if;


ARCHITECTURE behavior OF wait_in_if IS

  SIGNAL a : INTEGER := 4;

BEGIN

  PROCESS

  BEGIN

    IF a /= 0 THEN

      IF a MOD 2 = 0 THEN
        x <= x + 1;
        a <= a - 1;
        WAIT FOR 10 NS;
        y <= y + 2;

      ELSE
        a <= a - 1;
        WAIT FOR 5 NS;

      END IF;
```

```
        ELSE
          WAIT;

        END IF;

      END PROCESS;

    END Behavior;
```

8.3 State Delta Specification

The state delta wait_in_if.sd below, our specification for this example, is assumed to reside in a file wait_in_if.spec.

It expresses the claim:

> At any point at which the translation of wait_in_if.vhdl holds, there will be a later point at which the model will have been elaborated, such that at some later point, **x** will have been incremented by 2, **y** will have been incremented by 4, and the model will have completed execution.

```
wait_in_if.sd =

  [sd pre: (vhdl(wait_in_if.vhdl))
    comod:
      mod: (all)
     post: (vhdl_model_elaboration_complete(wait_in_if),
            [sd pre: (true)
              comod: (all)
                mod: (all)
               post: (#x = .x + 2,
                      #y = .y + 4,
                      vhdl_model_execution_complete(wait_in_if))])]
```

8.4 Batch Proof

A straightline symbolic execution proof is all that is needed for this example.

```
(defproof wait_in_if.proof
   "(setflag autoclose off,
     vhdltr \"testproofs/vhdl/wait_in_if.vhdl\",
     read \"testproofs/vhdl/wait_in_if.spec\",
     prove wait_in_if.sd
       proof:
         (go vhdl_model_elaboration_complete(wait_in_if),
          prove g(2)
            proof:
              (go vhdl_model_execution_complete(wait_in_if),
               close),
          close))")
```

9 WAIT in WHILE

9.1 General Description

The Stage 1 VHDL description for this example, displayed in Section 9.2, is assumed to reside in a file `wait_in_while.vhdl`.

The example illustrates the translation and proof of a Stage 1 VHDL description involving `WAIT` statements embedded in a `WHILE` statement. No loop statements of any kind — nor, perforce, `WAIT` statements embedded in loops — were allowed in Core VHDL.

The description itself is a behavioral model of an integer multiplier.

9.2 VHDL Code

```
PROGRAM wait_in_while IS

ENTITY wait_in_while IS
  PORT ( SIGNAL x : IN INTEGER := 2 ;
         SIGNAL m : IN INTEGER := 3 ;
         SIGNAL p: OUT INTEGER := 0 );
END wait_in_while;


ARCHITECTURE behavior OF wait_in_while IS
BEGIN

  mult : PROCESS
    VARIABLE i : INTEGER := 0;
  BEGIN

    WHILE (i < ABS(m)) LOOP
      p <= p + x;
      i := i + 1;
      WAIT FOR 10 NS;
    END LOOP;

    IF m < 0 THEN
      p <= - p;
    END IF;

    WAIT ON x, m;

  END PROCESS mult;

END behavior;
```

9.3 State Delta Specification

The state delta `wait_in_while.sd` below, our specification for this example, is assumed to reside in a file `wait_in_while.spec`.

It expresses the claim:

> At any point at which the translation of `wait_in_while.vhdl` holds, there will be a later point at which the model will have been elaborated, such that at some later point, the value of p will be the product of the input values of m and x, and the model will have completed execution.

```
wait_in_while.sd =

  [sd pre: (vhdl(wait_in_while.vhdl))
    comod:
      mod: (all)
     post: (vhdl_model_elaboration_complete(wait_in_while),
            [sd pre: (true)
              comod: (all)
                mod: (all)
               post: (#p = .m * .x,
                      vhdl_model_execution_complete(wait_in_while))])]
```

9.4 Batch Proof

A straightline symbolic execution proof is all that is needed for this example.

```
(defproof wait_in_while.proof
   "(setflag autoclose off,
     vhdltr \"testproofs/vhdl/wait_in_while.vhdl\",
     read \"testproofs/vhdl/wait_in_while.spec\",
     prove wait_in_while.sd
       proof:
         (go vhdl_model_elaboration_complete(wait_in_while),
          prove g(2)
            proof:
              (go vhdl_model_execution_complete(wait_in_while),
               close),
          close))")
```

9.5 Discussion

The input ports `x` and `m` in the above description are supplied with concrete values, which is what makes the proof by straightline execution possible. Symbolic values for the input ports would necessitate an induction over the loop; currently, we do not know how to perform such an induction in the presence of the embedded `WAIT` statement.

The difficulty is that we do not yet know how to perform inductions over simulation cycles of a VHDL description, partly because SDVS cannot yet reason about symbolic representations of VHDL time, and partly because it is not yet clear how to state invariants that must hold at the end of each simulation cycle. Attempting to induct over a loop with an embedded `WAIT` statement, however, would immediately entail just such an induction over simulation cycles: the "step" case of the induction would assume that a certain (unspecified) number of loop executions had been performed, hence a certain number of executions of the `WAIT` statement, and hence — since a `WAIT` suspends the parent process until a later simulation cycle — a certain number of simulation cycles for the whole description.

This is an issue for further research relative to the incorporation of VHDL into SDVS.

10 EXIT from Outer Loop

10.1 General Description

The Stage 1 VHDL description for this example, displayed in Section 10.2, is assumed to reside in a file `exit_outer_loop.vhdl`.

The example illustrates the translation and proof of a Stage 1 VHDL description involving an `EXIT` statement in a nested `WHILE` loop. No loop statements of any kind — nor, perforce, `EXIT`s from loops — were allowed in Core VHDL.

In the description itself, the `EXIT` request (made from the inner loop) is to exit the outer loop, which then results in the suspension of the process and the assignment of '1' to the zero-th order bit of the bitvector `v`. Because of its sensitivity to changes in `v`, the process executes a second time, whereupon the zero-th order bit of `v` is reset to '0'.

10.2 VHDL Code

```
PROGRAM exit_outer_loop IS

ENTITY exit_outer_loop IS

  PORT ( SIGNAL v : INOUT BIT_VECTOR(2 DOWNTO 0) := "000" );

END exit_outer_loop;


ARCHITECTURE behavior OF exit_outer_loop IS

BEGIN

  set_bit_0 : PROCESS (v)
    VARIABLE i, j : INTEGER;
  BEGIN

    i := 0;

    i_loop:
    WHILE (i < 3) LOOP

      j := 0;

      j_loop:
      WHILE (j <= 10) LOOP

        EXIT i_loop WHEN j > i;
```

```
            IF (j = i)
              THEN v(j) <= NOT(v(j));
            END IF;

            j := j + 1;

          END LOOP j_loop;

          i := i + 1;

        END LOOP i_loop;

      END PROCESS set_bit_0;

    END behavior;
```

10.3 State Delta Specification

The state delta **exit_outer_.sd** below, our specification for this example, is assumed to reside in a file **exit_outer_loop.spec**.

It expresses the claim:

> At any point at which the translation of **exit_outer_loop.vhdl** holds, there will be a later point at which the model will have been elaborated, such that at some later point, the value of bitvector **v** will be 1 in three bits, and at yet a later point, it will be 0 in 3 bits.

```
exit_outer_loop.sd =

  [sd pre: (vhdl(exit_outer_loop.vhdl))
    comod:
      mod: (all)
     post: (vhdl_model_elaboration_complete(exit_outer_loop),
            [sd pre: (true)
              comod: (all)
                mod: (all)
               post: (#v = 1(3),
                      [sd pre: (true)
                        comod: (all)
                          mod: (all)
                         post: (#v = 0(3))])])]
```

10.4 Batch Proof

A straightline symbolic execution proof is all that is needed for this example.

```
(defproof exit_outer_loop_test.proof
   "(setflag autoclose off,
     vhdltr \"testproofs/vhdl/exit_outer_loop.vhdl\",
     read \"testproofs/vhdl/exit_outer_loop.spec\",
     prove exit_outer_loop_test.sd
       proof:
         (go vhdl_model_elaboration_complete(exit_outer_loop),
          prove g(2)
            proof:
              (go #v = 1(3),
               prove g(3)
                 proof:
                   (go #v = 0(3),
                    close),
               close),
          close))")
```

11 Shift-and-Add Multiplier

11.1 General Description

The Stage 1 VHDL description for this example, displayed in Section 11.2, is assumed to reside in a file `shift_and_add_multiplier.vhdl`. Note that the syntax of Stage 1 VHDL differs slightly from that of IEEE Standard 1076-1987 VHDL [4] in the representation of `BIT_VECTOR` constants as strings (of bit characters). In a fully legal implementation of the language, we would need to write, e.g., B"00000000" rather than "00000000".[2]

This description serves as a low-level behavioral implementation for the design modeled by the description in Section 14 of [5]: a device for computing the product of two integer inputs — a nonnegative "multiplier" `m` and a "multiplicand" `x` — returning the result through an integer output port, `p`.

In the description of [5], the product was computed as the sum of the products of `x` with the terms in the binary expansion of `m`:

```
p = x * [(m mod 2) + ((m / 2) mod 2) * 2
                   + ((m / 4) mod 2) * 4
                   + (m / 8) * 8]

