User's Guide to ATV, an Abstract Timing Verifier

David E. Wallace

ABSTRACT

ATV, the Abstract Timing Verifier, is a program to perform static timing analysis of dependency graphs derived from logic designs, analyzing worst-case paths using an abstract representation of time and delays that enables a user to choose different representations of time and delays for the analysis.

This technical report describes how to use ATV to analyze designs, including clock phase length analysis of designs that include transparent latches. Background information on the principles behind ATV, including descriptions of several models not yet implemented in ATV, is presented in the main body of my dissertation, available as the companion technical report, *Abstract Timing Verification for Synchronous Digital Systems*.

User's Guide to ATV, an Abstract Timing Verifier

David Edward Wallace

Table of Contents

Table of Figures

List of Tables

Appendix A: User's Guide to ATV

1. INTRODUCTION TO ATV

ATV, the Abstract Timing Verifier, is a program that examines a dependency graph representing a digital logic design, and analyzes worst case paths from input events to outputs. Unlike other timing verifiers, such as Crystal, ATV allows the user to select the representation for time and delays used in the analysis. Thus an analysis could be run using single numbers, and then rerun using ranges (min-max) or statistical descriptions (mean, standard deviations) to represent signal arrival times. It is possible for the sophisticated user to extend the program by developing new models to meet special needs.

Also unlike Crystal, ATV does not run directly on a low-level description of the circuit (such as an extracted transistor list), but requires the input to the program to already be in the form of a dependency graph, with the individual delays already specified. At present, there are two possible sources for such a graph: the user can generate the graph by hand, or it can be generated automatically from an Oct description of a circuit that includes timing descriptions for its low-level blocks. Section 5 discusses how to generate a timing graph from an Oct description.

1.1. Concepts

Timing verification is primarily concerned with analyzing the stable/changing behavior of most signals, rather than looking at specific data values. The abstract timing verifier operates on a dependency graph derived from the logic design to be verified, as shown in Figure A-1. This is a directed graph, with nodes and arcs. Nodes are like SPICE nodes, with no internal delays. *Events* occur at the nodes of the graph. An event consists of an *edge*, which describes the type of transition occurring and an *event time*, a model-dependent description of when the event occurs. ATV currently supports four kinds of edges: S-CH (stable->changing), CH-S (changing->stable), R (rising), and F (falling). The first two define the beginning and end of periods when a signal may be changing its value; the last two define specific rising and falling behavior, and are generally used for clocks.

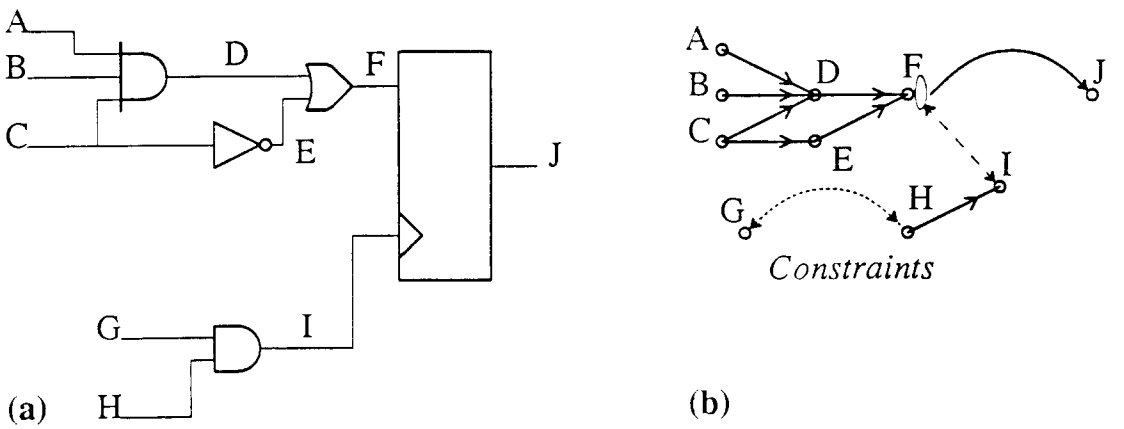

Figure A-1: (a): Logic diagram. (b): Corresponding dependency graph.

Arcs run from each node to the nodes it directly affects. The format used to represent delays is defined by the particular timing model used. Arcs come in two flavors: *ordinary arcs*, corresponding to pure combinational elements, which always transmit data, and *gated arcs*, corresponding to sequential elements, which are controlled by some clock node, and only transmit data when the clock node is in the appropriate state (see Figure A-2). Both types of arcs have *delays* attached to them, which specify how an event time at the tail of the arc is transformed into the consequent event time at the head of the arc. Although specific logic functions are not represented in the graph (since the timing verifier is generally only concerned with the stable/changing behavior of signals), arcs do have an *arc type* attached, which specifies the inverting properties of the arc. This arc type can have one of three values: INVERTING, NON-INVERTING, or UNKNOWN. An arc is inverting if any output transition must be opposite in direction to the input transition that caused it, non-inverting if it must be in the same direction, and unknown if it could be either. Thus for boolean gates, AND and OR would use non-inverting arcs, NOT, NAND and NOR would use inverting arcs, and XOR would use unknown arcs.

Gated arcs act as if they had a small gate at the tail of the arc, controlled by the specified clock node. They only transmit data when the gate is open. Such arcs have both opening and closing edges specified from the set {R, F}. The gate is open from the time the specified opening event occurs at the clock node until the specified closing event occurs (which may be the same event). Thus a level-sensitive

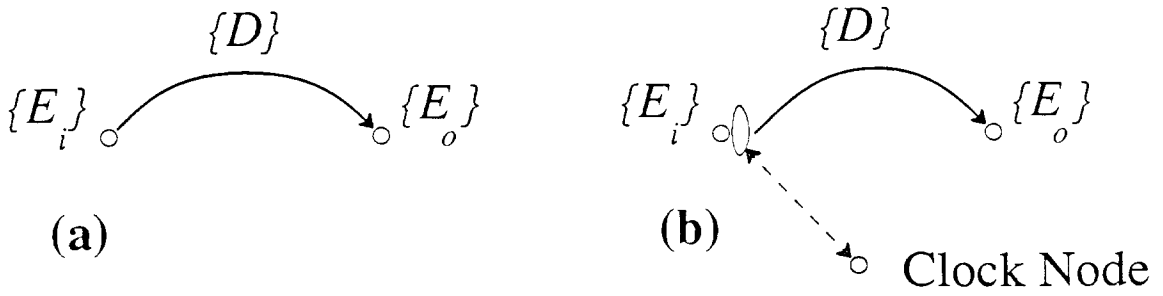

Figure A-2: (a): Ordinary arc. (b): Gated arc.

transparent latch open when the clock node is high would be modeled with a gated arc with OPEN=R, CLOSE=F, while a rising-edge triggered register would be modeled with a gated arc with OPEN=R, CLOSE=R. Clocks are distributed over ordinary arcs from user-specified origin nodes. For now, clocks must be distributed over pure fan-out trees from the origin nodes. If necessary, clock enable signals should be handled specially. In Figure A-1 above, signal G was a clock enable signal modeled by eliminating any direct arcs from it to the clock line, and instead establishing constraints between it and clock node H that will force G to be stable whenever H is high. A typical timing graph is shown in Figure A-3.

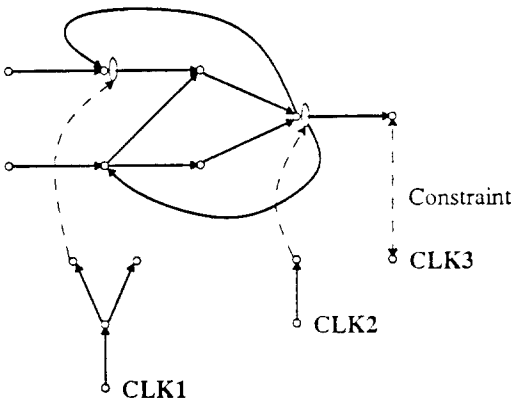

Figure A-3: Typical timing graph.

To support analyses where the clock schedule is not known in advance, ATV supports the idea of multiple *reference frames*. Events in one reference frame have a known relationship to one another, but different reference frames have different origins, which may have an unknown relationship to each other.

One of the objectives of clock-phase length analysis is to determine what relationships must hold between the different reference frames.

The sequence of clock edges is initially specified in the form of a *clock graph*, which has a node for each clock edge in a user-defined *cycle*, and directed arcs from each edge to its direct successors. In the course of the analysis, this cycle is repeated a user-specified number of times to form the *unrolled graph*, which has a distinct node for every clock edge in every cycle.

1.2. Types of Analysis

There are several types of analysis possible with ATV. ATV acts like a toolbox: it provides functions that can be used to do several types of analysis, but does not impose a particular way to use them.

One type of analysis is looking at purely combinational logic, specifying input events at the inputs to the graph, and looking at the resulting events at the outputs. Related to this is taking a graph which includes clocking, and only analyzing it as far as the next sequential element. This is most appropriate when there are no transparent latches (or similar elements) in the design.

Another type of analysis is clock phase length analysis. Here the program is given a design and asked "How fast can it go?" Here different clock edges are initially placed in different reference frames, and the program computes constraints on the translations between frames.

Yet another type of analysis is where the clock schedule is predetermined, and the program is asked to verify that the design will meet the given schedule. This is similar to a clock phase length analysis, but all clock edges are specified in a single reference frame.

1.3. Timing Models

ATV supports the ability to perform analyses using different timing models. A timing model specifies how event times and delays are represented, and defines a few fundamental operations on them. The timing models supplied with the current version of ATV are: (1): Single number (both delays and event times are single numbers), (2): Min-max (both delays and event times are (min, max) pairs), (3):

Simple mean, standard-deviation (delays and event times are means and standard deviations), and (4): Asymmetric rise-fall. In the asymmetric rise-fall model, event times are specified as a rising time and a falling time, where these are chosen from any other timing model, while delays are specified as a rising delay and a falling delay plus a delay type (which is like the arc-type defined above).

The general rule for timing models is that the type of event times entered by the user determine the model to use. When the program attempts to delay an event time along an arc, it will attempt to coerce whatever delay has been specified for that arc into the form called for by the event time. There may be more than one delay format specified for a given arc; they are collected in a special format called a delay-union, which can be coerced into any of its included types. In addition to the above delay formats, some others are defined with coercions to one or more of the standard types: min-typ-max delay takes three numbers and coerces automatically to a min-max delay, and there is an optional module that can be loaded to define a nominal-worst case delay which can be coerced to other forms. The general idea is to allow the delay to be specified in whatever its natural form is, and to then define coercions to get to whatever form is needed by a particular timing model.

There is also one special type of delay, called a null-delay. It is defined to have no effect on its input event time, always passing it through unchanged. It is useful for arcs that are not themselves being modelled with delays, but are just there to provide connectivity for the graph. One example is when assuming that all wire delays are zero (or folded in to the connected block delays), but arcs are still needed to connect the block arcs together. This is the default when extracting timing graphs from Oct (see Section 5).

1.4. Fundamental Operations and Critical Paths

Every timing model defines a minimum of four operations that apply to event times in that model. These are: *translating* an event time forwards or backwards in time by some real number, *delaying* an event time by some delay along an arc of the timing graph, *merging* multiple event times together at a node to produce a "worst-case" event time at that node (when there are multiple inputs to the node) and

comparing two event times together. Comparisons are determined by *constraints* which must hold between events; if the events in question are in different reference frames, the comparison produces a lower bound on the required translation between the two reference frames. If the events are in the same reference frame, a bound is produced between the frame and itself. Because the translation between any frame and itself is necessarily zero, a positive lower bound between the frame and itself indicates a timing error.

