Voss — A Formal Hardware Verification System User's Guide

Technical Report 93-45

Carl-Johan H. Seger
Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z4 Canada
Email: seger@cs.ubc.ca

December 6, 1993

Abstract

The Voss system is a formal verification system aimed primarily at hardware verification. In particular, verification using symbolic trajectory evaluation is strongly supported. The Voss system consists of a set of programs. The main one is called fl and is the core of the verification system. Since the metalanguage in fl is a fully general functional language in which Ordered Binary Decision Diagrams (OBDDs) have been built in, the verification system is not only useful for carrying out trajectory evaluation, but also for experimenting with various verification (formal and informal) techniques that require the use of OBDDs. This document is intended as both a user's guide and (to some extent) a reference guide. For the Voss alpha release, this document is still quite incomplete, but work is underway to remedy this.

Contents

Preface

This document is meant as an introduction to the Voss formal verification system. In particular, it is intended as an introduction to using the Voss system for symbolic trajectory evaluation. However, it is not meant to introduce the complete theory behind the system. For this, the reader is referred to [SegBry92].

Since the user interface to the Voss system is a complete, fully lazy, functional language similar to lazy-ML, this document begins by introducing the functional language. This is accomplished through a number of examples. For someone familiar with functional programming in a lazy language, these sections can be read very cursory. In the second part of the manual, verification tasks using the Voss system are explained and some secondary tools introduced. This section is by its very nature example oriented but I will try to be as precise as I can.

Since the Voss system is under active development, I cannot provide any guarantees for the correctness, suitability for any particular use. Since I am actively developing the system, I would appreciate bug reports and examples of dubious correctness. In return, I'll provide as rapid fixes and updates as humanly possible. Also, since the system is still evolving, it is quite possible that some examples do not correspond exactly to the way the current system works. However, I have tried to make the manual as up-to-date as I have been able to. In general, all the examples have been run using Voss 1.5.

Finally, I have often been asked what "Voss" really stands for, and I have heard various attempts containing the words Verification, Symbolic and Simulation, but the truth is that the name was chosen specifically not to be an acronym. Voss is a city in Norway between Bergen and Oslo and is prominently featured on virtually all weather forecasts in Norway. The city is at the west end of one of the main mountain ranges in Norway and has a special meaning to both my wife and myself. In recognition of this, I decided to call the verification system Voss.

Carl Seger

1 The Voss System—Background

The Voss system, hereafter referred to as Voss, started as a hardware verification system that supported symbolic trajectory evaluation. However, the main interface to the trajectory evaluator was a general purpose functional language with ordered binary decision diagrams built in. Consequently, Boolean functions could be represented, manipulated, and compared very efficiently. Since these capabilities are highly desirable in formal verification systems, it is perhaps not too surprising that Voss has become a prototype system for various forms of verification methods. In particular, there are now both symbolic model checkers as well as small theorem provers written in FL—the command language of Voss. However, since trajectory evaluation is less well known and also the main verification methodology supported in Voss, this manual will focus mostly on this technique.

Symbolic simulation is an offspring of conventional simulation. Like conventional simulation, it uses a built-in model of hardware behavior and a simulation engine to compute, on demand, the behavior of some design for some given inputs. However, it differs in that it considers symbols rather than actual values for the design under simulation. In this way, a symbolic simulator can simulate the response to entire classes of values with a single simulation run.

The concept of symbolic simulation in the context of hardware verification was first proposed by researchers at IBM Yorktown Heights in the late 1970's as a method for evaluating register transfer language representations. The early programs were limited in their analytical power since their symbolic manipulation methods were weak. Consequently, symbolic simulation for hardware verification did not evolve much further until more efficient methods of manipulating symbols emerged. The development of Ordered Binary Decision Diagrams (OBDDs) for representing Boolean functions radically transformed symbolic simulation.

Since a symbolic simulator is based on a traditional logic simulator, it can use the same, quite accurate, electrical and timing models to compute the circuit behavior. For example, a detailed switch-level model, capturing charge sharing and subtle strengths phenomena, and a timing model, capturing bounded delay assumptions, are well within reach. Also—and of great significance—the switch-level circuit used in the simulator can be extracted automatically from the physical layout of the circuit. Hence, the correctness results can link the physical layout with some higher level of specification.

The first "post-OBDD" symbolic simulators were simple extensions of traditional logic simulators. In these symbolic simulators the input values could be Boolean variables rather than only 0's, 1's as in traditional logic simulators. Consequently, the results of the simulation were not single values but rather Boolean functions describing the behavior of the circuit for the set of all possible data represented by the Boolean variables. By representing these Boolean functions as Ordered Binary Decision Diagrams the task of comparing the results computed by the simulator and the expected results became straightforward for many circuits. Using these methods it has become possible to check many (combinational) circuits exhaustively.

Recently, Bryant and Seger began developing a new generation of symbolic simulator based verifier. Since the method has departed quite far from traditional simulation, they called the approach symbolic trajectory evaluation. Here a modified version of a simulator establishes the validity of formulas expressed in a very limited, but precisely defined, temporal logic. This temporal logic allows the user to express properties of the circuit over trajectories: bounded-length sequences of circuit states. The verifier checks the validity of these formulas by a modified form of symbolic simulation.

Although the general theory underlying symbolic trajectory evaluation, as described in [SegBry92], is equally applicable to hardware as software systems, we will only describe a somewhat specialized version tailored specifically to hardware verification. For a more comprehensive discussion of the general case, the reader is referred to[SegBry92].

In symbolic trajectory evaluation the circuit is modeled as operating over logic levels 0, 1, a third level X representing an indeterminate or unknown level and a fourth value $\top$ representing overconstrained values. These values are partially ordered by their "information content" as $X \sqsubseteq 0$, $X \sqsubseteq 1$, $0 \sqsubseteq \top$, and $1 \sqsubseteq \top$, i.e., X conveys no information about the node value, 0 and 1 are fully defined values, and $\top$ represent an overconstrained value or a value that is both 1 and 0 at the same time [Normally, the $\top$ value is treated as an error condition]. The only constraint placed on the circuit model—apart from the obvious requirement that it accurately model the physical system—is monotonicity over the information ordering. Intuitively, changing an input from X to a binary value (i.e., 0 or 1) must not cause an observed node to change from a binary value to X or to the opposite binary value. In extending to symbolic simulation, the circuit nodes can take on arbitrary quaternary (four-valued) functions over a set of Boolean variables V. Symbolic circuit evaluation can be thought of as computing circuit behavior for many different operating conditions simultaneously, with each possible assignment of 0 or 1 to the variables in V indicating a different condition.

The biggest difference between trajectory evaluation and symbolic simulation is the way setting nodes to some value is accomplished. In a symbolic simulator, if the user requests the system to set the value on a node, say node A, to some value, say E, then this node takes on this value immediately, and if the node is an input node, keeps this value until the user requests the node to take on another value. In trajectory evaluation, on the other hand, the system only tries to set the node to the value E. In fact, it will set the node to the least element in the partial order that is consistent with both the current value on the node and the expression E. For example, if the node currently has the value bX (i.e., if the Boolean variable b is false, then the value on the node is 0, otherwise it is X), and we request the system to set the value on node A to cX, then the node will in fact take on the value bc$X$ (i.e., the node will be X unless at least one of b and c is false). Furthermore, in trajectory evaluation, inputs do not keep their values. If the user wants an input to a circuit to stay at 1 for 100 time units, he or she will have to state so explicitly in the antecedent. More about this later.

Properties of the system are expressed in a restricted form of temporal logic having just enough expressive power to describe both circuit timing and state transition properties, but remaining simple enough to be checked by an extension of symbolic simulation. The basic decision algorithm checks only one basic form, the assertion, in the form of an implication $[A \Longrightarrow C]$; the antecedent A gives the stimulus and current state, and the consequent C gives the desired response and state transition. System states and stimuli are given as trajectories over fixed length sequences of states.

Each of these trajectories are described with a temporal formula. The temporal logic used here, however, is very limited. A formula in this logic is either:

1. UNC (unconstrained),
2. (a) (n is 1) (node n is equal to 1)
   (b) (n is 0) (node n is equal to 0),
3. $F_1 \wedge F_2$ (F_1 and F_2 must both hold),
4. F when E (the property represented by formula F need only hold for those assignments satisfying the Boolean expression E),
5. NF (F must hold in the next state).

The temporal logic supported by the evaluator is far weaker than that of more traditional model checkers. It lacks such basic forms as disjunction and negation, along with temporal operators expressing properties of unbounded state sequences. The logic was designed as a compromise between expressive power and ease of evaluation. It is powerful enough to express the timing and state transition behavior of circuits, while allowing assertions to be verified by an extended form of symbolic simulation. Note however that the construct 4 above is very powerful. For example, suppose one would like to express the condition that

$$[A_1 \text{ "or" } A_2 \Longrightarrow C]$$

Clearly, this cannot be expressed directly in the logic. However, by introducing a new Boolean variable, say a, we could rewrite the above assertion as:

$$[(A_1 \text{when} a) \land (A_2 \text{when} \neg a) \Longrightarrow C]$$

Thus, at the cost of introducing one more Boolean variable, we can deal with disjunction too. However, since the number of Boolean variables used greatly affect the efficiency of the trajectory evaluation, this should be used sparingly.

The constraints placed on assertions make it possible to verify an assertion by a single evaluation of the circuit over a number of circuit states determined by the deepest nesting of the next-time operators. In essence, the circuit is simulated over the unique weakest (in information content) trajectory allowed by the antecedent, while checking that the resulting behavior satisfies the consequent. In this process a Boolean function is computed expressing those assignments for which the assertion holds.

The assertion syntax outline above is very primitive. To facilitate generating more abstract notations, the specification language can be embedded in a general purpose programming language. When a program in this language is executed, it automatically can generate the low-level temporal logic formulas and carry out the verification process.

The Voss system is a formal verification system based on symbolic trajectory evaluation developed by Dr. Carl Seger at University of British Columbia. Conceptually, the Voss system consists of two parts as shown in Fig. 1. The front-end is a compiler/interpreter for a small, fully lazy, functional language. A specification is written as a "program" in this language. When this specification program is executed, i.e., reduced to normal form, it builds up the simulation sequence that must be run in order to completely verify the specification.

The back-end of the Voss system is an extended symbolic simulator. The simulator uses an externally generated finite state machine description to compute the needed trajectories. This finite-state machine is a behavioral model of a digital circuit which can be generated from a variety of hardware description languages. In particular, the finite state machine can be generated from:

1. From a transistor netlist in .sim or .ntk format by a suite of programs called sim2ntk and ntk2exe.

2. From a gate netlist in a subset of Silos format by a program called silos2exe.

3. From a data-flow behavioral VHDL program, a structural VHDL program, or from an EDIF description, via a program called convert2fl.

Since we are using the ntk2exe tool[1] to pre-compile a switch-level netlist, the Voss system can carry out switch-level verification using the full MOSSIM II switch-level model. In addition, the

[1]The ntk2exe program is an extensive re-write of the ANAMOS tool as distributed in the COSMOS compiled switch-level simulator tool suite developed by Randy Bryant and associates at Carnegie Mellon University. Although virtually a complete re-write, the fundamental research ideas embedded in ntk2exe all have their roots in the ANAMOS system.

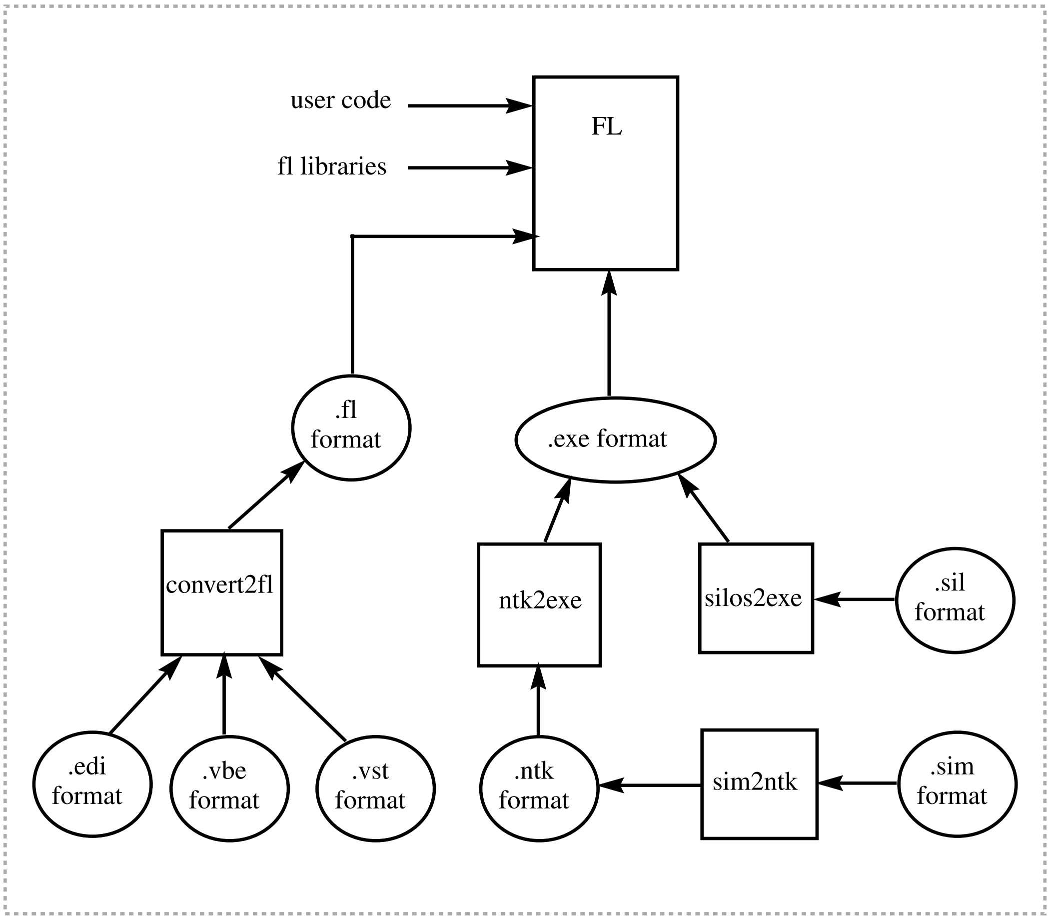

Figure 1: The Voss verification system.

finite state machine can be back-annotated with extracted delay values and thus fairly sophisticated delay simulation can also be carried out.

The gate level simulator is (roughly) functionally equivalent to the SILOS II simulator. In addition, fairly comprehensive delay modeling capabilities has been added for more accurate verification. In order to achieve good performance, the symbolic simulator employs event scheduling for both the circuit simulation as well as in maintaining the verification conditions

Behavioral and structural VHDL is currently supported through a translator program that is a derivative of the VHDL simulator distributed with the Alliance 1.1 tool suite. Thus, only dataflow behavioral VHDL programs are supported. An extensive rewrite of this part of the system is currently underway, but it is doubtful that it will be ready for general release until June 1994. If you desperately need a richer VHDL language to work in, please send me an email and I can inform you on the current status of the translator.

From the Voss user's point of view the basic verification command in the Voss system looks like:

FSM options fsm ant-list cons-list trace-list;

where options is a string that can give specification options to the trajectory evaluation simulator, fsm is a behavioral description of a finite state machine and ant-list and cons-list denote lists of atomic constraints used to express the verification conditions Each atomic constraint is a 5-tuples of the form (b,n,v,s,f) which, for a given trajectory, denotes the constraint that "if the Boolean expression b is true then the node named by n has the value v in all states of the trajectory from the start state s up to, but not including, the final state f". Finally, the trace-list is a list of triples of the form (n,f,t) requesting a trace of the node n from time f to time t. If the verification is successful (we will return to this in greater detail later), FSM will return T (true); otherwise it will return a boolean expression denoting the condition under which it is valid (if it is never valid, it will simply return F). If the verification fails for some reason, the system prints out a counter-example for the first node for which it encounters an incorrect value.

To give a very simple example, the command:

```
FSM "" (load_exe "inv.exe") [(T, "input", F, 0, 1)] [(T, "output", T, 1, 2)] [];
```

expresses a relationship between the input and output node of the circuit "inv.exe" for one particular input value and where the output value is delayed by one time unit.

A slightly more sophisticated approach is illustrated by the assertion:

```
let v = variable "v";
FSM ""(load_exe "inv.exe")[(T, ‘input‘, v, 0, 1)] [(T, ‘output‘, (NOT v), 1, 2)] [];
```

where the constants F and T have been replaced by the symbolic expressions v and (NOT v).

It may appear that the temporal scope of the above assertion is limited to the first two instants of discrete time—that is, "if the input at time 0 is v, then the output at time 1 will be (NOT v)." However, the temporal scope of this assertion actually extends infinitely along every trajectory of the finite-state machine. This is because the automatic verification procedure considers every state of the finite-state machine to be a possible initial state of the machine. At any point along any trajectory, the current state corresponds to the initial state of some other trajectory. Because the temporal scope of the above assertion extends infinitely along every trajectory, the assertion can be accurately interpreted to express the property that "for all times t, if the value of the input node of the inverter is v, then the value of the output node at time t+1 will be (NOT v)". We will return to the pragmatics of trajectory evaluation later in this document. For now we turn our attention to the interface language to the Voss system.

2 Fl—The Meta Language of Voss

In this section[2] we provide an introduction to the functional language FL.

Similar to many theorem provers (e.g., the HOL system[HOL]) the Voss command language for the verification system is a general purpose language. In fact, it shows a strong degree of similarity to the version of ML used in the HOL system. However, there are several differences: many syntactic but some more fundamental. In particular, the functional language used in Voss has lazy evaluation semantics. In other words, no expression is evaluated until it is absolutely needed. Similarly, no expression is evaluated more than once. Another difference is that Boolean functions are first-class objects and can be created, evaluated, compared and printed out. For efficiency reasons these Boolean functions are represented as ordered binary decision diagrams.

Fl is an interactive language. At top-level one can: 1) perform declarations, and 2) evaluate expressions. In this section we will introduce the language by several examples.

2.1 Invoking fl

If the Voss system is installed on your system and you have the suitable search path set up, it suffices to type fl to get a stand-alone version of Fl. In this manual, we have used Voss 1.5 throughout. In other words, typing fl yielded:

```
% fl
      /\
     /  \/\
 /\ /      \
/  Voss 1.5 \
VOSS-LIBRARY-DIRECTORY = /isd/local/generic/lib/vosslib

:
```

Note that the VOSS-LIBRARY-DIRECTORY is installation dependent. We will return to this later when we discuss the user defaults.

