CLOVER: A USER GUIDE

Dimitris Doukas
Andrea S. LaPaugh

CS-TR-345-91

August 1991

CLOVER: A User Guide

Dimitris Doukas and *Andrea S. LaPaugh*
Department of Computer Science
Princeton University

August 7, 1991

Abstract

CLOVER is a timing verification system for digital systems. It was designed to handle medium size asynchronous systems, particularly interface systems, but can be used with synchronous systems as well. A wide range of basic components can be used with CLOVER, including user-defined components. To use CLOVER, the designer provides a description of the design in the hardware description language PDL-e and a description of the timing constraints that the design should satisfy in CLOVER's constraint language ATCSL. CLOVER derives an *event graph* for the design using an event-based timing simulator and checks the satisfaction of each constraint within the event graph.

In this report we present a detailed description on the use of CLOVER from the user perspective. A detailed documentation of all CLOVER's features is given, accompanied by simple but illustrative examples.

CLOVER: A User Guide

Dimitris Doukas and *Andrea S. LaPaugh*
Department of Computer Science
Princeton University

August 7, 1991

1 Introduction

In this paper we present a detailed description of the use of CLOVER from the user perspective. A detailed documentation of all CLOVER's features is given, accompanied by simple but illustrative examples. A complete and formal presentation of the concepts and semantics of CLOVER can be found in [5]. Here we will give a brief summary of CLOVER's main features.

CLOVER is a system for the description and timing analysis of medium size asynchronous interface systems, where the basic components can be as simple as a logic gate or as complicated as a complex functional module. Of particular interest are the description and analysis of bus interface circuits where more complicated timing relations exist between signals. The components and the wires of the designs assume bounded delays; in the current version of the system wires may only assume constant delay values.

CLOVER is a verification system which draws advantages from both tools like simulators and timing analyzers and tools based on formal verification methods. Simulators and timing analyzers do not provide a sophisticated specification model, and therefore they can verify a limited number of timing relations. Formal verification proofs on the other hand can be quite tedious, unless the design is small. The timing behavior in CLOVER is described within a formal specification model. The model is based on dependency graphs and vectors of signal values over time. Relations specifying transition of signals to different values, timing bounds between transitions and sequencing of transitions over time (transition *history*) can be easily described within the model. The model is an improvement over existing temporal specification models (e.g., temporal logic [4]) because it defines a framework which allows the expression of timing relations over the absolute timing of signals and reference to different instances of the same signal over time.

The verification methodology of CLOVER is based on *event graphs*. We call this analysis method *event graph verification.* An event graph contains the necessary information to verify a timing constraint specified with our specification model. It contains both dependency and timing information about signals. Event graphs are derived from the implemented design using event-driven time simulating techniques.

In Figure 1 we schematically present the organizational structure of CLOVER depicting the main components of the system. During verification, the tool matches the intended temporal behavior of a design against the event graph of its implementation, identifying and reporting violations. The timing behavior is described within CLOVER's formal specification model. To derive the event graph of the implementation, we do an event-driven time simulation of the implemented design. The simulator is based on an extended six-value system (presented in Section 3); it takes as input the netlist of the implemented circuit and the behavioral descriptions of the design's constituent modules and derives as output the timing behavior of the implementation, in the form of an event graph. The netlist of the circuit is obtained after describing the design with the hardware description language PDL-e.

CLOVER runs under the UNIX operating system; we assume that the user is familiar with basic concepts of UNIX, such as directories, files and environment variables. In order to use the system, the user has to know how to program in the *C* programming language. Next we show how the depicted components of CLOVER in Figure 1 are interfaced from the user point of view.

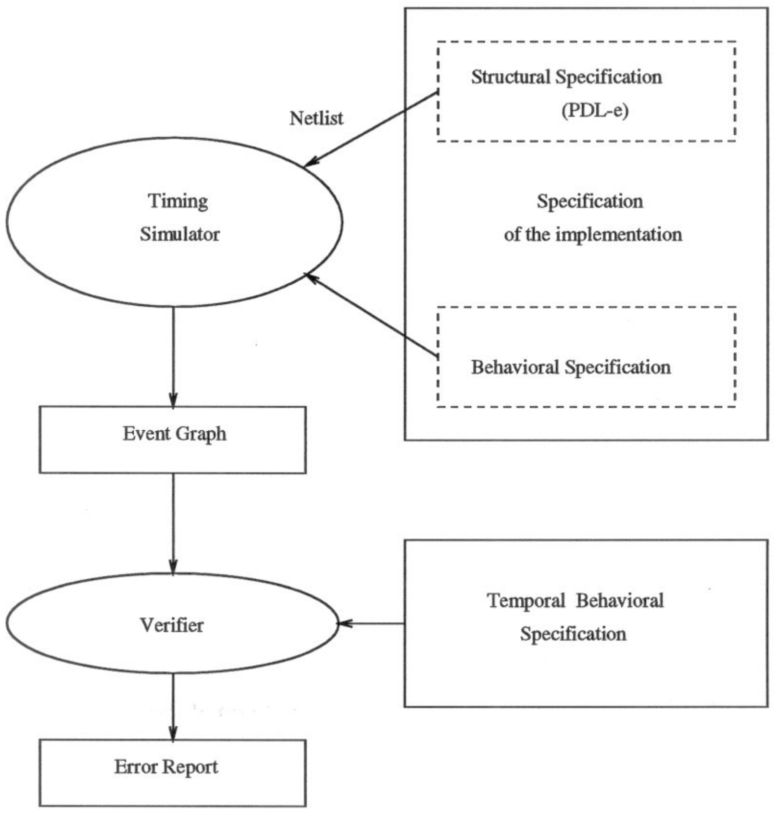

Figure 1: System Organization

1.1 Interfacing CLOVER's Components

Assuming that we want to verify a design named *foo*, the following steps need to be taken (a detailed description of each step will be presented in the following sections):

1. The design has to be structurally described. A design in the structural domain of description is viewed as a collection of functional modules interconnected with wires. The description is the equivalent of a graph with nodes representing the design's functional modules and edges representing the wires which interconnect them. We call this graph the *netlist* of the design. Assuming a direction on the flow of signals, from inputs to outputs, the graph becomes directed.

   To structurally describe a design, CLOVER uses a hardware description language: PDL-e. To specify the structure of the design *foo* the user has to write a PDL-e program. We call this program: *foo.pdl.* Because the simulator manipulates the structure of a design in the form of a netlist, the PDL-e description has first to be transformed to a netlist description. Program *pdl2net* is used to accomplish this transformation. The transformation is a two step operation: first, *pdl2net*, implicitly compiles and executes file *foo.pdl* to derive an intermediate netlist representation (we call it a *pif* representation) in a file called *foo.pif*; then, during the second step, it transforms the *pif* description to a representation readable by the simulator. This netlist representation is derived in a file called *foo.net.*

2. Next the behavior of the modules comprising the design has to be described. The *functional behavior* of a module can be defined as a function which transforms the discrete-valued input events of the module to discrete-valued output events. The input and output events assume values from the extended value system we use. Given a certain combination of input events the module responds after some time with a certain

combination of output events. The response times define the *timing behavior* of the module. The *behavior* of a module is the combination of its functional and timing behaviors. The behavioral description of a design is dependent on the behaviors of its constituent functional modules.

We distinguish two kind of modules: standard (i.e., standard gates, latches, flip-flops etc.) and user-defined (modules whose behavior is defined by the user). There is a built-in library containing the behavioral descriptions of standard-modules. If the design *foo* is comprised of standard modules, whose behavioral descriptions exist in the built-in library, then there is no need for user-intervention at this step. Otherwise, if user-defined modules are specified, then the user is responsible for specifying their behavior by writing *C* functions. All user-defined functions should be part of a user-defined library. The names of the functional modules in the PDL-e description should be the same as the names of the corresponding *C* functions which describe their behavior. Program *user_lib* will compile the user-defined library and link it to the standard-modules library.

3. Before running the simulator the user specifies in a command file, *foo.com*, the signal values for the design's primary inputs. In Section 3.4 we introduce a set of simple commands which guide the operation of the simulator and initialize the values of the primary input signals.

4. Having specified *foo* both structurally and behaviorally we are ready to run the simulator by supplying the input file *foo.com*. To run the simulator we execute program *clsim*. The simulator derives the event graph, recorded in the file *foo.sim*. In a report file, *foo.rep*, the simulator reports possible violations detected during simulation, for example, set-up or hold time constraint violations, oscillation, undefined signals, incorrect input values etc. Finally the *clsim* program, as it reads the netlist file *foo.net*, creates a file *foo.nam* which contains all the names of the signals defined in the *foo.pdl* file. This information will be used from the timing constraints specification language parser to identify the acceptable identifier names for signals and events.

5. The user has to specify the timing constraints which characterize the design *foo*, by writing an ATCSL program. We call this program: *foo.con*.

6. Finally, the user has to run the verifier. The inputs for the verifier are the constraint file, *foo.con*, and the event graph *foo.sim*. The verifier will parse the constraint file and after performing the verification process will report identified violations. The verifier produces two report files: errors are reported in file *foo.err*; warnings are reported in file *foo.war*. When trace analysis is requested by the user, signal traces are reported in file *foo.trc*. A trace is a detail description for every specified signal of the design. The description includes the events which constitute the signal, their timing characteristics and their dependency relations. A trace analysis will help the designer identify or better understand the reason for a violation. To run the verifier we execute program *verify*.

All the names of the files generated by CLOVER are comprised of two parts separated by a dot (.); we call them the prefix and the extension. For example, file *foo.con* has *foo* as its prefix and *con* as its extension. The prefix usually corresponds to the name of the design under verification. The extension characterizes the type of the file (constraint file, *pdl* file etc.). As a common procedure in order to run a CLOVER's program we need only to provide the name of the design under verification (i.e., *clover_program foo* in the previous discussion); implicitly CLOVER will append the appropriate extension. However, and in order to permit verification of the same design under different parameters each program provides certain options which permit the user to assign names to files with a name prefix different from the name of the design under verification. For example, we may want to verify design *foo* under two different set of constraints specified in files *foo1.con* and *foo2.con*. To do that the verification program *verify* provides an option (*-C*) to assign, as an argument, an explicit name for the constraint file. In that case we would run the verification program two times as: *verify -Cfoo1.con foo* and *verify -Cfoo2.con foo*.

The user, after specifying the necessary files, can either run the individual programs one by one (as indicated in the above steps) or can call *clover foo* to automatically apply all the necessary steps for the verification of the design named *foo*. *Clover* will automatically step through each verification step.

By running *clover* with the option *-L* a name for the user-defined library is specified. Thus *clover -Llibrary_name foo* will verify the design *foo* using *library_name* as the user-defined library. A textual interface based on UNIX *curses* (a package of screen functions with optimal cursor motion) has been implemented which allows the user to verify a design at once or apply the individual steps one by one according to the information provided in the next sections. The use of the interface is self-documented and it is initiated by calling the program *view_clover*. In Appendix B we summarize the necessary steps which need to be taken by the user for a complete verification session.

For every file used or created by the system (during verification of *foo*) we specify in the following table which component of CLOVER uses or creates that file.

File	Created by	Used by
foo.pdl	User	*pdl2net*
foo.pif	*pdl2net*	*pdl2net*
foo.net	*pdl2net*	*clsim*
foo.com	User	*clsim*
built-in library	System	*clsim*
user_library	User	*clsim*
foo.sim	*clsim*	*verify*
foo.rep	*clsim*	User
foo.nam	*clsim*	*verify*
foo.con	User	*verify*
foo.err	*verify*	User
foo.war	*verify*	User
foo.trc	*verify*	User

The details of the use of the programs implementing the desired functions of each particular component of CLOVER are presented in the following sections. The document is organized as follows: in Section 2 we present the hardware description language PDL-e. Section 3 describes the behavioral specifications of standard and user-defined hardware modules and contains the information needed to run the simulator. The timing constraints specification language, ATCSL, and the use of the verifier are presented in Section 4. Finally, in Section 5 we provide information needed for the initial setup of the system and the current status of its development.