Every event calculated in the course of the analysis has some *critical predecessor*: the immediately preceding event that gave rise to this event and was most critical to the operation that generated the current event. Delay operations have only one predecessor, which is by definition the critical predecessor; merge operations will usually have more than one predecessor (although a merge is done at each node even if there is only one input arc). By tracing back a chain of critical predecessors from any given event to some original input event, we can generate the *critical path* that gave rise to the given event. In any constraint between two events, there will be two critical paths, one for each event. Typically, one of these two events is an event at a clock node. The bound generated by the constraint may be improved either by shortening one of the paths or by lengthening the other.

2. A BRIEF GUIDE TO COMMON LISP

ATV is written in Common Lisp, and uses its features to interact with the user. You don't need to know Lisp to use ATV, but it certainly doesn't hurt. The commands you type to ATV are actually Lisp expressions that are being evaluated just like any other Lisp expression. This section is intended to provide a brief introduction to a few Lisp features that may be useful to understand what you're typing. A complete reference to Common Lisp is Guy L. Steele, Jr., *Common LISP: the Language*, but this is a reference book, not a tutorial.

2.1. Lisp Ideas

The main thing to keep in mind about Lisp is that just about everything you type at it is an expression to be evaluated. When you type an expression at Lisp, it responds by typing the value of that expression. Expressions generally come in two varieties: simple objects (numbers, character strings, variable names and the like), and function calls, which are parenthesized lists. Many useful commands are built-in Lisp functions.

Numbers are typed just as you might expect, and evaluate to themselves. Character strings are enclosed in double quotes ("), and also evaluate to themselves. Symbols (variable names) are collections of "pseudo-alphanumeric" characters -- those with no special meaning defined, unlike parentheses or blanks. Ordinarily, alphabetic characters in symbol names are mapped to upper case. If you want variable names with both upper and lower case, you have to escape the name of the symbol by enclosing it in vertical bars. Thus abc, Abc, ABC, and AbC all denote the symbol ABC, but |abc| is a different symbol, in lower case.

2.2. Evaluation and Quoting

Lisp function calls look like parenthesized lists of expressions. The first element of the list is a symbol that is the name of the function to be called, and subsequent elements are the arguments to the function. The general rule is that first all the arguments to the function are evaluated, and then the function is called with those arguments. Thus the function call: (a b c) calls a function named a with two arguments, the current values of the variables b and c. The arguments to a function call may themselves be function calls, thus (a b (+ 3 1)) calls function a with the first argument being the value of b, and the second argument being the number 4 (the result of the function call (+ 3 1). This example points out an important feature of Lisp: arithmetic operations are function calls too, and they are written in the same prefix notation as other Lisp function calls, not the more familiar infix notation a + b. The standard functions for addition, subtraction, multiplication, and division are +, -, *, and /, respectively. This takes some getting used to, but anyone who has been able to make the adjustment to using RPN on an HP

calculator should be able to make this adjustment, too.

The above rule works fine in many cases, but sometimes you want to talk about the actual symbol or list, not its value. In general, the way you do this in lisp is to precede the symbol or list with a single quotation mark. Thus the value of (a b c) is the result of calling function a with arguments of the value of b and the value of c, the value of (a 'b 'c) is the result of calling function a with arguments of the symbols B and C, and the value of '(a b c) is just the list of three symbols, (A B C). A quote before an outer layer of parentheses prevents the evaluation of anything inside the parentheses. Keywords (symbols that start with a :) never need to be quoted: they always evaluate to themselves whether they are quoted or not.

There are a few exceptions to the above where lists that look like function calls do not in fact evaluate all of their arguments in spite of their not being quoted (and in these cases, it is generally an error to quote the argument in question). Most of which will not concern the non-programming user of ATV. One that such as user will want to use is the assignment operation setf, described below.

Two special symbols that are frequently used are **t** and **nil**. They are used to stand for true and false, respectively. They both evaluate to themselves. It is an error to attempt to assign a new value to either of them. In boolean contexts, anything other than **t** or **nil** is generally interpreted as meaning true. The symbol **t** is just a convenient special case that is always guaranteed not to be **nil**.

2.3. Optional and Keyword Parameters

In addition to standard required parameters, some functions can take additional parameters that need not be supplied in every call. Optional parameters are positional parameters that follow the required parameters. A **&rest** parameter is a special kind of optional parameter that accepts any number of additional arguments to the call. Keyword parameters follow any optional parameters, and consist of a keyword (beginning with a colon) followed by the value of that parameter. They may occur in any order. In describing a function, a function description such as:

(**anyfunc** *a b* **&optional** *c d* **&key :e :f**)

means that *a* and *b* are required parameters, *c* and *d* are optional parameters, and *e* and *f* are keyword

parameters using the keywords **:e** and **:f**, respectively. All of the following would be legal calls to anyfunc:

```
(anyfunc 1 2)
(anyfunc 1 4 5)
(anyfunc 3 4 5 7)
(anyfunc 3 4 6 7 :f 9)
(anyfunc 3 4 4 1 :e 7 :f 8)
```

The special symbol **&allow-other-keys** in a function description means that the function can accept keyword parameters other than those listed, usually to pass on to some other function.

2.4. Useful Builtin Lisp Functions

Load

Many built-in Lisp functions are potentially useful to the ATV user. One example is the **load** function. Its full syntax is:

(**load** *filename* **&key :verbose :print :if-does-not-exist**)

Load loads and executes a file of lisp expressions as if they were typed at the terminal. This lets the user put a sequence of commands in a file, and then read them in. Load takes one required parameter, which should be the name of a file. If the file name contains lower case characters (as most do), it should be entered as a character string (in double quotes). The file name must either be relative to the directory where ATV was run from, or an absolute pathname (load doesn't grok tilde expansion). Load can also take a keyword parameter called **print**. If the value of the print expression is anything but **nil** (Lisp shorthand for false), each expression read from the file will be printed with its returned value. The

default is not to print them.

Examples:

```
(load "nwdelay.lsp")
(load "/usr/tmp/foobar")
(load "my.commands" :print t)
```

Setf

Another useful lisp function is the assignment function, **setf**. Setf does not evaluate its first argument, but takes it as a description of a place to put the value of its second argument. The syntax is:

(**setf** *place expression*)

It returns the value of that argument (i.e., the value that was assigned). If the first argument is a symbol, setf will assign a value to that symbol. Thus:

```
(setf a 3)   => 3
(setf b 7)   => 7
(+ a b)      => 10
```

A variant of setf that is often used (mostly for historical reasons) is **setq**. Setq works just like setf, but is limited to the case where the first argument is a symbol. Setf can have other expressions in that place: for example, if *a* is a vector, then

```
(setf (aref a i) 10)
```

sets the *i*th element of *a* to 10.

Most symbols can have values assigned to them by setf, but there are a few that have hardwired values that cannot be changed. The most common are **t** and **nil**, which are used to mean true and false.

Keywords (symbols starting with a colon) are hardwired to themselves, and cannot be assigned values. It is an error to attempt to assign a value to such a symbol.

Output Control

Two variables that are useful in controlling the amount of printout you see are ***print-level*** and ***print-length***. The variable *print-level* controls how many levels down Lisp will print a nested data structure (such as lists within lists); *print-length* controls how long a list will be printed. If a structure has more than *print-level* levels, the inner levels will be printed as a pound sign (#); if a list is longer than *print-length*, only the first *print-length* items will be printed, followed by ellipses (...). Thus if *print-level* was 2 and *print-length* was 4, a function returning a value:

```
(a (b (c d) e f g) h i j k)
```

would print the result as:

```
(A (B # E F ...) H I ...)
```

This is a little different in the ExCL version of Common Lisp (on the suns): here you have to set top-level:*print-level* and top-level:*print-length*, respectively, to affect the way Lisp prints the values of expressions you type.

Lisp always prints the value of expressions you type at it (and every such expression has a value). If you are typing a command that will produce a long-winded value that you don't want to see at all (say you are assigning something long-winded to a variable and don't want to see the value), the easiest thing to do is to surround the expression that does what you want with a function call that returns something innocuous, such as T or NIL. A good function to use for this purpose is **null**: it takes one argument, returning T if the value of that argument is NIL, NIL if the value was anything else. Thus:

```
(null (setf var1 big-long-hairy-expression))  => NIL
```

Help

Two built-in functions that are useful in finding help about the usage of other variables and functions are **apropos** and **describe**. Apropos prints out all the symbols that contain a given string. For example:

```
Lisp> (apropos "APROPOS")

Symbols in package USER containing the string "APROPOS":
  APROPOS-LIST, has a definition
  APROPOS, has a definition
Lisp> (apropos "DESCRIBE")

Symbols in package USER containing the string "DESCRIBE":
  PCL::DESCRIBE-CLASS, has a definition
  PCL::DESCRIBE-INSTANCE, has a definition
  PCL::|(METHOD DESCRIBE-CLASS NIL)|
  DESCRIBE, has a definition
Lisp>
```

(Note that the string argument should be entered in all upper-case, to match the automatic translation of symbol names to upper case in Common Lisp.) The result identifies symbols that have function definitions ("has a definition") and are active variables ("has a value"). The symbol `PCL::|(METHOD DESCRIBE-CLASS NIL)|` above is an example of a symbol that is escaped with vertical bars (because it contains embedded blanks). The `PCL::` on the front of several symbols above just identifies them as being part of the PCL package.

Describe takes a symbol as its argument, and prints out information about the function with that name (if any) and the usage of that name as a variable, including its current value. For example:

```
Lisp> (describe 'describe)

It is the symbol DESCRIBE
Package:  SYSTEM
Value:    unbound
Function: compiled
    DESCRIBE object

      This function displays a complete description of the object
      to *STANDARD-OUTPUT*.  No values are returned.

Lisp> (describe '*print-level*)

It is the symbol *PRINT-LEVEL*
Package:  SYSTEM
Value:    the symbol NIL
      This variable specifies how many levels of a nested data
      object are to be printed.  If it is NIL, the entire object is
      printed.  Otherwise its value should be an integer indicating
      the maximum level to be printed.
Function: undefined

Lisp>
```

Note that you need to quote the symbol you are passing to describe (so that you pass the symbol and not its value).

Saving a Core Image

Both VAXLISP and ExCL offer a way to save and restore a core image of the whole lisp process to disk. This should be used with caution, since it is a good way to chew up many megabytes of disk space. However, it can be invaluable for checkpointing a long-running analysis at crucial points (such as after reading in a large example), and for increasing the amount of memory available to the process (VAX-LISP only).

Under VAXLISP, the function to do this is called **suspend**. The usage is:

```
(suspend "filename").
```

Under ExCL, the function is called **dumplisp**. It takes a keyword argument, **name**, that is the filename to be written. Hence the ExCL usage is:

```
(dumplisp :name "filename").
```

The file produced by ExCL is executable directly: you just type the filename to get back to where you left off. The VAXLISP file must be supplied as a command-line argument to the vaxlisp command when restarting lisp. Note that ATV itself is implemented as just such a suspended image: on the VAX, the atv command is a shell script that invokes the suspended image. To restart your own image, you would just type the same kind of command, replacing the argument atv.sus with a path to your own suspended image.