  = (m mod 2) * x + ((m / 2) mod 2) * 2 * x
                  + ((m / 4) mod 2) * 4 * x
                  + (m / 8) * 8 * x
```

The algorithm embodied in the VHDL description below is identical, with the only difference being in the implementation of the multiplier `m` and the multiplicand `x` as bitvectors of length 4 (corresponding to integers in the range 0 to 15). The appropriate portions of VHDL code are modified to describe actual shifts (right and left) of bits, previously simulated by division and multiplication by 2.

We gratefully acknowledge Professor Richard Auletta of George Mason University for providing us with the VHDL code for this example, and for his collaboration on the proof [17]. Together, we defined the overall proof structure shown in Section 11.4. Professor Auletta proposed some integer and bitstring lemmas necessary for static portions of the proof, shown in Sections 11.5 and 11.6. Assuming the truth of these mostly unproved lemmas, the main proof was able to close.

With one exception, we have proved the required lemmas, as well as a number of subordinate static facts. As discussed in Section 11.7, however, the original statement of one key lemma (`xn`) was erroneous. While our effort to patch the resulting hole in the main proof has solved the problem in principle, an actual implementation of a working proof still needs to be carried out.

In our discussion, we shall concentrate on clarifying the various issues encountered in our attempt to complete this verification example.

[2]This has been rectified in Stage 2 VHDL [8].

11.2 VHDL Code

```
PROGRAM shift_and_add_multiplier IS


ENTITY shift_and_add_multiplier IS

  PORT ( SIGNAL x : IN BIT_VECTOR(3 DOWNTO 0);
         SIGNAL m : IN BIT_VECTOR(3 DOWNTO 0);
         SIGNAL p : BUFFER BIT_VECTOR(7 DOWNTO 0)
       );

END shift_and_add_multiplier;


ARCHITECTURE behavior OF shift_and_add_multiplier IS

  -- signals to act like a shift register
  SIGNAL sr8 : BIT_VECTOR(7 DOWNTO 0);
  SIGNAL s   : BIT_VECTOR(7 DOWNTO 0);

  -- clock signals
  SIGNAL ph1   : BIT := '0';
  SIGNAL add   : BIT := '0';
  SIGNAL reset : BIT := '1';

BEGIN


  sr_4 : PROCESS

    VARIABLE sr4 : BIT_VECTOR(3 DOWNTO 0) := "0000";

  BEGIN

    WAIT UNTIL ph1 = '1';

    IF reset = '1' THEN       -- load
      sr4 := m;
    ELSE                      -- shift right
      sr4(0) := sr4(1);
      sr4(1) := sr4(2);
      sr4(2) := sr4(3);
      sr4(3) := '0';
    END IF;
  add <= sr4(0);            -- serial output the shifted value

END PROCESS sr_4;


sr_8 : PROCESS

  VARIABLE tmp : BIT_VECTOR(7 DOWNTO 0);

BEGIN

  WAIT UNTIL ph1 = '1';

  tmp := sr8;

  IF reset = '1' THEN       -- load
    tmp(0) := x(0);
    tmp(1) := x(1);
    tmp(2) := x(2);
    tmp(3) := x(3);
    tmp(4) := '0';
    tmp(5) := '0';
    tmp(6) := '0';
    tmp(7) := '0';
  ELSE                      -- shift left
    tmp(7) := tmp(6);
    tmp(6) := tmp(5);
    tmp(5) := tmp(4);
    tmp(4) := tmp(3);
    tmp(3) := tmp(2);
    tmp(2) := tmp(1);
    tmp(1) := tmp(0);
    tmp(0) := '0';
  END IF;

  sr8 <= tmp;                 -- parallel output the shifted value

END PROCESS sr_8;


latch_8 : PROCESS

BEGIN

  WAIT UNTIL ph1 = '1';
```

```
    IF reset = '1' THEN             -- clear the data
      p <= "00000000";              -- parallel output the latched value
    ELSIF add = '1' THEN            -- latch the data
      p <= s;                       -- parallel output the latched value
    END IF;

  END PROCESS latch_8;


  adder_8 : PROCESS (p, sr8)

    VARIABLE sum  : BIT_VECTOR(7 DOWNTO 0);
    VARIABLE cout : BIT_VECTOR(7 DOWNTO 0);

  BEGIN

    sum(0)  := p(0) XOR  sr8(0);
    cout(0) := sr8(0) AND p(0);

    sum(1)  := p(1) XOR sr8(1) XOR cout(0);
    cout(1) := (p(1) AND sr8(1)) OR (p(1) AND cout(0)) OR (sr8(1) AND cout(0));

    sum(2)  := p(2) XOR sr8(2) XOR cout(1);
    cout(2) := (p(2) AND sr8(2)) OR (p(2) AND cout(1)) OR (sr8(2) AND cout(1));

    sum(3)  := p(3) XOR sr8(3) XOR cout(2);
    cout(3) := (p(3) AND sr8(3)) OR (p(3) AND cout(2)) OR (sr8(3) AND cout(2));

    sum(4)  := p(4) XOR sr8(4) XOR cout(3);
    cout(4) := (p(4) AND sr8(4)) OR (p(4) AND cout(3)) OR (sr8(4) AND cout(3));

    sum(5)  := p(5) XOR sr8(5) XOR cout(4);
    cout(5) := (p(5) AND sr8(5)) OR (p(5) AND cout(4)) OR (sr8(5) AND cout(4));

    sum(6)  := p(6) XOR sr8(6) XOR cout(5);
    cout(6) := (p(6) AND sr8(6)) OR (p(6) AND cout(5)) OR (sr8(6) AND cout(5));

    sum(7)  := p(7) XOR sr8(7) XOR cout(6);
    cout(7) := (p(7) AND sr8(7)) OR (p(7) AND cout(6)) OR (sr8(7) AND cout(6));

    s <= sum;

  END PROCESS adder_8;
```

```
    clock : PROCESS

    BEGIN

      ph1 <= NOT ph1 AFTER 40 NS;      -- '1' -> '0' clock tick
      reset <= '0' AFTER 80 NS;
      WAIT ON ph1;
      ph1 <= NOT ph1 AFTER 40 NS;
      WAIT ON ph1;

      ph1 <= NOT ph1 AFTER 40 NS;      -- '1' -> '0' clock tick
      WAIT ON ph1;
      ph1 <= NOT ph1 AFTER 40 NS;
      WAIT ON ph1;

      ph1 <= NOT ph1 AFTER 40 NS;      -- '1' -> '0' clock tick
      WAIT ON ph1;
      ph1 <= NOT ph1 AFTER 40 NS;
      WAIT ON ph1;

      ph1 <= NOT ph1 AFTER 40 NS;      -- '1' -> '0' clock tick
      WAIT ON ph1;
      ph1 <= NOT ph1 AFTER 40 NS;
      WAIT ON ph1;

      ph1 <= NOT ph1 AFTER 40 NS;      -- '1' -> '0' clock tick
      WAIT ON ph1;
      ph1 <= NOT ph1 AFTER 40 NS;
      WAIT ON ph1;

      WAIT;

    END PROCESS clock;


  END behavior;
```

11.3 State Delta Specification

The state delta `shift_and_add_multiplier.sd` below, our specification for this example, is assumed to reside in a file `shift_and_add_multiplier.spec`.

It expresses the claim:

At any point at which the translation of shift_and_add_multiplier.vhdl holds, there will be a later point at which the model will have been elaborated, such that at some later point the integer value of the output bitvector p will be the product of the integer values of the input bitvectors m and x; furthermore, at this point the model will be done executing.

```
shift_and_add_multiplier.sd =

[sd pre: (vhdl(shift_and_add_multiplier.vhdl))
  comod:
    mod: (all)
   post: (vhdl_model_elaboration_complete(shift_and_add_multiplier),
          [sd pre: (|.m| ge 0,
                    |.m| le 15,
                    |.x| ge 0,
                    |.x| le 15)
            comod: (all)
              mod: (all)
             post: (|#p| =  |.m| * |.x|,
                    vhdl_model_execution_complete(shift_and_add_multiplier))])]
```

11.4 Batch Proof

```
(defproof shift_and_add_multiplier.proof
   "(setflag autoclose off,,
     vhdltr \"testproofs/vhdl/shift_and_add_multiplier.vhdl\",
     read \"testproofs/vhdl/shift_and_add_multiplier.spec\",
     read \"testproofs/vhdl/shift_and_add_multiplier.integerlemmas\",
     read \"testproofs/vhdl/shift_and_add_multiplier.bitstringlemmas\",
     activate m,
     prove shift_and_add_multiplier.sd
       proof:
         (go vhdl_model_elaboration_complete(shift_and_add_multiplier),
          prove g(2)
            proof:
              (interpret proof.static,
               go,
               cases .p = val(.driver\\p,.vhdltime_previous)
                 then proof:
                   (go,
                    cases .sr8 = val(.driver\\sr8,.vhdltime_previous)
                      then proof:
                        (go,
                         interpret bitadd,
                         cases .add = 1(1)
                      then proof:
                        interpret proof.intermediate
                      else proof:
                        interpret proof.intermediate)
                 else proof:
                   (go,
                    interpret bitadd,
                    cases .add = 1(1)
                      then proof:
                        interpret proof.intermediate
                      else proof:
                        interpret proof.intermediate))
            else proof:
              (go,
               interpret bitadd,
               cases .add = 1(1)
                 then proof:
                   (go,
                    cases .p = val(.driver\\p,.vhdltime_previous)
                      then proof:
                        interpret proof.intermediate
                      else proof:
                        interpret proof.intermediate)
                 else proof:
                   (go,
                    interpret bitadd,
                    cases .sr8 = val(.driver\\sr8,.vhdltime_previous)
                      then proof:
                        interpret proof.intermediate
                      else proof:
                        interpret proof.intermediate))),
       date,
       close))")