The fl program can take a number of arguments. In particular,

-f n Start FL by first reading in the content of the file named n.

-I n Set the default search path to n.

-s i Set the default OBDD table to be of size 2^i, where i can range from 16 to 22. Normally, this is not needed. However, if a verification task will be needing more than 1/2 million OBDD nodes, setting i to some number above 19 will improve performance.

2.2 Expressions

The Fl prompt is : so lines beginning with this contain the user's input; all other lines are output of the system.

```
: 2+3;
5
```

[2] This chapter is to a large extent modeled after Chapter 1 in the HOL System DESCRIPTION from Cambridge University. In particular, many of the early examples are taken from this source.

Here we simply evaluated the expression 2+3 and FL reduced it to normal form; in this case computed the result 5. Note that fl does only support integers as number types. Furthermore, these integers are limited to +/- 536870912 (two's complement 30 bit numbers). This restriction on the numbers is likely to disappear shortly, but for Voss 1.5 it is a restriction that is important to remember.

2.3 Declarations

The declaration let x = e binds a computation of e to the variable x. Note that it does not evaluate e (since the language is fully lazy). Only if x is printed or used in some other expression that is evaluated will it be evaluated. Also, once e is evaluated, x will refer to the result of the evaluation rather than the computation. Hence, the expression e is evaluated at most once, but it may not be evaluated at all.

```
: let x = 3+3;
x::int
```

Note that when expressions are bound to variables, the system simply prints out the inferred type of the expression. We will return to the typing scheme in FL later. For now, it suffices to say that FL tries to find as general type as possible that is consistent with the type of the expression.

Contrary to ML, FL 1.5 does not allow simultaneous bindings. Hence, if we would like to bind the expressions 2 and 4-5 to the variables x and y respectively, we would have to write:

```
: let x = 2;
x::int
: let y = 4-5;
y::int
```

A declaration can be made local to the evaluation of an expression e by evaluating the expression decl in e. For example:

```
: let y = let x = 4 in x-5;
y::int
```

would bind the expression 4 to x only inside the expression bound to y. Thus, we get:

```
: let x = 2;
x::int
: let y = let x = 4 in x-5;
y::int
: x;
2
: y;
-1
```

FL is lexically scoped, and thus the binding in effect at the time of definition is the one used. In other words, if we write:

```
: let x = 2;
x::int
: let y = x*5;
y::int
: let x = 12;
x::int
```

and we then evaluate y we will get 10 rather than 60.

2.4 Functions

To define a function f with formal parameter x and body e one performs the declaration: let f x = e. To apply the function f to an actual parameter e one evaluates the expression f e.

```
: let f x = x+2;
f::(int) -> (int)
: f 4;
6
```

Note that the type inferred for f is essentially “a function taking an int as argument and returning an int”. Applications binds more tightly than anything else in FL; thus for example: f 3 + 4 would be evaluated as: ((f 3)+4) and thus yield 9.

Functions of several arguments can also be defined:

```
: let add x y = x+2*y;
add::(int) -> ((int) -> (int))
: add 1 4;
9
: let f = add 1;
f::(int) -> (int)
: f 4;
9
```

Applications associate to the left so add 3 4 means (add 3) 4. In the expression add 3, the function add is partially applied to 3; the resulting value is the function of type int-¿int which adds 3 to twice its argument. Thus add takes its arguments one at a time. We could have made add take a single argument of the cartesian product type (int#int):

```
: let add (x,y) = x+y;
add::((int # int)) -> (int)
: add (3,4);
7
: add 3;
===Type mismatch: (int # int) and int
#### Run-time error
---- Type error
```

As well as taking structured arguments (e.g. (3,4)) functions may also return structured results:

```
: let manhat_dist (x1,y1) (x2,y2) = (x2-x1, y2-y1);
manhat_dist::((int # int)) -> (((int # int)) -> ((int # int)))
: manhat_dist (1,1) (3,5);
(2,4)
```

Trying to print a function with insufficient number of actual arguments yield a dash for the function and the type of the expression is printed out. For example:

```
: (5, manhat_dist (1,2));
(5,-) ::(int # ((int # int)) -> ((int # int)))
```

2.5 Recursion

The following is an attempt to define the factorial function:

```
: let fact n = n=0 => 1 | n*fact (n-1);
#### Run-time error
---- Undefined variable (fact)
```

The problem is that any free variables in the body of a function have the bindings they had just before the function was declared; fact is such a free variable in the body of the declaration above, and since it is not defined before its own declaration, an error results. To make things clear consider:

```
: let f n = n+1;
f::(int) -> (int)
: let f n = n=0 => 1 | n*f (n-1);
f::(int) -> (int)
: f 3;
9
```

Here 3 results in the evaluation of 3*(f 2), but now the first f is used so f 2 evaluates to 2+1=3. To make a function declaration hold within its own body, letrec instead of let must be used. The correct recursive definition of the factorial function is thus:

```
: letrec fact n = n=0 => 1 | n*fact (n-1);
fact::(int) -> (int)
: fact 5;
120
```

It should be pointed out that FL currently does not allow direct definition of mutually recursive functions. One way around this is to define a "wrapper" function that takes as a parameter some number of name of the mutually recursive function that is intended. Mutually recursive function definitions is quite likely to be added to the next major release of the system.

2.6 Tuples

If $e_1, e_2, \ldots, e_n$ have types $t_1, t_2, \ldots, t_n$, then the FL expression $(e_1, e_2, \ldots, e_n)$ have type $t_1\texttt{\#}t_2\texttt{\#}\ldots\texttt{\#}t_n$. The standard functions on tuples are fst (first), snd (second), and the infix operation , (pair).

```
: let q = ((1,2),3);
q::((int # int) # int)
: let qq = (1,2,3);
qq::(int # (int # int))
: q;
((1,2),3)
: qq;
(1,2,3)
: let qqq = (1,"abc");
qqq::(int # string)
: qqq;
(1,"abc")
```

2.7 Lists

If $e_1, e_2, \ldots, e_n$ have type t, then the FL expression $[e_1, e_2, \ldots, e_n]$ has type (t list). The standard functions on lists are hd (head), tl (tail), [] (the empty list), and the infix operation : (cons). Note that all elements of a list must have the same type (compare this with a tuple where the size is determined but each member of the tuple can have different type).

```
: let l = [1,2,3,3,2,1,2];
l::(int list)
: hd l;
1
: tl l;
[2,3,3,2,1,2]
: 0:l;
[0,1,2,3,3,2,1,2]
: letrec (len [] = 0) /\ (len (a:rest) = 1+len rest);
len::((* list)) -> (int)
: len l;
7
```

2.8 Strings

A sequence of characters enclosed between " or ‘ is a string. The standard functions on strings are ^(catenation), explode (make string into list of strings) and implode (make list of strings into single string). There are also int2str and bool2str functions that create a string from an integer or an object of type boolean. We will return to these later.

```
: let q = "abc and _12!@@#";
q::string
: let qq = ‘qw"q qw‘;
qq::string
: q^qq;
"abc and _12!@@#qw"q qw"
: explode q;
["a","b","c"," ","a","n","d"," ","_","1","2","!","@","@","#"]
: implode ["1", "2"];
"12"
: int2str (1-34);
"-33"
: bool2str ((variable "a") AND (variable "b"));
"a&b"
```

2.9 Polymorphism

The list processing functions hd, tl, etc. can be used on all types of lists.

```
: hd [1,2,3];
1
: hd ["abc", "edf"];
"abc"
: (hd ["a", "b"]), hd [4,2,1];
("a",4)
: let q = [T,T,F];
q::(bool list)
: hd q;
T
```

Thus hd has several types; for example, it is used above with types (int list) `->` int, (string list) `->` string, and (bool list) `->` bool. In fact if ty is any type then hd has the type (ty list) `->` ty. Functions, like hd, with many types are called polymorphic, and FL uses type variables *, **, ***, etc. to represent their types.

```
: let f x = hd x;
f::((* list)) -> (*)
: letrec map fn [] = []
   /\    map fn (h:rest) = (fn h) : (map fn rest);
map::((*) -> (**)) -> (((* list)) -> ((** list)))
: letrec fact n = n=0 => 1 | n*fact (n-1);
fact::(int) -> (int)
: map fact [1,2,3,4,5,6,7];
[1,2,6,24,120,720,5040]
```

The FL function map takes a function f (with argument type * and result type **), and a list l (of elements of type *), and returns the list obtained by applying f to each element of l (which is a list of elements of type **). Map can be used at any instance of its type: above, both * and ** were instantiated to int; below, * is instantiated to (int list) and ** to bool. Notice that the instance need not be specified; it is determined by the type checker.

```
: let eq1 x = x=1;
eq1::(int) -> (bool)
: map eq1 [1,2,3,4,12,2,1,2];
[T,F,F,F,F,F,T,F]
```

It should be pointed out that FL has a polymorphic type system that is sightly different from standard ML's. In particular, only "top-level" user-defined functions can be polymorphic. In other words, the following works as we would expect.

```
: let null l = l = [];
null::((* list)) -> (bool)
: let f x y = null x OR null y;
f::((* list)) -> (((** list)) -> (bool))
: f [1,2,3] ["abc", "cdef"];
F
```

However, if we use the same declaration inside the expression, it must be monomorphic. In other words, the following example fails.

```
: let f x y =
    let null l = l = [] in
    null x OR null y;
f::((* list)) -> (((* list)) -> (bool))
: f [1,2,3] ["abc", "cdef"];
===Type mismatch: int and string
#### Run-time error
---- Type error
```

In this respect, FL is similar to the functional language called Miranda[3][**?**].

2.10 Lambda Expressions

The expression \x.e evaluates to a function with formal parameter x and body e. Thus the declaration let f x = e is equivalent to let f = \x.e. The character \ is our representation of lambda, and expressions like \x.e are called lambda-expressions.

[3]Miranda is a trademark.

```
: \x.x+1;
- ::(int) -> (int)

: let q = \x.x+1;
q::(int) -> (int)
: q 1;
2
: map (\x.x*x) [1,2,3,4,5];
[1,4,9,16,25]
```

2.11 Failures

Some standard functions fail at run-time on certain arguments, printing out a string (which is usually the function name) to identify the sort of failure. A failure with string "t" may also be generated explicitly by evaluating the expression error "t" (or more generally error e where e has type string).

```
: hd(tl [2]);
Failure:    ----  Cannot compute hd of the empty list

: 1/0;
Failure:    ----  Division by zero


: error "My message";
Failure:    ----   My message
```

A failure can be trapped by catch; the value of the expression e_1 catch e_2 is that of e_1, unless e_1 causes a failure, in which case it is the value of e_2. One important property of catch is that it is (very) strict in its first argument. In other words, (hd (e_1 catch e_2)) will completely evaluate e_1 even though only the first element in the list may be needed. In view of FL's lazy semantics, the use of catch should be very carefully considered. In particular, the bindings of catch is dynamic rather than static so the user beware! It is not unlikely that catch will disappear from FL in future versions.

2.12 Boolean Expressions

All Boolean expressions in FL are maintained as ordered binary decision diagrams. Hence, it is very easy to compare complex Boolean expressions and to combine them in different ways. Boolean variables are created by variable s, where s is of type string. The system uses name equivalence, and thus

```
: let v = variable "v";
v::bool
: v=v;
T
: variable "v" = variable"v";
T
```

The constants true and false are denoted T and F respectively. The standard boolean functions are available, i.e., AND, OR, NOT, XOR, and = are all defined for objects of type Boolean.. Furthermore, there is a special identity operator == that return true or false depending on whether the two arguments represent the same Boolean function or not.

Note that the variable ordering in the OBDD representation is defined by the order in which each variable function call *gets evaluated*. Since FL is a fully lazy language, and thus the order in which expressions are evaluated is often difficult to predict, it is strongly recommended that each variable declaration is forced to be evaluated before it is being used. In the standard library default.fl a function, called declare, is defined to simplify this task. We will return to this later when we discuss the various FL libraries. Also, note that once a variable function call has been evaluated for a specific string argument, the created variable has been placed in the variable order and thus consequent calls will return this variable. Consequently, the only way of changing the variable order after a variable has been created, is to quit FL and start it again.

```
: let a = variable "a";
a::bool
: let b = variable "b";
b::bool
: a AND b;
a&b
: a OR b;
a + b
: NOT a AND NOT b AND T;
a'&b'
: a = b;
a&b + a'&b'
:  a == b;
F
: (a=b) == (a AND b OR NOT a AND NOT b);
T
```

The default style for printing Boolean expressions is as a sum-of-products. Since this may require printing an extremely large expression, there is a user-setable limit on how many products that will be printed and the maximum size of a product. For more details how to modify these two parameters, see the section on the .vossrc file on page 4.

2.13 Quantifiers

There are several ways of using quantification. But the "traditional" !x. e (for all x) and ?x. e (there is an x) can be used as long as the type of x and e is bool. In addition, you can also quantify away a variable in an expression by quant_forall v e or quant_thereis v e.

```
: !a. ?b. (a XOR b);
T
: let a = variable "a"; let b = variable "b"; let c = variable "c";
a::bool
b::bool
c::bool
: : : a AND b OR c;
a&b + c
:  quant_forall a (a AND b OR c);
c
: quant_thereis (a OR c) (a AND b OR c);
T
```

In fact, quant_forall and quant_thereis quantifies away all variables in the first Boolean expression. For example:

```
: let v s = variable s;
v::(string) -> (bool)
: let a = v "a"; let b = v "b"; let c = v "c"; let d = v "d";
a::bool
b::bool
c::bool
d::bool
: : : : let ex = (a AND NOT b);
ex::bool
: ex;
a&b'
: let nex = ex AND (a=c) AND (b=d);
nex::bool
: quant_thereis (a AND b) nex;
c&d'
```

Note that the actual Boolean expression used as first argument is irrelevant. The only important fact is on what variables the expression depends.

2.14 Dependency

Sometimes it is useful to find out which Boolean variables a Boolean function actually depends on. The built-in function depends takes a list of elements of type bool and return the union of the variables these functions depend on. For example:

```
: let v s = variable s;
v::(string) -> (bool)
: let a = v "a"; let b = v "b"; let c = v "c"; let d = v "d";
a::bool
b::bool
c::bool
d::bool
: : : : let ex1 = (a=c) AND d;
ex1::bool
: let ex2 = a = b;
ex2::bool
: depends [ex1];
["a","c","d"]
: depends [ex1,ex2];
["a","b","c","d"]
```

Note that the order of the variables in the list returned by depends is the variable order of the OBDD representation.

2.15 Substitutions

Given a Boolean function represented as a OBDD, it is convenient to be able to apply the function to some arguments. This can be accomplished by the substitute command that takes a list of (variable name, expression) and an expression in which the simultaneous substitution is to be made. For example,:

```
: let v s = variable s;
v::(string) -> (bool)
: let a = v "a"; let b = v "b"; let c = v "c"; let d = v "d";
a::bool
b::bool
c::bool
d::bool
: : : : let ex = (a AND NOT b);
ex::bool
: ex;
a&b'
: substitute [("a", c), ("b", d)] ex;
c&d'
```

It should be pointed out that there are no restrictions on the expressions in the substitutions. In particular, it is possible to "swap" variables. We illustrate this by continuing the example above:

```
: ex;
a&b'
: substitute [("a", b), ("b", a)] ex;
a'&b
```

2.16 Type Abbreviations

Types can be given names:

```
: new_type_abbrev pair = int#int;
: let p = (1,2);
p::(int # int)
```

However, as can be seen from the example, the system does not make any distinction between the new type name and the actual type. It is purely a short hand that is useful when defining concrete types below.