2.5. Troubleshooting

2.5.1. The Debugger: Getting out

If an error occurs while you are in ATV, either because you typed something wrong, or a bug occurred in the program, you may be thrown into the Lisp debugger. This will give you a different kind of prompt, and will usually print some kind of error message. In VAXLISP, the debugger will prompt with **`Debug i>`**, where i is the depth of the debugger (errors occurring while you are in the debugger

can recursively invoke the debugger at a greater depth). For example (in VAXLISP):

```
Lisp> (bad-expression a)

Fatal error in function SYSTEM::%EVAL (signaled with ERROR).
Undefined function: BAD-EXPRESSION

Control Stack Debugger
Frame #1: (EVAL (BAD-EXPRESSION A))
Debug 1> error

Fatal error in function SYSTEM::%EVAL (signaled with ERROR).
Undefined function: BAD-EXPRESSION
Debug 1> quit
Lisp>
```

While the debugger is useful for programmers trying to track down the cause of a bug, you will generally just want to be able to find out what caused the error and get out. The commands to do this in VAXLISP are shown above: *error* prints the error message (in this example, VAXLISP had already printed the error message, but it doesn't always do so), and *quit* gets you out of the debugger altogether. If the problem was something you typed, such as in the above example, you should be able to keep going after leaving the debugger; if you don't understand the error message, or it looks like a bug in ATV, you should contact the ATV maintainer (atv-support@degas). Include the error message that you got, and an explanation of what you were doing at the time.

2.5.1.1. Continuing from certain errors

Certain errors give you the option of fixing the problem and continuing from where you left off. These errors will print an error message that includes a description of what will happen if you continue. If you do want to continue, type `continue` in VAXLISP.

2.5.1.2. Debugger on a Sun

The commands to use if you are running under ExCL (on the Suns) are different, but the effect is the same. Type `:error` to print the error message; type `:reset` to get back to the top level, and type `:cont` to continue.

3. USING ATV

To get into ATV, the command is `atv` on both the VAX and Sun versions. The command will start up ATV and provide you with a Lisp prompt. From here, you can type commands as appropriate to read in and analyze a graph, as described in the specific sections below. I have found it useful to run atv as a shell command under the Emacs editor: the ability to go back and cut and paste input and results from previous commands is extremely handy.

3.1. Leaving ATV

One of the most important things about using any program is how to get out. In VAXLISP, the function **exit** will end the program, no matter where you are (at the top level, in the debugger, whatever). Just type `(exit)`. Under ExCL, the command `:exit` will do the same thing.

3.2. Help functions

In ATV, **help** is a variable that will print a short description of most of the top-level functions available in ATV (just type `help`). The **describe** function, described above, can be used to get more information on any of these functions.

3.3. Setting up for Analysis

The function **reset-all** erases all the timing and clocking graphs before starting a new analysis. Although not strictly necessary when starting with a fresh ATV image, I generally put it at the top of each new example, to keep any pieces of an old timing graph from clobbering the analysis. The syntax is:

(reset-all)

If the graph will use a non-standard delay format (such as nominal-worst), it should be defined (by reading in the appropriate file) before the graph is read in. Any non-standard delay coercions may be loaded at this time.

The **model** command sets up a default timing model for events that do not specify a timing model. The syntax is:

(model *modname*)

where *modname* is the name of the model (current choices are **singnum, minmax,** or **meanstd**, for the single-number, min-max, and simple mean-standard deviation models, respectively). It is a symbol, so it should be quoted. This command sets the global variable ***default-zero-et*** to a default event time representing "zero," which is used for any input events that do not specify an event time. You can accomplish the effect of the model command by setting this variable yourself (with **setf**). You will need to do this if you are using a model other than one of the standard models recognized by the model command (unless you want to supply an explicit event time for every input event), e.g., when using asymmetric rise and fall times.

ATV can be set to print a warning when a new node name is used that it thinks should have been defined earlier (such as when defining constraints). This is sometimes useful in catching typos, but it can also be a pain when it warns about perfectly legitimate nodes that just weren't defined yet. This behavior is controlled by the variable ***warn-unspecified-node***: a warning is only printed if *warn-unspecified-node* has a non-**nil** value. The default value is **nil**. This feature is marginally useful, as there are only a few cases that will be checked even if *warn-unspecified-node* is not **nil**. If you want to be warned in

the cases that are checked, you should set the variable here.

3.4. Defining a Graph

Defining an Arc

The basic command used to define a timing graph is the **arc** command. This is called once for every arc in the timing graph. Usually the graph definition will be read in from a file, but you can define it directly to the program. The syntax is:

(**arc** *fname tname delayblk* **&key :name :type :clock-name :open :close**

:delay-cycles)

Fname and *tname* are the names of the from and to nodes, respectively. Technically, node names may be any lisp object that compares EQUAL to a newly read identical object (i.e., no structures), but they are generally either symbols or character strings. The interface program from Oct defines all node names to be symbols. Using symbols makes it easier to access individual nodes (see the **abbrev** command, below), but does make it difficult to have case-sensitive node names, since Common Lisp symbols are normally all uppercase. If it is important to have case-sensitive node names, you can use character strings instead of symbols, or escape the symbol name by enclosing it in vertical bars.

The *delayblk* defines all the different delays formats that are defined for this arc. It can be a delay, a list of delays, or a list of forms that will each be evaluated to produce a delay. The symbol ***Null-Delay*** is bound to a constant null-delay, so you can use it as the delayblk if you want an arc with a null delay. See the section "Delay Constructors," below.

The keyword parameters are all optional. You can give the arc a descriptive name with the **:name** parameter. Like node names, an arc name can be a string or a symbol, and should be unique in the whole timing graph. If you don't give the arc a name, the system will make one up for you that looks something like "ARC00035". You should avoid giving arcs names of this form, so they don't conflict with the automatically generated ones. The **:type** parameter defines the type of the arc. It can be: 'INVERTING,

'NON-INVERTING, or 'UNKNOWN, as defined above. If you don't supply an arc-type, the system will try to figure it out from the delays you specified (if there was an asymmetric delay, it will include the arc type). If it can't do this, it will assume the arc is of UNKNOWN type. This is illegal for clock distribution networks, so make sure that all your clock distribution arcs have an arc-type specified.

Gated arcs are defined with the three keyword parameters: **:clock-name, :open,** and **:close.** These three go together: either they must all be specified for a given arc, or none of them can be. The clock-name names the controlling node for this arc; :open and :close define the opening and closing edges for the arc on that node. They take the values 'R and 'F, for rising and falling, respectively.

The **:delay-cycles** parameter is used to specify a *cycle delay* on the arc. See "Specifying Cycle Delays," below.

Delay Constructors

The functions used to describe delays depend on the specific models being used. The following delay construction functions are supplied with ATV:

Delay Constructors

Type	Constructor
Delay Union	(**du-del** *delay-list*)
Asymmetric Delay	(**as-del** *rise-delay fall-delay delay-type*)
Single Number	(**sn-del** *num*)
Min-Max	(**mm-del** *min max*)
Min-Typ-Max	(**mtm-del** *min typ max*)
Mean-Standard dev.	(**ms-del** *mean dev*)
Nominal-Worst	(**nw-del** *nominal worst*)

Some of the above forms may need some explanation. The delay union is a way to express a choice between several different delay formats. The program will pick a delay format from the union based on the type of event time specified by the user. The argument to **du-del** is a list of other delays, which can be constructed using the built-in **list** function:

```
(du-del (list (sn-del 3) (mm-del 2 5)))
```

expresses a union between a single number delay of 3 and a min-max delay of (2 5). The first two arguments to **as-del** are the delay if the output of the arc is rising, and the delay if the output of the arc is falling. They can be any legal form of delay, including a delay union of several delays. The delay-type can be any of the three forms: 'INVERTING, 'NON-INVERTING, or 'UNKNOWN, as defined above. For example:

```
(as-del (mm-del 3 4) (mm-del 4 7) 'inverting)
```

defines an inverting asymmetric delay based on the min-max model. See also the examples in Section 5.

Defining a Constraint

Constraints between events are defined with the **constrain** command. Its syntax is:

(**constrain** *node1 edge1 cond node2 edge2 trans eq-clock* &**key**

:incomp-clock :accept-p)

The constrain command defines a relationship that must hold between *edge1* at *node* and *edge2* at *node2*. The edges can be: **'CH-S, 'S-CH, 'R,** or **'F.** *Cond* is the condition that must hold between them; it is either **:geq** or **:leq.** The (quoted) symbols >= and <= may be used instead of :geq and :leq, respectively. *Trans* is a number that is a translation to be applied to event 2 before the constraint is checked. The condition :geq means that event 1 must occur *no earlier* than event 2 plus its translation; :leq would mean that it must occur *no later*. Thus to express a set-up constraint that the data at some latch must be stable 3 time units before the rising edge of the clock, you could say:

```
(constrain 'data 'ch-s '<= 'clk 'r -3 t)
```

or equivalently:

```
(constrain 'clk 'r '>= 'data 'ch-s +3 t)
```

The last three arguments are used to screen out events that are obviously irrelevant, based on the last

clock edge known to have occurred for each of the two events. By default, events are checked if their last clock edges are in the same relation as the condition to be checked. *Eq-clock* is a flag that is **t** if events with the *same* last clock-edge are to be checked against each other, **nil** if not. Thus in the above example, only those changing to stable events at the data port that have a last clock edge on or before the rising edge of the clock will be checked against that rising edge. If eq-clock had been **nil**, events originating from any rising clock edge would not be checked against that same edge. If this was an edge-triggered register that was used in a feedback path to itself (say a state register in a finite-state machine), you would not want to check such events, and so would specify **nil** for eq-clock. (Otherwise, the event caused when the register was clocked would eventually cause an event on the register input that would clearly be later than the arrival of the clock, violating the set-up constraint.)

The last two parameters, both keyword parameters, will not often be used. If ***incomp-clock*** **is t,** it means that events whose last clock edges are incomparable are to be checked against each other (this will only happen if you have two distinct clock edges whose order is not strictly known). The default is not to check such events. *Accept-p* allows for more complex rules about what events to check. If specified, it is a user-defined function that takes two last clock edges and returns **t** if the corresponding events should be checked, **nil** if not.

Implied Constraints

Certain constraints can also be implied, rather than explicitly stated. If the global variable ***implied-constraints*** is non-**nil** (the default), every gated arc has an implied constraint of the form:

```
(constrain from-node 'ch-s '<= clock-node clock-edge 0 nil)
```

where from-node is the from node of the arc, and clock-node is the controlling clock node. If the global variable ***check-close-edge*** is non-**nil** (the default), the clock-edge in the above is the closing edge of the arc; otherwise the opening edge of the arc is used.

Constraining System Outputs

In addition to checking internal constraints such as set-up and hold times, constraints are also used to check the arrival times of system outputs relative to some reference point, such as a global clock edge.

Abbreviating Node Names

Node names sometimes get long-winded, particularly if they are derived from a long chain of instance names in a cell hierarchy. To reduce the awkwardness of such names, the **abbrev** command is used. It processes every node in the graph, giving each such node a new name that will be used when displaying the node, and can be used to refer to the node (the original name remains a valid way to refer to the node). Abbrev will take node names of the form: a/b/c/d/e, and strip them down to the last component (after the last slash), adding a suffix of the form #nn (where nn is a number) to make the name unique. Optionally, it will also create a symbol with this name, and assign the node to that symbol. The syntax is:

(**abbrev &optional** *assign*)

To get the assignment behavior, assign should be specified with a non-**nil** value. This assignment behavior is really orthogonal to the act of abbreviation, and should eventually be split off into a separate command.