2 PDL-e: A Hardware Description Language

To structurally describe a design, CLOVER uses a hardware description language, PDL-e, whose properties are presented in this section. PDL [10], is a generator language, developed at Princeton, for register transfer design. The user writes *C* programs that generate register-transfer machines and optimize the logic design using the Berkeley optimizer, misII [3]. PDL-e is an extension of PDL with different objectives: those of a hardware description language. Since PDL-e has been built on *C*, it offers all of the advantages of a high-level programming language. The use of functions makes hierarchical design feasible and natural. Regular structures are very easy and elegant to describe. For some designs, there is an analogy between the program that generates the structural description of the design and the one which specifies the behavior of it (see examples in [10]).

As we said earlier, a design in the structural domain of description is viewed as a collection of functional modules interconnected with wires. These wires are connected to the input and output ports of the modules. In order to specify wires, PDL-e offers a *wire* basic type. Variables referring to wires have to be declared as of type pointer to *wire*. For such a declared variable to represent a "physical" wire a built-in function (*get_wire*) exists which creates a unique "physical" wire. Furthermore, other provided built-in functions allow the explicit naming of a wire, the changing of a wire's name and the connection of two wires together (see next section for a complete presentation). For a *wire* variable to have some meaning it has to be associated with a module's input or output port. This is accomplished when we structurally describe a functional module. In the structural domain we can view a functional module as being a black box (in terms of its behavioral characteristics) with a determined number of input and output ports. In order to "create" the structural description of a functional module we need to specify the module's input and output ports. The built-in, PDL-e function, *create_fun* fulfill this purpose. *Create_fun* associates wire variables with the input and output ports of the module under description.

Given a design named *foo* the user writes the structural description of it in the file *foo.pdl*. In order to derive the netlist of the implementation first we run the program *pdl2net*. By calling *pdl2net foo* the program will look for a file named *foo.pdl* and it will produce an intermediate representation of the netlist in a file called *foo.pif*. This intermediate representation indicates the input and output wires for each defined functional module, and contains information about the connectivity of wires and the equivalence of names for wires assigned more than one names. The user though does not have to be aware of the contents of the *pif* file. The final form of the netlist which will be used as an input for the simulator is derived in the file *foo.net*.

2.1 PDL-e Built-in Functions

The *wire* is the basic type in PDL-e. Furthermore, PDL-e offers the user the ability to group a set of wires using the *wire_array* type. The built-in functions of PDL-e are basically functions to manipulate wires and wire_arrays. We should be able to create a wire, to name a wire, to change a wire's name and to connect wires together. Finally PDL-e defines the built-in function *create_fun* which takes a variable number of arguments; it is used to describe arbitrary functional modules with certain input and output ports.

wire *get_wire() The function *get_wire* returns a pointer to a wire, creating a unique wire. With this function a declared wire variable is instantiated as a real "physical" wire.

wire_array *get_wire_array(int n) The function *get_wire_array* returns a pointer to an array of *n* wires, creating *n* unique wires.

wire *name_wire(char *name) The function *name_wire* returns a pointer to a wire with name the string *name*. That way the user can assign names to input and output ports of modules. Note the distinction between the wire variable name and the wire *name*.

wire_array *name_wire_array(char *name, int n) The function *name_wire_array* returns a pointer to an array of *n* wires, with name the string *name*.

connect(wire *w1, wire *w2) The function *connect* connects wires *w1* and *w2*. This is the equivalent of connecting, through a wire, input and output ports of two modules.

connect_array(wire_array *w1, int f1, int t1, wire_array *w2, int f2, int t2) The function *connect_array* connects wires *w1[f1]* through *w1[t1]*, to wires *w2[f2]* through *w2[t2]*, respectively. Connecting two wire arrays with $t1 - f1 \neq t2 - f2$, results in a violation and the program is terminated.

wire *change_name(wire *w, char *name) The function *change_name* changes the name of wire *w* to *name* and returns a pointer to the renamed wire. The old name is still valid. We usually use this function to assign names to the output wires returned by the function *create_fun* (see description below).

wire_array *change_name_array(wire_array *w, char *name, int f, int t) The function *change_name_array* changes the name of wires *w[f]* through *w[t]* to *name*, and returns a pointer to wire_array *w*. The old name remains valid.

wire_array *create_fun(char *format, arg1, arg2, ...) The function is used for the structural description of user-defined or standard-defined modules. In Appendix A we present the standard-modules (and their names) currently defined in our built-in library.

Create_fun takes a variable number of arguments and returns a pointer to a wire_array which represents the output wires of the defined module. The first argument *format*, is a string constituted of conversion specifications each of which is used to interpret the rest of the arguments of *create_fun* (as does *printf()* in *C*). Each conversion specification begins with a % and ends with a conversion character. Six conversion characters are supported; with them the user can specify the name of the described module, the input and output ports of it (using wire or wire_array types) and the delays associated with the wires connected to the input and output ports of the module (only constant wire delay values are supported in the current version of the system). The following table explains in more details the use of each conversion character.

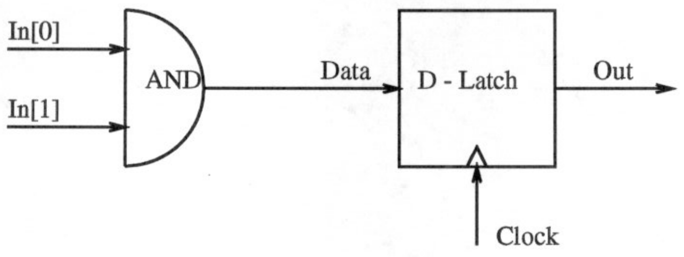

Figure 2: D-latch and AND gate in series

Conversion Char	Input Data; Argument Type
s	Input is a string characterizing the name of the described module
w	Input is a pointer to a wire representing an input wire. If the conversion character *w* is followed by the character *c* (*wc*) that will indicate that the corresponding input wire is a control signal. This information can be used when we determine timing causality dependencies.
W	Three arguments should correspond to this conversion character. The first two denote the wire_array limits (*from, to*). The last is a pointer to a wire_array representing input wires. *W* can be followed by the character *c* to indicate a wire_array of control signals (*Wc*).
d	If present, should immediately follow a %w conversion specification. Input is an integer which determines the delay, of a previously specified wire.
D	If present should immediately follow a %W conversion specification. Input is a set of integers which determine the delay of a previously specified wire_array. The first number of the set indicates the cardinality for the rest of the integer set.
O	Input is an integer which determines the number of output ports of the defined module.

2.2 PDL-e Specification Examples

As our first specification example we present the PDL-e description for the circuit shown in Figure 2 (an *AND* gate and a *D-latch* connected in series). We call this design an *and_dlatch.*

```
#include "pdlpif.c" /* Must appear in all pdl-e descriptions */

/* The macro D_latch(i, en) will "create" the structural description of the */
/* module D_latch, with two inputs of type wire. The macro is defined to */
/* return a variable of type wire because the D_latch has a single output port */
/* (this is accomplished by taking as output the first element (indexed by [0]) */
/* of the wire_array returned by create_fun). */
#define D_latch(i, en)      create_fun("%s %w %w %O", "D_latch", i, en, 1)[0]

main ()

{
wire *cl, *d, *out;
```

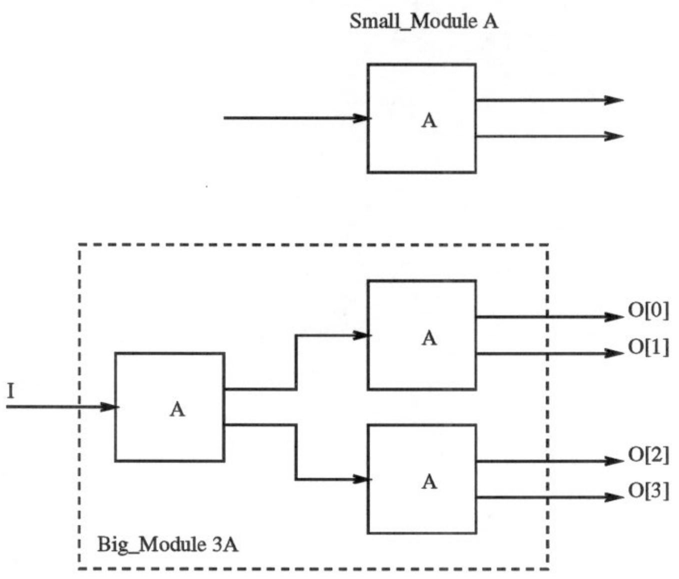

Figure 3: Hierarchical Connection of Modules

```
wire_array *i;


    /* Create a wire_array with two elements and name "In"  (In[0], In[1]) */
    i = name_wire_array("In", 2);

    /* Create a wire with name "Clock" */
    cl = name_wire("Clock");

    /* Create the module AND with output named "Data"); */
    /* AND is predefined in pdlpif.c */
    d = change_name(AND(i[0], i[1]), "Data");

    /* Create the module D_latch with inputs d, cl */
    /* and output named "Out" */
    out = change_name(D_latch(d, cl), "Out");

}
```

Note here that all *pdl* files should include the file *pdlpif.c*. In *pdlpif.c* there are predefined definitions for the standard gates AND, OR, NOT, NAND, NOR and XOR (their corresponding implementations are modules *_and, _or, _not, _nand, _nor* and *_xor* defined in Appendix A). As a result the user does not have to "create" them using the built-in function *create_fun*. The next example will show how we can take advantage of the high-level structure of PDL-e to hierarchically describe a design.

We want to describe the specification of the module *Big_Module 3A*, which consists of three modules *Small_module A*, connected as shown in Figure 3. The PDL-e description follows:

```
#include "pdlpif.c"

wire_array *Small_Module (wire *i)
{
    /* Create Module Small_Module */
    return create_fun("%s %w %O", "A", i, 2);
```

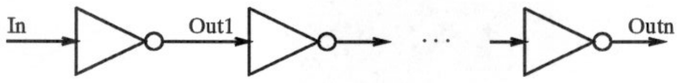

Figure 4: N inverters in series

```
}

wire_array *Big_Module (wire *i)
{
wire_array *tmp, *tmp_out;

    tmp = get_wire_array(4);
    /* Big_Module 3A is created by calling Small_Module()
      with a Small_Module output as an input */
    tmp_out = Small_Module(i);
    connect_array(tmp, 0, 1, Small_Module(tmp_out[0]), 0, 1);
    connect_array(tmp, 2, 3, Small_Module(tmp_out[1]), 0, 1);
    return tmp;
}

main ()

{
wire *i, *o;

    /* Name input wire as "I" */
    i = name_wire("I");
    /* Create module Big_Module with input "I" and output "O" */
    change_name_array(Big_Module(i), "O", 0, 3);

}
```

Finally the last example will show how naturally regular structures can be specified within PDL-e. We will specify N inverters connected in series as they appear in Figure 4.

```
#include "pdlpif.c"
#define N 4

main ()
{
wire *in, *tmp;
int i;
char *name[] = {"Out1", "Out2", "Out3", "Out4"};

    in = name_wire("In");
    for (i = 0; i < N; i++) {
        tmp = change_name(NOT(in), name[i]);
        in = tmp;
    }
}
```

3 Describing Behavior with CLOVER

In CLOVER we use the *C* programming language to program the desired behavior of a functional module. As a result the behavior of a module is represented as a *C* function. As stated earlier, we distinguish two kinds of functional modules: standard and user-defined. A built-in library exists which describes the behaviors for standard components like gates, latches, multiplexers. User-defined modules can range in complexity from a simple gate to a complex hardware component. The user has the ability to code the behavior for any user-defined component of the design, at different levels of detail. This is a very important feature for hierarchical analysis and verification. The names of the user-defined functions should be the same as the corresponding names used to "create" these modules in the *pdl* file. During the operation of the circuit, functional modules communicate with each other through their input and output ports. Output signals appear, after some propagation delay time, at the output ports as a response to an input signal vector. In describing the behavior of a module as a *C* function, the input and output ports of the module and the corresponding propagation delay times are arguments of the *C* function. The *C* function receives its input values through the arguments which correspond to the input ports of the module and returns its output values through the arguments which correspond to the output ports of the module. At the same time, the arguments of the *C* function which correspond to the propagation delays through the module return the propagation delay times for the reported output values. That way, two communicating modules are represented during simulation as two *C* functions which exchange information through their corresponding arguments. The parameter set of a user-defined behavioral description *C* function is presented in Section 3.1.