2.17 Concrete Types

New types (rather than mere abbreviations) can also be defined. Concrete types are types defined by a set of constructors which can be used to create objects of that type and also (in patterns) to decompose objects of that type. For example, to define a type card one could use the construct type:

```
: lettype card = king | queen | jack | other int;
other::(int) -> (card)
jack::card
queen::card
king::card
```

Such a declaration declares king, queen, jack and other as constructors and gives them values. The value of a 0-ary constructor such as king is the constant value king. The value of a constructor such as other is a function that given an integer value n produces other(n).

```
: king;
- ::card

: other 9;
- ::card
```

Note that there is no print routine for concrete types. If a print routine is desired, one has to define it. To define functions that take their argument from a concrete type, we introduce the idea of pattern matching. In particular

```
let f pat1 = e1
 /\ f pat2 = e2
 /\ ...
 /\ f patn = en;
```

denotes a function that given a value v selects the first pattern that matches v, say pati, binds the variables of pati to the corresponding components of the value v and then evaluates the expression ei. We could for example define a print function for the cards in the following way:

```
: let pr_card king = "K"
  /\  pr_card queen = "Q"
  /\  pr_card jack = "J"
  /\  pr_card (other n) = int2str n;
pr_card::(card) -> (string)
: pr_card king;
"K"
: pr_card queen;
"Q"
: pr_card jack;
"J"
: pr_card (other 5);
"5"
```

Mutually recursive types can also be defined. To do so, simply list the type names on the left hand side of the equality sign and list the type expressions on the right hand side. For example:

```
: lettype IExpr, BExpr = Ivar string | Plus IExpr IExpr,
                         And BExpr BExpr | GEQ IExpr IExpr;
GEQ::(IExpr) -> ((IExpr) -> (BExpr))
And::(BExpr) -> ((BExpr) -> (BExpr))
Plus::(IExpr) -> ((IExpr) -> (IExpr))
Ivar::(string) -> (IExpr)
```

2.18 Abstract Types

In FL one can also hide the definitions of types, type constructors, and functions. By enclosing a sequence of type declarations and function definitions within begin_abstype end_abstype elist, only the constructors and/or functions mentioned in the elist will be visible and accessible for other functions and definitions. Thus, one can protect a concrete type and only make some abstract constructor functions available. To illustrate the concept, consider defining a concrete type called theorem. The only way we would like the user to be able to create a new theorem is to give a Boolean expression that denotes a tautology (something always true). First we define the expression type.

```
: lettype expr =  Forall string expr |
                Thereis string expr |
                Var string |
                True |
                False |
                And expr expr |
                Or expr expr |
                Not expr;
Not::(expr) -> (expr)
Or::(expr) -> ((expr) -> (expr))
And::(expr) -> ((expr) -> (expr))
False::expr
True::expr
Var::(string) -> (expr)
Thereis::(string) -> ((expr) -> (expr))
Forall::(string) -> ((expr) -> (expr))
```

We then define the concrete type theorem and the constructor function is_taut. Note that we also define a couple of help functions. However, only the is_taut function is exported out of the abstract type, and thus is the only way of creating a theorem.

```
: begin_abstype;
: lettype theorem = Thm expr;
Thm::(expr) -> (theorem)
: letrec assoc x l = l = [] => error "assoc" |
                let h = hd l in
                (fst h) = x => (snd h) | assoc x (tl l);
assoc::(*) -> ((((* # **) list)) -> (**))
: letrec  (eval (Forall s e) al = !x.(eval e ((s,x):al))) /\
        (eval (Thereis s e) al = ?x.(eval e ((s,x):al))) /\
        (eval (Var s) al = (assoc s al) catch
                                   (error "Free variable")) /\
        (eval True al = T) /\
        (eval False al = F) /\
        (eval (And e1 e2) al = (eval e1 al) AND (eval e2 al))/\
        (eval (Or e1 e2) al = (eval e1 al) OR (eval e2 al)) /\
        (eval (Not e) al = NOT (eval e al));
eval::(expr) -> ((((string # bool) list)) -> (bool))
: let is_taut e = ((eval e []) == T) => Thm e |
                                       error "Not a tautology";
is_taut::(expr) -> (theorem)
end_abstype is_taut;
```

We can now use this very safe theorem system, since we can only generate theorems that are tautologies. For example

```
: let e = (Forall "a" (Thereis "b" (And (Var "a") (Var "b"))));
e::expr
: is_taut e;
Failure:    ----   Not a tautology


: let f = (Forall "a" (Thereis "b" (Or (Var "a") (Var "b"))));
f::expr
: is_taut f;
- ::theorem
```

2.19 Infix Operators

In order to make the FL code more readable, it is possible to declare a function to be infix (associating from the left), infixr (associating from the right), nonfix (really prefix), or postfix. For the infix and infixr directives, the precedence can be given as a number from 1 to 9, where a higher number binds tighter. Note that prefix and postfix functions bind higher than any infix function. Beware that the fixity declaration modifies the parser and thus remains in effect whether the function is exported out of an abstract data type or note. This "feature" is likely to be fixed fairly soon. As an illustration of this idea, consider the following example:

```
: lettype expr = Var int |
                 Mult expr expr |
                 Plus expr expr |
                 Negate expr;
Negate::(expr) -> (expr)
Plus::(expr) -> ((expr) -> (expr))
Mult::(expr) -> ((expr) -> (expr))
Var::(int) -> (expr)
: letrec eval (Var i) = i /\
         eval (Mult e1 e2) = (eval e1) * (eval e2) /\
         eval (Plus e1 e2) = (eval e1) + (eval e2) /\
         eval (Negate e1) = 0-(eval e1);
eval::(expr) -> (int)
: let ** a b = Mult a b;
**::(expr) -> ((expr) -> (expr))
: let ++ a b = Plus a b;
++::(expr) -> ((expr) -> (expr))
: infix 4 **;
: infix 3 ++;
: let ' i = Var i;
'::(int) -> (expr)
: let q = '1 ++ Negate ('2) ** Negate ('4);
q::expr
: eval q;
9
```

The next example illustrates how postfix declarations can make the code more readable.

```
: let ns i = 1000*i;
ns::(int) -> (int)
: postfix ns;
: let to a b = (a,b);
to::(*) -> ((**) -> ((* # **)))
: infix 3 to;
: 2 ns to 4 ns;
(2000,4000)
```

2.20 Circuit Models

Since the main use of the FL system, and its historical root, is related to hardware verification, there are a number of built-in functions specifically tailored towards hardware modeling and symbolic trajectory evaluation in particular. Internally, a circuit is represented by a list of nodes (names), and a next state function. The next state function is mapping the current state of the circuit (including the current values on the inputs) to a new state of the circuit. Since the circuit representation is intended for trajectory evaluation, the value domain for each node in the circuit is $\{0, 1, X, \top\}$ and thus the next state function consists of quaternary (four-valued) extensions of the usual Boolean function. The type of such circuit is fsm (for finite state machine) and, by default, it does not have a print function (since the machines are usually much too large to be meaningful anyway to the user. However, for the curious reader, the command print_fsm will print out a pretty complete version of the state machine. Note that for efficiency reasons, the next state function also contains delay and fan-in and fan-out information to aid in the efficient simulation.

In general, there are two ways of creating a fsm object:

1. Loading a pre-compiled version of a circuit in .exe format.

2. Converting a FL structure into an fsm.

2.20.1 Loading a .exe File

If the original circuit was described in Berkeley .sim format or as a SILOS II gate list, there are programs distributed with the system that can be used to compile an fsm model directly from these formats. The common format for such pre-compiled circuit model is a (binary) file with a .exe suffix. Loading in the .exe file and making it an fsm object simply involves calling the load_exe function.

For example, if there is a full_adder.exe file in the current working directory, the following command would create an fsm object of the circuit.

```
: let ckt1 = load_exe "full_adder.exe";
ckt1::fsm
```

Note that there is no theoretical limit on the number of circuit that can be loaded into the FL system at any particular point in time. However, since fsm models are often quite large, it is generally advisable not to load more models than absolutely necessary.

2.20.2 Creating an fsm Object Inside FL

Warning: This addition is fairly young and has thus not been extensively tested. Also, it leaves quite a bit to be desired in terms of not being very "clean and simple". For a reader that is more interested in using the Voss system rather than writing a new conversion program from some other netlist format, this section can be skipped.

The main routine for creating an fsm object is make_fsm. The type of make_fsm is (Set) -¿ (fsm)[4] Before going into how an object of type Set can be constructed, we need to make a small digression and introduce the quaternary logic type. For historical reasons, the name of the quaternary objects is tern, and tern is defined as the concrete type:

[4] Currently, when FL is invoked, the type of make_fsm is actually (*)->(fsm), but this is a bug that will be removed shortly.

```
: lettype tern =  One |
                Zero |
                X |
                Z |
                Val string |
                And tern tern |
                Or tern tern |
                Not tern;
Not::(tern) -> (tern)
Or::(tern) -> ((tern) -> (tern))
And::(tern) -> ((tern) -> (tern))
Val::(string) -> (tern)
Z::tern
X::tern
Zero::tern
One::tern
```

where One, Zero, X, And, Or, and Not are the obvious functions. Z is used to represent the top value. Finally, Val s is used to refer to the value on node s. Thus, as a concrete example, the next state function of a node "o" may be described as (Not (And (Val "in1") (Val "in2"))).

There are four constructor functions for an object of type Set:

1. Empty,
2. Element (string#((tern#tern) list)),
3. Union Set Set, and
4. Sequential Set Set.

The Empty is introduced to make writing functions that generate objects of type Set easier. Element is the constructor that actually introduces new nodes and also give driver functions for the node. In general, Element n gvl, will create a node named n. This node will at least (more about this later) the drivers listed in the gvl list. A driver is simply a pair of quaternary expressions: the first object of the pair being a guard, and the second part the value being driven when the guard is true. For a combinational node, the driver list is a single pair whose guard is One and whose value component is the next state function of the gate. For more complex nodes, like register nodes or bus nodes, the guard-value list is often much longer.

The Union construct is used to gather a collection of these Element declarations so that an fsm can eventually be made. Finally, the Sequential constructor takes two objects S1 and S2 of type Set and merges them into a single object of type Set. In that sense, Sequential behaves pretty much like Union. The difference is how the two deal with nodes defined in both set S1 and set S2 and for which both nodes may have at least one of their guards enabled at the same time. Union will find the greatest lower bound of the values being driven at the same time whereas Sequential will assume that the driver in the second set overrides the driver in the first. To illustrate the difference, consider the following example:

```
: let a = Element ("in1", [(One, One)]);
a::Set
: let b = Element ("in1", [(One, Zero)]);
b::Set
: let ex1 = Union a b;
ex1::Set
: let ex2 = Sequential a b;
ex2::Set
: let m1 = make_fsm ex1;
m1::fsm
: let m2 = make_fsm ex2;
m2::fsm
: m1;
```

Now m1 and m2 will both be fsm models with a single node called in1. However, in m1, the next state function of in1 will be X, since that is the most we can say from the inconsistent driver commands given in the example. On the other hand, the next state function of the in1 node in m2 will be Zero.

For an example of using this facility, the directory HDL in the demo distribution illustrates how one can go about defining a new description language in order to create an fsm object.

2.21 Queries to the Circuit Model

There are four built-in functions that are useful in dealing with an fsm model: nodes, fanin, and fanout, and get_node_val. Nodes takes an fsm model and returns a *list of lists* of circuit nodes. The reason for the double listing is that a node may have more than one name (only from .exe file translations). Fanin takes an fsm model and a name of a node and returns the list of node names that the next state function depend on. Fanout works in a similar way, but for the fanout nodes. Finally, get_node_val takes an fsm object and the name of a node and returns the encoded version of the current value of the node. The encoding used is: $X = (T,T)$, $1 = (T,F)$, $0 = (F,T)$, and $\top = (F,F)$. Of course, the two Boolean values are often Boolean functions over some variables. The main use of the get_node_val function is when the simulation is aborted for some reason. Get_node_val can then be used to probe the current state of the system. For example, assuming there is a full_adder.exe file in the current working directory, we would get:

```
: let ckt = load_exe "full_adder.exe";
ckt::fsm
: nodes ckt;
..[["cout"],["t4"],["t3"],["t2"],["result"],["cin"],["t1"],["b"],["a"]]
: fanin ckt "cout";
["t2","t3","t4"]
: fanout ckt "t3";
["cout"]
: fanout ckt "t4";
["cout"]
: fanout ckt "a";
["t3","t2","t1"]
: get_node_val ckt "t2";
..(T,T)
```

If the user also loads in the "default.fl" standard library, there is the very useful function *excitation* that also takes an fsm model and a node name, but that returns the next state function for binary inputs. Note, however, that the current version of the excitation function only works correctly for unit delay nodes.

```
: load "defaults.fl";
-Loading file defaults.fl
T

: excitation ckt "cout";
....
Trace started for node: cout
        Current value:X
.Time: 1
.Trace: Node cout  at time 1: t4' + t3' + t2'
Time: 2

Trace ended for node: cout
"t4' + t3' + t2'"
```

We will return to this in Section 4.

2.22 Symbolic Trajectory Evaluation

There is actually only one built-in command for symbolic trajectory evaluation called FSM. In general, FSM determines, through symbolic trajectory evaluation, whether an antecedent/consequent pair hold in for some circuit. FSM will return a Boolean function that gives the condition for the verification to succeed. For most applications the desired return value is T.

In general FSM is invoked as

```
FSM options fsm ant_list cons_list trace_list
```

where options is a string that can contain a combinations of the following flags:

-a Abort the verification at the first antecedent or consequent failure. If the verification is aborted, FSM will return a Boolean function that gives *the condition for this failure to manifest itself.* Note that this is contrary to FSM's usual behavior which is to return the Boolean function that gives the conditions for the verification to succeed.

-m n Abort the verification after reaching time n.

-i Allow antecedent failures. In other words, compute a straight implication. The normal behavior of the verification process is to disallow antecedent failures. Thus the default verification condition is both to check that every trajectory the circuit can go thorough that is consistent with the antecedent is also consistent with the consequent, and that there is at least one (real) circuit trajectory that is consistent with the antecedent.

-w Do not print out warning messages.

-t s In addition to printing out trace messages on stderr, also send the trace events in Postscript format to the file s. By previewing or printing out the file the user gets a waveform diagram for the traced signals.

-T s Same as -t, but generate Postscript code in landscape mode.

The second argument to FSM must be an object of the fsm type representing a circuit that is to be simulated.

The ant_list and cons_list are both lists of five-tuples. Each five-tuple is of the form (g, n, v, s, t), where g is a Boolean function denoting the domain for which this assertion/check should be carried out, n is the name of a node, v is the value to be asserted/checked, and s and t denote the start and stop times for this assertion/check respectively.

Finally, the last argument to FSM is a list of triples. Each triple is of the form (n, s, t), where n is a name of a node to be traced and s and t are the start and stop times for this trace respectively.

Of course, in practice, it would be quite tedious to have to write all specifications in terms of lists of five-tuples. Consequently, a small language (actually a small set of useful functions) has been defined in the library file "verification.fl". These functions make it much easier to write specification. However, it should be remembered that when the verification is actually performed, all these higher level constructs gets translated down to the two lists of five-tuples.

For more details on how to use the FSM function, we refer the reader to the tutorial section.

3 Syntax Summary

This is a (somewhat edited) version of the parser for FL. Since FL is still evolving, the actual syntax accepted by the program may differ slightly from this one. However, I have tried to make the grammar as close as possible to the parser in Voss 1.5.

```
/* Program */
pgm             : pgm ;  stmt
                | stmt

/* Statements */
stmt            : expr
                | decl
                | type_decl
                | print_all_fns
                | postfix VAR
                | nonfix VAR
                | infix NUMBER VAR
                | infixr NUMBER VAR
                | begin_adt
                | end_adt  var_list

var_list        : var_list  VAR
                | VAR

/* Function declarations */
decl            : let fn_defs
                | letrec fn_defs

fn_defs         : fn_def  /_  fn_defs
                | fn_def

fn_def          : VAR lhs_expr_list
                | ( VAR lhs_expr_list )

lhs_expr_list   : lhs_expr lhs_expr_list
                | = expr

lhs_expr        : NUMBER
                | T
                | F
                | []
                | STRING
                | VAR
                | ( lhs_expr0 )
                | [ expr_list ]

lhs_expr0       : lhs_expr1 ,  lhs_expr0
                | lhs_expr1

lhs_expr1       :  lhs_expr1 lhs_expr
                |  lhs_expr
```

```
/* Type declarations */
type_decl       : lettype  type_name = type_expr_list
                | new_type_abbrev = simple_type

type_name       : type_name ,  VAR
                | VAR

type_expr_list  : type_expr_list ,  type_expr
                | type_expr

type_expr       : type_expr |  type
                | type

type            : VAR type_list

type_list       : type_list simple_type
                | /* Empty */

simple_type     : VAR
                | simple_type -> simple_type
                | simple_type # simple_type
                | simple_type list
                | ( simple_type )

/* Expressions */
expr            : decl in expr
                | ! VAR . expr
                | ? VAR . expr
                | expr => expr | expr
                | _ VAR . expr
                | expr POSTFIX_VAR
                | expr INFIX_VAR expr
                | expr INFIXR_VAR expr
                | expr ,  expr
                | expr = expr
                | expr expr1
                | expr1

expr1           : [ expr_list ]
                | []
                | ( expr )
                | VAR
                | NUMBER
                | T
                | F
                | NIL
                | STRING
                | CONSTANT

expr_list       : expr ,  expr_list
                | expr
```

3.1 Reserved Words in FL

The following list contains all identifiers that are currently defined in FL. This list will likely change in future releases.

> **begin_abstype end_abstype forall_last in infix infixr let letrec lettype list new_type_abbrev nonfix postfix print_all_fns quit**

4 The .vossrc Default File

If the user puts a .vosrc file in his/her home directory or in the current directory, FL will read this file to set a number of defaults. Below we include a copy of the default .vossrc file which also include the acceptable alternatives.

```
############################
# Run time options for FL #
############################
#VOSS-LIBRARY-DIRECTORY =
#
# PRINT-ALIASES: should both primary node name and aliases be printed?
PRINT_ALIASES = TRUE
#
# PRINT-FORMAT for Boolean expressions: SOP (sum-of-products) INFIX TREE
PRINT-FORMAT = SOP
MAX-PRODUCTS-IN-SOP-TO-PRINT = 5
#
PRINT-TIME = TRUE
NOTIFY-OK_A-FAILURES = TRUE
NOTIFY-OK_C-FAILURES = TRUE
NOTIFY-TRAJECTORY-FAILURES = TRUE
NOTIFY-CHECK-FAILURES = TRUE
PRINT-FAILURE-FORMULA = TRUE
#
# Max number of steps to reach stability before setting to X
STEP-LIMIT = 100
#
# DELAY-MODEL is one of: UNIT-DELAY, MINIMUM-DELAY, MAXIMUM-DELAY,
#                        AVERAGE-DELAY, or BOUNDED-DELAY
DELAY-MODEL = UNIT-DELAY
```

5 Built-In Functions in FL

The following list contains all predefined functions in FL. The vast majority of these functions can be re-defined. In the list I also indicate whether the function is infix, whether it associates to the right and the precedence of the operator.

String manipulations

chr		Convert an integer to the ASCII character corresponding to it.
ord		Given a string returns the ASCII code for it.
explode		Convert string to list of single character strings.
implode		Takes a list of single character strings and catenates them together.
bool2str		Convert a Boolean to a string.
int2str		Convert integer to string for printing purposes.

General functions

catch	infix 2	Evaluate lhs, if it fails return e2 otherwise return result of lhs.
error		Fail and print out message.
empty		Applied to a list returns true if list is empty, false otherwise.
load		Re-direct standard input to this file;
print		Given a string, prints it out on stdout. Watch out for laziness!
time		Given an expression forces it to be completely evaluated and returns a pair of result, time pair.
seq	infix 1	Evaluate lhs first, throw away result and then evaluate rhs.

Boolean

<	infix 3	Less than.
<=	infix 3	Less than or equal.
==	infix 3	Identical.
!=	infix 3	Not equal.
>	infix 3	Greater than.
>=	infix 3	Greater than or equal to.
variable		Given a string returns the Boolean variable with this name.
AND	infix 4	Boolean conjunction.
OR	infix 3	Boolean disjunction
XOR	infix 3	Boolean exclusive or
NOT		Boolean negation.
!v.e		compute for all x in 0,1 e
?v.e		compute there is x in 0,1 e
bdd_size		Given a list of Boolean functions, returns the total size in number of BDD nodes
depends		Given a list of Boolean functions, returns a list of the Boolean variables the function depends on.
quant_forall		Universally quantify away all Boolean variables in the first argument from the expression in the second argument.
quant_thereis		Existentially quantify away all Boolean variables in the first argument from the expression in the second argument.
substitute		Applies a substitution to a Boolean expression.