Cycle Delays

This is an esoteric topic that only applies if you have long combinational paths in your design that deliberately extend over multiple machine cycles without passing through gated arcs in the middle. Since ATV uses the gated arcs a signal passes through to determine its last clock edge (used for the constraint screening), such paths will result in events at the end having last clock edges that are off by a full cycle or more. To correct this, you can specify a cycle delay for any arc along such a path. The cycle delay means that a specified number of machine cycles is assumed to have elapsed for any events passing over the given arc. The format is:

(**cycledelay** *arc cycles*)

where arc is the arc in question (either the actual arc, or its name), and cycles is the non-negative integer that is the cycle delay to be applied to this arc. Because the arc command returns the newly created arc, you can use cycledelay when originally creating the arc, e.g.,

```
(cycledelay (arc ...) 1)
```

This is equivalent to specifying the initial cycle delay with the delay-cycles keyword on the arc command:

```
(arc ... :delay-cycles 1)
```

You can also use it later on to add a cycle delay to an arc after some preliminary analysis indicates the problem. This use would be equivalent to saying:

```
(setf (delay-cycles (find-timing-arc arc)) cycles)
```

3.5. Defining the Clocking

ATV assumes that there is some basic clocking pattern, which is defined as the ***ClockGraph***, which is repeated once per cycle. The ***ClockGraph*** is a (usually) cyclic graph of clock edges, which is unrolled n times into the ***UnrolledGraph***, an acyclic graph that has distinct nodes for every clock edge in every cycle. Normally the ***ClockGraph*** will consist of a single loop of clock edges, but it is possible to create more complicated structures.

Defining the Basic Clock Sequence

The **clocksequence** command is used to define the ***ClockGraph***. The syntax is:

(**clocksequence** *clocklist*)

Clocklist is a list of lists. Each sublist is a pair or triple, consisting of a clock origin node, the edge at that node, and (optionally) an instance identifier that is used to distinguish multiple occurrences of a clock

edge within the basic cycle. The meaning is that the specified clock edges occur in the specified order. Usually, this will be specified as a loop, with the first clock edge repeated at the end. Multiple clocksequence commands may occur; if so, the effect is cumulative. For example,

```
(clocksequence '((clk1 r a)(clk1 f a)(clk2 r)(clk2 f)(clk1 r b)(clk1 f b)
    (clk1 r a)))
```

defines a basic cycle in which clk1 goes high twice per cycle, and clk2 goes high in between the first and second time that clk1 goes high. If we now add:

```
(clocksequence '((clk1 f a)(clk3 f)(clk3 r)(clk1 r b)))
```

it means that there is an additional clock, clk3, which goes low between the first and second occurrence of clk1, but which is (as yet) asynchronous with respect to clk2. If we wanted to say that the rising edge of clk2 and the falling edge of clk3 are asynchronous with respect to each other, but both occur before either of the following edges, we could add:

```
(clocksequence '((clk3 f)(clk2 f)))
(clocksequence '((clk2 r)(clk3 r)))
```

We could also define an additional clock, clk4, which only goes low in between the alternate instances of clk1 going high:

```
(clocksequence '((clk1 f b)(clk4 f)(clk4 r)(clk1 r a)))
```

Note that the instance identifiers on clk1 could be any symbol, but must be consistent across multiple clocksequence commands. Figure A-4 shows the clock ordering defined above. Figure A-5 shows the corresponding ***ClockGraph*** after the above commands have been executed.

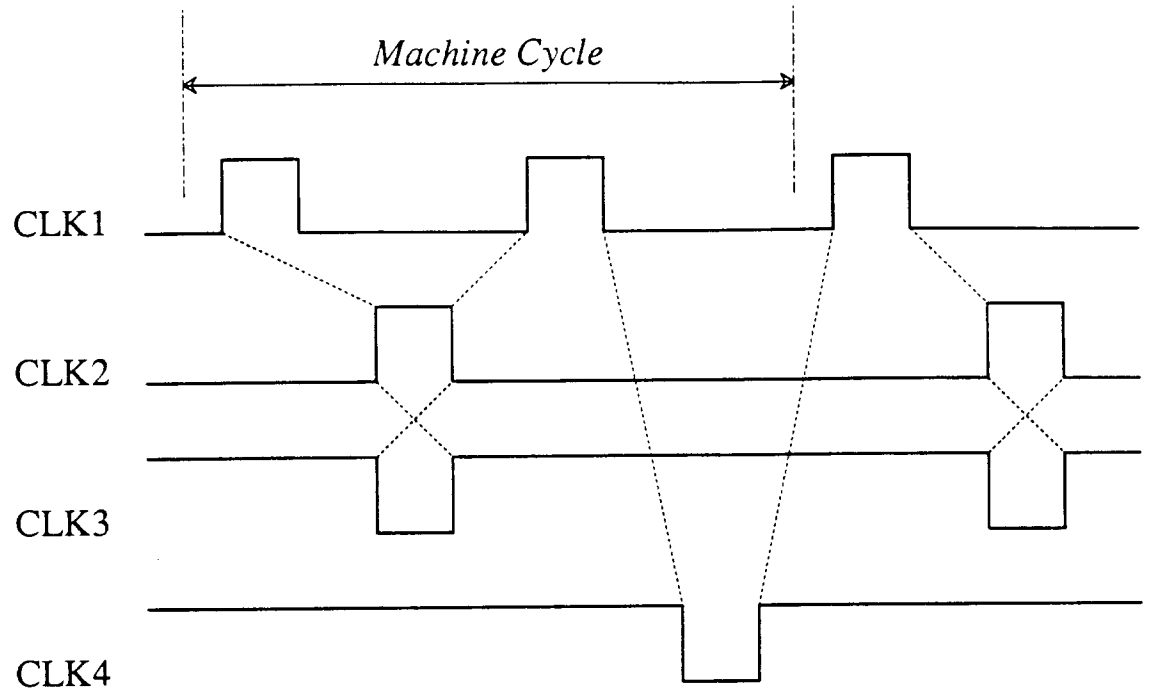

Figure A-4: A possible clock sequence.

Clock Aliases

Suppose in the above example, what we really meant was that clk2 rises *at the same time* that clk3 falls, and vice-versa. We could just use the first clocksequence command to define clk2 as a fundamental clock, and then define the appropriate clk3 edges to be *aliases* for the clk2 edges. To do this, we use the **clockalias** command. The syntax is:

(**clockalias** *clockedge1 clockedge2*)

where *clockedge1* and *clockedge2* have the same format as sublists in the clocksequence command. *Clockedge1* must already exist (defined through a previous clocksequence or clockalias command); *clockedge2* must be new and is defined as an alias for clockedge1. Thus to define the above aliases:

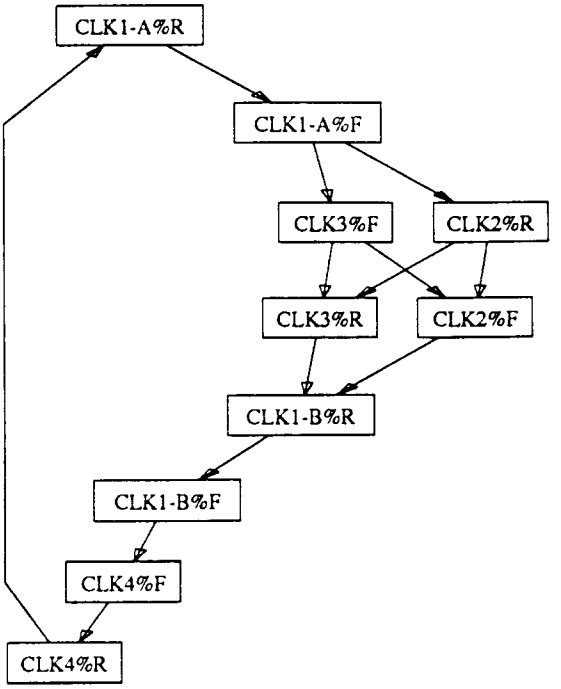

Figure A-5: The *ClockGraph* corresponding to the sequence shown in Figure A-4.

```
(clockalias '(clk2 r) '(clk3 f))
(clockalias '(clk2 f) '(clk3 r))
```

Here clk3 is effectively the negation of clk2, but originates from a different node in the graph.

Declaring Clock Origins

All the clock origin nodes specified by the user must be declared. This is currently done with the **declare-clock-source** command. The syntax is:

(**declare-clock-source** &**rest** *nodes*)

The main purpose of this command is to identify what nodes are clock sources, and to create any such nodes that may not have previously existed (if there were no arcs leading from such nodes). In principle,

clock sources should be precisely those nodes mentioned in the clockalias and clocksequence commands, and this command will be obsolete in a future version of the program.

Unrolling the Graph

The *UnrolledGraph*, a directed acyclic graph, determines the actual number and type of clock phases used in the analysis. It is generated from the *ClockGraph* by the **generate-unrolled-graph** command. The syntax is:

(**generate-unrolled-graph** *starting-edge* &**optional** *cycles*)

This generates the *UnrolledGraph* starting at *starting-edge*, which is a clock edge in *ClockGraph*, specified in the same form as a sublist in the clocksequence command, above. It unrolls the graph for *cycles* machine cycles (the default is 1), where a new cycle occurs every time *starting-edge* is reached in moving through the *ClockGraph*. For example the command:

```
(generate-unrolled-graph '(clk1 r a) 2)
```

on the above example generates the unrolled graph shown in Figure A-6.

3.6. Reference Frames and Phases

ATV conducts its analysis in multiple phases, using one phase per clock edge in the *Unrolled-Graph*. The number of distinct *reference frames*, however, is not predetermined, since it depends on your assumptions about which events are initially independent. You should set the variable *max-ref-frames* to the maximum number of reference frames you will be using. Generally this will either be the total number of phases (one phase per reference frame) or 1 (everything in the same reference frame: the default), but you could have some number different from either of these. Which reference frames are actually used depend on the numbers specified with the input events (see **set-event**, below).

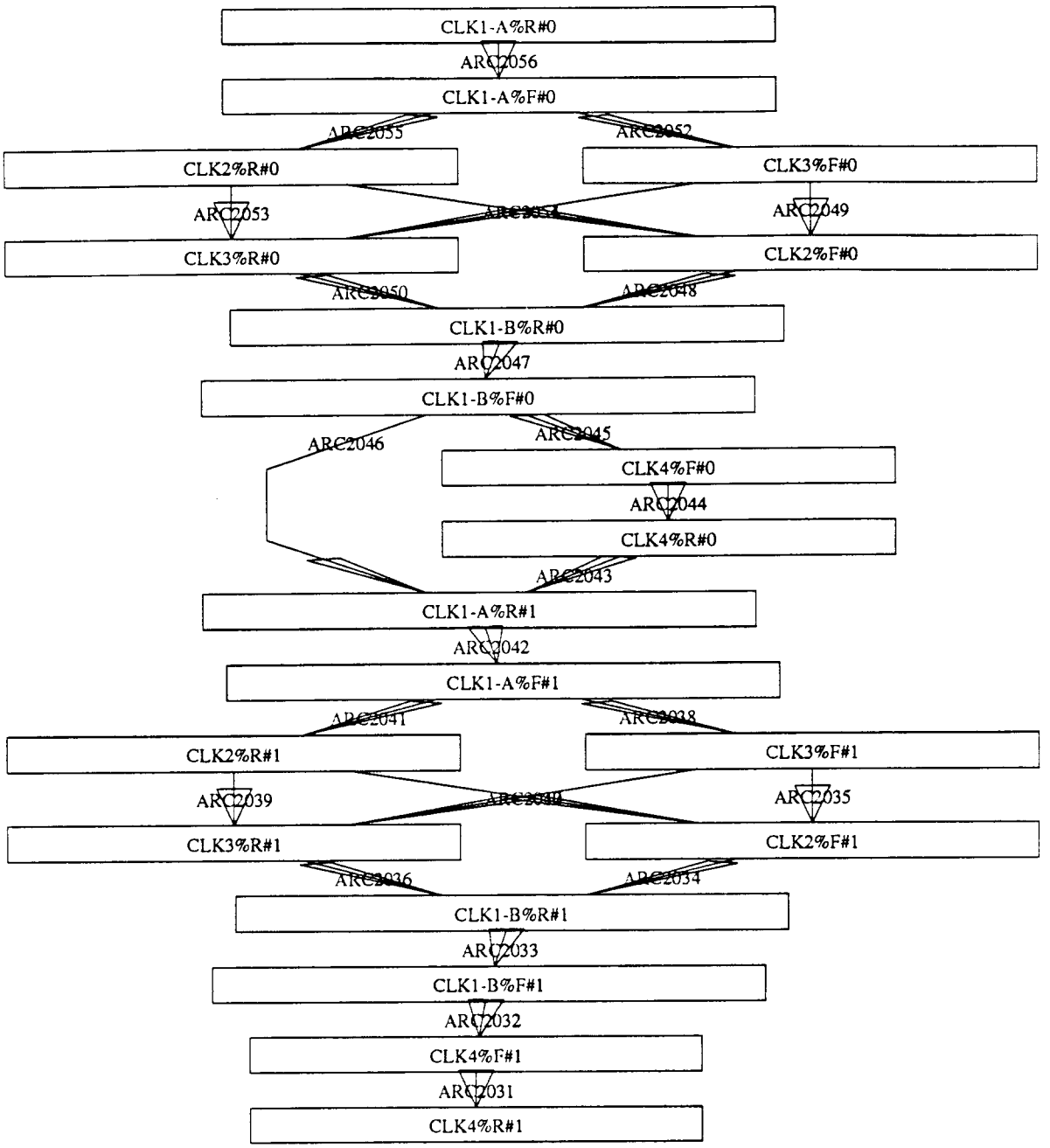

Figure A-6: The *UnrolledGraph* generated from the previous *ClockGraph*.