(defproof proof.static
   "(readaxioms \"axioms/div.axioms\",
     provebyaxiom (|.m| / 2 ^ 1) / 2 = |.m| / 2 ^ (1 + 1)
       using: divby2repeat,
     notice (|.m| / 2) / 2 = |.m| / 4,
     provebyaxiom (|.m| / 2 ^ 2) / 2 = |.m| / 2 ^ (2 + 1)
       using: divby2repeat,
     notice (|.m| / 4) / 2 = |.m| / 8,
     provebylemma (|.m| / (2 ^ (4 - 1))) mod 2 = |.m| / (2 ^ (4 - 1))
       using: maxfactors,
     notice (|.m| / 8) mod 2 = |.m| / 8,
provebylemma |.m| mod 2 ge 0 & |.m| mod 2 lt 2
  using: modposrange,
provebylemma (|.m| / 2) mod 2 ge 0 & (|.m| / 2) mod 2 lt 2
  using: modposrange,
provebylemma (|.m| / 4) mod 2 ge 0 & (|.m| / 4) mod 2 lt 2
  using: modposrange,
provebylemma (|.m| / 8) mod 2 ge 0 & (|.m| / 8) mod 2 lt 2
  using: modposrange,
provebylemma |.m| = (|.m| / 2) * 2 + |.m| mod 2
  using: remdef_with_mod,
provebylemma |.m| / 2 = ((|.m| / 2) / 2) * 2 + (|.m| / 2) mod 2
  using: remdef_with_mod,
notice |.m| / 2 = (|.m| / 4) * 2 + (|.m| / 2) mod 2,
notice |.m| = ((|.m| / 4) * 2 + (|.m| / 2) mod 2) * 2 + |.m| mod 2,
provebylemma ((|.m| / 4) * 2 + (|.m| / 2) mod 2) * 2 =
             (|.m| / 4) * 2 * 2 + ((|.m| / 2) mod 2) * 2
  using: multdistributeplus_right,
notice |.m| = ((|.m| / 4) * 4 + ((|.m| / 2) mod 2) * 2) + |.m| mod 2,
provebylemma |.m| / 4 = ((|.m| / 4) / 2) * 2 + (|.m| / 4) mod 2
  using: remdef_with_mod,
notice |.m| / 4 = (|.m| / 8) * 2 + (|.m| / 4) mod 2,
notice |.m| = (((|.m| / 8) * 2 + (|.m| / 4) mod 2) * 4 +
               ((|.m| / 2) mod 2) * 2) +
              |.m| mod 2,
provebylemma ((|.m| / 8) * 2 + (|.m| / 4) mod 2) * 4 =
             (|.m| / 8) * 2 * 4 + ((|.m| / 4) mod 2) * 4
  using: multdistributeplus_right,
notice |.m| = (((|.m| / 8) * 8 +
                ((|.m| / 4) mod 2) * 4) +
               ((|.m| / 2) mod 2) * 2) +
              |.m| mod 2,
provebylemma ((((|.m| / 8) * 8 +
                ((|.m| / 4) mod 2) * 4) +
               ((|.m| / 2) mod 2) * 2) +
              |.m| mod 2) * |.x|
             =
             (((|.m| / 8) * 8 +
               ((|.m| / 4) mod 2) * 4) +
              ((|.m| / 2) mod 2) * 2) * |.x| +
             (|.m| mod 2) * |.x|
  using: multdistributeplus_right,
provebylemma (((|.m| / 8) * 8 +
               ((|.m| / 4) mod 2) * 4) +
              ((|.m| / 2) mod 2) * 2) * |.x|
             =
             ((|.m| / 8) * 8 + ((|.m| / 4) mod 2) * 4) * |.x| +
                    (((|.m| / 2) mod 2) * 2) * |.x|
          using: multdistributeplus_right,
        provebylemma ((|.m| / 8) * 8 + ((|.m| / 4) mod 2) * 4) * |.x|
                    =
                    ((|.m| / 8) * 8) * |.x| + (((|.m| / 4) mod 2) * 4) * |.x|
          using: multdistributeplus_right,
        notice |.m| * |.x| = (|.m| / 8) * 8 * |.x| +
                            ((|.m| / 4) mod 2) * 4 * |.x| +
                            ((|.m| / 2) mod 2) * 2 * |.x| +
                            (|.m| mod 2) * |.x|,
        provebylemma |.m<0:0>| = (|.m| / 2 ^ 0) mod 2
          using: bitmod,
        provebylemma |.m<1:1>| = (|.m| / 2 ^ 1) mod 2
          using: bitmod,
        provebylemma |.m<2:2>| = (|.m| / 2 ^ 2) mod 2
          using: bitmod,
        provebylemma |.m<3:3>| = (|.m| / 2 ^ 3) mod 2
          using: bitmod)")


(defproof proof.intermediate
   "(go,
     interpret bitadd,
     cases .add = 1(1)
       then proof:
         (go,
          interpret bitadd,
          cases .add = 1(1)
            then proof:
              interpret proof.simple
            else proof:
              interpret proof.simple)
       else proof:
         (go,
          interpret bitadd,
          cases .add = 1(1)
            then proof:
              interpret proof.simple
            else proof:
              interpret proof.simple))")
```

```
(defproof proof.simple
   "(go,
     interpret bitadd,
     cases .add = 1(1)
       then proof: (go, close)
       else proof: (go, close))")


(defproof bitadd
   "(provebylemma (.p ++ .sr8)<0:0> = .p<0:0> usxor .sr8<0:0>
       using: x0,
     provebylemma (.p ++ .sr8)<1:1> =
                  (.p<1:1> usxor .sr8<1:1>) usxor
                  .sr8<1 - 1:1 - 1> && .p<1 - 1:1 - 1>
       using: x1,
     provebylemma (.p ++ .sr8)<2:2> =
                  (.p<2:2> usxor .sr8<2:2>) usxor .cout<2 - 1:2 - 1>
       using: xn,
     provebylemma (.p ++ .sr8)<3:3> =
                  (.p<3:3> usxor .sr8<3:3>) usxor .cout<3 - 1:3 - 1>
       using: xn,
     provebylemma (.p ++ .sr8)<4:4> =
                  (.p<4:4> usxor .sr8<4:4>) usxor .cout<4 - 1:4 - 1>
       using: xn,
     provebylemma (.p ++ .sr8)<5:5> =
                  (.p<5:5> usxor .sr8<5:5>) usxor .cout<5 - 1:5 - 1>
       using: xn,
     provebylemma (.p ++ .sr8)<6:6> =
                  (.p<6:6> usxor .sr8<6:6>) usxor .cout<6 - 1:6 - 1>
       using: xn,
     provebylemma (.p ++ .sr8)<7:7> =
                  (.p<7:7> usxor .sr8<7:7>) usxor .cout<7 - 1:7 - 1>
       using: xn,
     provebylemma .sr8[1] usxor .p[1] = .p[1] usxor .sr8[1]
       using: usxor\\commute,
     provebylemma .sr8[2] usxor .p[2] = .p[2] usxor .sr8[2]
       using: usxor\\commute,
     provebylemma .sr8[3] usxor .p[3] = .p[3] usxor .sr8[3]
       using: usxor\\commute,
     provebylemma .sr8[4] usxor .p[4] = .p[4] usxor .sr8[4]
       using: usxor\\commute,
     provebylemma .sr8[5] usxor .p[5] = .p[5] usxor .sr8[5]
       using: usxor\\commute,
     provebylemma .sr8[6] usxor .p[6] = .p[6] usxor .sr8[6]
       using: usxor\\commute,
    provebylemma .sr8[7] usxor .p[7] = .p[7] usxor .sr8[7]
      using: usxor\\commute,
    provebylemma |(.p ++ .sr8)<7:0>| =
                 |(.p ++ .sr8)<7:7> @ (.p ++ .sr8)<6:6> @
                  (.p ++ .sr8)<5:5> @ (.p ++ .sr8)<4:4> @
                  (.p ++ .sr8)<3:3> @ (.p ++ .sr8)<2:2> @
                  (.p ++ .sr8)<1:1> @ (.p ++ .sr8)<0:0>|
      using: add8)")
```

11.5 Integer Lemmas

```
;*****************************;
;*  Top-level Integer Lemmas  *;
;*****************************;