Finite State Machine Manipulation

load_exe		Read in exe file and return the fsm.

make_fsm	Converts FL description of system into fsm.
nodes	Given fsm returns a list of node lists. Each node list consists of all aliases for the node.
fanin	Given fsm model and node name returns a list of node names the next state function of the node depends on.
fanout	Given fsm model and node name returns a list of node names the nodes that depend on the value of this node.
get_node_val	Given fsm model and node name returns the encoded version of the current value on the node.
print_fsm	Print out an internal representation of FSM. Pretty obscure.
FSM	Basic trajectory evaluation function.

Dealing with Cartesian Products

e1 , e2	Returns the tuple (e1, e2)
fst	Returns the first element in tuple.
snd	Returns the second component of tuple.

Dealing with Lists

hd		Returns the first element in a list.
tl		Returns the tail of a list. Note that tl [] = [].
:	infixr 2	Corresponds to the CONS operator in LISP.

Arithmetic Functions

	infix 4	Multiplication.
/	infix 4	Integer division.
+	infix 3	Integer addition.
-	infix 3	Integer subtraction.
^	infix 3	String catenation.

6 Standard Libraries

Since Fl is a very young language, there are no extensive standard libraries and thus this section is very tentative and is likely to be modified significantly in future releases. All standard libraries reside in the vosslib directory. The easiest way to make sure this directory is in the search path for fl is to create a file called *.vossrc* in your home directory that contains a line

```
VOSS-LIBRARY-DIRECTORY = /path/where/voss/is/installed/vosslib
```

6.1 default.fl

This is a basic library that contains many useful general functions.

length l	Returns the length of the list l.
append l1 l2	Appends lists l1 and l2. Note '@' and 'and' are infix aliases to append.
el i l	Select element i in list l. List elements are numbered from 1.
last l	Return the last element in list l.
butlast l	Return the list l with the last element removed.
replicate x n	Return a list with n copies of x as elements.
map fn l	Apply the function fn to each element of the list l and return the resulting list.
itlist f l x	Combine all the elements in l with the function f, i.e., f (hd l) (f (el 2 l) (f (el 3 l) (... (f (last l) x)))...).
rev_itlist f l x	As itlist, but do it in reverse.
find p l	Return the first element in the list that makes the predicate p true.
exists p l	Determine whether an element exists in the list l that satisfies the predicate p.
forall p l	Determine whether all elements in in the list l satisfies the predicate p.
mem x l	Determines whether there is an element in l equal to x.
assoc x al	Return the second component of a pair in the list l whose first component is equal to x.
rev_assoc x l	Same as assoc but exchange meaning of fst and snd.
rev xl	Reverse list xl.
filter p l	Returns the list obtained by removing from l every element that does not satisfy p.
flat ll	Takes a list of lists and return the list obtained by merging all the lists.
interleave ll	Takes a list of lists and returns a single list that is the interleaving of each list.
combine l r	Takes lists l and r and creates a list of pairs whose first components are drawn from l and whose second components are drawn from r.
split pl	Takes a list of pairs and returns a pair of lists. The first list are all first components of the pairs and the second list contain all second components of the pairs.
s1 intersect s2	Return the list of elements common to both s1 and s2.
s1 subtract s2	Return the list of elements that are in s1 but not in s2.
s1 union s2	Return the union (no duplicates) of s1 and s2.
distinct l	Determines whether there are any duplicates in the list l. If so, returns false; otherwise returns true.
setify l	Take a list and make it into a set (no duplicates).

s1 set_equal s2	Determines whether the two sets s1 and s2 are equal.
declare vl	Takes a list of Boolean variables and forces them to be evaluated in the order they appear in the list. Useful in declaring Boolean variables for the OBDD ordering.
num2str n	Converts a number (positive) to a string.
lg n	Computes the number of bits requires to represent n as an unsigned binary number.
bdd_profile expr_list	Takes a list of Boolean expressions and prints out a histogram over the width of the combined OBDD forest.
excitation ckt nd	Computes the next state function for node nd in circuit ckt. *NOTE:* this function only works correctly when using the UNIT-DELAY model.
node_profile ckt node_list	Prints out a bdd profile for all the current values on the nodes in node_list. Mostly useful in conjunction with the '-m n' option to FSM (i.e., abort the simulation at a suitable time and check the size and profile of the OBDDs on the selected nodes.

6.2 verification.fl

This is the basic verification library that contains useful functions to make writing verification conditions much more convenient. It should be noted that this is an abstract data type so not all functions defined in the library are exported. In order to shorten the typing information, we use the following shorthand: voss_tuple = (bool, (string, (bool, (int, int)))). All these functions, with the exception of node_vector, variable_vector, verify and nverify returns lists of five-tuples, of the form described in the description of FSM. Briefly, the functions are as follows:

UNC	Unconstrained. Useful as padding when writing functions generating verification conditions.
n is v	Node n is asserted/checked to have the value v with guard true.
nv isv vv	Node list nv is asserted/checked to have the values in the value list vv with guard true.
node_vector s n	Create a list of strings of the form s catenated with the string representing integer i, where i goes from (n-1) to 0.
variable_vector s n	Create a list of Boolean variables of the form s catenated with the string representing integer i, where i goes from (n-1) to 0. Note that these variables are not declared until they are forced to be evaluated. See declare in defaults.fl for a function to do so.
vl when e	Imposes the domain constraint denoted by the Boolean expression e on all the five-tuples in vl.
vl from t	Set all the starting times in the five-tuples in vl to t.
vl to t	Set all the ending times in the five-tuples in vl to t.
during f t vl	Set all the starting and ending times in the five-tuples in vl to f and t respectively.
vl1 then vl2	Merge the lists together, but adjust the durations for the five-tuples in vl2 so that the "time 0" for vl2 is equal to the maximum time of any five-tuple in vl1
vl for t	Same as 'to' above.
verify fsm l ant cons trl	An old shorthand for (declare l) seq (FSM "" fsm ant cons trl. Probably should be removed.
nverify fsm l ant cons trl	Same as verify but the first argument is the option string to FSM. Again, should be viewed as obsolete.

SymbIndex nl addr fn	Symbolic indexing function. Will apply the function fn to every element i in nl and then apply a when condition to each result that requires addr to be equal to i.

6.3 arithm.fl

This is a library of bitvector functions. A bitvector is represented as a list of Booleans and is viewed as a big-endian vector, i.e, the head of the list is the most significant bit.

num2bv sz n	Convert the integer n to a bitvector of size sz.
bv2num bv	If the bitvector bv does not contain any Boolean variables, view it as an unsigned binary number and convert it to a number.
prefix n av	Return the n first elements of av.
suffix n av	Return the n last elements of av
av add bv	Add the two bitvectors together.
increment av	Add one to the bitvector av.
ones_complement av	Return the 1's complement of the bitvector av.
twos_complement av	Return the 2's complement of the bitvector av.
av subtract bv	Subtract bitvector bv from bitvector av.
av greater bv	Compute the Boolean expression for the number denoted by av is greater than the number denoted by bv. Both bitvectors are viewed as unsigned integers.
av equal bv	As for greater, but for equality.
av geq bv	As for greater, but for greater than or equal to.
av less bv	As for greater, but for less than.
av leq bv	As for greater, but for less than or equal to.
av bvAND bv	Bit-wise AND.
av bvOR bv	Bit-wise OR.
av bvXOR bv	Bit-wise XOR.
bvNOT av	Bit-wise NOT.

6.4 HighLowEx.fl

This library defines only two funcitons: Hexpl and Lexpl. The basic task of both is to take a Boolean function and return an assignments to some set of variables that would make the Boolean function evaluate to true. The two functions differ only in that Hexpl tries to find an assignment with as many 1's as possible, whereas Lexpl tries to assign as many 0's as possible. In order to make the output more readable, both functions take as first argument a list of pairs. The first element in the pair is a string and the second argument is a list of Boolean variables. The string will be used as a header for the assignemnts to the list of variables. For example, is i, Aa, Ab, a, b, d, and q denote lists of Booelan variables, and f denote some Boolean expression over these (and possibly other) variables, then we may get:

```
: Lexpl [("I",i),("Aa",Aa),("Ab",Ab),("a",a),("b",b),("d",d),("q",q)] f;
"
I = 001011000
Aa = 0000
Ab = 0001
a = 1111
b = 0000
d = 0000
q = ----

"
```

where 0's and 1's denote assignemnt to the corresponding variables to make f evaluate to 1, whereas $-$ denote don't care conditions.

7 Examples of Using the Voss System

In this section we will give some examples of symbolic trajectory evaluation and how the Voss system can be used for other verification tasks. The specification code and circuit descriptions for these examples are available in the directory demos in the Voss demo distribution.

7.1 AMD2901

Our first example of symbolic trajectory evaluation is the verification of two different[5] descriptions of the 2901 ALU bitslice from Advanced Micro Devices. In Fig. 2 we show a high level schematic of the circuit. As can be seen in the figure, the circuit contains both some non-trivial combinational circuitry and a fair amount of state storing registers. Of course, it is not a very complex design, but it is not an altogether trivial one either. For a typical, fairly informal, specification of the design, we refer the reader to Appendix A.

In this section we will go through the verification, and its various alternatives, in a fair amount of detail. In the later parts of the document, we will only highlight some specific characteristics for the other verification tasks.

Our first verification task is to verify that a behavioral VHDL model is correct. In particular, we will discuss how to derive the fsm model of the design, how to structure a specification/verification file, how to debug the circuit (and the specification!), and how to deal with the issue of variable ordering. The code for the tutorial is available in the directory demos/AMD2901/behavioral_VHDL in the demo distribution.

7.1.1 Creating the fsm Model

Before we can verify the circuit, we must obtain an fsm model that can be used in the symbolic trajectory evaluation. In this case, the behavioral VHDL model is defined in the file amd2901_beh.vbe. We are using the convention that behavioral VHDL models have filenames ending with .vbe, structural VHDL files have suffix .vst, EDIF files have suffix .edi, and sim and ntk files have suffices .sim and .ntk respectively. In this case, we are using the behavioral VHDL model. For a more detailed discussion on what subset of VHDL that is supported, we refer the reader to Appendix F.

The program convert2fl can be used to convert the .vbe (as well as .vst and .edi) file to a .fl file, ready to be loaded into FL.

```
% ls
amd2901_beh.vbe  spec.fl
% convert2fl amd2901_beh.vbe
% ls
amd2901_beh.fl   amd2901_beh.vbe  spec.fl
```

It is worth looking at the amd2901_beh.fl file a bit closer. The file looks roughly as shown in Fig. 3:

The file begins by encapsulating all the definitions inside an abstract data type. By only exporting the final result in the end, we effectively achieve information hiding. We then load in the library EXE.fl that contains all the functions used to create an object of type Set that eventually will be used to create the fsm object. The file then contains a large number of definitions used to create names that are easier to use in the translation. Finally, the real set of next state functions are given. There are four basic types of nodes: input, output, bus, and register nodes. An input

[5] We actually have three different netlist descriptions of the circuit: behavioral VHDL, structural VHDL, and EDIF. All of these are taken from the Alliance 1.1 distribution.

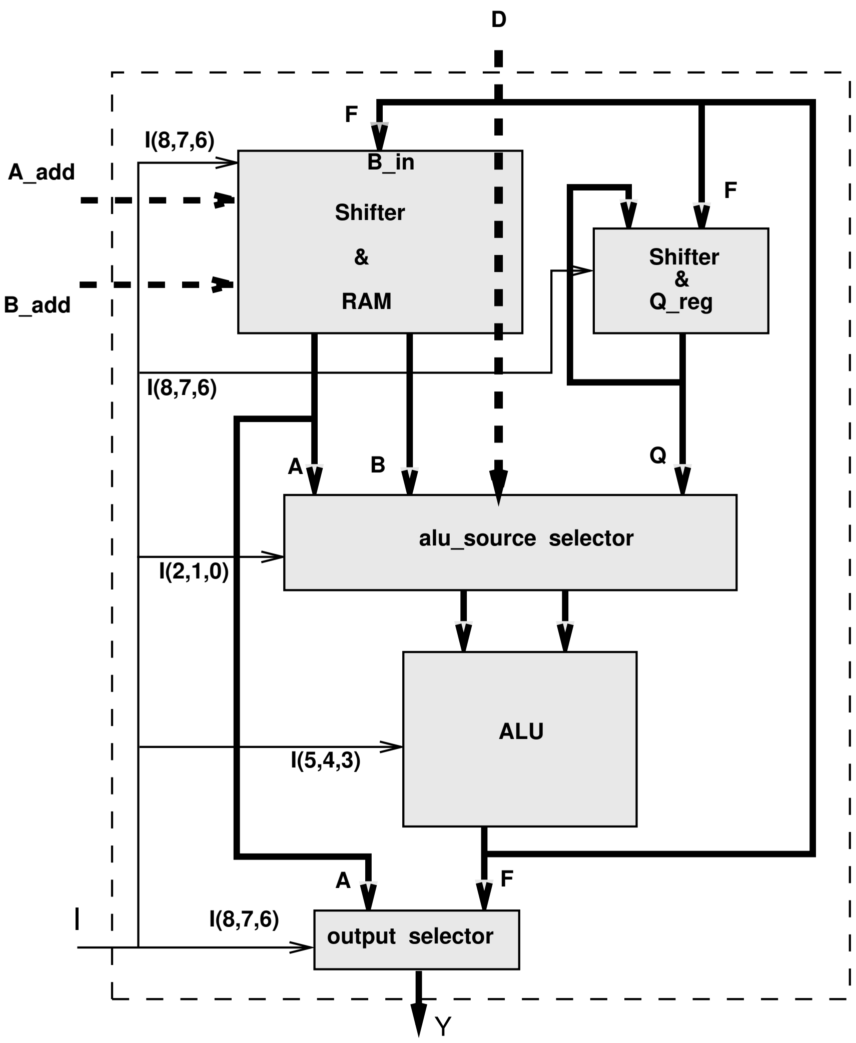

Figure 2: Block diagram of AMD 2901

```
begin_abstype;
load "EXE.fl";


// Behavioral Description for Entity amd


let VSSP = "vssp";
let VDDP = "vddp";
let VSS = "vss";
let VDD = "vdd";
let Y_0_ = "y[0]";
...
let ACCU_3_ = "accu[3]";
Set let AMD =
        (Output (COUT,
        (((Val C_SUMRS_3_) And (Not (Val I_3_) And Not (Val I_4_)
         And Not (Val I_5_))) Or (((Val C_DIFSR_3_) And ((Val I_3_)
         And Not (Val I_4_) And Not (Val I_5_))) Or ((Val C_DIFRS_3_)
         And (Not (Val I_3_) And (Val I_4_) And Not (Val I_5_)
        ))))))|_|
...
        (BusDrv(Y_1_, [(
        Not (Val NOE),
        (((Val RA_1_) And (Not (Val I_6_) And (Val I_7_) And
        Not (Val I_8_))) Or (((((((Val ALU_OUT_1_) And (Not (Val I_6_)
         And Not (Val I_7_) And Not (Val I_8_))) Or ((Val ALU_OUT_1_)
         And ((Val I_6_) And Not (Val I_7_) And Not (Val I_8_)
        ))) Or ((Val ALU_OUT_1_) And ((Val I_6_) And (Val I_7_)
         And Not (Val I_8_)))) Or ((Val ALU_OUT_1_) And (Not (Val I_6_)
         And Not (Val I_7_) And (Val I_8_)))) Or ((Val ALU_OUT_1_)
         And ((Val I_6_) And Not (Val I_7_) And (Val I_8_)))
        ) Or ((Val ALU_OUT_1_) And (Not (Val I_6_) And (Val I_7_)
         And (Val I_8_)))) Or ((Val ALU_OUT_1_) And ((Val I_6_)
         And (Val I_7_) And (Val I_8_))))))]))|_|
...
        (RegDrv(ACCU_3_, [(
        (Not (Val WCKACCU) And Not (Stable WCKACCU)),
        Not (Val ACCU_IN_3_))]));

let AMD = make_fsm AMD;
end_abstype AMD;
```

Figure 3: Structure of FL file obtained by convert2fl.

node has no next state function given, and thus will always become X unless the value on the node is asserted. An output node always take on the value of the next state function. A bus node has several drivers and takes on the value X if no-one is driving the node. Finally, a register node can also have several drivers. However, if no driver is active, it keeps its latest value. If there are more than one driver active for a bus or register node, the resulting value will be the greatest lower bound of the different values being driven. In other words, if all active drivers agree on the value, this will be the value of the next state function. On the other hand, if the active drivers disagree on the value, the next state will be X.

Once all nodes, and their associated next state functions, have been given, the FL program proceed to convert the FL Set object to an fsm object by invoking make_fsm. Finally, only the obtained fsm object is exported out from the abstract data type, and thus, loading amd2901_beh.fl will only define the fsm object named AMD.

7.1.2 Structuring the Specification File

Although strictly speaking not necessary, it has been our experience that following a fairly standardized style when writing a specification/verification program helps both in debugging as well as in breaking down the verification task into manageable sub tasks. The format we will describe here has been used quite successfully in teaching the Voss system to a couple of graduate classes and appears to work well.

Before we go into the details of the structure of the file, it is worth spending a moment discussing how to work with the fl system. The typical work mode in a window environment (or emacs) is to develop the specification script in a standard text editor and then cut and paste the code into a running copy of fl to make sure no syntax errors slip by and also to try things out. In general, for programmers used with compiled languages, the largest hurdle to overcome is the idea that you should *not* write the whole program before testing parts of it. In fact, it is often useful to test every new function defined with some arguments just to make sure they appear to be correct. Also, quitting the fl session once-in-a-while and reloading the definitions directly from the edited file, ensures that no definitions gets forgotten to be put it.

The structure of a specification/verification file is broadly divided into the following sections:

1. Loading of needed library files and circuit model(s).

2. Defining short hands for the actual names of nodes in the circuit that needs to be asserted/checked.

3. Defining the clocking scheme. In particular, define the length of a clock cycle, set-up and hold times etc.

4. Define timing and node abstraction functions that allows the high-level specification to be stated in terms of abstract entities, rather than be cluttered with details that are largely irrelevant.

5. Declaring the Boolean variables needed in the verification process. This also includes some function for declaring the variables in some suitable order to achieve acceptable OBDD performance.