3.7. Optimization: Simplifying the Graph

After the timing graph has been completely defined, including all the clocking, you may choose to simplify the graph. The graph simplification algorithm looks for consecutive delays that can be

combined into a single delay, eliminating extraneous arcs and nodes. This can reduce the time and space required for the subsequent analysis, and reduce the length of reported critical paths. It is necessary to have fully defined the clocking before simplification so that essential nodes are not eliminated from the graph. Note: the results are not necessarily transparent for all timing models (though they are for cases such as the basic min-max or single-number models). If you plan to do an analysis that will coerce delays into a different form than that in which they appear, you should be aware of this issue. The **simplify-graph** function does the simplification. The syntax is:

(simplify-graph)

3.8. Conducting the Analysis

As noted above, ATV conducts its analysis in multiple phases, using one phase per clock edge in the *UnrolledGraph*. Before beginning the analysis, you should specify the input events, any arcs to be cut, and any output constraints that are needed.

After constructing the unrolled graph, the function **init-phases** should be run before any of the other functions in this section. The syntax is:

(init-phases)

Init-phases looks at the unrolled graph, numbers the phases, and sets up several data structures for the subsequent analysis, assuming one phase per entry in the unrolled graph. It needs to be run before any of the functions that require phase numbers as one of their arguments. If there is nothing in the unrolled graph, init-phases assumes that there will be only one phase, phase 0.

Temporarily Cutting Arcs

In looking at the results of a timing verification, it is sometimes apparent that the program is looking at a "false path": going through two arcs that cannot both be active at the same time. Such examples are typically analyzed using "case analysis": looking separately at the case when first one arc is used, then the other. The **cut-arcs** command is used for this purpose. It declares that certain arcs are temporarily "cut" during a specified phase in the analysis. The syntax is:

(**cut-arc** *arc phase*)

where *arc* is the name of the arc, or the arc itself, and *phase* is the numerical phase when the arc is to be considered cut. Cut arcs act as if they do not exist for that particular phase.

Specifying Input Events

The **set-event** command is used to define input events to the *TimingGraph*. The syntax is:

(**set-event** *node edge* &**key** :**event-time** :**ref-frame** :**phase**)

The *node* and *edge* are required. *Edge* can be any one of: 'R, 'F, 'CH-S, or 'S-CH. If event-time is not specified, *default-zero-et* is used (see **model** command above). *Ref-frame* and *phase* are the numbers of the reference frame and phase the event is scheduled on. If unspecified, they both default to 0 (the first, or only, reference frame/phase). In general, specifying phases as numbers is not really satisfactory: you really want to specify a phase as a machine cycle plus a clock edge. This may be changed in the future.

Input clock events also have to be specified explicitly. Although they can be specified individually using set-event, one case that occurs frequently is when you want every clock phase to occur in its own reference frame, all with the same initial event-time (representing time ''zero'' in that frame). The **set-clock-events** function handles this special case. The syntax is:

(**set-clock-events** &**key** :**event-time**)

where *event-time*, if specified, is the event time to be assigned to all input clock events (the default is

default-zero-et). This function also sets *max-ref-frames*, making it equal to the number of nodes in the *UnrolledGraph*. If you are not using this function, you have to specify all the input clock events explicitly.

Event Time Constructors

The functions used to create event times depend on the particular models being used. The following event time construction functions are supplied with ATV:

Event Time Constructors	
Model	Constructor
Single-number	(**sn-et** *number*)
Min-Max	(**mm-et** *min max*)
Simple Mean/Std. Dev	(**ms1-et** *mean dev*)
Asymmetric Rise/Fall	(**as-et** *rise-et fall-et*)

Most of these should be self-explanatory. The asymmetric rise/fall constructor, as-et, takes two event times as its arguments; thus its use would be something like:

```
(as-et (sn-et 3) (sn-et 5))
```

to specify a asymmetric event time with a rising time of the single number 3 and a falling time of the single number 4. Both rising and falling event times should generally be specified in the same model.

Specifying Output Constraints

Any output constraints should be specified using the **constraint** command described above. If there is no particular constraint on an output, but you are still interested in finding the latest output, you could try creating a phony event in its own reference frame at a node with no other connections to the graph, and specifying constraints to that event

Setting Analysis Control Variables

Before proceeding with the analysis, you should set the global variable ***stop-clocked***. This is a boolean variable that determines whether the analysis stops automatically at gated arcs. If it is **t** (the default), the analysis will stop at any gated arcs encountered. This is suitable for analyzing the purely combinational paths in one phase only. If you are concerned with constraints extending through gated arcs (such as transparent latches), *stop-clocked* should be set to **nil**. The function **compute** will prompt you and set *stop-clocked* automatically.

Another variable that can be set is ***max-crit-paths***. This limits the number of potential critical paths that will be reported per condition (e.g., between two reference frames). In practice, this is used to determine how many potentially critical events to keep track of for each pair of reference frames. The default is 5.

Another important variable is ***init-clocked-events***. This is a boolean vector with one element per phase. It controls whether events are initiated by any gated arcs opening in that phase. The idea is that in a multiple-cycle analysis, you don't normally learn anything new from events that are initiated after the first cycle, since their effect is duplicated by events that were initiated in the first cycle (if some arcs are cut, this may not apply). By default, this vector is **t** for all phases. You can set some of these phases off by using the built-in **fill** command. Fill sets all of the elements of a sequence (list or vector) within a specified range to a given value. Thus to turn off *init-clocked-events* for phases 6 through 11, you would say:

```
(fill *init-clocked-events* nil :start 6 :end 12)
```

Conducting the Analysis

The functions **compute** and **compute-events** are used to actually perform the analysis. Neither takes any parameters; the syntax is:

(compute)

(compute-events)

Compute is intended for interactive use, prompting for variable settings from the console (currently, it only asks about *stop-clocked*). After setting the variables, it calls compute-events to do the computation. From a file, it is probably more appropriate to set the variables yourself and call compute-events directly. In either case, the function takes all the input events that have been specified, propagates them through the network, and analyzes the implicit and explicit constraints to determine constraint violations and required translations between reference frames. The number of constraint violations within each reference frame is summarized at the end of the analysis.

3.9. Examining Results

3.9.1. Printing Results

After the analysis is completed, you can use the **report-constraints** function to get a report on the worst paths for each pair of reference frames. The syntax is:

(report-constraints &key :from :to :count)

The report gives each pair of reference frames as [i j], and lists up to *max-crit-paths* pairs of events for each frame. Each pair of events originates from some constraint: there is an event from reference frame i, an event from reference frame j, and the necessary translation between reference frames i and j to satisfy the constraint (the origin of reference frame j must be at least x time units later than reference frame i). If i=j, a positive translation means that a constraint has been violated (since the translation from a reference frame to itself is always zero). You can then use the **get-event** and **critical-path** commands to find the critical paths leading up to the two events involved in a given constraint. *From* and *to*, if specified, are lists of numbers or single numbers to restrict the report to the specified reference frames (the default is all reference frames). To get reference frames 0 to n-1, you can use the *iota* function

(inspired by the APL function of the same name; it returns a list of the numbers 0 through n-1):

(iota n)

If *count* is specified, only *count* event pairs will be listed per reference frame pair (in any case, at most *max-crit-paths* will be listed).

Examples are:

`(report-constraints :from (iota 8) :to (iota 8) :count 1)` - report the most critical constraint between each pair of reference frames (i j), for all i and j from 0 to 7.

`(report-constraints :from 3 :to '(2 4))` - report on critical constraints between reference frame 3 and reference frames 2 and 4.

`(report-constraints :count 2)` - report the two most critical constraints for all pairs of reference frames.

Printing General Information

The main function to print general information about timing objects (nodes, arcs, events, etc.) is **atv-print**. The syntax is:

(**atv-print** *printlist* **&rest** *options*)

Printlist is a list of items: nodes, arcs, node names, arc names, events, and so on, a single item, or one of the special keywords: **:nodes, :arcs,** or **:all**, which print all the nodes, arcs, or both in the timing graph, respectively. Options are keyword options that control how these items are printed. If no options are supplied, only the printable name of the item is shown. If options includes the keyword **:all**, every field associated with each object is printed. Individual slots can be selected for printing by including the keyword form of their name in the option list. If **:dest** appears in the option list, the next item is taken as the destination for the output. By default, it goes to the users terminal, but it can also be re-directed into a file by supplying a stream as the argument. Basically, anything that can be used as a destination for the Common Lisp **format** command can be used here as the dest (except that output to a character string

isn't fully supported yet).

Examples are:

`(atv-print 'node1 :outarcs :node-events)` - print the fields outarcs and node-events of the node named node1.

`(atv-print '(node1 node2) :all)` - print all the fields of the nodes named node1 and node2.

```
(with-open-file (ofile "myfile" :direction :output)
    (atv-print :all :all :dest ofile))
```
- using the with-open-file form, write all fields of all nodes and arcs of the graph to the file named *myfile*.

3.9.2. Finding Critical Paths

Given an event produced in the analysis, we can get the critical path of events that produced this event. The **get-event** function can be used to extract a given event from a specified node. The syntax is:

(**get-event** *node* &**optional** *phase index*)

This function returns an event associated with the specified node. Phase and index, if supplied, are numbers restricting the event to a particular phase and the index of the event within that phase. If not supplied and there is more than one appropriate value (with events at that node), the options will be listed, and the user will be asked to select one of them.