(deflemma maxfactors
   "x ge 0  &  y gt 0  &  n ge 1  &  x lt y ^ n
      -->  (x / (y ^ (n - 1))) mod y = x / (y ^ (n - 1))"
   (x y n) nil nil nil
   :proof
   "(provelemma maxfactors
       proof:
         (provebyaxiom x / y ^ (n - 1)
                            = ((x / y ^ (n - 1)) / y) * y +
                                (x / y ^ (n - 1)) rem y
            using: remdef,
          provebylemma (x / y ^ (n - 1)) / y = x / y ^ n
            using: divbyylemma,
          provebyaxiom x / y ^ n = 0
            using: diveq0,
          provebyaxiom y * 0 = 0
            using: mult0,
          provebyaxiom y * 0 = 0 * y
            using: multcommute,
          provebyaxiom y ^ (n - 1) gt 0
            using: e1,
          provebyaxiom x / y ^ (n - 1) ge 0
            using: divge0,
          provebylemma (x / y ^ (n - 1)) mod y
                            = (x / y ^ (n - 1)) rem y
            using: newmodrem1))")
```

```
(deflemma modposrange
   "y gt 0 --> x mod y ge 0 & x mod y lt y"
   (x y) nil nil nil
   :proof "(readaxioms \"axioms/mod.axioms\",
            provelemma modposrange
              proof:
                (provebyaxiom x mod y ge 0
                   using: modpos,
                 provebylemma x mod y lt y
                   using: modposlt,
                 close))")


(deflemma remdef_with_mod
   "x ge 0 & y gt 0 --> x = (x / y) * y + x mod y"
   (x y) nil nil nil
   :proof "(readaxioms \"axioms/rem.axioms\",
            provelemma remdef_with_mod
              proof:
                (provebylemma x mod y = x rem y
                   using: mod_rem_eq,
                 provebyaxiom x = (x / y) * y + x rem y
                   using: remdef,
                 close))")


(deflemma multdistributeplus_right
   "(y + z) * x = y * x + z * x"
   (x y z) nil nil nil
   :proof "(activate m,
            readaxioms \"axioms/mult.axioms\",
            provelemma multdistributeplus_right
              proof:
                (notice (y + z) * x = x * (y + z),
                 notice y * x = x * y,
                 notice z * x = x * z,
                 provebyaxiom x * (y + z) = x * y + x * z
                   using: multdistributeplus,
                 close))")
```

```
;*******************************;
;*  Subordinate Integer Lemmas  *;
;*******************************;


;; For positive y, non-negative x, and n greater than 1,
;; the quotient of x and y to the (n-1)th power divided by y
;; is equal to the quotient of x and y to the nth power.
;;
;; Used in proof of lemma: maxfactors

(deflemma divbyylemma
   "(n ge 1 & y gt 0) & x ge 0
       --> (x / y ^ (n - 1)) / y = x / y ^ n"
   (x y n) nil nil nil
   :proof "(provelemma divbyylemma
              proof:
                (provebyaxiom y ^ 1 * y ^ (n - 1) = y ^ (1 + (n - 1))
                   using: e3,
                 provebylemma y ^ 1 = y
                   using: tothezerothlemma,
                 provebyaxiom y ^ (n - 1) gt 0
                   using: e1,
                 provebylemma (x / y ^ (n - 1)) / y
                                = x / (y ^ (n - 1) * y)
                   using: divdivlemma))")


;; y raised to the first power is y.
;;
;; Used in proof of lemma: divbyylemma

(deflemma tothezerothlemma
   "y ^ 1 = y"
   (y) nil nil nil
   :proof "(provelemma tothezerothlemma
              proof:
                cases y = 0
                  then proof:
                  else proof:
                    (provebyaxiom y ^ 1 = y * y ^ (1 - 1)
                       using: expmult,
                     provebyaxiom y ^ 0 = 1
                       using: e4))")
```

```
;; For b,c positive and a non-negative,
;; division a by b and then by c is equal to dividing a by (b*c).
;;
;; Used in proof of lemma: divbyylemma
;;
;; N.B.: this proof may have to be modified as a bug
;;       in the axiom divdist1 was discovered after it was proved.

(deflemma divdivlemma
   "(a ge 0 & b gt 0) & c gt 0 --> (a / b) / c = a / (b * c)"
   (a b c) nil nil nil
   :proof "(provelemma divdivlemma
              proof:
                (provebyaxiom a = (a / b) * b + a rem b
                   using: remdef,
                 provebyaxiom a / b
                                 = ((a / b) / c) * c + (a / b) rem c
                   using: remdef,
                 provebyaxiom b *
                                (((a / b) / c) * c + (a / b) rem c)
                         = b * (((a / b) / c) * c) +
                                    b * ((a / b) rem c)
                   using: multdistributeplus,
                 provebylemma ((a / b) rem c) * b + a rem b lt b * c
                   using: remlesslemma,
                 provebyaxiom b * c gt 0
                   using: multgt0,
                 provebyaxiom a / b ge 0
                   using: divge0,
                 provebyaxiom (a / b) / c ge 0
                   using: divge0,
                 provebyaxiom ((b * c) * ((a / b) / c) +
                                 (b * ((a / b) rem c) + a rem b)) / (b * c)
                                 = (a / b) / c
                   using: divdist1))")


;; Used in proof of lemma: divdivlemma

(deflemma remlesslemma
   "(a ge 0 & b gt 0) & c gt 0
       --> ((a / b) rem c) * b + a rem b lt b * c"
   (a b c) nil nil nil
   :proof "(provelemma remlesslemma
              proof:
```

```
cases a = 0
  then proof:
    (provebyaxiom a / b = 0
       using: diveq0,
     provebyaxiom (a / b) rem c = 0
       using: rem0,
     provebyaxiom a rem c = 0
       using: rem0,
     provebyaxiom 0 rem b = 0
       using: rem0,
     provebyaxiom b * c gt 0
       using: multgt0)
  else proof:
    (provebyaxiom abs(a rem b) = a rem b
       using: rempos,
     provebylemma a rem b ge 0
       using: absxeqxlemma,
     provebyaxiom a / b ge 0
       using: divge0,
     cases a / b = 0
       then proof:
         (provebyaxiom 0 rem c = 0
            using: rem0,
          provebyaxiom abs(a rem b) lt abs(b)
            using: remub,
          read \"axioms/mult.axioms\",
          provebyaxiom b * c ge b * 1
            using: multge)
       else proof:
         (provebyaxiom abs((a / b) rem c) = (a / b) rem c
            using: rempos,
          provebylemma (a / b) rem c ge 0
            using: absxeqxlemma,
          provebyaxiom abs(a rem b) lt abs(b)
            using: remub,
          provebyaxiom abs((a / b) rem c) = (a / b) rem c
            using: rempos,
          provebylemma (a / b) rem c ge 0
            using: absxeqxlemma,
          provebyaxiom abs((a / b) rem c) lt abs(c)
            using: remub,
          provebyaxiom b * (c - 1) ge b * ((a / b) rem c)
            using: multge,
          provebyaxiom b * (c - 1)
                         = b * c - b * 1
            using: multdistributeminus)))")
```

```
;; For nonnegative y:  x mod y lt y.
;;
;; Used in proof of lemma: modposrange

(deflemma modposlt
   "y gt 0 --> x mod y lt y"
   (x y) nil nil nil
   :proof
   "(read \"axioms/rem.axioms\",
     read \"axioms/mod.axioms\",
     provelemma modposlt
       proof:
         cases x = 0
           then proof: provebyaxiom x mod y = 0
                          using: mod0
           else proof:
             cases (x / y) * y = x or x gt 0
               then proof:
                 (provebylemma x mod y = x rem y
                    using: newmodrem1,
                  provebyaxiom abs(x rem y) lt abs(y)
                    using: remub,
                  provebylemma abs(x mod y) ge x mod y
                    using: absgelemma)
               else proof:
                 (provebylemma abs(x rem y) = abs(y) - abs(x mod y)
                    using: newmodrem2,
                  provebyaxiom abs(x rem y) ge 0
                    using: remlb,
                  provebyaxiom x = (x / y) * y + x rem y
                    using: remdef,
                  provebylemma abs(x rem y) ~= 0
                    using: absne0lemma,
                  provebylemma abs(x mod y) ge x mod y
                    using: absgelemma))")


;; Used in proof of lemma: remdef_with_mod

(deflemma mod_rem_eq
   "x ge 0 & y gt 0 --> x mod y = x rem y"
   (x y) nil nil nil
   :proof "(readaxioms \"axioms/mod.axioms\",
            readaxioms \"axioms/rem.axioms\",
            provelemma mod_rem_eq
              proof:
                (cases x le 0
                   then proof:
                     (notice x = 0,
                      provebyaxiom x mod y = 0
                        using: mod0,
                      provebyaxiom x rem y = 0
                        using: rem0,
                      close)
                   else proof:
                     (provebyaxiom x mod y = x rem y
                        using: modrem1,
                      close),
                 close))")


;; Various facts about absolute values.


;; The absolute value of x is greater than x.
;;
:: Used in proof of lemma: modposlt.