6. High-level specification functions that denotes the desired behavior of the system.

7. Verification conditions in the form of antecedent and consequent definitions.

Loading the library files and the circuit model is usually quite straightforward. In our case we simply have:

```
load "verification.fl";
load "arithm.fl";

// ----------------------------------------------------------------
// Load fsm model (called AMD)
load "amd2901_beh.fl";
```

Finding the names of circuit nodes is usually tedious, but not overly difficult. Also, it is highly dependent on the source of the circuit and how well it is documented. One major practical difference between traditional simulation and trajectory evaluation, is the need to find names on internal state storing elements in the circuit. In other words, finding the names of the nodes in some of the latches that store important state. It should be emphasized though that the registers that has to be exposed are the ones that naturally would be discussed when describing the behavior of the circuit. Thus, fortunately, it is often the case that some of the internal registers in the control logic never need to be exposed.

In our case, we need to find the names of all inputs, outputs, and all the names of the RAM cells and the Q register cells. There are several ways of doing this, but a combination of looking at the .vbe and the translated .fl file in addition to loading the file and giving the command: `nodes AMD`, will suffice to find the names of the nodes. In Fig. 4 we show the definitions we use. Note that we actually explicitly listed all the nodes. If there are very many nodes, it is often more convenient to define a function that creates these lists of nodes. For example, we could have replaced the big listing of all the RAM cell names with the (equivalent) definition

```
let ram =
        let rv = rev (node_vector "ram" 16) in
        let mk_nd_vec name = [name^"[3]",name^"[2]",name^"[1]",name^"[0]"] in
        map mk_nd_vec rv;
```

Defining the clocking scheme and timing parameters are often relatively straightforward as well. When using a unit delay model, the only constraint on the length of the cycle is usually that it must be long enough for the circuit to stabilize between consecutive clock signals. Of course, if a more accurate delay model is used, the exact value of the clock cycle must be in terms of the basic time unit that was used in the circuit description. In the alpha release of Voss, only circuit descriptions in Silos netlist format can use average, minimum, maximum and bounded delay timing models[6] and thus usually the values for clock cycle length etc. are fairly arbitrary. For our example, we use the following definitions:

```
// Clocking scheme
let PHASE     = 50;
let CYCLE     = 2*PHASE;
let LATCH_DEL = 5;
let HOLD      = 5;
let cycle n   = (n-1)*CYCLE;
let phiL n    = (n-1)*CYCLE;
let phiH n    = (n-1)*CYCLE+PHASE;
```

Note that we also define some convenient functions for abstracting the time references. Thus, we define functions that map from an abstract cycle count and relative position in the clock cycle,

[6]Actually, switch-level models can also use this feature if the .exe file is accompanied by a .del file that contains min/max rise and fall delays for all the nodes in the circuit. For more details, the reader is referred to page 65.

```
// Short-hands for circuit nodes
let I     = ["i[8]", "i[7]", "i[6]", "i[5]", "i[4]",
             "i[3]", "i[2]", "i[1]", "i[0]"];
let Aadd  = ["a[3]", "a[2]", "a[1]", "a[0]"];
let Badd  = ["b[3]", "b[2]", "b[1]", "b[0]"];
let D     = ["d[3]", "d[2]", "d[1]", "d[0]"];
let Y     = ["y[3]", "y[2]", "y[1]", "y[0]"];
let RAM0  = "r0";
let RAM3  = "r3";
let Q0    = "q0";
let Q3    = "q3";
let CLK   = "ck";
let C0    = "cin";
let OEbar = "noe";
let C4    = "cout";
let Gbar  = "ng";
let Pbar  = "np";
let OVR   = "ovr";
let F3    = "signe";
let F30   = "zero";

let FUNC  = "fonc";
let TEST  = "test";

// Names of register nodes
let Q   = [  "accu[3]", "accu[2]", "accu[1]", "accu[0]"  ];
let ram = [ ["ram0[3]", "ram0[2]", "ram0[1]", "ram0[0]"],
            ["ram1[3]", "ram1[2]", "ram1[1]", "ram1[0]"],
            ["ram2[3]", "ram2[2]", "ram2[1]", "ram2[0]"],
            ["ram3[3]", "ram3[2]", "ram3[1]", "ram3[0]"],
            ["ram4[3]", "ram4[2]", "ram4[1]", "ram4[0]"],
            ["ram5[3]", "ram5[2]", "ram5[1]", "ram5[0]"],
            ["ram6[3]", "ram6[2]", "ram6[1]", "ram6[0]"],
            ["ram7[3]", "ram7[2]", "ram7[1]", "ram7[0]"],
            ["ram8[3]", "ram8[2]", "ram8[1]", "ram8[0]"],
            ["ram9[3]", "ram9[2]", "ram9[1]", "ram9[0]"],
            ["ram10[3]", "ram10[2]", "ram10[1]", "ram10[0]"],
            ["ram11[3]", "ram11[2]", "ram11[1]", "ram11[0]"],
            ["ram12[3]", "ram12[2]", "ram12[1]", "ram12[0]"],
            ["ram13[3]", "ram13[2]", "ram13[1]", "ram13[0]"],
            ["ram14[3]", "ram14[2]", "ram14[1]", "ram14[0]"],
            ["ram15[3]", "ram15[2]", "ram15[1]", "ram15[0]"] ];
```

Figure 4: Definition of short-hands for the node names.

to the actual circuit time. For example, the function phiH maps an abstract cycle count n to the actual circuit time when the clock signal goes high in cycle n.

The next part of the specification file—abstraction functions for inputs, outputs, and latch signals—is often the most difficult to get right. In particular, unless the design is using a very well defined clocking methodology, it is usually non-trivial to determine on what node at what time a latch is "storing its value". However, there is a very useful trick in determining this information—simulation. We will return to this topic when we discuss how to debug circuits and specifications.

For our verification effort, we first define a clocking function that, given a parameter n, will create the assertion list needed for assuming that the input signal "ck" (named CLK above) takes on the proper values at the proper times. To illustrate the definition, below we include both the definition of clock_cyc, as well as an example of applying clock_cyc to the argument 2.

```
: letrec clock_cyc n = (n = 0) => UNC |
    (CLK is F from (phiL n) to (phiH n))@
    (CLK is T from (phiH n) to (cycle (n+1)))@
    (clock_cyc (n-1));
clock_cyc::(int) -> (((bool # (string # (bool # (int # int)))) list))
: clock_cyc 2;
[(T,"ck",F,100,150),(T,"ck",T,150,200),(T,"ck",F,0,50),
 (T,"ck",T,50,100),(F,"",F,0,0)]
```

The input and output signal abstraction functions are fairly straighforward. The only subtle point is that they need to take set-up and hold times into account. In this case, we can get by with a set-up time of 0.

```
// Input signals timing
let inB cyc = (phiL cyc)+HOLD;
let inE cyc = (phiL (cyc+1))+HOLD;
let AaddIs addr cyc = Aadd isv addr from (inB cyc) to (inE cyc);
let BaddIs addr cyc = Badd isv addr from (inB cyc) to (inE cyc);
let D_is val cyc = D isv val from (inB cyc) to (inE cyc);
let I_is val cyc = I isv val from (inB cyc) to (inE cyc);
let C0_is val cyc = C0 is val from (inB cyc) to (inE cyc);
let Q0_is val cyc = Q0 is val from (inB cyc) to (inE cyc);
let Q3_is val cyc = Q3 is val from (inB cyc) to (inE cyc);
let RAM0_is val cyc = RAM0 is val from (inB cyc) to (inE cyc);
let RAM3_is val cyc = RAM3 is val from (inB cyc) to (inE cyc);

// Output signals timing
let Y_is val cyc = Y isv val from (phiH (cyc-1)) to (phiL cyc);
```

The abstraction functions for the latches is more intricate. There are three properties an abstraction function must answer:

1. On what node(s) is this value stored.
2. When is the value stable.
3. What encoding is used.

If we first look at the accumulator register definition Q_is,

```
let Q_is val cyc = Q isv (bvNOT val)
    from ((phiL cyc)+LATCH_DEL) to ((phiL cyc)+LATCH_DEL)+1;
```

we can see that the nodes that correspond to these signals are called "accu[3]", "accu[2]", "accu[1]", and "accu[0]". Furthermore, the signals are stable from the phiL plus latch delay for one time unit. Finally, the values are actually stored complemented on the node.

```
let RamIs addr val cyc =
    let ram_is n = (n isv (bvNOT val))
                   from ((phiL cyc)+LATCH_DEL) to ((phiL cyc)+LATCH_DEL+1) in
    SymbIndex ram addr ram_is;
```

The RamIs function above illustrates a more sophisticated abstraction mapping. The function takes three arguments: a 4-bit address, a value to be asserted/checked, and the abstract cycle in which the addressed nodes should take on this value. What makes the function more involved is that the address argument can be a vector of Boolean functions. Thus, at simulation time it may be impossible to determine which RAM cell is intended. More precisely, the address may refer to more than one location depending on the assignments to some set of Boolean variables. This situation is a typical example of *symbolic indexing*—selecting an element in a list by providing a symbolic address. The solution to this common case is to use the when condition in a subtle way. Rather than trying to figure out which node is actually referred to by the address, we will create a five-tuple assertion/check for *each* node in the list. However, the five-tuple for cell i will have as its guard a Boolean expression that is only true for interpretations in which the number represented by the address vector equals i.

To better illustrate symbolic indexing, consider a somewhat simpler example. Suppose we have a list with four nodes: a0, a1, a2, and a3. Assume furthermore that we would like to say that the node on address i should be asserted/checked to take on the Boolean value u for 100 time units. However, the address i is given as a bitvector and may contain Boolean variables. Thus depending on the values on these Boolean variable, we may in fact select different nodes in the list. Here we show how SymbIndex can be used to derive a list of five-tuples that indeed matches this intuition.

```
: let ex_arr = ["a0", "a1", "a2", "a3"];
ex_arr::(string list)
: SymbIndex ex_arr [F,T] (\n. n is (variable "u") for 100);
[(F,"a0",u,0,100),(T,"a1",u,0,100),(F,"a2",u,0,100),(F,"a3",u,0,100)]
: SymbIndex ex_arr [variable "i1", variable "i0"]
      (\n. n is (variable "u") for 100);
[(i0'&i1',"a0",u,0,100),
 (i0&i1',"a1",u,0,100),
 (i0'&i1,"a2",u,0,100),
 (i0&i1,"a3",u,0,100)]
```

Note that both invocations result in one five-tuple for each node in the list. However, the guard expression differ. In fact, for the first example, where the address is fully defined, all but one node have their guard equal to false. For the second example, the guard for node i is a boolean expression that must hold for the address to be equal to i.

After the abstraction functions are defined, we go on to introduce the Boolean variables needed for the verification. In general, since the complexity of the verification task depend very greatly on the number of Boolean variables, it is often extremely useful to formulate the correctness statement in such a way that it minimizes the number of Boolean variables needed. The verification of the AMD2901 contains an excellent example of this. Consider verifying that the built-in RAM gets updated properly. One way of doing this would be to assert that each RAM cell contained a distinct Boolean variable before an instruction is performed and that every memory cell not addressed in the operation keep its value and the destination register(s) take on their proper values. However, this would require at least $16*4+4 = 70$ Boolean variables. On the other hand, we could rephrase the correctness statement in the following way: Suppose RAM-cell i, for some arbitrary address i between 0 and 15, contains some value u, then after performing an operation, the content of word i should still be u, unless i was the destination address of the operation in which case word i should contain the result of the computation. If we now represent the address i as a vector of four Boolean variables, we will be able to carry out the verification using only $4+4 = 8$ variables—a reduction by more than 60 variables! In general, this approach of using symbolic indexing and Boolean variables to select the different cases, is the single most powerful aspect of symbolic trajectory evaluation. In general, it allows us to verify properties of circuits much larger than what more traditional symbolic model checking algorithms can handle.

Returning to our example, we have chosen not to completely minimize the number of Boolean variables used, but rather keep the number small, except when it is more convenient to use a larger number of variables. In particular, we use a fully symbolic version of the instruction word, input addresses, and what is stored in the two addresses and what is currently stored in the accumulator register.

```
// Boolean variables
// Instruction
let i = variable_vector "i" 9;
// Addresses
let Aa = variable_vector "Aa" 4;
let Ab = variable_vector "Ab" 4;
// Data
let a = variable_vector "a." 4;
let b = variable_vector "b." 4;
let d = variable_vector "d." 4;
let q = variable_vector "q." 4;
let c = variable "c";
let q0 = variable "q0";
let q3 = variable "q3";
let ram0 = variable "ram0";
let ram3 = variable "ram3";
```

The next task to accomplish is to force the evaluation of these variable declarations so that a suitable variable ordering for the OBDD routines is accomplished. This task is fairly ad-hoc. However, there are some “rules-of-thumb” that appears to work well. First of all, any variable vectors that will be added or subtracted should have their variables interleaved in the variable ordering with their most significant bits first. Secondly, variables that greatly affect the control actions of the system should appear early in the ordering. One point making is that one does not have to declare an ordering for all variables since an undeclared variable will eventually be declared automatically when it is used. However, by declaring the variables, the user retains control over the ordering and thus can more easily vary the ordering if that is deemed necessary. In our case, the variable ordering we selected it pretty much the obvious first attempt. In Section 7.1.5 we return to this topic with some techniques that can be helpful in determining acceptable variable orderings. Again we see the benefits of using as few Boolean variables as possible—here fewer variables have to be ordered.

```
// Variable ordering (could be tuned!)
let var_order = i @ (interleave [a,b,d,q]) @ (interleave [Aa,Ab]);
declare var_order;
```

We are now ready for defining the *desired* behavior of the circuit. Here we are separating the abstract functionality description from the actual timing of the various signals. Thus we start by defining functions that denote the desired behavior of the circuit. In order to make this functional specification as readable as possible, we begin by introducing some helpful functions.

```
// Useful help functions
let getALUsrc  [I8,I7,I6,I5,I4,I3,I2,I1,I0] = [I2,I1,I0];
let getALUfun  [I8,I7,I6,I5,I4,I3,I2,I1,I0] = [I5,I4,I3];
let getALUdest [I8,I7,I6,I5,I4,I3,I2,I1,I0] = [I8,I7,I6];
let ALUsrc = getALUsrc i;
let ALUfun = getALUfun i;
let ALUdest = getALUdest i;
let member iv lv = itlist (\e.\r. (iv equal e) OR r) lv F;
let ITEv c t e = (map (\v. c AND v) t) bvOR (map (\v. (NOT c) AND v) e);
```

Next we define functions that are direct translations of the various tables used in the informal specification of the AMD2901, as given in Appendix A. Thus we first define two functions that compute what the arguments to the ALU should be.

```
let RE = ITEv (member ALUsrc [[F,F,F],[F,F,T]]) a
        (ITEv (member ALUsrc [[F,T,F],[F,T,T],[T,F,F]]) [F,F,F,F]
        (d));

let S  = ITEv (member ALUsrc [[F,F,F],[F,T,F],[T,T,F]]) q
        (ITEv (member ALUsrc [[F,F,T],[F,T,T]]) b
        (ITEv (member ALUsrc [[T,F,F],[T,F,T]]) a
        ([F,F,F,F])));
```

Next, we take these results and apply the proper function to compute the desired output of the operation. There is only one subtle point in this example: it would have been tempting to use the name F for the result and in fact FL allows you to do so. However, that would mean that there would be no way to refer to "false" after re-defining F. Consequently, we use `Fr` as the name of the result.

```
let Fr = ITEv (ALUfun equal [F,F,F]) (RE add S add [F,F,F,c])
        (ITEv (ALUfun equal [F,F,T]) ((bvNOT RE) add S add [F,F,F,c])
        (ITEv (ALUfun equal [F,T,F]) (RE add (bvNOT S) add [F,F,F,c])
        (ITEv (ALUfun equal [F,T,T]) (RE bvOR S)
        (ITEv (ALUfun equal [T,F,F]) (RE bvAND S)
        (ITEv (ALUfun equal [T,F,T]) ((bvNOT RE) bvAND S)
        (ITEv (ALUfun equal [T,T,F]) (RE bvXOR S)
        (bvNOT (RE bvXOR S))))))));
```

We are now ready to state the various correctness statements and the verification conditions that we wish to check. This is done by defining a collection of antecedent/consequent pairs. Intuitively, one can view an antecedent/consequent pair (A,C) as saying: if, during the lifetime of this system, we ever encounter a sequence of states satisfying the formula A, then that same sequence of states should also satisfy the formula C. In our case, two antecedent/consequent pairs suffice: the first one deals mostly with that all "good" things happen as they should, the second verifies that no "bad" things happen.

Intuitively, the first assertion is of the form:

> Assume the circuit is clocked properly and all the inputs signals take on their respective values at the correct time. Assume also that the address inputs are Aa and Ab respectively and that word Aa in the RAM contains the word a and that word Ab in the RAM contains the word b. Finally, assume the accumulator contains the value q. Then, one cycle later, depending on the destination field of the instruction being executed, the output, the accumulator, or the word Ab in the RAM will hold the proper values.