An extended value system is used for the characterization of the module's output values. The following table defines the values of the extended value system and their meaning:

VALUE	MEANING
0	Value stable at zero
1	Value stable at one
r	Signal monotonically increasing from zero, called *rising*
f	Signal monotonically decreasing from one, called *falling*
c	Signal is changing, any transitions acceptable
s	Signal is stable either at zero or one
u	Same as changing, but it is used only for initialization, called *undefined*

The user does not have to use the undefined value. The simulator uses this value internally for initialization purposes. As an example, we present below the extended functional table of the function NOR, based on the extended value system:

NOR	0	1	r	f	c	s	u
0	1	0	f	r	c	s	u
1	0	0	0	0	0	0	0
r	f	0	f	c	c	c	u
f	r	0	c	r	c	c	u
c	c	0	c	c	c	c	u
s	s	0	c	c	c	s	u
u	u	0	u	u	u	u	u

To illustrate how we program the behavior of a module, we present the program for the behavioral description of an *AND* gate with two inputs and one output. We assume that the gate propagation delay can range from *10* to *15ns* for rising transitions, and from *9* to *15ns* for falling transitions. For simplicity we assume a two-value system with the values zero and one. The use of the parameter *mode* will be explained in Section 3.2.

```
int and (int *in, int **out, int **tmin, int **tmax, int *mode)
{
    /* Time low to high minimum = 10 - maximum = 15 ns */
    /* Time high to low minimum = 9 - maximum = 15 ns  */
    /* *mode is initialized to -1 by the simulator     */

    if (in[0] == '0' || in[1] == '0') {
        /* if either input is zero the output will become zero */
        o[0][0] = '0'; tmin[0][0] = 9; tmax[0][0] = 15;
    }
    else {
        /* if both inputs are one the output will become one */
        o[0][0] = '1'; tmin[0][0] = 10; tmax[0][0] = 15;
    }
}
```

Since the behavior of the functional modules is programmable, we can explore ways to describe more accurately the behavior of the analyzed design. Consider for example, the behavior of a flip-flop. We know that if the set-up constraint is violated the flip-flop's output may fall into a metastable state for an unbounded (practically "large") amount of time. In Section 3.2 we show how violations of the set-up and hold time constraints of standard modules like flip-flops and latches can be detected (a detailed description of the behavior of a *D-latch* is presented in Section 3.2). A flip-flop behavioral description which accounts for a set-up or hold time constraint violation will determine dynamically the output of the module when the violation is detected. A metastable state, in this case, can be represented by assigning to the flip-flop's output signal a changing value for a "large" amount of time; the exact amount of time is also programmable. A static approach will detect a violation without propagating its effects to the rest of the circuit. That way a less accurate behavior of the design will be obtained. The dynamic approach will propagate the effects of the violation to the rest of the circuit. That way, however, we may inhibit the verification of constraints not dependent on the detected timing violations. For example when a changing value is used to represent the output signal value of a flip-flop in a metastable state, constraints which use the same output signal with zero or one value will fail to evaluate because these values will never appear. Since the behavior is programmable the static or the dynamic approach can be used; their combination may provide the designer with a better understanding of the design.

3.1 Parameters Definition for a User-Defined Function

The parameters and the meaning of a user-defined behavioral description function, are presented next:

int *in *In* is a pointer to the array of the module's input values. The size of the array equals the number of the module's input ports (this information is contained in the netlist of the design).

int **out *Out* is an array of pointers. The size of the array equals the number of the module's output ports. Each pointer points to an array of signal values. Each array is the response to *in* on a particular output port. It is thus clear that we can model a behavior where one input transition causes more than one output transitions on the same port.

int **tmin *Tmin* is an array of pointers with size equal to the number of the module's output ports. Each pointer points to an array which contains the minimum propagation delay times for every output signal on every output port. The designer has total freedom in assigning these delays based on the input values. For example differences between the delays of rising and falling transitions can be very easily coded in the behavior.

int **tmax Same as *tmin* to determine the maximum propagation delay time for the output signals.

int *mode *Mode* is an array of pointers with size equal to the number of the module's output ports. *Mode* is used to control delayed evaluation discussed in Section 3.2. The simulator initially initializes the elements of the array to value -1. The following interaction takes place during simulation: when the functional module wishes to delay evaluation for some output signal at port *out[i]*, the corresponding element of **mode*, *mode[i]*, is set to a positive value t which denotes (to the simulator) that the output signal at port *out[i]* will be evaluated in delayed mode after time t (see next section). For user-defined modules the user's C code for the module behavior is responsible for assigning the delayed time t. The simulator will delay the evaluation for time t and will assign to *mode[i]* the negative value -2. After time t the delayed evaluation will take place; the negative value -2 of *mode[i]* will indicate (to the module) that the incoming evaluation is one that had been rescheduled for delayed evaluation at the output port *out[i]*. Note here that we do not carry along the value of the delayed time t. As a result, we cannot distinguish between more than one incoming delayed evaluation referring to the same output signal. However, this situation never occurs in practice; incoming events scheduled at the same port at time t_1 will be preempted by earlier events scheduled at time t_2 when $t_2 > t_1$. Unless the functional module again assigns *mode[i]* a positive value, the simulator will reset *mode[i]* to -1.

3.2 Handling State Information During Event Driven Simulation

An important shortcoming of the standard event-driven process is that it is memoryless. An event may have an effect on the circuit only at the time it comes out of the event queue. After that time, the event is lost. If, during the behavioral evaluation of a module, we allow events which come out of the event queue to reenter the queue and reappear (delayed) after some specified time, then some sort of memory (state) is added on the standard event-driven mode of operation. In that case we say the the module is evaluated in ***delayed*** mode. The mode of evaluation is determined by the use of the *mode* parameter, which either is negative meaning standard mode, or is an integer which specifies the time the event should reappear in the event queue. The default state of operation is the standard mode; the system initializes the *mode* parameter to a negative value (-1).

Consider as an example the behavioral description of a *D-latch*. In the standard mode of operation (no delayed mode) the *D-latch* will pass its input value to its output port whenever the enable signal is "on" (enable is "on" only when it assumes the value one). At the same time we would like to ensure that the last input signal (while the enable was "on") assumed its value at least a *set-up* time before the falling edge of enable (that is when enable went "off"). Finally for the ***hold*** time to be satisfied the input which changed last before the enable went "off" should retain its value at least a ***hold*** time after the falling edge of the enable signal.

From the above description it is clear that in order to ensure that no *set-up* time violation exists when an input signal appears at the input port of the *D-latch*, we should wait until at least a *set-up* time or until the falling edge of the enable signal appears. Since the standard event-driven mode of operation is memoryless this check cannot be made during running time and as a result the behavioral description of the *D-latch* will be static (the response of the *D-latch* will not depend on whether a *set-up* time violation occurred or not). To allow for a dynamic behavioral description we should evaluate the *D-latch* module in ***delayed*** mode.

In a dynamic description, the *D-latch* operation can be described in two states. In the initial state the input event appears for the first time; If the enable signal is "on" we reenter the input event into the queue specifying that the *D-latch* output should be evaluated for this event in delayed mode after a time equal with the *set-up* time. Next time this event appears we evaluate the module in delay mode and we check for a possible *set-up* time violation. If the enable signal is "off" and we are in the initial state then we should check for a ***hold*** time violation. If the enable signal is "off" and we are in delayed evaluation mode then we should check for a *set-up* time violation. In case a new input event appears before the delayed event then the state of the operation is reinitialized to the starting state. The old delayed event will be preempted when it comes out of the queue since it was originally scheduled before the new input event. Note that if two events occur simultaneously, e.g., both enable and input change at the same time, the events will come out of the queue for processing one at a time in arbitrary order. At the end of this section we give the code for the complete behavioral description of a *D-latch*.

It is interesting to see what else we have accomplished here: the behavior of a *D-latch* is described in a way that the simulator can check the standard timing constraints of the *D-latch* (set-up and hold time) while evaluating the *D-latch* description. In this way, a verification for these constraints is provided for free, as part of the simulation process. Any violation found is reported in the simulator's report file. The same standard constraints can be expressed within CLOVER's specification model and verified by the verifier. However, it would be inconvenient for the user to specify the same constraints for every standard module like a latch or flip-flop of a design's implementation and name every input and output signal of these modules (signals should be named in our specification model).

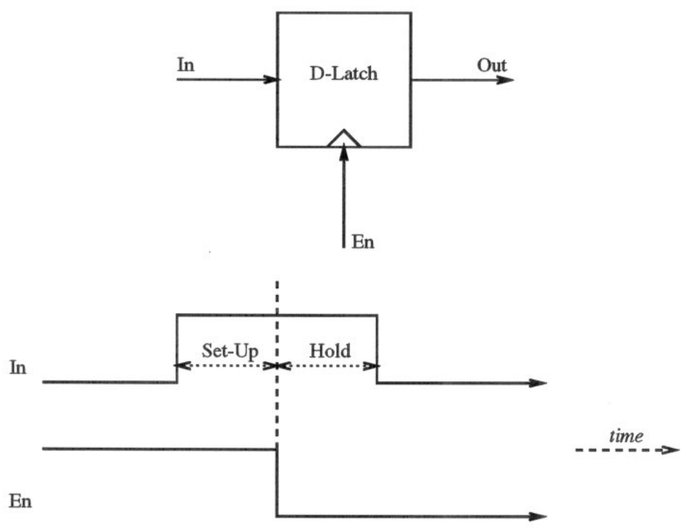

Figure 5: D-Latch Set-up and Hold Times

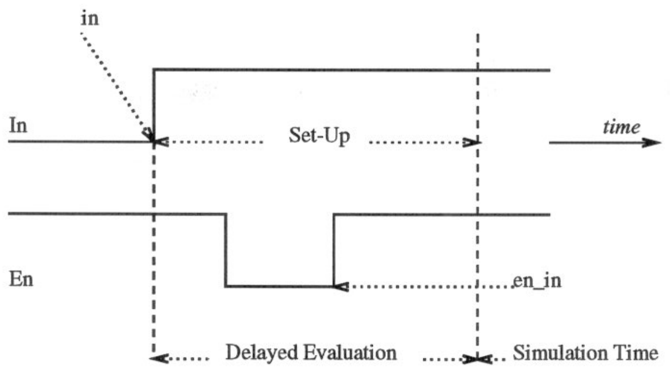

Figure 6: Set-up Violation

Behavioral Description of a D-Latch

```
extern int get_ev_info(char, int, char, int, int *, int *, int *);
extern int retain_value();
extern int SIM_TIME;

int D_latch (int *in, int **out, int **tmin, int **tmax, int *mode)
{
int in, width, elem;
int en_in, en_width, en_elem;
int set_up = 5, hold = 20;

    /* SN74LS373; CL = 45pf; Rl = 667 w */
    /* Time low-to-high (data) typical = 12 - maximum = 18; */
    /* Time high-to-low (data) typical = 12 - maximum = 18; */
    /* Time low-to-high (enable) typical = 20 - maximum = 30; */
    /* Time high-to-low (enable) typical = 18 - maximum = 30; */
    /* set_up = 5; hold = 20; (See Figure 5) */
    /* Data come on input port zero */
/* Enable comes on input port one */

/* The get_ev_info function returns the initial time, */
/* the width and the index of an event by using */
/* information from the event graph */
get_ev_info('i', 0, in[0], -1, &in, &width, &elem);
get_ev_info('i', 1, in[1], -1, &en_in, &en_width, &en_elem);

if (in[1] == '1') { /* Enable is on */
    /* Check if enable or data signal changed last */
    if (en_in == SIM_TIME) { /* Enable signal changed last */
        if (in[0] == '1') { tmin[0][0] = 20; tmax[0][0] = 30; }
        else
        if (in[0] == '0') { tmin[0][0] = 18; tmax[0][0] = 30; }
        else { /* For every other value assume worst case values */
            tmin[0][0] = 18; tmax[0][0] = 30;
        }
        out[0][0] = in[0];
    }
    else { /* Data signal changed last or Delayed Evaluation */
        if (mode[0] == -2) {
            /* in is the original initial time of the delayed event */
            if (in < en_in) {
                /* Set-up constraint is violated and we report it (not shown) */
                /* A glitch should have occurred on enable (Figure 6) */
                /* Keep the previous value */
                out[0][0] = retain_value(); return;
            }
            /* The set-up time constraint was not violated */
            out[0][0] = in[0];
            tmin[0][0] = 12 - set_up; tmax[0][0] = 18 - set_up;
        }
        else {
            /* Delay evaluation for a set_up time period */
            mode[0] = set_up;
            /* The simulator puts the delayed input value */
            /* into the simulation queue */
        }
    }
}
else { /* Enable is not one */
    if (en_in != SIM_TIME) { /* Data signal changed last */
        /* Check for set-up time violation */
        if (mode[0] == -2) {
            /* Set-up constraint is violated and we report it (not shown) */
            /* The input signal is not latched and its effect is lost */
            /* Keep the previous value */
            out[0][0] = retain_value(); return;
        }
        /* Check for hold time violation */
        if (SIM_TIME - en_in < hold) {
            /* Hold time is violated, output becomes not stable */
            out[0][0] = 'c'; return;
        }
    }
```

```
        /* Check if delayed input value occurs simultaneously with enable */
        if (en_in == set_up + in && mode[0] == -2) {
            /* Latch the delayed input value */
            out[0][0] = in[0];
        }
        else {
            /* Keep the previous value */
            out[0][0] = retain_value();
        }
    }
}
```