(deflemma absgelemma
   "abs(x) ge x"
   (x) nil nil nil
   :proof "(provelemma absgelemma
              proof:
                (cases x ge 0
                   then proof:
                   else proof: ))")


;; x not equal to 0 implies the absolute value of x is not equal to 0.
;;
;; Used in proof of lemma: modposlt

(deflemma absne0lemma
   "x ~= 0 --> abs(x) ~= 0"
   (x) nil nil nil
   :proof "(provelemma absne0lemma
              proof:
                (cases x gt 0
                   then proof:
                   else proof: ))")
```

```
;; If the absolute value of x is equal to x, then x is non-negative.
;;
;; Used in proof of lemma: remlesslemma

(deflemma absxeqxlemma
   "abs(x) = x --> x ge 0"
   (x) nil nil nil
   :proof "(provelemma absxeqxlemma
              proof:
                (read \"axioms/abs.axioms\",
                 cases x ge 0
                   then proof:
                   else proof: provebyaxiom abs(x) = -x))")



;******************************;
;*  Proposed Axiom Revisions  *;
;******************************;


;; Proposed revision for existing axiom: modrem1 in bitstring.axioms
;; Fixes bug in cases where x is a multiple of y.
;;
;; Used in proofs of lemmas: maxfactors and modposlt.

(deflemma newmodrem1
   "((x / y) * y = x or 0 gt x & 0 gt y) or x gt 0 & y gt 0
       --> x mod y = x rem y"
   (x y) nil nil nil)


;; Proposed revision for existing axiom: modrem2 in bitstring.axioms
;; Fixes bug in cases where x is a multiple of y.
;;
;; Used in proof of lemma: modposlt

(deflemma newmodrem2
   "(x / y) * y ~= x & (0 gt x & y gt 0 or x gt 0 & 0 gt y)
       --> abs(x rem y) = abs(y) - abs(x mod y)"
  (x y) nil nil nil)
```

11.6 Bitstring Lemmas

```
;********************************;
;*  Top-level Bitstring Lemmas  *;
;********************************;


;; The value of the nth bit of m is
;; the quotient of the value of m and 2 to the nth mod 2.

(deflemma bitmod
   "|m<n:n>| = (|m| / 2 ^ n) mod 2"
   (n m) nil nil nil
   :proof
   "(provelemma bitmod
       proof:
         cases n ge lh(m)
           then proof:
             (provebyaxiom lh(m<n:n>) =
                           max(0, 1 + (min(lh(m) - 1,n) - max(0,n)))
                using: lh\\ussub,
              provebyaxiom max(0,n) ge n
                using: lemax,
              provebyaxiom n ge max(0,n)
                using: gemax,
              provebyaxiom min(lh(m) - 1,n) ge lh(m) - 1
                using: lemin,
              provebyaxiom lh(m) - 1 ge min(lh(m) - 1,n)
                using: minle,
              provebyaxiom 2 ^ lh(m) gt |m|
                using: usval\\lt\\lh,
              provebyaxiom 2 ^ n ge 2 ^ lh(m)
                using: e2,
              read \"axioms/div.axioms\",
              provebyaxiom |m| / 2 ^ n = 0
                using: diveq0)
           else proof:
             cases n lt 0
               then proof:
                 (provebyaxiom lh(m<n:n>) =
                               max(0, 1 + (min(lh(m) - 1,n) - max(0,n)))
                    using: lh\\ussub,
                  provebylemma 2 ^ n = 0
                    using: negexptlemma,
                  provebyaxiom |m| / (-0) = -(|m| / 0)
                    using: divneg2)
```

```
else proof:
  (provebylemma |m| = (2 ^ (n + 1) * |m<lh(m) - 1:n + 1>| +
                       2 ^ n * |m<n:n>|)
                      + |m<n - 1:0>|
     using: valfraglemma,
   provebyaxiom 2 ^ n * 2 ^ 1 = 2 ^ (n + 1)
     using: e3,
   provebyaxiom 2 ^ n * (2 ^ 1 * |m<lh(m) - 1:n + 1>| +
                         |m<n:n>|)
                =
                2 ^ n * (2 ^ 1 * |m<lh(m) - 1:n + 1>|) +
                2 ^ n * |m<n:n>|
     using: multdistributeplus,
   provebyaxiom 2 ^ n * (2 ^ 1 * |m<lh(m) - 1:n + 1>|) =
                (2 ^ n * 2 ^ 1) * |m<lh(m) - 1:n + 1>|
     using: multassoc,
   let front = |m<lh(m) - 1:n + 1>|,
   let back = |m<n - 1:0>|,
   let nthbit = |m<n:n>|,
   cases n = 0
     then proof:
       (read \"axioms/mod.axioms\",
        read \"axioms/rem.axioms\",
        provebylemma |m<n:n>| = 0 or |m<n:n>| = 1
          using: bitvaluecases,
        cases nthbit = 0
          then proof:
            provebyaxiom 0 mod 2 = (0 + front * 2) mod 2
              using: modmult
          else proof:
            cases nthbit = 1
              then proof:
                provebyaxiom 1 mod 2 = (1 + front * 2) mod 2
                  using: modmult
              else proof: )
     else proof:
       (provebyaxiom lh(m<n - 1:0>) =
                     max(0, 1 + (min(lh(m) - 1,n - 1) - max(0,0)))
          using: lh\\ussub,
        provebyaxiom 2 ^ lh(back) gt |back|
          using: usval\\lt\\lh,
        provebyaxiom 2 ^ n ge 2 ^ 0
          using: e8,
        provebyaxiom (2 ^ n * (2 * front + nthbit) + back) / 2 ^ n =
                     2 * front + nthbit
          using: divdist1,
                                   cases nthbit = 0
                                     then proof:
                                       provebyaxiom 0 mod 2 = (0 + front * 2) mod 2
                                         using: modmult
                                     else proof:
                                       cases nthbit = 1
                                         then proof:
                                           provebyaxiom 1 mod 2 = (1 + front * 2) mod 2
                                             using: modmult
                                         else proof:
                                           provebylemma |m<n:n>| = 0 or |m<n:n>| = 1)))")


;; Half-Adder Bit-Zero Lemma
;;
;; Also used in proof of lemma: x1

(deflemma x0
   "lh(x) ge 1 & lh(y) ge 1 --> (x ++ y)<0:0> = x<0:0> usxor y<0:0>"
   (x y) nil nil nil
   :proof
   "(provelemma x0
       proof:
         (provebyaxiom lh(x<0:0>)
                           = max(0, 1 + (min(lh(x) - 1,0) - max(0,0)))
            using: lh\\ussub,
          provebyaxiom lh(y<0:0>)
                           = max(0, 1 + (min(lh(y) - 1,0) - max(0,0)))
            using: lh\\ussub,
          provebyaxiom x<0:0> usxor y<0:0> = (x<0:0> ++ y<0:0>)<0:0>
            using: usxor\\usplus,
          provebyaxiom (x usxor y)<0:0> = x<0:0> usxor y<0:0>
            using: ussub\\usxor,
          provebyaxiom (x<0:0> ++ y<0:0>)<0:0> = (x ++ y)<0:0>
            using: ussub\\usplus\\ussub))")

;; Full-Adder Bit-One Lemma

(deflemma x1
   "(lh(x) gt 1 & lh(y) gt 1) & lh(x) = lh(y)
       --> (x ++ y)<1:1> = (x<1:1> usxor y<1:1>) usxor y<0:0> && x<0:0>"
   (x y) nil nil nil
   :proof "(provelemma x1
              proof:
                (rewritebylemma (x ++ y)<1:1>
                   using: addlemma,
rewritebylemma (x ++ y)<0:0>
  using: x0,
rewritebyaxiom lh(x<1:1>)
  using: lh\\ussub,
rewritebyaxiom lh(y<1:1>)
  using: lh\\ussub,
rewritebyaxiom lh(x<0:0>)
  using: lh\\ussub,
rewritebyaxiom lh(y<0:0>)
  using: lh\\ussub,
mcases
  (case: ((x<1:1> = 0(1) & y<1:1> = 0(1)) & x<0:0> = 0(1)) &
        y<0:0> = 0(1)
    proof:
   case: ((x<1:1> = 0(1) & y<1:1> = 0(1)) & x<0:0> = 0(1)) &
        y<0:0> = 1(1)
    proof:
   case: ((x<1:1> = 0(1) & y<1:1> = 0(1)) & x<0:0> = 1(1)) &
        y<0:0> = 0(1)
    proof:
   case: ((x<1:1> = 0(1) & y<1:1> = 0(1)) & x<0:0> = 1(1)) &
        y<0:0> = 1(1)
    proof:
   case: ((x<1:1> = 0(1) & y<1:1> = 1(1)) & x<0:0> = 0(1)) &
        y<0:0> = 0(1)
    proof:
   case: ((x<1:1> = 0(1) & y<1:1> = 1(1)) & x<0:0> = 0(1)) &
        y<0:0> = 1(1)
    proof:
   case: ((x<1:1> = 0(1) & y<1:1> = 1(1)) & x<0:0> = 1(1)) &
        y<0:0> = 0(1)
    proof:
   case: ((x<1:1> = 0(1) & y<1:1> = 1(1)) & x<0:0> = 1(1)) &
        y<0:0> = 1(1)
    proof:
   case: ((x<1:1> = 1(1) & y<1:1> = 0(1)) & x<0:0> = 0(1)) &
        y<0:0> = 0(1)
    proof:
   case: ((x<1:1> = 1(1) & y<1:1> = 0(1)) & x<0:0> = 0(1)) &
        y<0:0> = 1(1)
    proof:
   case: ((x<1:1> = 1(1) & y<1:1> = 0(1)) & x<0:0> = 1(1)) &
        y<0:0> = 0(1)
    proof:
   case: ((x<1:1> = 1(1) & y<1:1> = 0(1)) & x<0:0> = 1(1)) &
        y<0:0> = 1(1)
```