There is one subtle point in the above formulation. What if Aa is equal to Ab? In other words, what happens if both address lines point to the same word in memory. Clearly, this must mean that a and b must be equal (since a and b are meant to represent the current values in words Aa and Ab in the RAM). We deal with this subtle point by defining a "consistent" predicate that we use as a guard in several "when" conditions to ensure inconsistent assignments are ignored. More specifically, we define:

```
let consistent = (NOT (Ab equal Aa)) OR (a equal b);
```

With this in place, we define the first antecedent as:

```
let ant1 = (FUNC is T for (cycle 3)) @
           (TEST is F for (cycle 3)) @
           (OEbar is F for (cycle 3)) @
           (clock_cyc 2) @
           (I_is i 1) @
           (AaddIs Aa 1) @
           (BaddIs Ab 1) @
           (RamIs Aa a 1) @
           ((RamIs Ab b 1) when consistent) @
           (D_is d 1) @
           (Q_is q 1) @
           ((C0_is c 1) when (member ALUfun [[F,F,F],[F,F,T],[F,T,F]])) @
           ((Q3_is q3 1) when (ALUdest equal [T,F,F])) @
           ((Q0_is q0 1) when (ALUdest equal [T,T,F])) @
           ((RAM3_is ram3 1) when (member ALUdest [[T,F,F],[T,F,T]])) @
           ((RAM0_is ram0 1) when (member ALUdest [[T,T,F],[T,T,T]]));
```

and the first consequent:

```
let cons1 = (
             // Check outputs
             ((Y_is Fr 2) when (NOT (ALUdest equal [F,T,F]))) @
             ((Y_is a 2) when (ALUdest equal [F,T,F])) @
             // Check accumulator
             ((Q_is Fr 2) when (ALUdest equal [F,F,F])) @
             ((Q_is (q3:(butlast q)) 2) when (ALUdest equal [T,F,F])) @
             ((Q_is ((tl q)@[q0]) 2) when (ALUdest equal [T,T,F])) @
             ((Q_is q 2) when
                   (member ALUdest [[F,F,T],[F,T,F],[F,T,T],[T,F,T],[T,T,T]])) @
             // Check RAM
             ((RamIs Ab b 2) when (member ALUdest [[F,F,F],[F,F,T]])) @
             ((RamIs Ab Fr 2) when (member ALUdest [[F,T,F],[F,T,T]])) @
             ((RamIs Ab (ram3:(butlast Fr)) 2)
                          when (member ALUdest [[T,F,F],[T,F,T]])) @
             ((RamIs Ab ((tl Fr)@[ram0]) 2)
                          when (member ALUdest [[T,T,F],[T,T,T]]))
            ) when consistent;
```

We then call the FSM function to carry out the symbolic trajectory evaluation and return the expression for which this assertion to holds in the circuit AMD. Thus, we define:

```
let check1 = FSM "" AMD ant1 cons1 [];
```

For the "nothing bad happens" verification, we specify a simpler assertion and calls FSM.

```
// If Ram[Aaddr]=a / ((NOT load_regfile) OR (Baddr != Aaddr))
//   then we should have Ram[Aaddr]=a one cycle later
let ant2 = (FUNC is T for (cycle 3)) @
           (TEST is F for (cycle 3)) @
           (clock_cyc 2) @
           (I_is i 1) @
           (BaddIs Ab 1) @
           (RamIs Aa a 1);

let cons2 = (RamIs Aa a 2) when ((member ALUdest [[F,F,F],[F,F,T]]) OR
                                  NOT (Aa equal Ab));

let check2 = FSM "" AMD ant2 cons2 [];
```

Finally, we get the final correctness statement, in which we require both ckeck1 and check2 to hold:

```
check1 AND check2;
```

7.1.3 Carrying out the Verification

Once the specification script has be written, running the verifier is straightforward: simply load the file into FL. In our case we would get an output like the one shown in Fig. 5.

As can be seen, the final result of the verification is T, indicating that indeed the circuit satisfies both verification conditions *for every possible assignment to the Boolean variables.* Since we are using 38 Boolean variables, we actually verify more than 10^{11} different antecedent/consequent pairs with this verification run!

Before we go into how to find errors in the circuit and/or specification, it is worth while explaining the output of the verification process. First a work about garbage collection. There are two types of user-visible garbage collections: garbage collection related to executing the functional language, and garbage collection related to the OBDD representation. The first one is virtually always very quick and can usually be ignored. The second type, OBDD garbage collection is much more time consuming. It is often a sign of a poor variable ordering. However, a single OBDD garbage collection is usually acceptable. If there are more than one during a single verification attempt, experimenting with different variable orderings can pay off handsomely.

During the verification process[7] the system outputs a period for every unit delay that it has to run the simulator. Since the simulator and the assert/check procedures are using event-scheduling, the system will often reach "stable" states where nothing is going to happen until the next change in an assert or check. When the system reaches such a state, it prints out a "stable at time" message, and jumps ahead to the next "interesting" point in time. Finally, at every time an assert or check event occurs, the system will print out a "Time:" message. Although these periods and time commands convey relatively little information, they are often very useful in gouging the progress of the verification process. In fact, they are often the first sign of poor variable orderings, since the simulation process appears to have ground to a halt. When this happens, it is often useful to abort the run, restart it but with a "-m' command to the FSM function to abort the simulation run in a proper manner, and use some of the OBDD profiling functions that are defined in the dafault.fl library. We will return to this later.

7.1.4 Debugging a Design and/or Specification

Although the above script looked quite simple and straightforward and the verification only took a few minutes (if even that), clearly it is not always this easy. In fact, in practice, what counts more

[7] Actually during the evaluation of the FSM command.

```
% fl -f spec.fl
      /\
     /  \/\
 /\ /      \
/  Voss 1.5 \
VOSS-LIBRARY-DIRECTORY = /isd/local/generic/lib/vosslib

-Loading file spec.fl
-Loading file verification.fl
: T
-Loading file defaults.fl
T
-Loading file arithm.fl
T
-Loading file defaults.fl
T
-Loading file amd2901_beh.fl
T
-Loading file EXE.fl
T
Start garbage collection ...(Used=21138(Shared=303) Freed:33616)...done
T
"Verify the circuit"
..Start garbage collection ...(Used=24543(Shared=814) Freed:46596)...done
......  stable at time 3
Time: 5
.Time: 6
..........  stable at time 15
Time: 50
....  stable at time 53
Time: 100
...Start garbage collection ...(Used=24458(Shared=738) Freed:46681)...
Start bdd garbage collection.
Start with: 189328(189327) bdd nodes in use
Finished bdd garbage collection.
Currently: 136506(136506) bdd nodes in use
done
..Time: 105
.Time: 106
....  stable at time 109
Time: 150
....  stable at time 153
Time: 200
......  stable at time 3
Time: 5
.Time: 6
..........  stable at time 15
Time: 50
....  stable at time 53
Time: 100
.....Time: 105
.Time: 106
....  stable at time 109
Time: 150
....  stable at time 153
Time: 200
T
```

Figure 5: Output of FL running the AMD2901 spec.fl file.

than verification speed is how difficult it is to discover and track down errors in the circuit and/or the specification[8]. This is one area where trajectory evaluation is quite powerful. Part of this is of course that the approach resembles simulation to a large extent, and thus is fairly natural to many designers.

To illustrate some of the techniques that can be employed using the Voss system, we will return to the AMD 2901 verification. This time, however, we will use an incorrect specification and show how to track this down. In the file "err_spec.fl" in the demos/AMD2901/behavioral_VHDL directory, we have a specification file that contains two errors: the Q_is abstraction function does not say that the values are stored negated, and the timing of the carry-in signal is incorrect. When loading this program and forcing `check1` to be evaluated, we get a response that (after being cut down significantly) looks like:

```
: check1;
..Start garbage collection ...(Used=24497(Shared=817) Freed:46642)...done
......  stable at time 3
Time: 5
.Time: 6
..........  stable at time 15
Time: 50
.Warning: Consequent failure at time 50 on node y[3]
Current value:i8&i5&i4&i3&i2&i1&i0&d.3' + i8&i4&i3&i2&i1&i0'&d.3&q.3' +
   i8&i5&i4&i3&i1&i0'&d.3'&q.3 + i4&i3&i2&i1'&i0&a.3&d.3 +
   i8&i5&i4&i3&i2&i0&a.3'&d.3' OR ...  +
 X(i8&i5'&i4&i3'&i2&i1&i0&d.2'&d.1'&d.0' +
   i8&i5'&i4&i3'&i2&i1&i0'&d.2&q.2'&d.1&q.1'&d.0&q.0' +
   i8&i5'&i4&i3'&i2&i1&i0'&d.2&q.2'&d.1&q.1'&d.0'&q.0 +
   i8&i5'&i4&i3'&i2&i1&i0'&d.2&q.2'&d.1'&q.1&d.0&q.0' +
   i8&i5'&i4&i3'&i2&i1&i0'&d.2&q.2'&d.1'&q.1&d.0'&q.0 OR ... )
Expected value:i5&i4&i3&i2&i1&i0&a.3&b.3&d.3'&a.2&b.2&a.1&b.1&a.0&b.0 +
   i5&i4&i3&i2&i1&i0&a.3&d.3'&Aa0&Ab0' + i5&i4&i3&i2&i1&i0&a.3&d.3'&Aa0'&Ab0 +
   i5&i4&i3&i2&i1&i0&a.3&d.3'&Aa1&Ab1' + i5&i4&i3&i2&i1&i0&a.3&d.3'&Aa1'&Ab1
   OR ...  +
  X(a.0&b.0'&Aa3&Ab3&Aa2&Ab2&Aa1&Ab1&Aa0&Ab0 +
    a.0&b.0'&Aa3&Ab3&Aa2&Ab2&Aa1&Ab1&Aa0'&Ab0' +
    a.0&b.0'&Aa3&Ab3&Aa2&Ab2&Aa1'&Ab1'&Aa0&Ab0 +
    a.0&b.0'&Aa3&Ab3&Aa2&Ab2&Aa1'&Ab1'&Aa0'&Ab0' +
    a.0&b.0'&Aa3&Ab3&Aa2'&Ab2'&Aa1&Ab1&Aa0&Ab0 OR ... )
Strong disagreement when:i8&i5&i4&i1&i0'&a.3&b.3&a.2&b.2&a.1&b.1&a.0&b.0 +
    i8&i5&i4&i1&i0'&Aa0&Ab0' + i8&i5&i4&i1&i0'&Aa0'&Ab0 +
    i8&i5&i4&i1&i0'&Aa1&Ab1' + i8&i5&i4&i1&i0'&Aa1'&Ab1 OR ...

 * * *
...  stable at time 202
Time: 205

------WARNING: Some errors not reported
i6&i5&i0 + a.0&b.0'&Aa3&Ab3&Aa2&Ab2&Aa1&Ab1&Aa0&Ab0 +
a.0&b.0'&Aa3&Ab3&Aa2&Ab2&Aa1&Ab1&Aa0'&Ab0' +
a.0&b.0'&Aa3&Ab3&Aa2&Ab2&Aa1'&Ab1'&Aa0&Ab0 +
a.0&b.0'&Aa3&Ab3&Aa2&Ab2&Aa1'&Ab1'&Aa0'&Ab0' OR ...
```

First of all, the system complains at the first node it finds an error. For every error it find (up to a user setable limit), the system will print out

1. The current value on the node.

[8]Although, in theory, the specification should always be correct and only the circuit contain errors, in practice it is very common to have errors in both.

2. The expected value on the node.
3. The condition for this error to show up.

For the first two values, the print routine prints out the values in the form `f1 + X(f2)`, where `f1` and `f2` look like Boolean expressions, but should be read as quaternary extensions of the Boolean expression. One should read this formula as: if `f1` is equal to 1, then the expression is equal to 1. If both `f1` and `f2` are equal to 0, then the expression is equal to 0. In all other cases the expression is equal to X.

The third statement that is printed out at a consequent failure is the Boolean condition that must hold for this error to manifest itself. Here we distinguish between two types of errors: weak and strong disagreements. A strong disagreement means that there is some assignment to all the Boolean variables in currently used so that the node value is 1 and the expected value is 0, or vice versa. This is clearly an error. A weak disagreement, on the other hand, signifies that the value on the node is X when a Boolean value was expected. This error condition is not a clear-cut as the strong disagreement, since it is possible that the pessimism inherent with using X as an unknown value, may generate responses that have more X's than absolutely necessary. However, in practice, it usually means that a dependency on some signal was forgotten and thus this signal did not have a value asserted and consequently stayed at X.

Finally, the result of the verification (after all the error messages) is the Boolean condition for when the whole verification is successful. This result can sometimes give a clue to what went wrong.

Although the above expressions are often very helpful, they are very difficult to read and understand. Consequently, we load in the HighLowEx.fl library to get access to some concrete example generating functions. For example, we would get:

```
: load "HighLowEx.fl";
-Loading file HighLowEx.fl
T
-Loading file defaults.fl
T

: let vl = [(" I",i),("Aa",Aa),("Ab",Ab),(" a",a),(" b",b),(" d",d),(" q",q)];
vl::((string # (bool list)) list)
: Lexpl vl (check1 AND consistent);
"
 I = 001011000
Aa = 0000
Ab = 0001
 a = 1111
 b = 0000
 d = 0000
 q = ----

"
```

showing that one instruction that does work as specified is the OR function between word 0 in the RAM and the content of the accumulator assuming the value stored in the word 0 consists of all 1's.

Although the above may help us pinpoint the error, it is usually easier to catch the error as soon as they happen. Consequently, we will modify the FSM command and give it the -a option that will force the symbolic simulation to abort at the first error encountered. We now get:

```
: let trac1 = FSM "-a" AMD ant1 cons1 [];
trac1::bool
: trac1;
......  stable at time 3
Time: 5
.Time: 6
..........  stable at time 15
Time: 50
.Warning: Consequent failure at time 50 on node y[3]
Current value:i8&i5&i4&i3&i2&i1&i0&d.3' + i8&i4&i3&i2&i1&i0'&d.3&q.3' OR ... +
  X(i8&i5'&i4&i3'&i2&i1&i0&d.2'&d.1'&d.0' +
    i8&i5'&i4&i3'&i2&i1&i0'&d.2&q.2'&d.1&q.1'&d.0&q.0' OR ... )
Expected value:i5&i4&i3&i2&i1&i0&a.3&b.3&d.3'&a.2&b.2&a.1&b.1&a.0&b.0 +
  i5&i4&i3&i2&i1&i0&a.3&d.3'&Aa0&Ab0' OR ...  +
 X(a.0&b.0'&Aa3&Ab3&Aa2&Ab2&Aa1&Ab1&Aa0&Ab0 +
   a.0&b.0'&Aa3&Ab3&Aa2&Ab2&Aa1&Ab1&Aa0'&Ab0' OR ... )
Strong disagreement when:i8&i5&i4&i1&i0'&a.3&b.3&a.2&b.2&a.1&b.1&a.0&b.0 +
   i8&i5&i4&i1&i0'&Aa0&Ab0' + i8&i5&i4&i1&i0'&Aa0'&Ab0 +
   i8&i5&i4&i1&i0'&Aa1&Ab1' + i8&i5&i4&i1&i0'&Aa1'&Ab1 OR ...
```

The important point here is that the return value of FSM with the -a flag is the condition under which the error manifests itself. Thus, using the Lexp or Hexpl functions can give us a concrete example to run to see what went wrong.

```
: Lexpl vl trac1;
"
 I = 000000000
Aa = 0000
Ab = 0000
 a = 0000
 b = 0000
 d = 0000
 q = -001

c = 0
"
```

The best approach now is to re-run the verification but instead of using all the Boolean variables, we would use these values. A useful trick here is to over-ride the definitions of the variable shorthands. For example, we modify err_spec.fl as follows:

```
let i = variable_vector "i" 9;
let i = [F,F,F,F,F,F,F,F,F];
// Addresses
let Aa = variable_vector "Aa" 4;
let Aa = [F,F,F,F];
let Ab = variable_vector "Ab" 4;
let Ab = [F,F,F,F];
// Data
let a = variable_vector "a." 4;
let a = [F,F,F,F];
let b = variable_vector "b." 4;
let b = [F,F,F,F];
let d = variable_vector "d." 4;
let d = [F,F,F,F];
let q = variable_vector "q." 4;
let q = [F,F,F,T];
let c = variable "c";
let c = F;
```

Since FL is lexically scoped, the file must be reloaded. However, before doing so, it is convenient also to add some tracing commands in the verification run to get a picture of what is going on. There are two additions we must to for this to happen. First we must select some set of nodes to be traced. In this case it is natural to chose the clock and the y outputs. In order to get an easy to read waveform diagram, we also give a -T plot.ps (or -t plot.ps if we do not want the plot to be in landscape mode) to the options of the FSM command. In other words, we will have:

```
let tr_list = [CLK] @ Y;
let trac1 = FSM "-T plot2.ps" AMD ant1 cons1
(map (\node. (node,0,cycle 3)) tr_list);
trac1;
```

at the end of the specification file. Once this file has been loaded, a postscript file containing the waveform diagram shown in Fig. 6 will be generated. From this file, we can see that `y[0]` has an

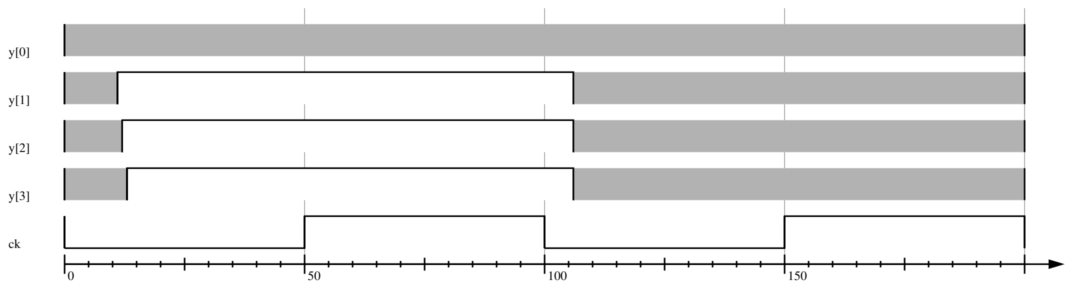

Figure 6: Waveform diagram generated from trace commands.