Single Critical Paths

Given an event, the **critical-path** function returns a list of events that forms the most critical path for the event in question. Its syntax is:

(**critical-path** *event*)

It can be combined with get-event to find the critical path for a specified event:

```
(critical-path (get-event 'node1))
```

If you want a list of the particular nodes and arcs on a critical path (for example, to generate a subgraph as described below), you can use the built-in Lisp function **mapcar** to apply the function **arc-node** to each event, as shown here:

```
(mapcar #'arc-node (critical-path (get-event 'node2)))
```

Multiple Critical Paths

ATV can also generate more than one critical path for a given event, in increasing order of total path slack (global slack). The **first-critical-path** function returns the most critical path for a given event, like the critical-path function described above, but it also returns a generator that can be passed to the function **next-critical-path** to produce additional critical paths for that event. If the input event has a non-zero slack with respect to some other global criteria (for example, if this event is generating a constraint that has additional slack with respect to the overall constraint), this slack can be passed as an optional argument to **first-critical-path** so that generated slacks will incorporate this initial slack. **Next-critical-path** takes a generator as its only argument, and returns the next most critical path and its global slack, updating the generator in the process. The built-in Lisp function **multiple-value-setq** can be used to accept multiple returned values from a function, such as the path and generator returned by **first-critical-path**. The syntax for these commands is:

(**first-critical-path** *event* **&optional** *slack*)

(**next-critical-path** *generator*)

For example:

```
(multiple-value-setq (p1 s1 q) (first-critical-path e1))
```

sets p1 to the first critical path for event e1, s1 to its slack (0 in this case), and sets q to the generator. Subsequently:

```
(multiple-value-setq (p1 s1) (next-critical-path q))
```

sets p1 to the next most critical path for this event, and s1 to its slack, updating the generator, q.

3.9.3. Displaying Graphs and Subgraphs

To generate a visual representation of any of the above graphs (*TimingGraph*, *ClockGraph*, or *UnrolledGraph*), or any of their subgraphs, the **showgraph** function is used. Its syntax is:

(showgraph &key :graph :filename :display-prog)

Showgraph prints out the arcs and nodes of the graph in **sungrab** format to the specified filename, and then invokes display-prog on that file. You can use the graph manipulation functions described in Section 6 to create subgraphs of the original graphs for display purposes: for example, to show only a critical path. If you are running on a Sun under SunTools, you can set the display-prog to the sungrab program, and have it open a window displaying the graph directly on your display. Otherwise, you can set the display-prog to something harmless, like /bin/true, and process the file yourself outside of ATV. Section 7 describes an RPC application called the digraph-maker that runs under the graphics editor VEM to display the graph when running under X windows. If the graph is not specified, it defaults to *TimingGraph*. If the filename is not specified, it defaults to "current.graph" in the current directory. If the display-prog is not specified, it defaults to the current value of the global variable ***display-prog***.

3.10. Defining and Using Delay Coercions

Right now, the delay coercion mechanism is rather primitive. Coercions are defined between different delay formats, and you can load them or not as you choose in order to turn them on or off. For details, see Section 8.2, "Defining Coercions Between Delay Classes."

3.11. Summary of Variables

Variable Name	Type	Default	Description
check-close-edge	bool	t	t if implied constraints check against closing arc edge.
ClockGraph	graph	empty	Description of clock edge precedence for 1 cycle.
ConstraintGraph	graph	empty	Graph containing all the explicit constraints between nodes of *TimingGraph*
CycleLimit	integer	1	Number of machine cycles covered by analysis.
default-zero-et	event-time	none	Default event-time used for input events when no event time specified.
display-prog	string		Default display program pathname for showgraph function.
implied-constraints	bool	t	t if all gated arcs imply a constraint check.
init-clocked-events	vector	all t	Boolean vector: do opening arcs in this phase initiate new events?
max-crit-paths	integer	5	Maximum number of potential critical events per pair of reference frames that will be kept.
max-ref-frames	integer	1	The maximum number of reference frames to be used by the analysis.
Null-Delay	delay	constant	Null delay used in defining arcs.
stop-clocked	bool	t	Stop analysis at gated arcs?
TimingGraph	graph	empty	The main graph for the analysis.
warn-unspecified-node	bool	nil	t if program prints warning when encountering nodes not previously defined.
UnrolledGraph	graph	empty	*ClockGraph* unrolled for *CycleLimit* machine cycles (by unroll-graph)
help	string	constant	Short description of user functions for interactive help.

3.12. Summary of User Functions

Function Name	Description
abbrev	Abbreviate *TimingGraph* node and arc names
arc	Create an arc in the timing graph
atv-Print	Print information about nodes or arcs.
clear	Clear all events from all nodes
clocksequence	Define list of clock edges
clockalias	Establish new clock edge as an alias for old
compute	interactive wrapping for compute-events
compute-events	compute event-times for nodes
constrain	Set constraint between events

Function Name	Description
critical-path	Return most critical path of events leading up to specified event
cut-arc	Cut an arc in the specified phase
cycledelay	Delay events on arc by # of cycles
declare-clock-source	Declare clock sources.
find-timing-arc	Find arc of *TimingGraph* with specified name
find-timing-node	Find node of *TimingGraph* with specified name
first-critical-path	Get most critical path, return generator for more
generate-unrolled-graph	generate the *UnrolledGraph* from *ClockGraph*
get-event	Extract specified event from a given node
init-phases	Set up for phase-based analysis.
model	Set timing model to the indicated name
next-critical-path	Return next most critical path produced by generator
report-constraints	Report on constraints between all reference frames
reset-all	Erase all stored timing graphs
set-event	Set an event at a given node
set-clocked-events	Set the same event times for all clock inputs
simplify-graph	Simplify the timing graph
showgraph	Display the specified graph

4. EXAMPLES

4.1. Transparent Latch Example 1

;;; This example is based on an example of transparent latch analysis

;;; from Norm Jouppi's thesis. Note: this is the original form of the problem,

;;; using the min-max model and without the added constraints ATV uses to

;;; analyze the loops!

```
Lisp> (reset-all)

NIL

Lisp> (arc 'a1 'b1 '(mm-del 25 40) :clock-name 'a :open 'r :close 'f)

#<Arc ARC2057 from A1->B1 in *TimingGraph*>

Lisp> (arc 'a1 'c1 '(mm-del 20 30) :clock-name 'a :open 'r :close 'f)
```

```
#<Arc ARC2058 from A1->C1 in *TimingGraph*>
Lisp> (arc 'a1 'a2 '(mm-del 12 20) :clock-name 'a :open 'r :close 'f)
#<Arc ARC2059 from A1->A2 in *TimingGraph*>
Lisp> (arc 'a2 'a1 '*Null-Delay* :delay-cycles 1)
#<Arc ARC2060 from A2->A1 in *TimingGraph*>
Lisp> (arc 'b1 'c1 '(mm-del 20 35) :clock-name 'b :open 'r :close 'f)
#<Arc ARC2061 from B1->C1 in *TimingGraph*>
Lisp> (arc 'c1 'a1 '(mm-del 15 20) :clock-name 'c :open 'r :close 'f)
#<Arc ARC2062 from C1->A1 in *TimingGraph*>

;;; define the default event time.
Lisp> (model 'minmax)
#S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0)
Lisp> (declare-clock-source 'a 'b 'c)
(#<Node C in *TimingGraph*> #<Node B in *TimingGraph*> #<Node A in *TimingGraph*

;;; Now I'm about to make a mistake: I'm going to try unrolling
;;; the clock graph before putting anything in it (with clocksequence).
;;; This would be ok if I wasn't doing anything involving clocking (i.e.,
;;; just analyzing combinational logic between latches), but not here.

Lisp> (generate-unrolled-graph '(a r) 2)
NIL
Lisp> (init-phases)
NIL
Lisp> (fill *init-clocked-events* nil :start 6 :end 12)
```

```
;;; And now the mistake I made earlier catches up to me and throws me
;;; into the Lisp debugger.  In this case, the debugger doesn't print the
;;; error message until I ask for it (with error).
Control Stack Debugger
Frame #3: (FILL #(T) NIL :START 6 :END 12)
Debug 1> error

Fatal error in function FILL (signaled with ERROR).
End: 12 must be less than, or equal to, the length of the sequence: 1.

;;; This suggests the cause of the error: for some reason,
;;; *init-clocked-events* is the wrong length.  You could just exit from
;;; the debugger, but I try evaluating a couple of expressions to find out
;;; what the problem is:
Debug 1> (length (all-nodes *UnrolledGraph*))
0
;;; There aren't any nodes in the *UnrolledGraph* !!
Debug 1> (length (all-nodes *ClockGraph*))
0
;;; And this is why: there aren't any nodes in *ClockGraph* either.
;;; Now that I've identified the problem, I'll exit from the debugger, and
;;; go back to fix the problem by backing up to where I should have defined
;;; the clock sequence.

Debug 1> quit
Lisp> (clocksequence '((a r) (a f) (b r) (b f) (c r) (c f) (a r)))
T
```

```
Lisp> (generate-unrolled-graph '(a r) 2)
#<Node A%R#0 in *UnrolledGraph*>
Lisp> (length (all-nodes *UnrolledGraph*))
12
Lisp> (init-phases)
NIL
Lisp> (fill *init-clocked-events* nil :start 6 :end 12)
#(T T T T T T NIL NIL NIL NIL NIL NIL)

;;; Now this is ok. The first six phases will initiate clocked events;
;;; the others won't.

Lisp> (set-clock-events)
NIL
Lisp> (compute)
Ignore this question. [Y] :~&Should analysis stop at clocked arcs? [N] :n
;;; The prompter is a little buggy right now, asking that extraneous
;;; question at the beginning. It provides a default value in brackets
;;; that is used if you just hit return when it asks the question.
;;; You could just as well have set *stop-clocked* yourself and called
;;; (compute-events) instead, but eventually (compute) will step you through
;;; more of the process automatically.

Entering section: SETUP
Leaving section: SETUP, time = 0.02 seconds.
Entering section: ANALYSIS
Leaving section: ANALYSIS, time = 3.22 seconds.
```

```
Entering section: CONSTRAINT-ANAL

Leaving section: CONSTRAINT-ANAL, time = 0.3 seconds.

NIL

;;; Now report on what the constraints between reference frames were.

;;; The first column is the pair of reference frames affected; the second

;;; is the required translation between frames (i.e., the origin of frame 3

;;; must be at least 40 time units after the origin of frame 0), and the

;;; third is the pair of events that gave rise to the constraint.

Lisp> (report-constraints :count 1)

[3 0]:   40    Event CH-S on #<Node B1 in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 25 MAX-TIME 40), last 0, frame 0
               Event F on #<Node B in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 3, frame 3

[5 0]:   75    Event CH-S on #<Node C1 in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 45 MAX-TIME 75), last 2, frame 0
               Event F on #<Node C in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 5, frame 5

[5 2]:   35    Event CH-S on #<Node C1 in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 20 MAX-TIME 35), last 2, frame 2
               Event F on #<Node C in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 5, frame 5

[7 0]:   95    Event CH-S on #<Node A1 in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 35 MAX-TIME 95), last 4, frame 0
               Event F on #<Node A in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 7, frame 7

[7 2]:   55    Event CH-S on #<Node A1 in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 35 MAX-TIME 55), last 4, frame 2
```

```
                Event F on #<Node A in *TimingGraph*>
                     at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 7, frame 7
[7 4]:   20     Event CH-S on #<Node A1 in *TimingGraph*>
                     at #S(MIN-MAX-ET MIN-TIME 15 MAX-TIME 20), last 4, frame 4
                Event F on #<Node A in *TimingGraph*>
                     at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 7, frame 7
[9 0]:   135    Event CH-S on #<Node B1 in *TimingGraph*>
                     at #S(MIN-MAX-ET MIN-TIME 37 MAX-TIME 135), last 6, frame 0
                Event F on #<Node B in *TimingGraph*>
                     at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 9, frame 9
[9 2]:   95     Event CH-S on #<Node B1 in *TimingGraph*>
                     at #S(MIN-MAX-ET MIN-TIME 60 MAX-TIME 95), last 6, frame 2
                Event F on #<Node B in *TimingGraph*>
                     at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 9, frame 9
[9 4]:   60     Event CH-S on #<Node B1 in *TimingGraph*>
                     at #S(MIN-MAX-ET MIN-TIME 40 MAX-TIME 60), last 6, frame 4
                Event F on #<Node B in *TimingGraph*>
                     at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 9, frame 9
[11 0]:  170    Event CH-S on #<Node C1 in *TimingGraph*>
                     at #S(MIN-MAX-ET MIN-TIME 57 MAX-TIME 170), last 8, frame 0
                Event F on #<Node C in *TimingGraph*>
                     at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 11, frame 11
[11 2]:  130    Event CH-S on #<Node C1 in *TimingGraph*>
                     at #S(MIN-MAX-ET MIN-TIME 80 MAX-TIME 130), last 8, frame 2
                Event F on #<Node C in *TimingGraph*>
                     at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 11, frame 11
[11 4]:  95     Event CH-S on #<Node C1 in *TimingGraph*>
```

```
                    at #S(MIN-MAX-ET MIN-TIME 60 MAX-TIME 95), last 8, frame 4
               Event F on #<Node C in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 11, frame 11
NIL
;;; Now just look at those constraints from frame 7
Lisp> (report-constraints :from 7 :count 1)
[7 0]:   95   Event CH-S on #<Node A1 in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 35 MAX-TIME 95), last 4, frame 0
               Event F on #<Node A in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 7, frame 7
[7 2]:   55   Event CH-S on #<Node A1 in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 35 MAX-TIME 55), last 4, frame 2
               Event F on #<Node A in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 7, frame 7
[7 4]:   20   Event CH-S on #<Node A1 in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 15 MAX-TIME 20), last 4, frame 4
               Event F on #<Node A in *TimingGraph*>
                    at #S(MIN-MAX-ET MIN-TIME 0 MAX-TIME 0), last 7, frame 7
NIL
;;; I want to look at that first pair of events in more detail.
;;; Here I goof again: I ask for the event on node A1 from reference frame 0,
;;; but the events are indexed by phase (last 4), not reference frame.
;;; I outsmarted myself here: if I left the phase off altogether, the function
;;; would have told me which phases had events at that node, and prompted
;;; me for one, unless there was only one such phase.
Lisp> (get-event 'a1 0)
```

```
Control Stack Debugger
Frame #3: (GET-EVENT A1 0 NIL)
Debug 1> error

Fatal error in function GET-EVENT (signaled with ERROR).
No events for this phase!
Debug 1> quit
```