```
                         proof:
                        case: ((x<1:1> = 1(1) & y<1:1> = 1(1)) & x<0:0> = 0(1)) &
                             y<0:0> = 0(1)
                         proof:
                        case: ((x<1:1> = 1(1) & y<1:1> = 1(1)) & x<0:0> = 0(1)) &
                             y<0:0> = 1(1)
                         proof:
                        case: ((x<1:1> = 1(1) & y<1:1> = 1(1)) & x<0:0> = 1(1)) &
                             y<0:0> = 0(1)
                         proof:
                        case: ((x<1:1> = 1(1) & y<1:1> = 1(1)) & x<0:0> = 1(1)) &
                             y<0:0> = 1(1)
                         proof: )))")


;; Full-Adder Bit-Slice Lemma
;;
;; Original lemma is erroneous: consider x = 1(3), y = 2(3), c = 3(3), n = 2
;;
;;(deflemma xn
;;   "(lh(x) gt 1 & lh(y) gt 1) & lh(x) = lh(y) & lh(c) gt 1 & lh(x) = lh(c) &
;;                     c<n-1:n-1> = x<n - 1:n - 1> && y<n - 1:n - 1> usor
;;                                  x<n - 1:n - 1> && c<n - 2:n - 2> usor
;;                                  y<n - 1:n - 1> && c<n - 2:n - 2>
;;       --> (x ++ y)<n:n>
;;            = (x<n:n> usxor y<n:n>) usxor c<n-1:n-1>"
;;   (c x y n) nil nil nil)


;; State delta correction of lemma xn
;;
;; Suitably modified, can replace xn in shift_and_add_multiplier.proof

(defsd xn.sd
   "[sd pre: ((((lh(x) gt 1 & lh(x) = lh(y)) & lh(y) = lh(c)) & n ge 0) &
              forall n (c<n:n>
                        = (x<n:n> && y<n:n> usor
                               x<n:n> && c<n - 1:n - 1>) usor
                             y<n:n> && c<n - 1:n - 1>))
        post: ((x ++ y)<n:n>
               = (x<n:n> usxor y<n:n>) usxor c<n - 1:n - 1>)]")
```

```
;; Proof of xn.sd

(defproof xn.sd-proof
  "(prove xn.sd
     proof:
       (natural induction on: m
          formulas:    ((x ++ y)<m:m>
                           = (x<m:m> usxor y<m:m>) usxor
                               c<m - 1:m - 1>)
          base proof:  rewritebylemma (x ++ y)<0:0>
                         using: addlemma
          step proof:
            mcases
              (case: m lt 0
                proof:
               case: m = 0
                proof:
                  (rewritebyaxiom lh(c<-1:-1>)
                     using: lh\\ussub,
                   rewritebyaxiom lh(x<0:0>)
                     using: lh\\ussub,
                   rewritebyaxiom lh(y<0:0>)
                     using: lh\\ussub,
                   provebyinstantiation
                     using: q(1)
                     substitutions: (n=0),
                   rewritebylemma (x ++ y)<0:0>
                     using: addlemma,
                   rewritebylemma (x<0:0> usxor y<0:0>) usxor 0(0)
                     using: nilusxorzerolemma,
                   rewritebylemma (x ++ y)<1:1>
                     using: addlemma,
                   rewritebylemma (x<0:0> usxor y<0:0>) usxor (x ++ y)<0:0>
                     using: usxorcancellemma,
                   rewritebylemma x<0:0> && 0(1)
                     using: usandzerolemma,
                   rewritebylemma y<0:0> && 0(1)
                     using: usandzerolemma)
              case: m gt 0 & m lt lh(x)
                proof:
                  (rewritebyaxiom lh(c<m - 1:m - 1>)
                     using: lh\\ussub,
                   rewritebyaxiom lh(x<m:m>)
                     using: lh\\ussub,
                   rewritebyaxiom lh(y<m:m>)
```

```
                        using: lh\\ussub,
                      provebylemma c<m - 1:m - 1> =
                                   (x<m:m> usxor y<m:m>) usxor (x ++ y)<m:m>
                        using: shiftclemma,
                      provebyinstantiation
                        using: q(1)
                        substitutions: (n=m),
                      rewritebylemma (x ++ y)<m + 1:m + 1>
                        using: addlemma)
                  case: m ge lh(x)
                   proof:
                     (rewritebyaxiom lh((x ++ y)<m + 1:m + 1>)
                        using: lh\\ussub,
                      rewritebyaxiom lh(x<m + 1:m + 1>)
                        using: lh\\ussub,
                      rewritebyaxiom lh(y<m + 1:m + 1>)
,                        using: lh\\ussub,
                      rewritebyaxiom lh(c<m:m>)
                        using: lh\\ussub)),
          provebyinstantiation
            using: forall m (m ge 0
                        --> (x ++ y)<m:m>
                               = (x<m:m> usxor y<m:m>) usxor
                                   c<m - 1:m - 1>)
            substitutions: (m=n)))")


;; USXOR is commutative.

(deflemma usxor\\commute
   "x usxor y = y usxor x"
   (x y) nil nil nil
   :proof
   "(provelemma usxor\\commute
       proof:
         (provebylemma x usxor y
                         = x && y usor ~x && ~y
            using: usxor\\usand\\usor,
          provebylemma y usxor x
                         = y && x usor ~y && ~x
            using: usxor\\and\\or,
          provebyaxiom x && y = y && x
            using: commuteusand,
          provebyaxiom ~x && ~y = ~y && ~x
            using: commuteusand))")
```

```
; Eight-Bit Adder Lemma without Carry-Out

(deflemma add8
   "lh(x) = 8 & lh(y) = 8
       --> |(x ++ y)<7:0>|
              = |(((((((x ++ y)<7:7> @ (x ++ y)<6:6>) @
                                (x ++ y)<5:5>) @
                              (x ++ y)<4:4>) @
                            (x ++ y)<3:3>) @
                          (x ++ y)<2:2>) @
                        (x ++ y)<1:1>) @
                      (x ++ y)<0:0>|"
  (x y) nil nil nil
  :proof
     "(provelemma add8
       proof: )")


;**********************************;
;*  Subordinate Bitstring Lemmas  *;
;**********************************;

;; 2 to any negative power is 0.
;;
;; Used in proof of lemma: bitmod

(deflemma negexptlemma
   "x lt 0 --> 2 ^ x = 0"
  (x) nil nil nil
  :proof "(read \"axioms/exp.axioms\",
      provelemma negexptlemma
       proof:
          (provebyaxiom 1 gt 2 ^ x
            using: e6,
          provebylemma 2 ^ x ge 0
            using: pospowerlemma))")


;; A positive number raised to any power is non-negative.
;;
;; Used in proof of lemma: negexptlemma

(deflemma pospowerlemma
   "b gt 0 --> b ^ x ge 0"
   (b x) nil nil nil
   :proof "(provelemma pospowerlemma
              proof:
                cases x ge 0
                  then proof: provebyaxiom b ^ x gt 0
                                using: e1
                  else proof:
                    (provebyaxiom b ^ (-x) gt 0
                       using: e1,
                     provebyaxiom b ^ x * b ^ (-x) = b ^ (x - x)
                       using: e3,
                     provebyaxiom b ^ 0 = 1
                       using: e4,
                     read \"axioms/div.axioms\",
                     provebylemma (b ^ (-x) * b ^ x) / b ^ (-x) = b ^ x
                       using: improveddivmulteq,
                     provebyaxiom 1 / b ^ (-x) ge 0
                       using: divge0))")


;; Integer division by non-zero a is inverse to multiplication by a.
;;
;; Used in proof of lemma: pospowerlemma

(deflemma improveddivmulteq
   "a ~= 0 --> (a * b) / a = b"
   (a b) nil nil nil
   :proof "(provelemma improveddivmulteq
              proof:
                cases a ge 1
                  then proof: provebyaxiom (a * b) / a = b
                                using: divmulteq
                  else proof:
                    cases a le -1
                      then proof:
                        (provebyaxiom ((-a) * b) / (-a) = b
                           using: divmulteq,
                         provebyaxiom (-a) * b = -(a * b)
                           using: multminus,
                         provebyaxiom (-(a * b)) / (-a)
                                         = -((a * b) / (-a))
                           using: divneg1,
                         provebyaxiom (a * b) / (-a) = -((a * b) / a)
                           using: divneg2)
                      else proof: )")
```

```
;; The value of a bitstring in terms of the nth bit
;; and the parts of the string before and after the nth bit.
;;
;; Used in proof of lemma: bitmod