X value when it should be 0, and that all the other y nodes have their wrong values. Although strong disagreements are often easier to trace, in view of the fact that the instruction that we are considering is addition, it makes sense to find the root of the incorrect value on `y[0]` first. In this case, the obvious method is to trace all fanin nodes to `y[0]`. However, it is also useful to see the functionality of the next state function for the node. In the following small script we work ourselves backwards by using the built-in `fanin` function and the `excitation` function defined in default.fl.

```
: fanin AMD "y[0]";
["i[7]","i[6]","i[8]","alu_out[0]","ra[0]","noe"]
: excitation AMD "y[0]";
..
Trace started for node: y[0]
        Current value:X
.Time: 1
.Trace: Node y[0]  at time 1: i[6]&alu_out[0]&noe' + i[8]&alu_out[0]&noe'
    + i[7]&i[6]'&i[8]'&ra[0]&noe' + i[7]'&alu_out[0]&noe' + X(noe)
Time: 2

Trace ended for node: y[0]
"i[6]&alu_out[0]&noe' + i[8]&alu_out[0]&noe' +
 i[7]&i[6]'&i[8]'&ra[0]&noe' + i[7]'&alu_out[0]&noe' + X(noe)"
: fanin AMD "alu_out[0]";
["i[4]","i[3]","i[5]","s[0]","r[0]","difrs[0]","difsr[0]","sumrs[0]"]
: FSM "-T plot2.ps" AMD ant1 cons1 (map (\node. (node,0,cycle 3)) tr_list);
```

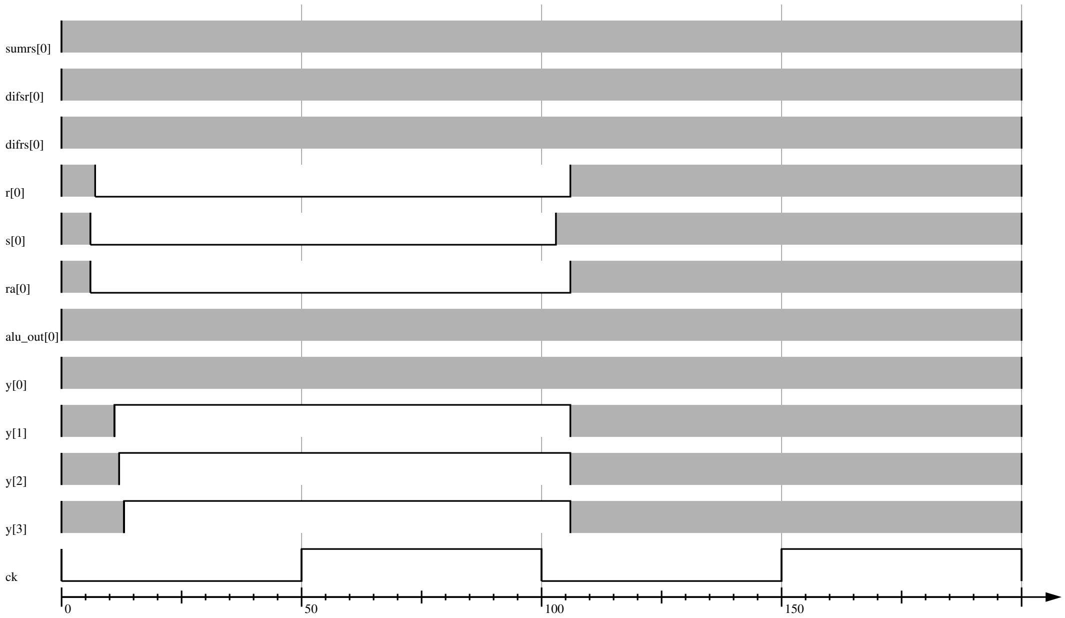

Figure 7: Second waveform diagram generated from trace commands.

After this, we would get a waveform diagram like the one shown in Fig. 7. Continuing like this, by finding the fanin nodes, tracing them, possibly finding the next state function, we can eventually determine that the `cin` node has the wrong value and thus discover that error in the specification. Similarly, one can relatively easily find the error in the Q_is function. Of course, in practice there are often errors in the circuit design itself too, but the same methodology often works well also for this type of errors.

Finally a word about the waveform diagrams. In Table 8 we give a key for interpreting the different patterns that can emerge. Note that for symbolic expressions, the information printed out to stderr is often needed to fully interpret the waveform diagram. Nevertheless, being able to see the waveforms are often invaluable when determining what values are stored where and when.

Pattern	Interpretation
grey	X
low solid line	0
high solid line	0
low and high solid line	symbolic expression that can be 0 or 1
grey with solid low line	symbolic expression that can be 0 or X
grey with solid high line	symbolic expression that can be 1 or X
grey with solid high and low lines	symbolic expression that can be 0, 1 or X
black	overconstrained signal ($\top$)

Figure 8: Patterns and their meaning in waveform diagrams.

7.1.5 Variable Ordering

One of the most mysterious aspects of symbolic trajectory evaluation for the novice is the need for variable ordering. Although the concept is easy to understand, coming up with good orders sounds very difficult. However, experience has show this not to be so difficult as it first appears. First of all, since the Boolean variables are used in a very explicit fashion in symbolic trajectory evaluation, the user has a much better understanding of the use of the various variables. Also, the fact that we often reduce the number of variables needed very significantly by using symbolic indexing, means that there are fewer variables around and thus the need for good ordering is less critical (even in the presence of exponential blow-up if you are only using a logarithmic number of variables you may still have acceptable performance!).

To simplify the task of selecting good variable ordering there are some useful functions provided in the Voss system. Perhaps the most useful ones are: bdd_size and bdd_profile. Both take a list of Boolean functions as argument. Bdd_size will return the width of the multi-root OBDD defined by these Boolean functions for each variable at least one of the Boolean functions depends on. Bdd_profile, on the other hand, prints out a histogram to give a quick visual picture of the shape of the OBDDs. To illustrate this, consider the AMD2901 example again and concentrate on a good variable ordering for the `Fr` values. We will show two OBDD profiles: the first one with the instruction variables last, the second one with these variables first. As will be seen, the difference is very significant!

```
: var_order;
[a.3,b.3,d.3,q.3,a.2,b.2,d.2,q.2,a.1,b.1,d.1,q.1,a.0,b.0,d.0,q.0,
 Aa3,Ab3,Aa2,Ab2,Aa1,Ab1,Aa0,Ab0,i8,i7,i6,i5,i4,i3,i2,i1,i0]
: bdd_profile Fr;
"
a.3* 1
b.3* 2
d.3* 4
q.3* 8
a.2* 17
b.2* 34
d.2* 68
q.2* 136
a.1** 273
b.1*** 546
d.1***** 964
q.1********* 1928
a.0************** 2961
b.0************************* 5282
d.0*************************************** 8260
q.0******************************************************************* 14536
 i5*************************************************************** 13840
 i4*************************************************************** 13808
 i3********************************************************************* 14946
 i2******************** 4157
 i1* 180
 i0* 12
  c* 2
"
```

```
: var_order;
 [i8,i7,i6,i5,i4,i3,i2,i1,i0,a.3,b.3,d.3,q.3,a.2,b.2,d.2,q.2,
  a.1,b.1,d.1,q.1,a.0,b.0,d.0,q.0,Aa3,Ab3,Aa2,Ab2,Aa1,Ab1,Aa0,Ab0]
: bdd_profile Fr;
"
 i5**** 4
 i4******* 8
 i3************* 16
 i2************************ 31
 i1**************************************** 52
 i0********************************************************************** 92
a.3******************** 26
b.3********* 11
d.3************** 18
q.3************* 17
a.2************************************** 50
b.2****************** 23
d.2**************************** 36
q.2*************************** 35
a.1************************************** 50
b.1****************** 23
d.1**************************** 36
q.1*************************** 35
a.0******************************** 42
b.0********* 11
d.0***************** 22
q.0********* 11
  c** 2
"
```

7.1.6 Structural VHDL Description

Given the above verification script, it is natural to ask how much of this work can be re-used for verification tasks later in the design process. What is pleasant about the methodology described above is that almost all of the specification file can remain unchanged! To illustrate this, consider verifying a complete structural implementation of the AMD 2901 circuit. In demos/AM2901/structural_VHDL we have the code for this design. We also have the spec.fl file which contains the verification script. What is satisfying about the example is the fact that the only thing that has changed from the behavioral verification script is the node name section.

8 A greater then B circuit

In the directory demos/AgreaterB there are three subdirectories: behavioral_vhdl, switch_level, and comparison. In behavioral_vhdl there is a behavioral description of a circuit that reads in two 32-bit unsigned integers a and b and determines whether $a > b$ and $b > 0$. In the same directory there is also a small verification script that verifies that this is indeed the behavior of the program.

In switch_level there is a fairly complex switch-level implementation of the same circuit. The implementation uses pre-charged domino-CMOS logic and requires a fairly sophisticated switch-level simulator to be simulated. In the same directory there is also a spec.fl file that contains a small verification script that verifies that if the circuit is clocked properly and the input signals are stable at the right time, the output of the circuit does indeed take on the correct value.

Finally, in the directory comparison there is a small verification script that can be used to compare two implementations of combinational logic. However, it is more sophisticated than that since it allows either implementation to be clocked. In the file "check_equality.fl" we use this function to compare the two implementations.

9 Binary2BCD

In the directory demos/Binary2BCD there is a structural VHDL implementation of a circuit that takes an 8-bit unsigned binary input and after some 11(?) clock cycles have converted this value into a three digit BCD number. What is interesting in this verification task is that we use a *relational* specification. In other words, we never say what the output should be, we only say that it should have certain properties. In particular, if we convert both the BCD output and the original input to decimal numbers, we should always get the same number.

For specifications where it is difficult to define a function that actually computes the desired result, using a relational style can often be very helpful. Note, however, that we must use much more Boolean variables is this style, and thus it is only recommended for specifications in which the desired function is difficult to compute.

10 Mead and Conway Stack

In the directory demos/MC_stack there is an NMOS stack, as described by Mead and Conway, and a verification script. The verification is another example of using symbolic indexing.

11 Tamarack3

A complete verification of a switch-level implementation of the Tamarack III processor. The specification uses a style that is very natural for micro-coded designs. The specification is also interesting because it illustrates the use of an invariant. In this case the invariant is simply that the micro-program counter is always 0 after each complete instruction.

12 UART

A structural VHDL implementation of a programmable UART circuit. The spec.fl file contains a partial specification that illustrates the first step away from traditional simulation. Again this is accomplished by an invariant.

13 McMillan

A complete verification of a pipelined integer unit of a typical RISC processor. The datapath has a four-stage pipeline with by-pass and stall logic. The circuit is modeled at the switch-level. This verification is the most complex among all the examples in this demo directory. In particular, the abstract specification is un-pipelined and uses the recent history of the inputs to compute a quite sophisticated abstraction function for where (and when) a register contains some value. In particular, depending on the previous two instructions and the previous cycles stall signal, the content of a register i may either be in one of the pipe-registers or be in the register file.

14 Model Checking

In the directory demos/ModelChecking there are a couple of files that illustrates the ease in which symbolic model checking can be implemented in FL. Although the next state relation is actually given explicitly in the two examples (of a transition arbiter), there is nothing stopping you from using trajectory evaluation to derive the next state relation. An example of this will very likely be in the next release.

A Informal specification of AMD2901

The Am 2901 four-bit microprocessor slice (from Advanced Micro Devices Inc.) is a high-speed cascadable ALU intended for use in CPUs, peripheral controllers and programmable microprocessors. The data is 4 bits wide at all points.

Functional Blocks

The main functional blocks in AMD2901 are as follows:

1. A 16-word by four bit two port RAM, with an up/down shifter at the input.
   (a) Port A is an output port.
   (b) Port B is a bidirectional port.
2. A register (called Q) with an up/down shifter at the input
3. An ALU source selector which select two inputs out of the following :
   (a) Port A of RAM
   (b) Port B of RAM
   (c) Q register output
   (d) External data input
   (e) Logical 0
4. A 4-bit ALU, capable of performing arithmetic and logical functions on the selected source words.
5. A destination selector which decides :
   (a) whether to load the ALU output (with or without shifting) into the RAM.
   (b) whether to load the ALU output (with or without shifting) or the Q register contents (with shifting) into the Q register
   (c) whether the ALU output or the Port A contents should be forwarded to the External data output.

NOTE: The destination selector is not shown as an explicit block in the figure. Its parts are included in the RAM, Q-register and output selector.

Inputs and Output Ports

Port	Type	Bit width	Description
I	in	9	Instruction word (discussed later)
Aadd	in	4	Address input to RAM (for READ)
Badd	in	4	Address input to RAM (for READ / WRITE)
D	in	4	Data input to chip
Y	out	4	Data output from chip
RAM0	inout	1	Up/down shifter port connected to LSB of RAM
RAM3	inout	1	Up/down shifter port connected to MSB of RAM
Q0	inout	1	Up/down shifter port connected to LSB of Q-register
Q3	inout	1	Up/down shifter port connected to MSB of Q-register
CLK	in	1	clock
C0	in	1	Carry input to ALU
OEbar	in	1	Tri-state driver input (if this is not asserted, the data output Y will be tri-stated to HIGH-Z)
C4	out	1	Carry output from ALU
Gbar	out	1	Generate term from ALU for carry lookahead
Pbar	out	1	Propagate term from ALU for carry lookahead
OVR	out	1	Overflow output from ALU (this signals that an overflow has occurred, while performing the operation)
F3	out	1	MSB of the ALU output
F30	out	1	Zero signal (asserted if the all 4 bits of ALU output are zero).

The Instruction Set

The Am 2901 has a 9-bit instruction, which has three 3-bit fields whose functions are as follows :

1. I2 downto I0: controls ALU source selector
2. I5 downto I3: controls ALU function
3. I8 downto I6: controls destination selector

A.0.7 ALU Source Operands Selected

Bit field	ALU source operands selected	
I2 I1 I0	RE	S
000	A	Q
001	A	B
010	0	Q
011	0	B
100	0	A
101	D	A
110	D	Q
111	D	0

Note: RE and S are the two outputs of the ALU source selector.

A.0.8 ALU Function

Bit field	ALU function (output --> F)	
I5 I4 I3	C0 = 0	C0 = 1
000	RE + S	RE + S + 1
001	S - RE - 1	S - RE
010	RE - S - 1	RE - S
011	RE or S	RE or S
100	RE and S	RE and S
101	not(RE) and S	not(RE) and S
110	RE xor S	RE xor S
111	RE xnor S	RE xnor S

Note: C0 is the carry-in input. F is the output of the ALU.

A.0.9 ALU Destination

Bit field	RAM function		Q-REG. function		Y	RAM shifter		Q-REG. shifter	
I8 I7 I6	SHIFT	LOAD	SHIFT	LOAD		RAM0	RAM3	Q0	Q3
000	—	—	—	F	F	—	—	—	—
001	—	—	—	—	F	—	—	—	—
010	—	F	—	—	A	—	—	—	—
011	—	F	—	—	F	—	—	—	—
100	down	F/2	down	Q/2	F	out	in	out	in
101	down	F/2	—	—	F	out	in	out	—
110	up	2F	up	2Q	F	in	out	in	out
111	up	2F	—	—	F	in	out	—	out

Note: Data that is loaded into the RAM is written at the RAM word pointed to by the address input Badd. Note also that the bidirectional ports are active only when some shifting is being done. Finally, whenever a bidirectional port is NOT being used as an output by the Am2901, the Am2901 chip tristates it from its own side. Then, it can be used as an input from the external world or it may be left inactive.

B Switch-Level Model

In this appendix we discuss the basic switch-level model used by the switch-level compilers sim2ntk and ntk2exe. Since sim2ntk is the standard sim2ntk program from the COSMOS system developed by Randy Bryant and associates at Carnegie Mellon University and the ntk2exe program has been heavily influenced by the anamos tool from the same system, the switch-level models are essentially the same as the ones supported by the COSMOS system. Furthermore, this section is heavily modeled after the user's guide of the COSMOS system. In particular, the structure and the examples all follow closely the user guide. For a more formal treatment of the underlying algorithms employed, the reader is referred to [**?**] and [**?**].

B.1 Circuit Model

A switch-level circuit consists of nodes and transistors. The ntk2exe program partitions the nodes and transistors into subsets that are called channel-connected subnetworks. The basic idea is to collect all nodes and transistors that are connected through source/drain connections that do not go through the supply nodes (power and ground). Each such channel-connected subnetwork is then analyzed in isolation and a behavioral model is automatically derived for the subnetwork. The behavior of the whole circuit is then derived from the interconnection of these behavioral models of the sub-networks.

B.1.1 Node Model

Each node can take on four different values $0, 1, X$, and $\top$. Normally, $\top$—the overconstrained value—cannot be generated by the circuit itself. However, it can be introduced by the simulator and thus the behavior of the network must be able to handle this value as well. The value X is used to denote an invalid, uninitialized, or changing value.

One basic assumption in the switch-level analysis is that the complete circuit is available. Thus, if a node is meant to be tri-state (high-impedance), the user must model this by attaching a dummy driver circuit. We will return to this later.

Note also that the analysis carried out by ntk2exe takes all size and strength effects into consideration. Hence, the four values mentioned above suffice for modeling the complete switch-level model with a number of different sizes and strengths etc.

There are two types of nodes:

input nodes Provide strong signals from sources external to the network. Examples of this type of nodes are power, ground, clock and data inputs. Note that power and ground nodes are treated specially as having fixed logic values 1 and 0 respectively.

storage nodes These nodes have their value determined by the switch-level analysis and (unless they have size 0) will retain the their current values in the absence of applied signals.

Each storage node is assigned a size in the set 0,...,maxnode to indicate (in a simplified way) its capacitance relative to other nodes with which it may share charge[9] When a set of connected storage nodes is isolated from any input nodes, they are charged to a logic state dependent only on the state(s) of the largest node(s). Thus the value on a larger node will always override the value on a smaller one. Many networks do not depend on charge sharing for their logical behavior and

[9]This description is accurate for the current release of the sim2ntk and ntk2exe programs. However, a new version is in early stages of testing that uses a partially ordered size and strength sets. This modified version will be in the next Voss release.

hence can be simulated with only one node size (maxnode = 1). In general, at most two node sizes (maxnode = 2) will suffice with high capacitance nodes (e.g. pre-charged busses) assigned size 2 and all others assigned size 1.

Node size 0 indicates that the node cannot retain stored charge. Whenever such a node is isolated, its state becomes X. This size is useful when modeling static circuits. By assigning size 0 to all storage nodes, the simulation is more efficient, and unintended uses of dynamic memory can be detected.

The symbolic analyzer ntk2exe attempts to identify and eliminate storage nodes that serve only as interconnections between transistor sources and drains in the circuit. It retains any node that it considers "interesting," i.e., whose state affects circuit operation. Interesting nodes include those that act as the gates of transistors, as inputs to functional blocks, or as sources of stored charge to other interesting nodes. Sometimes a node whose state is not critical to circuit operation, however, may be of interest to the simulator user. The user must take steps to prevent ntk2exe from eliminating these nodes, by identifying them as "visible." A node can be so identified by an attribute in the .sim file, or with a command line option to ntk2exe.

B.1.2 Transistor Model

A transistor is a three terminal device with node connections gate, source, and drain. Normally, there is no distinction between source and drain connections–the transistor is a symmetric, bidirectional device. However, transistors can be specified to operate unidirectionally to overcome limitations of the network model. That is, a transistor can be forced to pass information only from its source to its drain, or vice-versa. Unidirectional transistors are required only rarely in such circuits as sense amplifiers and pass transistor exclusive-or circuits. Excessive use of unidirectional transistors can cause the simulator to overlook serious design errors. Any circuit simulated with unidirectional transistors should be thoroughly analyzed with a circuit simulator such as SPICE.