;;; I try again. This time I get the phase right:

```
Lisp> (get-event 'a1 4)
Node has 6 events for this phase:
0: #<Event S-CH on #<Node A1 in *TimingGraph*> at #S(MIN-MAX-ET MIN-
TIME 35 MAX-TIME 55), frame 2>
1: #<Event CH-S on #<Node A1 in *TimingGraph*> at #S(MIN-MAX-ET MIN-
TIME 35 MAX-TIME 95), frame 0>
2: #<Event CH-S on #<Node A1 in *TimingGraph*> at #S(MIN-MAX-ET MIN-
TIME 35 MAX-TIME 55), frame 2>
3: #<Event CH-S on #<Node A1 in *TimingGraph*> at #S(MIN-MAX-ET MIN-
TIME 15 MAX-TIME 20), frame 4>
4: #<Event S-CH on #<Node A1 in *TimingGraph*> at #S(MIN-MAX-ET MIN-
TIME 35 MAX-TIME 95), frame 0>
5: #<Event S-CH on #<Node A1 in *TimingGraph*> at #S(MIN-MAX-ET MIN-
TIME 15 MAX-TIME 20), frame 4>
Which one?1
#<Event CH-S on #<Node A1 in *TimingGraph*> at #S(MIN-MAX-ET MIN-TIME 35 MAX-TI
```

;;; After prompting, I finally get the event I wanted. In the

;;; next couple of commands, I take advantage of a Common Lisp feature:

*;;; the special symbol * always has the value returned by the command you*

;;; just executed (this may differ a little while you are in the debugger).

*;;; Likewise, ** has the value from two commands ago, and *** has the*

;;; value from three commands ago. Since the command I just executed returned

*;;; the event I wanted to look at, * currently holds that event. Thus*

;;; I can ask for its critical path as I do below. I could also have

;;; accomplished this by typing (critical-path (get-event 'a1 4 1)),

;;; or by saving the original result in a variable and using it here,

;;; but this is a handy feature to know about.

```
Lisp> (critical-path *)
(#<Event R on #<Node A in *TimingGraph*> at #S(MIN-MAX-ET MIN-TIME 0
MAX-TIME 0), frame 0>
     #<Event R on #<Node A in *TimingGraph*> at #S(MIN-MAX-ET MIN-TIME
0 MAX-TIME 0), frame 0>
     #<Event CH-S on #<Arc ARC2057 from A1->B1 in *TimingGraph*> at
#S(MIN-MAX-ET MIN-TIME 25 MAX-TIME 40), frame 0>
     #<Event CH-S on #<Node B1 in *TimingGraph*> at #S(MIN-MAX-ET
MIN-TIME 25 MAX-TIME 40), frame 0>
     #<Event CH-S on #<Arc ARC2061 from B1->C1 in *TimingGraph*> at
#S(MIN-MAX-ET MIN-TIME 45 MAX-TIME 75), frame 0>
     #<Event CH-S on #<Node C1 in *TimingGraph*> at #S(MIN-MAX-ET
MIN-TIME 45 MAX-TIME 75), frame 0>
     #<Event CH-S on #<Arc ARC2062 from C1->A1 in *TimingGraph*> at
#S(MIN-MAX-ET MIN-TIME 60 MAX-TIME 95), frame 0>
     #<Event CH-S on #<Node A1 in *TimingGraph*> at #S(MIN-MAX-ET
```

```
MIN-TIME 35 MAX-TIME 95), frame 0>)
;;; Now I get a list of the arcs and nodes on that path, by using the
;;; mapcar function to extract the arc-node of each event, as described
;;; earlier.
Lisp> (mapcar #'arc-node *)
(#<Node A in *TimingGraph*> #<Node A in *TimingGraph*> #<Arc ARC2057
from A1->B1 in *TimingGraph*> #<Node B1 in *TimingGraph*> #<Arc
ARC2061 from B1->C1 in *TimingGraph*> #<Node C1 in *TimingGraph*>
#<Arc ARC2062 from C1->A1 in *TimingGraph*> #<Node A1 in *Timing-
Graph*>)
;;; Now I use the create-subgraph function (described in Section 6)
;;; to create a subgraph of *TimingGraph* consisting of the nodes and arcs
;;; along the critical path, and then use the showgraph function to
;;; produce a file "crit.path.1" of this subgraph that I can then display,
;;; as described in Section 7. If I had also dumped the *TimingGraph*,
;;; I could use the :select-subgraph command of the digraph-maker described
;;; in Section 7 to highlight the critical path while looking at a graph
;;; of the whole *TimingGraph*.

Lisp> (setf subgraph1 (create-subgraph *TimingGraph* *))
#<Subgraph NIL "in graph *TimingGraph*" 4260765>
Lisp> (showgraph :graph subgraph1 :filename "crit.path.1")
NIL
Lisp>
```

4.2. Transparent Latch Example 2

```
;;; Based on Jouppi's loop example using a single-number model.
;;; This is the version I refer to in Chapter 4 of my dissertation.
Lisp> (reset-all)

NIL
Lisp> (setq *stop-clocked* nil)
NIL
Lisp> (arc 'a1 'a2 '*Null-Delay* :clock-name 'a :open 'r :close 'f)
#<Arc ARC1304 from A1->A2 in *TimingGraph*>
Lisp> (arc 'a2 'b1 '(sn-del 40))
#<Arc ARC1305 from A2->B1 in *TimingGraph*>
Lisp> (arc 'a2 'a1 '(sn-del 12) :delay-cycles 1)
#<Arc ARC1306 from A2->A1 in *TimingGraph*>
Lisp> (arc 'b1 'c1 '(sn-del 35) :clock-name 'b :open 'r :close 'f)
#<Arc ARC1307 from B1->C1 in *TimingGraph*>
Lisp> (arc 'c1 'a1 '(sn-del 15) :clock-name 'c :open 'r :close 'f)
#<Arc ARC1308 from C1->A1 in *TimingGraph*>
;;; Short loop constraint.
Lisp> (constrain 'a 'f '<= 'a1 's-ch 0 nil
        ;; Check the falling edge of A against S-CH edges from the next
      ;; machine cycle (only). Hacked to hardwire the edges and screen
      ;; out other comparisons. This is an example of the use of the
      ;; accept-p escape to define a comparison condition that is more
      ;; complicated than just what order the edges fall in.
        :accept-p #'(lambda (a-edge a1-edge)
              (and (= a-edge 1)
```

```
                (= a1-edge 6))))
#<Arc ARC1309 from #<Node A in *TimingGraph*>->#<Node A1 in *TimingGraph*> in *C
;;; Long loop constraint
Lisp> (constrain 'a1 'ch-s '<= 'a 'r 0 t)
#<Arc ARC1310 from #<Node A1 in *TimingGraph*>->#<Node A in *TimingGraph*> in *C
Lisp> (clocksequence '((a r)(a f)(b r)(b f)(c r)(c f)(a r)))
T
Lisp> (model 'singnum)
(SN-ET 0)
Lisp> (declare-clock-source 'a 'b 'c)
(#<Node C in *TimingGraph*> #<Node B in *TimingGraph*> #<Node A in *TimingGraph*
Lisp> (generate-unrolled-graph '(a r) 2)
#<Node A%R#0 in *UnrolledGraph*>
Lisp> (init-phases)
NIL
Lisp> (fill *init-clocked-events* nil :start 6 :end 12)
#(T T T T T T NIL NIL NIL NIL NIL NIL)
Lisp> (setq *max-ref-frames* 12)
12
Lisp> (set-clock-events)
NIL
Lisp> (compute-events)
Entering section: SETUP
Leaving section: SETUP, time = 0.02 seconds.
Entering section: ANALYSIS
Leaving section: ANALYSIS, time = 1.42 seconds.
Entering section: CONSTRAINT-ANAL
```

```
Leaving section: CONSTRAINT-ANAL, time = 0.25 seconds.

NIL

Lisp> (report-constraints :from (iota 8) :to (iota 8) :count 1)

[0 1]:   -12   Event F on #<Node A in *TimingGraph*>
                    at (SN-ET 0), last 1, frame 1
               Event S-CH on #<Node A1 in *TimingGraph*>
                    at (SN-ET 12), last 6, frame 0

[3 0]:    40   Event CH-S on #<Node B1 in *TimingGraph*>
                    at (SN-ET 40), last 0, frame 0
               Event F on #<Node B in *TimingGraph*>
                    at (SN-ET 0), last 3, frame 3

[5 0]:    75   Event CH-S on #<Node C1 in *TimingGraph*>
                    at (SN-ET 75), last 2, frame 0
               Event F on #<Node C in *TimingGraph*>
                    at (SN-ET 0), last 5, frame 5

[5 2]:    35   Event CH-S on #<Node C1 in *TimingGraph*>
                    at (SN-ET 35), last 2, frame 2
               Event F on #<Node C in *TimingGraph*>
                    at (SN-ET 0), last 5, frame 5

[6 0]:    90   Event CH-S on #<Node A1 in *TimingGraph*>
                    at (SN-ET 90), last 4, frame 0
               Event R on #<Node A in *TimingGraph*>
                    at (SN-ET 0), last 6, frame 6

[6 2]:    50   Event CH-S on #<Node A1 in *TimingGraph*>
                    at (SN-ET 50), last 4, frame 2
               Event R on #<Node A in *TimingGraph*>
                    at (SN-ET 0), last 6, frame 6
```

```
[6 4]:    15     Event CH-S on #<Node A1 in *TimingGraph*>
                      at (SN-ET 15), last 4, frame 4
                 Event R on #<Node A in *TimingGraph*>
                      at (SN-ET 0), last 6, frame 6
[7 0]:    90     Event CH-S on #<Node A1 in *TimingGraph*>
                      at (SN-ET 90), last 4, frame 0
                 Event F on #<Node A in *TimingGraph*>
                      at (SN-ET 0), last 7, frame 7
[7 2]:    50     Event CH-S on #<Node A1 in *TimingGraph*>
                      at (SN-ET 50), last 4, frame 2
                 Event F on #<Node A in *TimingGraph*>
                      at (SN-ET 0), last 7, frame 7
[7 4]:    15     Event CH-S on #<Node A1 in *TimingGraph*>
                      at (SN-ET 15), last 4, frame 4
                 Event F on #<Node A in *TimingGraph*>
                      at (SN-ET 0), last 7, frame 7
NIL
Lisp>
```

5. GENERATING ATV INPUT FROM OCT

The interface program **oct2atv** will generate a timing graph description of an Oct cell provided it is built out of lower level cells that each have a timing model, as described below. The output of oct2atv are arc and constraint commands that can be used as input to ATV to define the timing graph. The cell should generally be run through the flattener, **flattiming**, first. Oct2atv can handle most cases of hierarchical cells, at the expense of adding some extra arcs for the interconnect, but there are some cases, such as cells wiring two ports together internally, that the flattener will catch and oct2atv will just ignore.