3.3 Simulator Built-in Functions

For a more complete and accurate description of the behavior of the circuit components, access to global simulation time is provided and there are built-in functions which give the user the capability to obtain information about the event graph at any state of the simulator. Thus information about the initial time and the width of events as well as the dependency relation between events can be obtained. This information can be used as conditional qualification in the behavioral description of the user-defined functional modules. Their definition follows:

int get_ev_info(char inout, int p, char c, int index, int *in, int *w, int *elem) This function returns timing information about specific events appearing on the input-output ports of the circuit modules. The *inout* parameter determines if the event appears on an input or output port *p* of the module (we also represent that as a *[inout, p]* port). It takes values from the set { *'i', 'o'* } where *'i'* denotes an input port and *'o'* denotes an output port. The port numbers follow the index numbers of their corresponding input or output signals. With the parameter *c* the value of the event is specified. It takes values form the set { *'0', '1', 'r', 'f', 's', 'c'* }. Parameter *index* specifies the index of the event. When it assumes the value -1 then we refer to the latest event created up to this point of the simulation. The initial time, width and index of the event are returned with the parameters *in*, *w* and *elem* respectively.

We saw in the *D-latch* behavioral description that we used this function to obtain information about the timing characteristics of the enable and the input signal. We were able to determine which of these two signals changed last and describe accordingly the latch behavior.

char get_value(char inout, int p) The function returns the value of the latest event which appears on port *[inout, p]*. This function can be particularly useful in cases where the behavior of a module depends not only on its current inputs but also on its previous outputs (e.g., an RS flip-flop). Then we can use *get_value()* to find out the value of the old outputs and use it to calculate the new ones.

int retain_value() This function is used to indicate no change to an output value. It is useful to describe behavior of memory elements like latches and flip-flops, for example, where the output signal retains its value when then enabling signal is “off”.

int cau_depend(char *port1, char c1, int index1, char *port2, char c2, int index2) The function returns one(1) if the event with value *c1* and index *index1* on port *port1* timing causes the event with value *c2* and index *index2* on port *port2*. Otherwise the function returns zero(0). By *port1* we refer to the port where the signal with name *port1* appears (similarly for *port2*).

One of the problems with min-max delay simulators is the report of spurious errors (e.g., glitches) in cases where in reality these may never take place (e.g., the problem of hazard detection due to reconvergent fanouts). The reason for that is because of the pessimism involved in the min-max delay calculations. By using the function *cau_depend* we can improve the accuracy of the reported errors by excluding from consideration sequences of signals which can never occur. For, example if we know that signal B timing causes signal C then we know that C would never precede B. If B and C were input events to a functional module $\mathcal{M}$, then we can program the behavior of $\mathcal{M}$ accordingly (using *cau_depend* in order to avoid spurious errors involving signal C preceding signal B).

SIM_TIME This is not a function but rather a global variable. It gives the user access to the internal global time of the simulator.

Before running the simulator and if new user-defined modules have been defined, we have to compile the user-defined library and link it to the standard modules library. To do that the user uses the program *user_lib*. By calling *user_lib library_name*, the program will parse the user-defined library, *library_name*, identifying the names of user-defined modules and subsequently will compile and link them with the standard library. The parse program will create a *C* structure relating user-defined functions and their names. The first field of the structure is a character string where the name of the user-defined module (in the *pdl* description) is stored. The second field is a pointer to the function which specifies the behavior of the same module. This structure is recorded in a file named *fun_library_name.c* where *library_name* is the name of the user defined library. Note that *library_name* can either be a plain file name or a full path name. After linkage, an executable version of the simulator will be created at the user's current working directory.

3.4 Running the Simulator

In order to run the simulator we have built a small set of commands to guide its operation. These commands are presented next. In the description of a command's format we adopt the following convention: everything in bold letters represent characters which should be typed as they appear; everything in italics are variables assigned values from the user, according to the specifications of the command.

set The *set* command is used to assign signals (initial values) to the primary inputs of the circuit. The format of the *set* command is:

set *signal_name n* $value_0$ $start\_time_0$ $\ldots$ $value_n$ $start\_time_n$

By *signal_name* we specify the name of the wire where the specified values appear. Number *n* is an integer which specifies the number of distinct events (values) assigned to *signal_name*. The values of the events and their starting times are specified next. For example, the command:

set *Signal 2 r 0 f 10*

will create event *Signal.r[0]* with starting time *0* and event *Signal.f[0]* with starting time *10*.

set_array The *set_array* command has similar functionality as the *set* command but it is used in order to assign values to an array of signals. The format of the *set_array* command is:

set_array *from-to signal_name value_assignments*

The numbers *from* and *to* are integers specifying the array bounds. The part *value_assignments* is similar to the part of the *set* command which follows the *signal_name* and it should assign values to the elements of the signal array *signal_name* from *from* to *to* (events of first array element are assigned first, events of second array element are assigned second and so on). For example, the command:

set_array *0-1 Signal 2 r 0 f 10 2 c 10 s 20*

will create the events *Signal[0].r[0], Signal[0].f[0], Signal[1].c[0]* and *Signal[1].s[0]* with starting times *0, 10, 10* and *20* (time units) respectively.

periodic Special provision exist in order to give a periodic signal as input, something which is necessary for synchronous designs with clocks. The format of the *periodic* command is:

periodic *signal_name period iteration value_assignments*

period is an integer which determines the period of the periodic signal. *Iteration* is an integer which if negative specifies infinite periodicity and if positive specifies the number of the desired iterations of the periodic signal. The part *value_assignments* is similar to the part of the *set* command which follows the *signal_name* and it should assign values to the events (events of the first period) of the periodic signal *signal_name.*

periodic_array The *periodic_array* command has similar functionality as the *periodic* command but it is used in order to assign values to an array of periodic signals. The format of the *periodic_array* command is:

periodic_array *from-to signal_name value_assignments*

The numbers *from* and *to* are integers specifying the array bounds. The part *value_assignments* is similar to the part of the *periodic* command which follows the *signal_name* and it should assign values to the elements of the signal array *signal_name* from *from* to *to*. Each signal of the array can be assigned its own period.

print The *print* command is used to print the waveforms of signals selected by the user. The format of the *print* command is:

print *signal_name value*

The simulator reports in the *rep* file the timing characteristics (starting time and ending time) for all events with name *signal_name* and value *value.* Violations for non-existent signals are also reported in *rep*.

print_array The *print_array* command is used to print the waveforms of array of signals selected by the user. The format of the *print_array* command is:

print_array *from-to signal_name value*

The integers *from* and *to* specify the array bounds. Out of bound indices are reported in the *rep* file.

time The *time* command is used to print the current simulation time. The format of the command is:

time

clock When we run the simulator for a certain number of clock cycles, the *clock* command is used to determine the desired clock period. The format of the command is:

clock *clock_period*

where *clock_period* is an integer specifying the desired clock period.

run The simulator starts running using the *run* command. The command has three possible formats:

run time *time*

Time is an integer which specifies the maximum simulator time the simulator is allowed to run. The second format of the command is:

run cycle *cycle_number*

The simulator runs for a number of cycles equal to *cycle_number.* The clock period had to be given before with the *clock* command. The last format of the command is:

d	MEANING
1	unit delay model
2	minimum - maximum delay model
3	probabilistic delay model
4	slope delay model

Table 1: Delay Models

run for

The simulator runs "forever" until its event queue becomes empty. In reality there is an internal time-out period after which the simulator stops running; its current value is 3000 time units. Note that infinite execution can still occur because simulator time need not progress.

erase The *erase* command initializes the state of the simulator, and it is used when we want to test the design for more than one input vector (case analysis). Case analysis may be necessary in order to obtain more accurate results and to avoid problems like false-paths [11]. The format of the *erase* command is:

erase

Expressions can be commented in a *.com* file using symbol # as a delimiter.

Now we are ready to run the simulator. The simulator program is called *clsim*. Given the design *foo* the command *clsim foo* will run the simulator producing an event graph. Following the same convention we used for the other programs of CLOVER, the user is able to use names for the files used by or derived from the simulator with a prefix part different from *foo*. The following options are used for that purpose:

-N The format *-Nfoo1* will result in using file *foo1.net* as the input netlist file for the simulator.

-C The format *-Cfoo1* will result in using file *foo1.com* as the command file for the simulator.

-S The format *-Sfoo1* will derive the event graph for *foo* in file *foo1.sim*

-R The format *-Rfoo1* will result in using file *foo1.rep* as the report file of the simulator.

The simulator supports both the inertial and the transport timing models [1]. Transport model is the default timing model. Running *clsim* with the option *-i* switches the timing model of the simulator to the inertial model. Finally the user is able to determine the delay model used during simulation. This is accomplished with the option *-d*. The format used is *-ddelay_model* where *delay_model* is an integer which determines the desired delay model. Table 1 associates *d* with different delay models. The default model and the only currently implemented model is the min-max model.

Coming back to the example in Figure 2 we create the following *com* file to simulate design *and_dlatch*:

```
# Initialize the array In; In[0] is stable between 0 and 80ns
and changing the rest of the time. In[1] is stable between
20 and 95ns and changing the rest of the time. #
set_array 0-1 In 2 s 0 c 80 3 c 0 s 20 c 95

# The Clock has a period of 100ns. For the first
period it assumes value one between 50 and 70 ns. #
periodic Clock 100 -1 3 0 0 1 50 0 70
```

```
# run the simulator for two cycles;
every cycle has a period of 100 ns. #
clock 100
run cycle 2

# Print the timing characteristics for signal
Out with values stable and changing. #
print Out s
print Out c
```

After running *clsim and_dlatch* we will get the event graph for *and_dlatch* in file *and_dlatch.sim* and the report file for the simulation in file *and_dlatch.rep.* In file *and_dlatch.nam* the names of the signals defined in *foo.pdl* are recorded. In this case four names are recorded: *In, Out, Data* and *Clock.* The *rep* file contains the result of the *print* command. For each valued signal specified in *print* the interval of time during which this signal holds is reported. The report file is presented next:

```
Signal: Out Value: s
80 168
Signal: Out Value: c
68 80, 168 infinity
```

By *infinity* (an **INT_MAX** integer) we indicate that signal *Out.c* was holding value *changing* when the simulation was terminated.

4 The ATCSL Language and the Verifier

We will start by presenting the basic concepts of events and signals as these concepts are defined in the specification model of CLOVER (formal definitions can be found in [6]). An informal presentation of the basic operators of the language will be given next, followed (in Section 4.1) by a formal syntax specification of ATCSL.