Each transistor has a strength in the set 1,...,maxtran. The strength of a transistor indicates (in a simplified way) its conductance when turned on relative to other transistors which may form part of a ratioed path. When there is at least one path of conducting transistors to a storage node from some input node(s), the node is driven to a logic state dependent only on the strongest path(s), where the strength of a path equals the minimum transistor strength in the path. Thus, a stronger signal will always override a weaker one. Most CMOS circuits do not involve ratioing, and hence can be simulated with one transistor strength (maxtran = 1). However, circuits involving multiple degrees of ratioing may require more strengths. Ntk2exe will utilize as many node sizes and transistor strengths as are used in the network file with the limitation that maxnode + maxtran ¡ 16.

The simulator models three types of transistors: n-type, p-type, and depletion. A transistor acts as a switch between source and drain controlled by the state of its gate node as follows: When a transistor is in an "unknown" state it forms a conductance of unknown value between (inclusively) its conductance when "open" (i.e. 0.0) and when "closed". The simulator models these transistors in such a way that any node with state sensitive to their actual conductances is set to X.

Normally, transistor switching is simulated with a unit delay model. That is, one simulation time unit elapses between when the gate node of a transistor changes state, and the subcircuit containing the source and drain nodes of the transistor is evaluated. However, a transistor can be specified to have zero delay, meaning that the subcircuit will be evaluated immediately. The implications of the transistor delay are discussed more in Section B.1.4. Zero delay transistors are required only in rare cases to correct for the effects of circuit delay sensitivities. They can also be used to speed up the simulation, by creating rank-ordered evaluation of the circuit components.

gate	n-type	p-type	depletion
0	open	closed	closed
1	closed	open	closed
X	unknown	unknown	closed

Table 1: Transistor State as a Function of Gate Node State

B.1.3 Circuit Partitioning

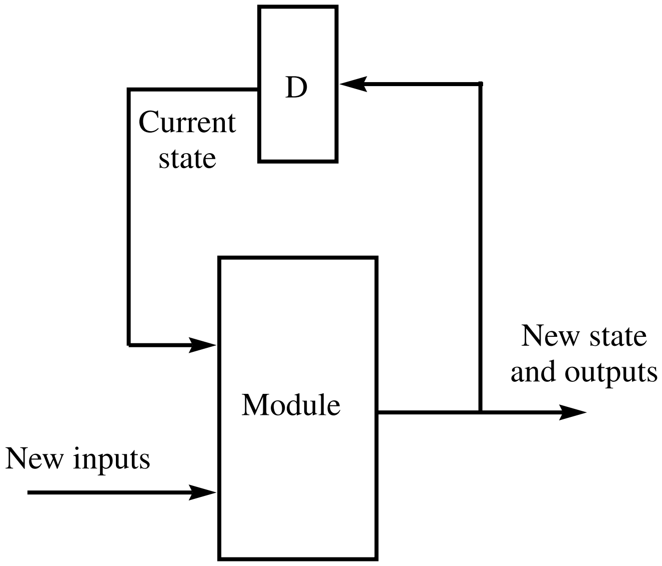

Figure 9: Finite State Behavior of a Circuit Module.

Ntk2exe partitions the initial circuit description into a set of modules. Each module corresponds to a transistor subnetwork. A subnetwork consists of a set of storage nodes connected by sources and drains of transistors, along with all transistors for which these nodes are sources or drains. Observe that an input node is not in any subnetwork, but a transistor for which it is a source (or drain) will be in the subnetwork containing the drain (or source) storage node.

The behavior of a module is described by a procedure generated automatically for the corresponding subnetwork.

As illustrated in Fig. 9, each module behaves like a finite state machine, computing new values for the results as a function of the old values on the results and inputs. The box labeled with D in the figure represent the delays of the various nodes and is always at least one time unit long. We will return to this shortly.

The partitioned circuit must obey the following rules:

1. A node can be a result connection of at most one module.

2. There can be no zero-delay cycles, i.e., every cycle in the set of interconnected modules must be broken by at least one unit delay.

B.1.4 Timing Model

.

The simulation program is designed primarily for simulating clocked systems, where there is a well defined timing regime for when signals should be stable and when outputs are expected. Although the sim2ntk and ntk2exe is only able to derive unit/zero delay models, the user can back annotate the generated .exe file by creating a file with the same name as the .exe file, but with a .del suffix. In this file, the user can list node names and bounds on the rise and fall delays of the nodes. An entry for a node in the .del file is of the format:

```
node_name |min_rise_delay max_rise_delay min_fall_delay max_fall_delay|
```

where the rise and fall delays must be integer multiples of the basic time unit.

The basic Fl system is able to analyze circuits using several timing models:

1. unit delay model
2. nominal delay
3. minimum delay
4. maximum delay
5. bounded delay

In the presence of a .del file and the appropriate delay model chosen in the .vossrc file, the FSM command in Voss will carry out the chosen delay analysis.

B.2 Circuit Examples

In the next release of this document we will provide some examples to illustrate the switch-level model.

C .sim format

The simulation tools crystal(1) and sim2ntk(1) accept a circuit description in .sim format. There is a single .sim file for the entire circuit, unlike Magic's ext(5) format in which there is a .ext file for every cell in a hierarchical design.

A .sim file consists of a series of lines, each of which begins with a key letter. The key letter beginning a line determines how the remainder of the line is interpreted. The following are the list of key letters understood.

— units: s tech: tech If present, this must be the first line in the .sim file. It identifies the technology of this circuit as tech and gives a scale factor for units of linear dimension as s. All linear dimensions appearing in the .sim file are multiplied by s to give centimicrons. Sim2ntk ignores the technology identifier.

type g s d l w x y g=gattrs s=sattrs d=dattrs Defines a transistor of type type. Currently, type may be e or d for NMOS, or p or n for CMOS. The name of the node to which the gate, source, and drain of the transistor are connected are given by g, s, and d respectively. The length and width of the transistor are l and w. The next two tokens, x and y, are optional. If present, they give the location of a point inside the gate region of the transistor. The last three tokens are the attribute lists for the transistor gate, source, and drain. If no attributes are present for a particular terminal, the corresponding attribute list may be absent (i.e, there may be no g= field at all). The attribute lists gattrs, etc. are comma-separated lists of labels. The label names should not include any spaces, although some tools can accept label names with spaces if they are enclosed in double quotes. Sim2ntk(1) documents the transistor attributes recognized by sim2ntk.

C n1 n2 cap Defines a capacitor between nodes n1 and n2. The value of the capacitor is cap femtofarads. NOTE: since many analysis tools compute transistor gate capacitance themselves from the transistor's area and perimeter, the capacitance between a node and substrate (GND!) normally does not include the capacitance from transistor gates connected to that node. If the .sim file was produced by ext2sim(1), check the technology file that was used to produce the original .ext files to see whether transistor gate capacitance is included or excluded; see "Magic Maintainer's Manual 2: The Technology File" for details. Sim2ntk only considers capacitors where one terminal is ground. It computes transistor gate capacitances based on the transistor constructs.

A node attrs Associates the set of attributes attr for node node. The attributes are specified by a comma-separated list of strings containing no blanks. Sim2ntk(1) documents the node attributes recognized by sim2ntk.

= node1 node2 Each node in a .sim file is named implicitly by having it appear in a transistor definition. All node names appearing in a .sim file are assumed to be distinct. Some tools, such as esim(1) (and sim2ntk), recognize aliases for node names. The = construct allows the name node2 to be defined as an alias for the name node1. Aliases defined by means of this construct may not appear anywhere else in the .sim file.

@ filename Redirects input to the specified file. When the end-of-file is reached, input reverts to the current file at the following line. Input redirection can be nested.

D .ntk Format

An .ntk file consists of a series of commands, each of which begins with a key character and terminates with a semicolon ';'. A period '.' terminates the circuit declaration. The key character determines how the command is interpreted. Elements of a command are separated by spacechar characters (see below). Note that there must even be space before the terminating semicolon. All names and key characters are case sensitive.

An .ntk file declares a circuit as a set of nodes, transistors, and vectors. A node is an electrical node of type input or storage. A transistor is a MOSFET with gate, source and drain nodes.

In the following description, items enclosed in braces { } may be repeated any number of times (including 0). Items enclosed in brackets [ ] are optional. When there is a list separated by vertical bars —, any item from this list may appear. Parentheses ( ) indicate grouping.

The following syntactic elements are referred to in the document.

spacechar A space character. These are: blank, tab, new-line, carriage-return, and new-page.

noderef A reference to a node. Circuit nodes are numbered from 1 up to the number of nodes. A reference to the ith node is of the form #i. No node may be referenced before it is defined.

attrs An attribute list. This is a sequence of the form /char value, where char is a single character attribute identifier, and value is the attribute value.

The components of the file are:

(i—+—-) { name } [noderef] ; [attrs ;] Defines an input node. Node type 'i' denotes an ordinary (e.g., data or clock) input node. Node types '+' and '-' denote power and ground nodes, respectively. The optional list of names declares a set of names for the node. All node names in the circuit must be unique. The noderef serves only as documentation. It must refer to the node being defined. The optional attribute list defines additional properties of the node. Currently, the only node attribute recognized is /c, followed by the node capacitance in picofarads (using the C syntax for floats.)

(s — S) size { name } [noderef] ; [attrs ;] Defines a storage node. Node type 's' denotes an ordinary storage node. Ntk2exe may optimize such a node out of the network, unless it is the gate of a transistor, occurs in a vector, is an input or output to a functional block, or can affect the value on some other node. Node type S denotes a visible storage node. Such a node cannot be optimized away under any condition. The node size is a nonnegative integer (typically small) specifying the node's precedence when sharing charge with other nodes. Node size 0 indicates that the node does not store charge. Any time such a node is isolated, its state is set to the unknown value X. The optional list of names declares a set of names for the node. All node names in the circuit must be unique. The noderef serves only as documentation. It must refer to the node being defined. The optional attribute list defines additional properties of the node. Currently, the only node attribute recognized is /c, followed by the node capacitance in picofarads (using the C syntax for floats.)

drainref ; [attrs ;](n — p — d) [U — Z] [¿ — ¡] strength gateref sourceref Defines a transistor of type 'n', 'p', or 'd'. The transistor may be specified to have unit ('U') or zero ('Z') delay. Unit delay is the default. The transistor may optionally be specified to conduct information in only one direction, either from the source to the drain ('¿') or to the source from the drain ('¡'). If no direction is specified, the transistor is bidirectional. The strength is a small, positive integer specifying the transistor's precedence in ratioed circuits. The gate, source, and drain nodes of the transistor are each specified by a noderef. The optional attribute list specifies addtional properties of the transistor. Currently, the only transistor attribute recognized is /r, followed by the effective resistance of the transistor in kilo-ohms (using the C syntax for floats.)

— **comment ;** All text up to the terminating semicolon is ignored. The comment string must not contain any semicolons.

. Terminates the .ntk description. Any following text is ignored. A file that does not contain a termination command is considered incorrect.

E .sil format

The .sil file format is reserved for SILOS II gate netlists. In fact, the silos2exe converter supports only a quite small subset of the SILOS II format.

To introduce the format, consider the following small example:

```
$********* GAte macros with delay parameters *******************
.macro CMOS4Xor2 i1 i2 / o
        o .xor 390,25.0 410,22.0 2.5*i1 2.5*i2
.eom

.macro CMOS4Nand2 i1 i2 / o
        o .nand 150,19.0 210,22.0 2.0*i1 2.0*i2
.eom

.macro CMOS4Nand3 i1 i2 i3 / o
        o .nand 180,21.0 240,24.0 2.0*i1 2.0*i2 2.0*i3
.eom


$********* Main circuit *******************
.TITLE ADDER   (On: Mon Feb  8 16:57:12 1993
(X1 CMOS4Xor2 a b t1
(X2 CMOS4Xor2 t1 cin result
(n1 CMOS4Nand2 a b t2
(n2 CMOS4Nand2 a cin t3
(n3 CMOS4Nand2 cin b t4
(n4 CMOS4Nand3 t2 t3 t4 cout

$********* End of netlist file *******************
```

First, the .xor, and .nand commands refer to two of the 9 built-in function types. The numbers after the .xor component represent the rise-delay and the fall-delay of the component and is calculated as the first number plus the second number times the fanout load. The multiplicative factors in front of input signals are used to denote the load factor. The macro definitions can be nested. However, the scoping rules are not well defined (silos2exe uses dynamic scoping), and thus it is recommended that all macros have distinct names. Finally, the format of a macro call is: (instance-name name-of-macro followed by arguments.

E.1 Syntax of .sil format supported by silos2exe

```
line            : alias_line
                | macro_def
                | component
                | macro_call
                |

alias_line      : $ string = string

macro_def       : .macro string string+
                          line*
                  .eom

macro_call      : ( string string input+

component       : string .buf delays input+
                | string .clk number string
                | string .or delays input+
                | string .nor delays input+
                | string .nand delays input+
                | string .and delays input+
                | string .xor delays input+
                | string .inv delays input+
                | string .tbuf delays input+

delays          : delay delay
                | /* empty */

delay           : number
                | number , number
                | number , float

input           : load string
                | load - string

load            : number *
                | float *
                | /* empty */
```

F VHDL Support

The support for behavioral and structural VHDL is through a translation program derived from the Alliance 1.1 distribution. Since the Alliance 1.1 tools are distributed under the Free Software Foundations license agreement, the source to this translator is available to whoever wants it.

The VHDL subset supported is fully compatible with the IEEE VHDL standard ref. 1076 (1987). Hopefully this means that any program that is acceptable to the convert2fl program, will also be acceptable to commercial synthesis and simulation tools. However, I don't give any guarantees. Below we outline the main restrictions of the VHDL subset we support.

A VHDL description of a circuit consists of two parts: the external view and the internal view. The external view defines the name of the circuit and its interface. The interface is a list of ports. Each port is specified by its name, mode, type, possible constraint, and its kind. The mode of a port depends only on the manner the port is used inside the circuit (in the internal view of the circuit). If the value is to be read in the view of the description, the port must be declared with the mode *in*. If the value is to be written by the internal view, the port must be declared with the mode *out*. If the internal view will both read and write to the port, the mode of the port must be *inout*.

Only *structural* and *behavioral data flow* representations are supported as internal view. Furthermore, it is not possible to mix behavioral and structural descriptions of a single entity. The convert2fl program requires also that each entity is contained in a file with the same name as the entity (lower case letters only!). If the description of the entity is structural, the suffix must be .vst, whereas if the description if behavioral, the suffix should be .vbe.

A typical VHDL description of a circuit will consist of a collection of files creating a structural hierarchy of .vst files with behavioral .vbe descriptions of the leaf nodes in the hierarchy. Note that the convert2fl program will first search for a behavioral description of an entity. Only if this fails, will the program look for a structural description of the entity[10].

F.1 Types Supported

The following set of predefined types has been defined. No other user-defined types are currently supported.

bit The predefined standard bit type ('0' or '1'). In the Voss system, this type is monotonically extended to the quaternary domain.

bit_vector An array of bits.

mux_bit A resolved subtype of bit using the *mux* resolution function. This function computes the greatest lower bound of the actively driven signals. If all drivers are disconnected, the value of the signal will be *X*. Note that signal of type mux_bit must be declared with the kind *bus*.

mux_vector An array of mux_bits.

reg_bit A resolved subtype of bit using the *mux* resolution function. This function computes the greatest lower bound of the actively driven signals. If all drivers are disconnected, the value of the signal will retain its old value. Note that signal of type reg_bit must be declared with the kind *register*.

[10]If convert2fl fails to find either a .vbe or .vst description, it looks for an EDIF description (.edi suffix)

F.2 Structural VHDL Supported

The declaration part of a structural description includes signal declarations and component declarations. A signal can be declared to have any of the types mentioned above.

A component declaration must be declared with exactly the same port description as in its entity specification. This means that local ports are to be declared with the same name, type, kind, and in the same order as in the entity specification.

A structural description is a set of component instantiation statements. The ports of the instance are connected to each other through other signals through a port map specification. Both explicit and implicit port map specifications are supported. The current version does not allow unconnected ports (the *open* mode is not supported).

Finally, only the catenation operator (&) can be used in the actual pat (effective signal connected to a formal port) in a port map specification.

Note that the *generate* statement is not currently supported (unfortunately!).

F.3 Behavioral VHDL Supported

The only type of statements supported by convert2fl are the following concurrent statements:

1. simple signal assignment
2. conditional signal assignment
3. selected signal assignment
4. block statement

When using concurrent statements, an ordinary signal can be assigned only once. The value of the signal must be explicitly defined by the signal assignment (for example, in a selected signal assignment, the value of the target signal must be defined for every value that the select expression can take on).

The above constraint is often too harsh when designing hardware that have their control distributed (e.g., precharged lines, distributed multiplexors, busses, etc.). To remedy this, VHDL uses guarded-resolved signals. A resolved signal is a signal declared with a resolved subtype. A resolved subtype is a type together with a resolution function. A resolved signal can be assigned by multiple signal assignments. Depending on the value of each driver, the resolution function determines the effective value of the signal.

A guarded signal is a resolved signal with drivers that can be disconnected. A guarded signal must be assigned inside a *block* statement through a *guarded* signal assignment.

To illustrate this, consider the following example of a distributed multiplexor:

```
signal DistributedMux : mux_bit bus;

begin
    FirstDriver: block (Sel1 = '1' )
    begin
        DistributedMux <= guarded Data1;
    end block;

    SecondDriver: block (Sel2 = '1' )
    begin
        DistributedMux <= guarded Data2;
    end block;
end
```

Sequential elements must be explicitly declared using he type *reg_bit* or *reg_vector* (and must be of kind *register*). A sequential element must be assigned inside a *block* statement by a *guarded* signal assignment. For example, a falling edge triggered D flip-flop could be defined as:

```
signal Reg : reg_bit register;

begin
    flip_flop: block (ck = '0' and not ck'STABLE)
    begin
        Reg <= guarded Din;
    end block;
end;
```

On the other hand, a rising edge triggered D flip-flop with asynchronous reset (active low) may be defined as:

```
signal Reg : reg_bit register;

begin
    flip_flop : block ((ck = '0' and not ck'stable) or (resetb = '0'))
    begin
        Reg <= guarded (resetb and Din);
    end block;
```

Finally, level sensitive latch can be defined as:

```
signal Reg : reg_bit register;

begin
    latch : block (ck = '1')
    begin
        Reg <= guarded Din;
    end block;
```

The subset of VHDL supported by convert2fl includes only the following built-in VHDL operators: **not, and, or, xor, nor, nand, &, =, /=**. These operators can be applied on all types supported. Note that other standard VHDL operators (most notably the arithmetic and comparison operators) are not supported in this release.