(deflemma valfraglemma
   "n ge 0 & n lt lh(x)
       --> |x| = (2 ^ (n + 1) * |x<lh(x) - 1:n + 1>| +
                  2 ^ n * |x<n:n>|) +
                 |x<n - 1:0>|"
   (x n) nil nil nil
   :proof "(provelemma valfraglemma
             proof:
               (provebyaxiom lh(x<n:0>)
                               = max(0, 1 + (min(lh(x) - 1,n) - max(0,0)))
                  using: lh\\ussub,
                provebyaxiom x<lh(x) - 1:n + 1> @ x<n:0>
                               = x<lh(x) - 1:0>
                  using: squash,
                provebyaxiom x = x<lh(x) - 1:0>
                  using: ussub\\total,
                provebyaxiom |x<lh(x) - 1:n + 1> @ x<n:0>|
                               = |x<lh(x) - 1:n + 1>| * 2 ^ lh(x<n:0>) +
                                 |x<n:0>|
                  using: usval\\usconc,
                provebyaxiom lh(x<n - 1:0>)
                               = max(0, 1 + (min(lh(x) - 1,n - 1) - max(0,0)))
                  using: lh\\ussub,
                provebyaxiom x<n:n> @ x<n - 1:0> = x<n:0>
                  using: squash))")


;; The value of a legitimate bit in a bitstring is 0 or 1.
;;
;; Used in proof of lemma: bitmod

(deflemma bitvaluecases
   "n ge 0 & n le lh(x) - 1 --> |x<n:n>| = 0 or |x<n:n>| = 1"
   (n x) nil nil nil
   :proof "(provelemma bitvaluecases
             proof:
               (provebyaxiom lh(x<n:n>)
                               = max(0, 1 + (min(lh(x) - 1,n) - max(0,n)))
                  using: lh\\ussub,
                provebylemma |x<n:n>| = 0 or |x<n:n>| = 1
                  using: zerooronelemma))")
```

```
;; Used in proof of lemma: bitvaluecases

(deflemma zerooronelemma
   "x ge 0 & x lt 2 --> x = 0 or x = 1"
   (x) nil nil nil
   :proof "(provelemma zerooronelemma
              proof:
                cases x = 0
                  then proof:
                  else proof:
                    cases x = 1
                      then proof:
                      else proof: provebylemma x = 0 or x gt 0)")


;; Strictly to force contradiction in zerooronelemma

(deflemma trivlemma
   "x ge 0 --> x = 0 or x gt 0"
   (x) nil nil nil
   :proof "(provelemma trivlemma
              proof: )")


;; Working definition of bitstring addition
;;
;; Used in: proof of lemma x1, xn.sd-proof

(deflemma addlemma
   "(x ++ y)<n:n>
       = (x<n:n> usxor y<n:n>) usxor
           ((x<n - 1:n - 1> && y<n - 1:n - 1>) usor
                (x<n - 1:n - 1> &&
                  (x<n - 1:n - 1> usxor y<n - 1:n - 1> usxor
                     (x ++ y)<n - 1:n - 1>)) usor
              (y<n - 1:n - 1> &&
                (x<n - 1:n - 1> usxor y<n - 1:n - 1> usxor
                   (x ++ y)<n - 1:n - 1>)))"
  (x y n) nil nil nil)
```

```
;; Definition of USXOR in terms of USAND, USOR, and USNEG.
;;
;; Used in proof of lemma: usxor\commute

(deflemma usxor\\usand\\usor
   "x usxor y = x && y usor ~x && ~y"
   (x y) nil nil nil)


;; Lemmas used in xn.sd-proof


(deflemma nilusxorzerolemma
   "lh(x) = 1 & lh(y) = 1 --> (x usxor y) usxor 0(0) = x usxor y"
   (x y) nil nil nil
   :proof "(provelemma nilusxorzerolemma
              proof: )")


(deflemma usxorcancellemma
   "(lh(x) = 1 & lh(y) = 1) & x usxor y = z --> (x usxor y) usxor z = 0(1)"
   (x y z) nil nil nil
   :proof "(provelemma usxorcancellemma
              proof:
                mcases
                  (case: lh(x) ~= 1
                    proof:
                   case: x = 0(1)
                    proof: mcases
                             (case: lh(y) ~= 1
                               proof:
                              case: y = 0(1)
                               proof:
                              case: y = 1(1)
                               proof: )
                   case: x = 1(1)
                    proof: mcases
                             (case: lh(y) ~= 1
                               proof:
                              case: y = 0(1)
                               proof:
                              case: y = 1(1)
                               proof: )))")
```

```
(deflemma usandzerolemma
   "lh(z) = 1 --> z && 0(1) = 0(1)"
   (z) nil nil nil
   :proof "(provelemma usandzerolemma
              proof: )")