Signal & Event Under CLOVER's specification model an event is defined as the occurrence of a new value assumed by a circuit port at some point in time during the operation of the circuit. An event is characterized by a 5-tuple $< P, V, I, T, W >$ where P is the port assumed the new value, V is the value assumed by the port, I is the index of the event (the $I + 1th$ time, counting from $I = 0$, during the operation of the circuit where port P assumed the value V), T is the starting time of the event and W is the width of the event: the time during which the event retains the value assumed at time T.

A signal S on a port P is defined as the set of events, over time, which occur on port P. In particular we define a *V-signal* SV on a port P as the set of events which occur on port P and assume the same value V. The timing characteristics of a V-signal (width and initial time) are defined as vectors with the corresponding timing characteristics of the events which constitute the V-signal as elements. A V-signal characterized by the 5-tuple $< P, V, n, \vec{T}, \vec{W} >$ is represented as $P.V$ (in ATCSL a port is described by its name), where n is the number of events which comprise V-signal $P.V$. For example to refer to a V-signal on a port S with value rising(r), we say: $S.r$. For simplicity we will usually call $P.V$ a signal rather than a V-signal. Any event appearing on S with value r will be one of the n events which comprise signal $S.r$. We will refer to them as: *S.r[i]* where $0 \leq i < n$. We refer to the initial time of signals and events using the function *start.* For example the initial time of event *S.r[i]* is represented as: *start(S.r[i]).* Similarly we refer to the width of a signal or event using the function *width.* For example the width of signal $S.r$ is represented as: *width(S.r).* Finally the number n of events *S.r[i]* which comprise signal $S.r$ (cardinality of $S.r$) is referred by *card(S.r).*

Arithmetic Operators ATCSL supports the standard four arithmetic operations.

OPERATOR	MEANING
+	Addition
-	Subtraction
*	Multiplication
/	Division

Relative Operators ATCSL supports six relative order operations.

OPERATOR	MEANING
>	Greater
>=	Greater than or Equal
<	Less
<=	Less than or Equal
=	Equal
!=	Not Equal

Timing Assertion We use the arrow operator ($\longrightarrow$) to express a timing assertion relation. The definition of the arrow operator is that of the assertion relation. We represent the relation that signal $P_1.V_1$ *asserts*, signal $P_2.V_2$ within a time range from time *min* to time *max* as: $P_1.V_1 \longrightarrow$ *[min max]* $P_2.V_2$. This means that for each index i, $0 \leq i \leq$ *min(card($P_1.V_1$), card($P_2.V_2$))* if $P_1.V_1[i]$ occurs then $P_2.V_2[i]$ occurs and $\{start(T_2[i]) \geq start(T_1[i])+min\} \bigwedge \{start(T_2[i]) \leq start(T_1[i])+max\}$.

Composite Operators ATCSL provides a small set of 2-operand operators over signals and events, whose semantics can be defined using the arithmetic and relative operators of the language and the initial times and widths of the operands. These operands are provided because they capture composite concepts which are very commonly used in specification problems. The first three operators can be defined over two signals, two events or one signal and one event operands. We first define these three operators over two event operands. Then we adapt the definitions for the case of two signal operands and the case of one signal and one event operand. The fourth operator, *SYNC*, is defined only over two signal operands. For the definitions we will assume two event operands $E_1 :< P_1, V_1, I, T_1, W_1 >$ and $E_2 :< P_2, V_2, J, T_2, W_2 >$, and two signal operands $S_1 :< P_1, V_1, n, \vec{T_1}, \vec{W_1} >$ and $S_2 :< P_2, V_2, m, \vec{T_2}, \vec{W_2} >$, where $n \leq m$.

1. **WHILE**: We define E_1 *WHILE* E_2 as: $\{T_2 \leq T_1\} \bigwedge \{T_2 + W_2 \geq T_1 + W_1\}$. The operator *WHILE* specifies that the event E_1 appears and holds while signal E_2 holds.
2. **BEFORE**: We define E_1 *BEFORE* E_2 as: $\{T_1 < T_2\} \bigwedge \{T_1 + W_1 < T_2\}$. The operator *BEFORE* specifies that the event E_1 should start and cease before the appearance of the event E_2.
3. **OVERLAP**: We define E_1 *OVERLAP* E_2 as: $\{T_1 \leq T_2\} \bigwedge \{T_2 \leq T_1 + W_1 < T_2 + W_2\}$. The operator *OVERLAP* specifies that the event E_1, which starts earlier than event E_2, ceases to hold while the E_2 is still holding its value.
4. **SYNC**: We define S_1 *SYNC [from to]* S_2 as: $\forall i, 0 \leq i < n, \exists j, 0 \leq j < m$ such that $\{T_1[i] \geq T_2[j]+ from\} \bigwedge \{T_1[i] \leq T_2[j]+to\}$. The operator *SYNC* is used to specify that the events of signal S_1 appear in synchrony with and within a time range of the events of signal S_2. This is a very commonly used specification in synchronous designs, where usually the signal S_2 is a clock signal. The time bounds *from* and *to* may also assume negative values.

When the operands of the operators *WHILE, BEFORE, OVERLAP* are either two signals or one signal and one event the definitions are adapted as follows (We use *OP* to declare any one of the above three operators). We define S_1 *OP* S_2 as: $\forall i, 0 \leq i < n, P_1.V_1[i]$ *OP* $P_2.V_2[i]$. We define S_1 *OP* E_1 as: $\forall i, 0 \leq i < n, P_1.V_1[i]$ *OP* E_1.

WHILE, BEFORE, OVERLAP, SYNC are also defined over what we call *value_intervals*. A *value_interval* is a set of time intervals during which two signals assume or do not assume the same value. Given signals S_1 and

S_2 we characterize the former condition as: *val(* S_1 *)* = *val(* S_2 *)* and the latter as: *val(* S_1 *) != val(* S_2 *)*. For example, we can describe the specification that V-signal S_3 should appear *while* signals S_1 and S_2 assume the same value as follows:

$$S_3 \ WHILE \ (\ val \ (S_1) = val \ (S_2))$$

Like V-signals, value_intervals can be indexed to select a particular time interval.

Timing Causality Assuming two signals $< P_1, V_1, n_1, \vec{T_1}, \vec{W_1} >$ and $< P_2, V_2, n_2, \vec{T_2}, \vec{W_2} >$ we represent the relation that $P_1.V_1$ *timing causes* $P_2.V_2$ as:

$$P_1.V_1 \Longrightarrow P_2.V_2$$

Similarly for two events (the causality operator $\Longrightarrow$ is typed as the combination of symbols = and >).

Strong Dependency We represent the relation that signals

$$< OP_1, OV_1, m_1, O\vec{T_1}, O\vec{W_1} >, \ \ldots, < OP_k, OV_k, m_k, O\vec{T_k}, O\vec{W_k} >$$

strongly depend on signals

$$< IP_1, IV_1, n_1, I\vec{T_1}, I\vec{W_1} >, \ \ldots, < IP_l, IV_l, n_l, I\vec{T_l}, I\vec{W_l} >$$

as:

$$\text{INPUT}(IP_1.IV_1, \ldots, IP_l.IV_l) \Longrightarrow \text{OUTPUT}(OP_1.OV_1, \ldots, OP_k.OV_k)$$

Similarly for events.

Weak Dependency We represent the relation that signals

$$< OP_1, OV_1, m_1, O\vec{T_1}, O\vec{W_1} >, \ \ldots, < OP_k, OV_k, m_k, O\vec{T_k}, O\vec{W_k} >$$

weakly depend on signals

$$< IP_1, IV_1, n_1, I\vec{T_1}, I\vec{W_1} >, \ \ldots, < IP_l, IV_l, n_l, I\vec{T_l}, I\vec{W_l} >$$

as:

$$\text{PATH}(IP_1.IV_1, \ldots, IP_l.IV_l) \Longrightarrow \text{TO}(OP_1.OV_1, \ldots, OP_k.OV_k)$$

Similarly for events.

Logical Operators ATCSL supports three logical operations.

OPERATOR	MEANING
&&	Logical AND
||	Logical OR
!	Logical NOT

Universal Quantification is defined over the indices of the events which appear in a relation. It is represented in ATCSL using the command *FOR* of the form:

FOR (i; from; to; step;) { *Timing or Dependency Relations* }

Where $from \leq i < to$ and i increases each time by *step*.

Existential Quantification is defined (like universal quantification) over the indices of the events which appear in a relation. It is represented in ATCSL using the command *THERIS* of the form:

THERIS **list** { *Timing or Dependency Relations* }

Where in **list** we specify indices of events and their range. An example of a **list** can be:
i [from1 : to1], j [from2 : to2].

Conditional Evaluation We define two classes of conditionally evaluated relations: *conditional timing relations* (CTR) and *conditional dependency relations* (CDR). A *conditional timing relation* (CTR) is defined as follows:

1. *IF (conditional predicate)* { *Timing Relations or CTRs* }
2. *IF (conditional predicate)* { *Timing Relations or CTRs* } *ELSE* { *Timing Relations or CTRs* }
3. *FOR (i; from; to; step;)* { *Timing Relations or CTRs* }
4. *THERIS* **list** { *Timing Relations or CTRs* }

Similarly, a *conditional dependency relation* (CDR) is defined as follows:

1. *IF (conditional predicate)* { *Dependency Relations or CDRs* }
2. *IF (conditional predicate)* { *Dependency Relations or CDRs* } *ELSE* { *Dependency Relations or CDRs* }
3. *FOR (i; from; to; step;)* { *Dependency Relations or CDRs* }
4. *THERIS* **list** { *Dependency Relations or CDRs* }

The *conditional predicate* can be a timing or a dependency relation. The interesting point is that a set of conditionally evaluated timing relations may assume a dependency relation as a predicate and, similarly, a set of conditionally evaluated dependency relations may assume a timing relation as a conditional predicate. This is the only (restricted) form allowed where timing and dependency relations may be mixed.

To separate timing and dependency relations in an ATCSL program, we use the keywords **TIMING** and **DEPENDENCY** (if both are present, timing relations should precede dependency relations). The following is an example syntax:

```
TIMING
{
    timing relations
}
DEPENDENCY
{
    dependency relations
}
```

4.1 Formal Specification of the Language

In the formal specification of the language below, keywords are shown in bold font, alternatives are separated by vertical bars, parentheses indicate grouping, optional clauses are indicated by brackets, and optional repetition is indicated by braces.

var_decl:
    index_identifier [= *scalar-constant*] { , *index_identifier* [= *scalar-constant*] }
index_identifier:
    identifier
range:
    scalar-constant | [–]**MIN** | [–]**MAX**
value:
    1 | **0** | **r** | **f** | **s** | **c**
signal:
    identifier **.** *value*
signal_event:
    signal [[*index_ari_exp*]]
event_signal_exp:
    start (*signal_event*)
    width (*signal_event*)
    card (*signal_event*)
val_interval:
    (**val** (*identifier*) (= | ! =) **val** (*identifier*)) [[*index_ari_exp*]]
index_ari_exp:
    card (*signal_event*)
    index_identifier
    scalar-constant
    (*index_ari_exp*)
    index_ari_exp (+ | - | * | /) *index_ari_exp*
tim_ari_exp:
    event_signal_exp
    index_identifier
    scalar-constant
    (*tim_ari_exp*)
    tim_ari_exp (+ | - | * | /) *tim_ari_exp*
tim_rel_exp:
    tim_ari_exp (> | >= | < | <= | = | ! =) *tim_ari_exp*
interval_exp:
    (*signal_event* | *val_interval*) **WHILE** (*signal_event* | *val_interval*)
    (*signal_event* | *val_interval*) **BEFORE** (*signal_event* | *val_interval*)
    (*signal_event* | *val_interval*) **OVERLAP** (*signal_event* | *val_interval*)
    (*signal* | *val_interval*) **SYNC** [*range range*] (*signal* | *val_interval*)
timing_assertion:
    signal_event – > [*range range*] *signal_event*
list:
    identifier [*range* : *range*]
tim_existential_exp:
    THERIS *list* { , *list* } (*timing*)
tim_universal_exp:
    FOR (*identifier* ; *index_ari_exp* ; *index_ari_exp* ; *index_ari_exp*) *timing*
tim_rel:
    tim_rel (&& | || | !) *tim_rel*
    tim_rel_exp
    interval_exp
    timing_assertion
    tim_existential_exp

tim_universal_exp
(*tim_rel*)

timing:
timing { *timing* }
{ *timing* }
tim_rel ;
IF (*tim_rel* | *dep_rel*) *timing* [**ELSE** *timing*]

causal_rel:
signal_event => [[*range range*]] *signal_event*

strong_rel:
INPUT (*signal_event* { , *signal_event* }) => **OUTPUT** (*signal_event* { , *signal_event* })

weak_rel:
PATH (*signal_event* { , *signal_event* }) => **TO** (*signal_event* { , *signal_event* })

dep_existential_exp:
THERIS *list* { , *list* } (*dependency*)

dep_universal_exp:
FOR (*identifier* ; *index_ari_exp* ; *index_ari_exp* ; *index_ari_exp*) *dependency*

dep_rel:
dep_rel (&& | || | !) *dep_rel*
causal_rel
strong_rel
weak_rel
dep_existential_exp
dep_universal_exp
(*dep_rel*)

dependency:
dependency { *dependency* }
{ *dependency* }
dep_rel ;
IF (*tim_rel* | *dep_rel*) *dependency* [**ELSE** *dependency*]

timing constraint:
[*var_decl* ;] **TIMING** *timing* **end**
[*var_decl* ;] **DEPENDENCY** *dependency* **end**
[*var_decl* ;] **TIMING** *timing* **DEPENDENCY** *dependency* **end**

A scalar-constant is an integer number. An identifier is an alphanumeric string. An index identifier can be any alphanumeric string; a signal or event name can be any string contained in the file *foo.nam* created by the simulator. Declared variables are not initialized by the system. The order of evaluation for the expressions is left-to-right. Comments are supported and commented expressions are described between the delimiters `/*` and `*/` (like in *C*). The **MIN** and **MAX** keywords correspond to minus and plus infinity respectively. Internally they are assigned the values **INT_MIN** and **INT_MAX**.

4.2 Using the Verifier

Having specified the timing constraints in ATCSL and derived the event graph using the simulator, we are ready to run the verification program *verify*. By calling *verify foo* the verification program will use files *foo.con, foo.sim and foo.nam* and will produce error or warning reports (if any) in files *foo.err* and *foo.war* respectively. Following the same convention we used for the other programs of CLOVER, the user is able to use names for the files used by or derived from the simulator with a prefix part different from *foo*. The following options are used for that purpose:

-A The format *-Nfoo1* will result in using file *foo1.nam* as the file where the acceptable signal names are recorded. This file will be used while parsing the constraint file.

-C The format *-Cfoo1* will result in using file *foo1.con* as the file where the timing constraints, specified in ATCSL, are recorded.

-S The format *-Sfoo1* will result in using file *foo1.sim* as the file where the event graph is recorded.

-E The format *-Efoo1* will result in using file *foo1.err* as the error report file and file *foo1.war* as the warning report file.

In order to produce a trace file we have to run the verification program with the option *-t*: *verify -t foo*. This will produce the trace file *foo.trc*. Again we can use the *-E* option to change the default name.
As an example we will verify the circuit of Figure 2, given the event graph produced in the previous section, against the following timing requirements specified in the *and_dlatch.con* file:

```
TIMING
{
    /* Width of Output stable should be grater than 200 ns */
    width(Out.s) > 200;
}
DEPENDENCY
{
    /* In[1] should cause the output to become stable */
    In[1].s => Out.s;
}
end
```

We run the verification program by calling *verify and_dlatch* and we get the following error report in file *and_dlatch.err*:

```
Timing Relations
----------------

**** width(Out.s) > 200
Signal Out.s index: 0 from: 80 to: 168
Violate(s) rel_operator: '>' against the number: 200

Dependency Relations
--------------------

**** In[1].s => Out.s
No existing causality path between events: In[1].s[0] and Out.s[0]
There is a strong dependency path between the two events
```

The error in the timing causality relation is due to the fact that in the *D_latch* the latest arriving input signal enabling the output is the control signal (here the clock) rather than the *Data.s* signal. However, the output signal assumes the value of the input signal. In case we want to verify dependency relations without taking into consideration control signals, CLOVER permits the user to explicitly define control signals in the PDL-e description of the design. For example, we can specify a *D_latch* module as follows:

```
#define D_latch(i, en)        create_fun("%s %w %wc %O", "D_latch", i, en, 1)[0]
```

Here we used the *%wc* specifier to specify that the *en* input in a *D_latch* is a control signal. Now the verification of the *and_dlatch* design will show no dependency violations:

```
Timing Relations
----------------

**** width(Out.s) > 200
Signal Out.s index: 0 from: 80 to: 168
```

```
Violate(s) rel_operator: '>' against the number: 200

Dependency Relations
--------------------

No errors
```

5 Initial Setup

The following default locations are used to install CLOVER's programs:

Executable Files *CLOVER/bin/cputype*

Standard-Modules Library *CLOVER/lib/cputype*

Include Files *CLOVER/include*

Source Files *CLOVER/src*

The prefix *CLOVER* is a location chosen by the installer of the system. It is set (by the installer) in the makefile (*clover_make*) which does the installation (see below). The suffix *cputype* in the directories for the executable and library files of CLOVER is provided to allow for different binaries of CLOVER corresponding to different *cpu* types. The environment variable **CLOVER** is a user-defined variable which should assume the same value as the corresponding value of the prefix *CLOVER*. This variable is to be used by the *pdl2net* program (see the corresponding manual page for more details).

The following table indicates the files used by the system and their corresponding location.

Executable Files (CLOVER/bin/cputype)	*pdl2net, user_lib, clsim verify, clover, grammar parse, look, view_clover*
Standard-Modules Library (CLOVER/lib/cputype)	*chip_lib.o, simulator.o*
Include Files (CLOVER/include)	*declarations.h, sim.h, net.h*
Source Files (CLOVER/src)	*comparison.c, simulator.c, constraint.y interface.c, pif2graph.c, parse.c search.c, user_lib.c, clover.c, pdl2net.c my_alloc.c, my_error.c, my_string.c i_node.c, pdlpif.c, declarations.c chip_lib.c, fun_chip_lib.c, usleep.c*

To create the executable files the following makefiles are provided. The table below indicates the correspondence between makefiles and created executables.

clover_make	Create files *clover, verify, pdl2net, clsim, grammar, view_clover, search, user_lib simulator.o, chip_lib.o*
cmp_make	Creates file *verify*
pdl_make	Creates file *pdl2net*
sim_make	Creates file *clsim, simulator.o, chip_lib.o*
cns_make	Creates file *grammar*
int_make	Creates file *view_clover*
parse_make	Creates file *parse*
user_lib_make	Creates file *user_lib*

To initially install the system, the user who does the installation has first to create directory *CLOVER/src* and then has to put all the makefile, source and include files of CLOVER under it. Next, the following three steps need to be taken in order:

1. *make -f clover_make setup*
   During this step the *bin*, *lib* and *include* directories of CLOVER are created. Furthermore, all include files are copied in the corresponding include directory of CLOVER.

2. *make -f clover_make*
   This command builds all the binary files of CLOVER and puts them under the installer's current directory (which should be the directory *CLOVER/src*).

3. *make -f clover_make install*
   During this step the binary files created in step 2 are placed under their corresponding directories. To account for different *cpu* types, *clover_make* installs the executable and library files of CLOVER under the right subdirectory of *bin* and *lib* by using the command *cputype* found in */usr/local/bin*. In case this command is missing the installer has to do the corresponding actions manually or he/she can use another command with similar functionality (the variable *BIN* in *clover_make* would have to be changed accordingly).

The variables of the Makefile (*clover_make*) *CC, CLOVER* permit the installer to choose *C* compiler (it should be *ANSI*), and prefix part (*CLOVER*) for the directories of CLOVER.

The makefile *cns_make* analyzes syntactically (calling *yacc* [9]) the constraint expressions written in ATCSL. It creates the executable file *grammar* which is called internally by *verify* and creates a parse tree of the ATCSL expressions. The parse tree is recorded under the */usr/tmp* directory and it is removed, after being read, by *verify*. The user should not be aware of the specifics of *grammar* (therefore *grammar* is undocumented).

The makefile *parse_make* creates the executable file *parse*. *Parse* is called internally by *user_lib*; it parses the user-defined library to identify the names of the user-defined functions (*parse*, like *grammar*, is undocumented).

To run *clover* the core simulator and the standard-defined module libraries have to be linked with the user-defined modules. The linkage takes place only if a user-defined library name is provided through the $-L$ option of *clover* (see Appendix B). The user should have defined the environment variable **CLOVERSYSLIB** in order for the linkage to be successful (otherwise he/she should have a copy of *simulator.o* and *chip_lib.o* in his/her working directory). This variable should assume as a value the path name of the library directory where the files *simulator.o* and *chip_lib.o* are located (*CLOVER/lib/cputype*). The system assumes by default that the user's include and library files necessary to compile user-defined modules are found in the user's current working directory. However, for a user-customed environment, the environment variables **CLOVERLIB** and **CLOVERINC** are provided allowing the user to designate alternate places for the locations of his/her include and library files. After linkage, an executable version of the simulator will be created at the user's current working directory (by default). The option $-B$ is provided with *clover* and *user_lib* to permit the user to indicate an alternate location for the derived executable (*clsim*) of the simulator. In order to refer to the derived *clsim* instead to the *clsim* found in *CLOVER/bin/cputype*, the directory where the derived *clsim* will be placed should precede in the user's path declaration the path defined by *CLOVER/bin/cputype*.

5.1 Current Status

CLOVER has been used to specify and verify two rather large hardware designs: the first design is the SPUR PCC-SBC interface [8] which interconnects two synchronous subsystems that have asynchronous clocks. The other is the Multibus Design Frame [2], a synchronous interface attached to the asynchronous Multibus. We are still working on refining and debugging the system as well as augmenting the standard-module defined library. Forward any question, remark or bug reports to *aslp@princeton.edu*.

Acknowledgements This research was partially supported by DARPA/ONR contract N00014-88-K-0459. The authors would like to thank Wayne Wolf for the insightful discussions we had and his useful comments. We would like to thank also Richard Lipton for giving us the motivation to search for a good specification method for timing constraints.

A Standard-Defined Modules

The following table contains the list with the standard modules currently defined by CLOVER. The first column characterizes the module and the second column describes the name by which the module is identified by CLOVER. This name should be used in a *create_fun* definition for a PDL-e description.

Standard Module	Name
AND gate with two inputs	*_and*
AND gate with two inputs (fast technology)	*_fand*
AND gate with eight inputs (fast technology)	*_fand8*
OR gate with two inputs	*_or*
OR gate with two inputs (fast technology)	*_for*
NOR gate with two inputs	*_nor*
NOR gate with two inputs (fast technology)	*_fnor*
NOR gate with three inputs	*_nor3*
NAND gate with two inputs	*_nand*
NAND gate with two inputs (fast technology)	*_fand*
Exclusive Or	*_xor*
Exclusive Or (fast technology)	*_fxor*
Inverter	*_not*
Inverter (fast technology)	*_fnot*
Inverter with open collector	*_not_oc*
Inverting Buffer	*_not_buffer*
Buffer	*_buffer*
D-latch	*D_latch*
D-latch (fast technology)	*D_flatch*
D Flip-Flop	*D_FF*
D Flip-Flop (fast technology)	*D_fFF*
Synchronizer	*_sync*
Synchronizer with Clear Signal	*_syncc*
RS Latch	*R_S*
RS Flip-Flop	*RS_FF*

In order to facilitate the search for the characteristics of a particular standard-defined module, we have implemented a command which can be used with an on-line library containing information about each defined-module's properties. Such properties may include load and temperature information, functional tables, and delay characteristics in minimum-maximum form. Currently only a skeletal library showing the format of information is provided. To do a search, the user has to call the program *look*. By calling *look search_string* the program will search the information library with name *library_information* in the current working directory for properties regarding a module identified by *search_string*. *Search_string* can either be the name of a module (as this name has been defined in the above table) or can be the serial number characterizing such a module under a certain technology (i.e., SN74F30 for an AND gate implemented with fast technology). In case we want to use a different name for the information library we can call *look* with the option *-f*: *look -flibrary_name* will read the library *library_name* instead of the *library_information* library (a full path name can also be used). The *look* program will search for an exact match of the searching string. When a partial matching is requested then a set of possible modules matching the requested test will be displayed. This can be accomplished by running the *look* program with the option *-p* (for partial matching). The information is represented in a form which can be included as a comment in a *C* file (for example it can be included as a comment in a behavioral specification of a user-defined module).