(deflemma shiftclemma
   "((lh(x) = 1 & lh(y) = 1) & lh(z) = 1) & w = (x usxor y) usxor z
       --> z = (x usxor y) usxor w"
   (x y z w) nil nil nil
   :proof "(provelemma shiftclemma
              proof:
                mcases
                  (case: ~((lh(x) = 1 & lh(y) = 1) & lh(z) = 1)
                     proof:
                   case: (x = 0(1) & y = 0(1)) & z = 0(1)
                     proof:
                   case: (x = 0(1) & y = 1(1)) & z = 0(1)
                     proof:
                   case: (x = 0(1) & y = 1(1)) & z = 1(1)
                     proof:
                   case: (x = 1(1) & y = 0(1)) & z = 0(1)
                     proof:
                   case: (x = 1(1) & y = 1(1)) & z = 0(1)
                     proof:
                   case: (x = 0(1) & y = 0(1)) & z = 1(1)
                     proof:
                   case: (x = 1(1) & y = 0(1)) & z = 1(1)
                     proof:
                   case: (x = 1(1) & y = 1(1)) & z = 1(1)
                     proof: ))")
```

11.7 Discussion

The high-level view of the **shift_and_add_multiplier.proof** is that symbolic execution proceeds until special handling is required in either of two situations: (1) an argument by **cases** must be made on whether or not an *event* (change in value) has occurred for a signal to which a **PROCESS** statement is sensitive (determining whether or not, respectively, the process resumes execution), or (2) static facts must be verified in order for usable (true) state deltas to become applicable. In addition, certain static facts relating integer division, multiplication, and modulus are established "up front" by interpreting **proof.static**; these are subsequently available to the remainder of the overall proof. Portions of the proof that have identical structure are isolated and named separately, for invocation via the **interpret** command (a kind of "macro expansion").

Our discussion will focus on three main areas: specific issues that arose in the course of proving the lemmas required by the main proof; reasons why the present incarnation of the proof is only "almost" complete and our plans for its completion; and some general issues of a heuristic nature deriving from our work on this example.

To recall Sections 11.5 and 11.6 above, the lemmas fall into two categories: integer lemmas and bitstring lemmas. Of the integer lemmas, the lemma maxfactors was by far the hardest to prove. It states that for nonnegative x, positive y, and n at least 1, if x is less than y^n, then the y-modulus of the integral quotient of x by y^{n-1} is precisely the integral quotient itself (in effect, $x/y^{n-1} < y$). The proof is complex primarily because integer division does not enjoy the pleasant properties of ordinary (real number) division. Interestingly, the proof of the main subordinate lemma (divdivlemma) was suggested by a pictorial argument in which a fixed area is subdivided in two different ways.

The bitstring lemma bitmod states that the integer value of the nth bit of bitstring m is the 2-modulus of the quotient of the integer value of m by 2^n. Its proof, while more straightforward than that of maxfactors, is complicated by having to consider cases where the index n is "out of range" — negative or exceeding the length of m — whence m<n:n> denotes the empty bitstring rather than a "legitimate" bit. The let command was used to assign meaningful identifiers to certain terms, improving readability.

In fact, the proof of bitmod for negative n relies upon the fact that currently, it is possible for SDVS to deduce the equality |m|/0 = 0. Though surprising, this does not lead to any contradictions: within the SDVS axiom base, arithmetic facts that exclude fractions with zero denominator (e.g., a/b = c --> a = b*c) are stated in such a way that they cannot be instantiated with 0 in the denominator (for example, the above fact would be expressed by the axiom "b ne 0 implies (a/b = c implies a = b*c)").

Whether the axioms should be further modified to exclude the deducibility of |m|/0 = 0 is under consideration. However, even if the system were to be so modified, this would not have negative consequences for our example. Specifically, the way bitmod is used in the main proof relies only on "legitimate" bits, so a lemma without these cases could easily be formulated to serve the identical purpose. This suggests a possible SDVS rule-of-thumb: other things being equal, some care should be taken to state lemmas as weakly as possible for the main proof to go through, in order to avoid the superfluous effort required by the consideration of irrelevant cases.

When the bitstrings or substrings of interest are of known (finite) length, a proof by cases may be feasible. For example, in the bitstring lemma x1, all the relevant substrings are of length 1, so the lemma can be proved by a separate argument for each bit combination. In each case, the Simplifier can reduce the desired consequent to true, closing the proof automatically. It might be worthwhile to investigate the possibility of implementing a general command to handle this sort of situation.

Bitstring lemma xn states that the nth bit of the sum of two bitstrings is the exclusive-or of three bits: the nth bits of the summands and the n-1st carry bit. Unfortunately, the original version of the lemma — employed as an unproved lemma in the main proof — was incorrect: see the "deflemma xn" form (commented with semicolons) in Section 11.6. A counterexample is provided by x = 1(3), y = 2(3), c = 3(3), and n = 2.

A proper formulation of lemma **xn** would require a recursive definition, involving quantification, of the carry bitstring **c**. Because SDVS does not accept quantifiers in lemmas, however, we had to recast the corrected lemma as the state delta **xn.sd** displayed in Section 11.6.

Our intended strategy was to substitute "applys of **xn.sd**" for the "**provebylemmas** using **xn**" occurring in the main proof (see the proof fragment **bitadd** in Section 11.4). It turns out that this strategy cannot be carried through: the user cannot instantiate variables in an **apply** command in the same way that **provebylemma** allows for variable instantiation. Specifically, an application of **xn.sd** can result only in the assertion of its literal postcondition, referring to terms **x**, **y**, **c**, and **n**, rather than the assertion of a postcondition involving terms **.p**, **.sr8**, **.cout**, and an integer constant instantiated for **n**, as required by the proof fragment **bitadd**.

As a result, the proof is currently unfinished, but we have a plan, in the short term, for completing it. We shall use (already proven) state deltas exactly like **xn.sd**, but with **.p**, **.sr8**, **.cout**, and integer constants substituted for **x**, **y**, **c**, and **n**, and then suitably modify the main proof to **apply** the appropriate one of these state deltas in lieu of invoking the **provebylemma** command using lemma **xn**.

Though relatively straightforward, these modifications are somewhat tedious to implement, because ideally it should be arranged that at each such point in the proof, the correct state delta to apply will be the highest applicable one. In the longer term, this particular sort of problem will be eliminated by enhancing the SDVS lemma capability to accept quantifiers.

Regarding bitstring lemma **usxor\commute**, an attempted proof of the commutativity of **usxor** by induction from the existing axioms uncovered several bugs in the implementation of the SDVS **natinduct** command. After these were rectified, induction on the length of the bitstrings was frustrated by the existing SDVS mechanisms for proofs involving quantified statements. There may be a way of working around the problem with the existing proof commands, but it is not obvious and probably involves knowing the details of how SDVS interacts with EKL, its predicate logic solver inherited from Stanford University [18]. For the time being, the proof goes through by treating **usxor** as a derived operator (i.e., one defined in terms of **usand**, **usor**, and **usneg**).

Another challenge that had to be met was the lack of sufficiently strong axiomatic characterizations of bitstring addition. All existing SDVS axioms characterize **usadd** only in the absence of carries. This stems from the fact that an appropriate axiomatization of bitstring addition apparently requires a recursive definition, and proving properties of functions defined in this way requires something at least as strong as the **natinduct** command, a relatively recent addition to the SDVS prover. A suitable (albeit nonintuitive) definition of bitstring addition, *viz.* lemma **addlemma**, is needed for the proofs of lemma **x1** and state delta **xn.sd**.

We shall now explore a few general issues that arose in the course of proving the integer and bitstring lemmas. To begin with, SDVS currently provides no libraries of established lemmas that may be searched and employed in a new proof: for each example, all necessary facts must be proved "from scratch." This impedes the proof of theorems in a top-down fashion. The user cannot be sure that a natural strategy of decomposition into lemmas will make the proof substantially easier. The low-level lemmas proved herein provide a good start towards producing such libraries. It is expected that this situation will be remedied as SDVS becomes more widely used, and as larger applications are tackled. This will, in turn, lead to the need for more sophisticated search routines.

Another important issue relates to the fact that in SDVS, assertions may be added as needed to the axiom sets when they are felt to be consistent with the existing axioms. While convenient, and motivated by the SDVS emphasis on proving properties of programs rather than general theorems, this approach opens the door to the possibility of introducing inconsistencies into the axiom base. Moreover, the fact that axioms have been added to the axiom base on a largely *ad hoc* basis means that the axioms are not as well-organized as might be desired, and tend to be overly specific; this occasionally makes searching for appropriate axioms difficult. We consider it advisable either to find a standard model for the existing axioms, or to find an existing well-understood axiomatization for the SDVS operators and then try to prove the SDVS axioms from them.

Finally, we have identified certain behaviors of SDVS which, while presenting no real impediment during proof development, might nevertheless be improved. One example is that the "multiple cases" command `mcases` does not merely check that the precondition implies the conjunction of the cases, but requires that the conjunction of the cases be true. This can be handled by including a case for the negation of the precondition (as in the proof of bitstring lemma `shiftclemma`).

12 Conclusion

The examples considered in this report give the flavor of our second VHDL language subset, Stage 1 VHDL. Their specifications and correctness proofs, ranging from the very simple to the complex, are indicative of the capability of SDVS for dealing with Stage 1 VHDL. The correctness proof of the "shift-and-add multiplier" example, while worked out in principle and mostly implemented, will require one more revision for a complete, functioning implementation.

Stage 2 VHDL, implemented in 1992, is a considerably more powerful behavioral subset of the language, extending Stage 1 VHDL with the addition of the following VHDL language features: (restricted) design files, declarative parts in entity declarations, package `STANDARD` (containing predefined types `BOOLEAN`, `BIT`, `INTEGER`, `TIME`, `CHARACTER`, `REAL`, `STRING`, and `BIT_VECTOR`), user-defined packages, `USE` clauses, array type declarations, certain predefined attributes, enumeration types, subprograms (procedures and functions, excluding parameters of object class `SIGNAL`), concurrent signal assignment statements, `FOR` loops, octal and hexadecimal representations of bitstrings, ports of default object class `SIGNAL`, and general expressions of type `TIME` in `AFTER` clauses.

In 1993 we will be implementing a translator for the Stage 3 VHDL language subset. Stage 3 VHDL is expected to include constructs for structural descriptions (e.g., *component declarations*, *component instantiation statements*, and *configuration declarations*). Furthermore, SDVS will be enhanced with proof capabilities enabling both more general specifications and more tractable proofs of VHDL hardware descriptions. Two enhancements of particular importance will be the ability to translate structural descriptions, and the ability to reason about symbolic representations of VHDL time.

References

[1] L. G. Marcus, "SDVS 11 Users' Manual," Technical Report ATR-92(2778)-8, The Aerospace Corporation, September 1992.

[2] R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, (Norwell, MA: Kluwer Academic Publishers, 1989).

[3] D. L. Perry, *VHDL*, (New York: McGraw-Hill, Inc., 1991).

[4] IEEE, *Standard VHDL Language Reference Manual*, 1988. IEEE Std. 1076-1987.

[5] I. V. Filippenko, "Some Examples of Verifying Core VHDL Hardware Descriptions Using the State Delta Verification System (SDVS)," Technical Report ATR-91(6778)-6, The Aerospace Corporation, September 1991.

[6] I. V. Filippenko, "A Formal Description of the Incremental Translation of Stage 1 VHDL into State Deltas in the State Delta Verification System (SDVS)," Technical Report ATR-91(6778)-7, The Aerospace Corporation, September 1991.

[7] T. Aiken, I. Filippenko, B. Levy, and D. Martin, "A Formal Description of the Incremental Translation of Core VHDL into State Deltas in the State Delta Verification System (SDVS)," Technical Report ATR-89(4778)-9, The Aerospace Corporation, September 1989.

[8] I. V. Filippenko, "A Formal Description of the Incremental Translation of Stage 2 VHDL into State Deltas in the State Delta Verification System (SDVS)," Technical Report ATR-92(2778)-4, The Aerospace Corporation, September 1992.

[9] Vantage Analysis Systems, Inc., 42808 Christy Street, Ste 200, Fremont, CA 94538, *VHDL Basics Lecture Manual*, 1991.

[10] B. H. Levy, "Feasibility of Hardware Verification Using SDVS," Technical Report ATR-88(3778)-9, The Aerospace Corporation, September 1988.

[11] D. F. Martin and J. V. Cook, "Adding Ada Program Verification Capability to the State Delta Verification System (SDVS)," in *Proceedings of the 11th National Computer Security Conference*, National Bureau of Standards/National Computer Security Center, October 1988.

[12] I. V. Filippenko, "The Partition of VHDL into Language Subsets for the State Delta Verification System (SDVS)," Technical Report ATR-90(5778)-7, The Aerospace Corporation, September 1990.

[13] S. H. Kelem and B. H. Levy, "Preliminary Definition, Examples, and Specifications of Core VHDL," Technical Report ATR-88(3778)-8, The Aerospace Corporation, September 1988.

[14] L. Marcus and B. H. Levy, "Specifying and Proving Core VHDL Descriptions in the State Delta Verification System (SDVS)," Technical Report ATR-89(4778)-5, The Aerospace Corporation, September 1989.

[15] J. V. Cook, "Final Report for the C/30 Microcode Verification Project," Technical Report ATR-86(6771)-3, The Aerospace Corporation, September 1986.

[16] J. V. Cook, "Verification of the C/30 Microcode Using the State Delta Verification System (SDVS)," in *Proceedings of the 13th National Computer Security Conference*, (Washington, D. C.), pp. 20–31, National Institute of Standards and Technology/National Computer Security Center, October 1990.

[17] R. J. Auletta, "Application of SDVS Verification to VHDL Design and Synthesis," technical report, Electrical and Computer Engineering, George Mason University, 1991. Navy-ASEE Faculty Research Program.

[18] J. Ketonen and J. Weening, "EKL—An Interactive Proof Checker User's Reference Manual," Technical Report STAN-CS-84-1006, Dept. of Computer Science, Stanford University, June 1984.