B Steps for a Complete Verification Session

We assume again that the user wants to verify a design named *foo*. He/she first has to create the files *foo.pdl* (structural description of *foo*), *foo.com* (commands to run the simulator) and *foo.con* (specification of timing constraints). If the user-defined library assumes the name *library_name* then all the user has to do to verify design *foo* is to type:

clover -Llibrary_name foo

The command should be executed in the user's directory where the files *foo.pdl, foo.con* and *foo.com* reside. All the other files created by *clover* will be stored in the same directory too. In the user's path, this directory must be searched before *CLOVER/bin/cputype* to find the correct version of *clsim*. If the user does not have or does not want to use his/her user-defined library then he/she can simply type:

clover foo

While the program is executed messages printed on the screen indicate the current phase of the verification procedure:

```
> clover foo
Create foo.net file
Parse library_name
Compile user-defined library: library_name
Link standard, user-defined libraries and simulator object file
Create foo.sim file
Parsing Constraint file foo.con
Verify - create foo.err - foo.war file{s}
```

Initially the structural description of *foo* is translated into a netlist description (file *foo.net*) to be used by the simulator. Then the user-defined library is parsed to identify the names of the user-defined functions and link these names to the corresponding names of the user defined-modules defined in the *pdl* structural description. A *C* structure is used to record this correspondence. The first field of the structure is a character string where the name of the user-defined module (in the *pdl* description) is stored. The second field is a pointer to the function which specifies the behavior of the same module. For a user-defined library with name *library_name* this structure is stored in a file named *fun_library_name* This file will be compiled and linked with the standard and user-defined libraries and the simulator object code file. After linkage, an executable version of the simulator will be created at the user's current working directory, called *clsim*. Next, the simulator runs and produces the event graph of *foo* in file *foo.sim*. Finally the constraint file *foo.con* is parsed and together with file *foo.sim* is used by the verifier to report error or warning messages.

By executing *clover foo* all the intermediate files created or used by the system assume the same prefix: *foo*. The following options are provided to permit the user to indicate a different prefix part for a particular file used during some phase of the verification:

-A The format *-Afoo1* will result in using file *foo1.nam* as the file where the acceptable signal names are recorded. This file will be used while parsing the constraint file.

-C The format *-Cfoo1* will result in using file *foo1.con* as the file where the timing constraints, specified in ATCSL, are recorded.

-E The format *-Efoo1* will result in using file *foo1.err* as the error report file and file *foo1.war* as the warning report file.

-N The format *-Nfoo1* will result in using file *foo1.net* as the input netlist file for the simulator.

-M The format *-Mfoo1* will result in using file *foo1.com* as the command file for the simulator.

-S The format *-Sfoo1* will derive the event graph for *foo* in file *foo1.sim*

-R The format *-Rfoo1* will result in using file *foo1.rep* as the report file of the simulator.

When running *clover* we assume by default that the simulator uses the transport timing model and that the delay model is the min-max model. As was explained in Section 3.4 we can use options *-i, -d* to alter these defaults. Executing *clover -i foo* will switch the timing model of the simulator to the inertial one. Executing *clover -ddelay_model foo* (where *delay_model* is an integer from one to four) a delay model will be assumed according to the Table 1. However, only the minimum-maximum delay model is implemented. Finally in order to produce a trace file we have to run *clover* with the option *-t*: *clover -t foo*. This will produce the trace file *foo.trc*. Again we can use the *-E* option to change the default name.

Clover in reality is a script which executes step by step the programs corresponding to each phase of the verification. The user is able to execute these programs independently. The following is a list of independent executions (with the acceptable options for each program) which result in a complete verification session:

```
pdl2net [-c] foo
user_lib [-Bc] library_name
clsim [-NCSRdi] foo
verify [-ACSEt] foo
```

CLOVER: A User Manual

NAME

pdl2net—Transforms a *pdl* description to a netlist (*net* description)

SYNTAX

pdl2net [options] **design_name**

DESCRIPTION

Pdl2net derives the netlist of the design *design_name*. The program takes as input a structural description of *design_name* written in the PDL-e hardware description language and stored in file *design_name.pdl*. By calling *pdl2net design_name* the program will look for a file named *design_name.pdl* and it will produce the netlist of the design in a file called *design_name.net*. Before creating the netlist of the design, *pdl2net* will have first to compile file *design_name.pdl*. All *pdl* files should include the file *pdlpif.c*. *Pdlpif.c* is located in directory *CLOVER/src*, where the value of *CLOVER* is determined by the installer; for the compiler to find this include file the user can either obtain a copy of the file on his/her current or include directory (designated by **CLOVERINC**), or he/she can declare in his/her environment the environment variable **CLOVER** assuming the value *CLOVER*.

OPTIONS

The following option is recognized by the *pdl2net* command.

-ccompiler_name The default C-compiler used by *pdl2net* for the compilation of a *pdl* file is the ANSI-C compiler *lcc* [7]. With the *-c* option the user is able to use the compiler of his/her choice (for example *compile_name*).

FILES

Design_name.{pif, net}

SEE ALSO

user_lib, clsim, verify, clover

NAME

user_lib—Compiles the user-defined library and links it to the standard modules library

SYNTAX

user_lib [options] **library_name**

DESCRIPTION

User_lib will parse, calling *parse* (undocumented), the user-defined library, *library_name*, identifying the names of user-defined modules and subsequently will compile and link them with the standard library. The user should have defined the environment variable **CLOVERSYSLIB** in order for the linkage to be successful (otherwise he/she should have a copy of *simulator.o* and *chip_lib.o* in his/her working directory). This variable should assume as a value the path name of the library directory where the files *simulator.o* and *chip_lib.o* are located (*CLOVER/lib/cputype*). After linkage, an executable version of the simulator will be created at the user's current working directory (by default). The option $-B$ (see below) is provided with *user_lib* to permit the user to indicate an alternate location for the derived executable (*clsim*) of the simulator. In order to refer to the derived *clsim* instead to the *clsim* found in *CLOVER/bin/cputype*, the directory where the derived *clsim* will be placed should precede in the user's path declaration the path defined by *CLOVER/bin/cputype*.

The parse program will create a *C* structure relating user-defined functions and their names. This structure is recorded in a file named *fun_library_name* where *library_name* is the name of the user defined library. By default the user-defined library is located under the user's current working directory. A full path name instead of a simple file name (for the user-defined library) can be used to designate an alternate location. The user's current working directory is also assumed to be the default place for the location of the user's include and library files necessary to compile user-defined modules. For a user-customed environment, the environment variables **CLOVERINC** and **CLOVERLIB** are provided allowing the user to designate alternate places for the locations of his/her include and library files.

OPTIONS

The following options is recognized by the *user_lib* command.

-ccompiler_name The default C-compiler used by *user_lib* for the compilation of the user's-defined library is the ANSI-C compiler *lcc*. With the *-c* option the user is able to use the compiler of his/her choice (for example *compile_name*).

-Bpath_name After linkage, an executable version of the simulator will be created at the user's current working directory (by default). The option $-B$ permits the user to indicate an alternate location for the derived executable (*clsim*) of the simulator, indicated by the string *path_name*. The string should be a full pathname name.

FILES

Fun_*library_name*.c

SEE ALSO

pdl2net, clsim, verify, clover

NAME

clsim—Derives the event graph of a design after event-driven simulation

SYNTAX

clsim [options] **design_name**

DESCRIPTION

Clsim performs event-driven simulation on the design *design_name* and derives the event graph of it, in the file *design_name.sim*. *Clsim* uses by default the minimum-maximum delay model and the transport timing model. Before starting the simulation the design first has to be structurally (*design_name.net* file) and behaviorally (standard and user-defined libraries) specified. The user guides the operation (non interactively) of the simulator by writing a command file *design_name.com*. In a report file, *design_name.rep*, the simulator reports possible violations detected during simulation, for example, set-up or hold time constraint violations, oscillation, undefined signals, incorrect input values etc. Finally the *clsim* program, as it reads the netlist file *design_name.net*, creates a file *design_name.nam* which contains all the names of the signals defined in the *design_name.pdl* file. This information will be used by the timing constraints specification language parser to identify the acceptable identifier names for signals and events.

OPTIONS

The following option are recognized by the *clsim* command.

-Nfoo The option will result in using file *foo.net* as the netlist file for the simulator.

-Cfoo The option will result in using file *foo.com* as the command file for the simulator.

-Sfoo The option will derive the event graph in file *foo.sim*.

-Rfoo The option will result in using file *foo.rep* as the report file of the simulator.

-i The simulator supports both the inertial and the transport timing models. Transport model is the default timing model. Running *clsim* with the option *-i* switches the timing model of the simulator to the inertial model.

-ddelay_model *delay_model* is an integer which determines the desired delay model. The next table associates *d* with different delay models. The default model and the only currently implemented is the min-max model.

d	MEANING
1	unit delay model
2	minimum - maximum delay model
3	probabilistic delay model
4	slope delay model

FILES

Design_name.{sim, rep, nam}

SEE ALSO

pdl2net, usel_lib, verify, clover

NAME

verify—Verifies a set of ATCSL specification against an event-graph implementation of a design

SYNTAX **verify** [options] **design_name**

DESCRIPTION

Verify verifies an ATCSL specification of *design_name* (file *design_name.con*) against an event-graph (stored in file *design_name.sim*) of an implementation of the design *design_name*. The verification program will use files *design_name.con, design_name.sim and design_name.nam* and will produce error or warning reports (if any) in files *design_name.err* and *design_name.war* respectively. To syntactically parse the ATCSL expressions and create a parse tree, *verify* calls internally program *grammar* (undocumented).

OPTIONS

The following option are recognized by the *verify* command.

-Afoo The option will result in using file *foo.nam* as the file where the acceptable signal names are recorded. This file will be used while parsing the constraint file (*con* file).

-Cfoo The option will result in using file *foo.con* as the file where the timing constraints, specified in ATCSL, are recorded.

-Sfoo The option will result in using file *foo.sim* as the file where the event graph is recorded.

-Efoo The option will result in using file *foo.err* as the error report file and file *foo.war* as the warning report file.

-t This option will derive a trace file for *design_name*: *design_name.trc*. Again we can use the -*E* option to change the default name.

FILES

Design_name.{err, war, trc}

SEE ALSO

pdl2net, usel_lib, clsim, clover

NAME

clover—Produces an event graph of a design using event-driven timing simulation techniques and verifies the behavior of the design (specified in ATCSL) against the derived event graph.

SYNTAX

clover [options] **design_name**

DESCRIPTION

Clover is a script which executes step by step individual programs corresponding to different phases of the verification process. Initially the structural (*pdl* file) description of the design *design_name* will be transformed to a netlist description (*net* file) which will be fed to the event-driven simulator. All *pdl* files should include the file *pdlpif.c*. *Pdlpif.c* is located in directory *CLOVER/src*, where the value of *CLOVER* is determined by the installer; for the compiler to find this include file the user can either obtain a copy of the file on his/her current or include directory (designated by **CLOVERINC**), or he/she can declare in his/her environment the environment variable **CLOVER** assuming the value *CLOVER*.

Before starting the simulation and if the $-L$ option has been used, *clover* will parse, calling *parse* (undocumented), the user-defined library identifying the names of user-defined modules and subsequently will compile and link them with the standard library. The user should have defined the environment variable **CLOVER-SYSLIB** in order for the linkage to be successful (otherwise he/she should have a copy of *simulator.o* and *chip_lib.o* in his/her working directory). This variable should assume as a value the path name of the library directory where the files *simulator.o* and *chip_lib.o* are located (*CLOVER/lib/cputype*). After linkage, an executable version of the simulator will be created at the user's current working directory. The option $-B$ (see below) is provided with *clover* to permit the user to indicate an alternate location for the derived executable (*clsim*) of the simulator. In order to refer to the derived *clsim* instead to the *clsim* found in *CLOVER/bin/cputype*, the directory where the derived *clsim* will be placed should precede in the user's path declaration the path defined by *CLOVER/bin/cputype*.

The parse program will create a *C* structure relating user-defined functions and their names. This structure is recorded in a file named *fun_library_name.c* where *library_name* is the name of the user-defined library. By default the user-defined library is located under the user's current working directory. A full path name instead of a simple file name (for the user-defined library) can be used to designate an alternate location. The user's current working directory is also assumed to be the default place for the location of the user's include and library files necessary to compile user-defined modules. For a user-customed environment, the environment variables **CLOVERINC** and **CLOVERLIB** are provided allowing the user to designate alternate places for the locations of his/her include and library files.

After simulation the behavior of the design is recorded in the form of an event graph in file *design_name.sim*. Finally the verification program will verify a set of specifications for *design_name* expressed in ATCSL (file *design_name.con*), against the derived event graph and will report possible violations

OPTIONS

The following options are recognized by the *clover* command.

-Llibrary_name Specifies the name of the user-defined library to be: *library_name*.

-Bpath_name If the $-L$ option is used, after linkage, an executable version of the simulator will be created at the user's current working directory (by default). The option $-B$ permits the user to indicate an alternate location for the derived executable (*clsim*) of the simulator, indicated by the string *path_name*. The string should be a full pathname name. Option $-B$ has no effect if option $-L$ is not also used.

-ccompiler_name The default C-compiler used by *clover* for the compilation of the user's-defined library and the user's *pdl* files is the ANSI-C compiler *lcc*. With the *-c* option the user is able to use the compiler of his/her choice (for example *compile_name*). This should be another *ANSI C* compiler.

-Nfoo The option will result in using file *foo.net* as the netlist file.

-C<u>foo</u> The option will result in using file *foo.con* as the file where the timing constraints, specified in ATCSL, are recorded.

-M<u>foo</u> The option will result in using file *foo.com* as the command file for the event-driven simulator.

-S<u>foo</u> The option will derive the event graph in file *foo.sim.*

-R<u>foo</u> The option will result in using file *foo.rep* as the report file of the simulation.

-A<u>foo</u> The option will result in using file *foo.nam* as the file where the acceptable signal names are recorded. This file will be used while parsing the constraint file (*con* file).

-E<u>foo</u> The option will result in using file *foo.err* as the error report file and file *foo.war* as the warning report file of the verification procedure.

-t This option will derive a trace file for *design_name*: *design_name.trc*. Again we can use the *-E* option to change the default name.

-i The simulator of *clover* supports both the inertial and the transport timing models. Transport model is the default timing model. Running *clsim* with the option *-i* switches the timing model of the simulator to the inertial model.

-d<u>delay_model</u> *delay_model* is an integer which determines the desired delay model. The next table associates *d* with different delay models. The default model of *clover* and the only currently implemented is the min-max model.

d	MEANING
1	unit delay model
2	minimum - maximum delay model
3	probabilistic delay model
4	slope delay model

FILES

Design_name.{pif, net, sim, nam, rep, err, war, trc}, Fun_*library_name*.c

SEE ALSO

pdl2net, usel_lib, clsim, verify

NAME

look—Searches a library containing information about properties of standard and user-defined functional modules

SYNTAX

look [options] **search_string**

DESCRIPTION

CLOVER provides an on-line library containing information about the properties of each defined functional module (standard or user-defined). Such properties include load and temperature information, functional tables, and delay characteristics in minimum-maximum form. *Look* facilitates the search through this library. By calling *look search_string* the program will search (in the current working directory) the information library with the default name *library_information*, for properties regarding a module identified by *search_string*. *Search_string* can either be the name of a module or can be the serial number characterizing such a module under a certain technology (i.e., SN74F30 for an AND gate implemented with TTL fast technology). The information in the library is represented in a form which can be included as a comment in a *C* file (for example it can be included as a comment in a behavioral specification of a user-defined module).

OPTIONS

The following option is recognized by the *look* command.

-fnew_name This option permits the use of a new name (here *new_name*) other than the default as the name of the information library. A full path for the name of the library can also be used.

-p The *look* program searches for an exact match of the searching string. When a partial matching is requested then a set of possible modules matching the requested test will be displayed. This can be accomplished by running the *look* program with the option *-p* (for partial matching).

NAME

get_wire, get_wire_array, name_wire, name_wire_array, connect_wire, connect_wire_array, change_name, change_-name_array, create_fun—These are the built-in functions of PDL-e. They manipulate wire and wire_arrays. With them we can create a wire, name a wire, change a wire's name and connect wires together. The function *create_fun* is used to describe arbitrary functional modules with certain input and output ports

SYNTAX

```
#include "pdlpif.c"

wire *get_wire()

wire_array *get_wire_array(n)
int n;

wire *name_wire(name)
char *name;

wire_array *name_wire(name, n)
char *name;
int n;

void connect(w1, w2)
wire *w1, *w2;

void connect_array(w1, f1, t1, w2, f2, t2)
wire_array *w1, *w2;
int f1, t1, f2, t2;

wire *change_name(w, name)
wire *w;
char *name;

wire_array *change_name_array(w, name, f, t)
wire_array *w;
char *name;
int f, t;

wire_array *create_fun(format, [, arg] ...)
char *format;
```

DESCRIPTION

The function *get_wire* returns a pointer to a wire, creating a unique wire. With this function a declared wire variable is instantiated as a real "physical" wire.

The function *get_wire_array* returns a pointer to an array of *n* wires, creating *n* unique wires.

The function *name_wire* returns a pointer to a wire with name *name*. That way the user can assign names to input and output ports of modules.

The function *name_wire_array* returns a pointer to an array of *n* wires, with name *name*.

The function *connect* connects wires *w1* and *w2*. This is the equivalent of connecting, through a wire, input and output ports of two modules.

The function *connect_array* connect wires *w1[f1]* through *w1[t1]*, to wires *w1[f2]* through *w2[t2]*, respectively. Connecting two wire arrays with $t1 - f1 \neq t2 - f2$, results in a violation and the program is terminated.

The function *change_name* changes the name of wire *w* to *name* and returns a pointer to the renamed wire. The old name is still valid. We usually use this function to assign names to the output wires returned by the function *create_fun* (see description next).

The function *change_name_array* changes the name of wires *w[f]* through *w[t]* to *name*, and returns a pointer to wire_array *w*. The old name remains valid.

The function *create_fun* is used for the structural description of user-defined or standard-defined modules. *Create_fun* takes a variable number of arguments and returns a pointer to a wire_array which represents the output wires of the defined module. The first argument *format*, is a string constituted of conversion specifications each of which is used to interpret the rest of the arguments of *create_fun* (as does *printf()* in *C*). Each conversion specification begins with a % and ends with a conversion character. Six conversion characters are supported; with them the user can specify the name of the described module, the input and output ports of it (using wire or wire_array types) and the delays associated with the wires connected to the input ports of the module (only constant wire delay values are supported in the current version of CLOVER). The following table explains in more details the use of each conversion character.

Conversion Char	Input Data; Argument Type
s	Input is a string characterizing the name of the described module
w	Input is a pointer to a wire representing input wire. If the conversion character *w* is followed by the character *c* (*wc*) that will indicate that the corresponding input wire is a control signal. This information can be used when we determine timing causality dependencies.
W	Three arguments should correspond to this conversion character. The first two denote the wire_array limits (*from, to*). The last is a pointer to a wire_array representing input wires. *W* can be followed by the character *c* to indicate a wire_array of control signals (*Wc*).
d	If present, should immediately follow a %w conversion specification. Input is an integer which determines the delay, of a previously specified wire.
D	If present should immediately follow a %W conversion specification. Input is a set of integers which determine the delay of a previously specified wire_array. The first number of the set indicates the cardinality for the rest of the integer set.
O	Input is an integer which determines the number of output ports of the defined module.

NAME

get_ev_info, get_value, retain_value, cau_depend — These functions are used in the description of user-defined modules and return information about the initial time and the width of events as well as information about the kind of dependency relation between events, while these events are generated during simulation. They can be used for the behavioral description of user-defined functional modules

SYNTAX

int get_ev_info(inout, p, c, index, in, w, elem)
int p, index;
char inout, c;
int *in, *w, *elem;

get_value(inout, p)
int p;
char inout;

retain_value()

cau_depend(port1, c1, index1, port2, c2, index2)
int index1, index2;
char *port1, *port2, c1, c2;

DESCRIPTION

The function *get_ev_info* returns timing information about specific events appearing on the input-output ports of the circuit modules. The *inout* parameter determines if the event appears on an input or output port *p* of the module (we also represent that as a *[inout, p]* port). It takes values from the set { *'i', 'o'* } where *'i'* denotes an input port and *'o'* denotes an output port. The port numbers follow the index numbers of their corresponding input or output signals. With the parameter *c* the value of the event is specified. It takes values from the set { *'0', '1', 'r', 'f', 's', 'c'* }. Parameter *index* specifies the index of the event. When it assumes the value -1 then we refer to the latest event created up to this point of the simulation. The initial time, width and index of the event are returned with the parameters *in*, *w* and *elem* respectively. The function returns -1 if the event we are interested in does not exist; an error message is reported in the simulator's report file (*rep* file).

The function *get_value* returns the value of the latest event which appears on port *[inout, p]*. This function can be particularly useful in cases where the behavior of a module depends not only on its current inputs but also on its previous outputs (e.g., an RS flip-flop). Then we can use *get_value()* to find out the value of the old outputs and use it to calculate the new ones. The function returns -1 when an event cannot be found on port *[inout, p]*.

The function *retain_value* is used to indicate no change to an output value. It is useful to describe behavior of memory elements like latches and flip-flops, for example, where the output signal retains its value when then enabling signal is "off".

The function *cau_depend* identifies if there is a timing causality dependency between two events. It returns one(1) if the event with value *c1* and index *index1* on port *port1* timing causes the event with value *c2* and index *index2* on port *port2*. Otherwise the function returns zero(0). If either one of the requested events cannot be identified the function returns -1 and an error message is reported in the simulator's report file (*rep* file). Only signals which appear at *named* ports can be checked for timing causality.

References

[1] Larry Augustin. Timing Models in VAL/VHDL. In *IEEE International Conference on Computer-Aided Design*, 1989.

[2] G. Borriello and R. Katz. Design Frames: A New System Integration Methodology. In *Chapel Hill Conference on VLSI*, May 1985.

[3] R. K. Brayton et al. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on CAD*, November 1987.

[4] M. Browne, E. Clarke, and D. Dill. Automatic Circuit Verification Using Temporal Logic: Two New Examples. In G. J. Milne and P. A. Subramanyan, editors, *Formal Aspects of VLSI Design*. Elsevier Science Publishers, 1986.

[5] D. Doukas. *A New Specification Model for Timing Constraints and Efficient Methods for their Verification.* PhD thesis, Princeton University, CS department, January 1991.

[6] D. Doukas and A. LaPaugh. CLOVER: A Timing Constraints Verification System. Technical Report CS-TR-274-90, Princeton University, CS department, 1990.

[7] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):to appear, October 1991.

[8] M. Hill. Design Decisions in SPUR: A VLSI Mutliprocessor. *IEEE Computer*, November 1986.

[9] Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler.

[10] R. Lipton, D. Serpanos, and W. Wolf. PDL++: An Optimizing Generator Language for Register Transfer Design. In *International Symposium on Circuits and Systems*, New Orleans, LI, May 1990.

[11] P. McGeer and R. K. Brayton. Efficient Algorithms to Find the Longest Viable Path in a Combinational Circuit. In *26th Design Automation Conference*, 1989.