Usage of these programs is:

oct2atv [-d delaystring] [-f flushlevel] cell [view [facet [version]]]

flattiming [-l level] [-o cell:view] cell:view

-d provide debugging information

-o specify output facet

-l n flatten until ...

n=0 there are no instances (default)

n=1 all instances have VIEWTYPE = PHYSICAL

n=2 user function returns true

n=3 all instances have CELLCLASS = LEAF

Flattiming should be run with the option -l 2 (it is a special version of the general flattener, **octflat**). The -d option for oct2atv allows you to specify a delay string that will be used for the delay block for all interconnect arcs that are generated. The default is *Null-Delay*. Neither flattiming nor oct2atv object if some cell is totally missing a timing model: they will just ignore it silently. This can be a source of errors.

5.1. Oct Timing Model Description

This is intended to provide a uniform description for annotating timing models on instances, interface facets and contents facets. If there are problems with this (i.e., bags can't be attached to instances, the MODEL bag (which is part of symbolic policy) can only be attached to an interface facet, etc.), we will change it (the first problem could be addressed by converting all bags into properties, the second by attaching the timing model directly to the [instance/interface facet/ contents facet] rather than the MODEL bag). We should find out which of these are real problems that need to be addressed.

The semantics of having a timing model attached are that a timing model on the instance or interface facet is a model describing the complete timing behavior of that instance/cell and everything below it, while a timing model on the contents facet describes delays of the interconnect in that facet, which are to be composed with timing models of the included instances. An active timing model on the instance overrides an active timing model on the interface facet, which overrides an active timing model on the contents facet, which overrides the standard default assumptions about interconnect delays.

Prefixing convention: I understand that "unofficial" properties, etc. in Oct are supposed to be prefixed with a tool abbreviation. I have only followed this for the ATV-TIMING bag and the ATV-TIMEGRP property, since these are the only objects that are attached to "official" oct objects. Everything else occurs only within a timing model, and doesn't have any attachments that go outside the model: if you're inside a timing model, you know where you are, and that everything you see is "unofficial". Adding prefixes to all these objects would be counterproductive, since many of the names of bags, etc. are themselves effectively parameters with semantics determined by user-defined models; it wouldn't do to go changing these names out from under the program that works on them (the only other option would be to re-write the interface program oct2atv to strip off a leading ATV- on everything it sees).

Ok, here we go. Assuming that we can attach a MODEL bag to instances and contents facets as well as interface facets, a timing model consists of:

I.: a MODEL bag attached to the instance/facet, which contains:

A: an ATV-TIMING bag.

I.A.: The ATV-TIMING bag contains:

1: an ACTIVEMODEL sprop (optional),

2: an ARCS bag, and

3: a CONSTRAINTS bag (optional).

I.A.1.: the ACTIVEMODEL property, if present with value "F", means "Ignore this model." It provides a way to turn off a timing model to get at the lower-level stuff

underneath it (e.g., for top-down refinement). This property is currently ignored.

I.A.2: The ARCS bag contains:

a: individual arc bags. The name of each bag is the name of the arc.

I.A.2.a: Each individual arc bag contains:

(1): a FROMGRP sprop,

(2): a TOGRP sprop,

(3): a DELAYS bag,

(4): a CLKGRP sprop (optional),

(5): an OPEN sprop (optional), and

(6): a CLOSE sprop (optional).

I.A.2.a.(1,2,4): The FROMGRP, TOGRP, and CLKGRP properties describe the nodes that the arc is from, to, and clocked by, respectively. Ordinarily these will be the names of terminals on the instance/facet containing the timing model; however they may also be nodes not corresponding to any terminal (indicated by providing a name that doesn't match any terminal), or a node corresponding to a group of terminals (established by the ATV-TIMEGRP sprop attached to the terminals).

I.A.2.a.(5,6): The OPEN and CLOSE properties describe the opening and closing edges of a clocked arc. For now, they can take the values "R" (for rising) or "F" (for falling).

I.A.2.a.(3): The DELAYS bag contains a (potentially complex) structure that expresses all the different delay descriptions for that arc. It deserves a separate section all by itself (Section III below).

I.A.3: The CONSTRAINTS bag format contains:

a: individual constraint bags.

I.A.3.a: Each individual constraint bag contains:

(1): a FROMGRP sprop,

(2): a FROMEDGE sprop,

(3): a COND sprop,

(4): a TOGRP sprop,

(5): a TOEDGE sprop,

(6): a DELTA sprop,

(7): a EQ-CLOCK sprop,

(8): a INCOMP-CLOCK sprop (optional), and

(9): an ACCEPT-P sprop (optional).

These correspond to the individual elements of an ATV *constraint* command. All but DELTA are quoted automatically in assembling the constraint command. DELTA is not quoted so that it can be a general expression to be evaluated.

II. Terminal Groups.

A timing model may chose to group two or more terminals together. This is done by attaching an ATV-TIMEGRP sprop to terminals of the instance/facet. The value of ATV-TIMEGRP attached to a terminal is the name of the timing group that terminal belongs to. It replaces the terminal name for references such as the FROMGRP, TOGRP and CLKGRP properties of arcs (above). If this feature is not used, each terminal is simply referenced by it's own name.

III. DELAYS bag.

III.A. Motivations:

The motivations for the structure of delay descriptions are: (1): It should be totally user-extensible. While some pre-defined delay formats are supplied, there should be no artificial limits on the user's ability to define new formats. (2): It should be self-defining. Because of (1), it is impossible for the interface program oct2atv to know what each delay format should look like. So oct2atv has to be able to figure out how to translate a delay format into something atv can read just by looking at

the structure in oct. (3): Delays and groups of delays should be able to be components of higher-level delay formats. The classic example is asymmetric-delay, which has components rise-delay and fall-delay, each of which can be an arbitrary grouping of delay formats. (4): Delay components should be able to be arbitrary expressions, which can be written in terms of variables passed from the surrounding oct environment (such as LOADCAP).

III.B. Current format:

The current format is that the DELAYS bag has several sprops attached. The names of these sprops are currently irrelevant. The value of each sprop is a string which produces a delay when evaluated by atv (in lisp). This will generally be a parenthesized list with the name of a delay-constructor function first, followed by the arguments to that function. Some built-in constructor functions currently supplied with atv that take positional parameters are:

1. sn-del - single number delay, takes 1 argument (the number).
2. mm-del - min-max delay, takes 2 args min and max.
3. mtm-del - min-typ-max delay, takes 3 args min typ and max.
4. ms-del - mean-standard deviation delay, takes 2 args mean and std. dev.
5. as-del - asymmetric rise-fall delay. Takes 3 args: a rising delay, a falling delay, and a delay-type ('INVERTING, 'NON-INVERTING, or 'UNKNOWN). Rising-delay and falling-delay are based on the direction of the output transition.
6. du-del - delay union. Takes 1 arg, which is a list of one or more delays, when evaluated by lisp. This lets you group alternate delay formats together, with the choice to be made by atv.

Examples of current format:

Property Value:

"(sn-del 3)" - Specifies a single-number delay of 3.

"(sn-del (+ 5.3 (* 2.0 7)))" - Specifies a single-number delay, with value equal to the expression 5.3 + 2.0 * 7 = 19.3. By itself, this is not very useful; but it can be extended to allow for parameters:

"(sn-del (+ 5.3 (* 2.0 LOADCAP)))" - Specifies a single-number delay, with value of 5.3+2.0*LOADCAP. In order for this to work properly, the interface program has to put out some surrounding code that will assign a value to LOADCAP at the time this expression is read by atv. This is not yet supported, but could be added.

"(as-del (sn-del 3) (sn-del 5) 'INVERTING)" - Specifies an asymmetric-delay, with a single-number rising delay of 3, a single-number falling delay of 5, and a delay-type of INVERTING (input transition is opposite in direction to output transition).

"(as-del (du-del (list (sn-del 3) (mm-del 2 4)))
(du-del (list (sn-del 5) (mm-del 4 6))) 'INVERTING)"
- Specifies a more complex asymmetric-delay, where the rising delay is either the single number 3 or the min-max delay 2-4, and the falling delay is either the single number 5 or the min-max delay 4-6.

6. GRAPH MANIPULATION FUNCTIONS

This section describes the external classes and methods of the graph package, which runs under Portable Common Loops in CommonLisp. All the major graphs in ATV use this package, so the functions described here should work for any of them. They are intended to be further specialized as needed by individual applications. The syntax used to describe methods is similar to that used in Common Lisp: the Language by Guy Steele, Jr. *[GVar]* means that the method can be used as a locator in a setf expression.

6.1. Basic Classes

abstract-graph *[Class]*

basic-graph *[Class]*

subgraph *[Class]*

node *[Class]*

arc *[Class]*

abstract-graph is a class with no methods and no instance variables that is a superclass of both **basic-graph** and **subgraph**. It allows methods to be written that will accept either of these, since neither is a proper subclass of the other. **basic-graph** is the basic class out of which graphs are built. Nodes and arcs are always contained within some basic-graph. The node "foo" within one basic-graph is not the same as the node "foo" within another basic-graph. A **subgraph** is built on top of either a basic-graph, or another subgraph. It specifies some subset of nodes and arcs from the graph it is based on, and consists of those nodes and arcs. Nodes and arcs can only be created or destroyed in a basic-graph, though nodes and arcs can be added or deleted from the list of nodes belonging to a subgraph. If a node or arc is removed from a subgraph, it is also removed from all subgraphs directly or indirectly based on that subgraph.

6.2. Instance Creation Functions

(make-basic-graph &key :name :default-dir :multiple-arcs *[Method]*

:dup-arc-action :self-arcs-allowed :node-creator :arc-creator)

This function creates a basic-graph. Explanations of the keyword parameters and default values may be found in the section on instance variables and pseudo-instance variables.

(create-node-in-graph *graph name* **&key :dup-err** *[Method]*

:node-creator &allow-other-keys)

This function creates a node within the basic-graph *graph*, using the node-creator function of *graph*, and names it *name*. *Name* can be any lisp object, but it should be printable. It is used to identify the created node, and to look it up in the graph (using **equal** as the test). Many applications will use a character string for *name*, but it could be a symbol or other lisp object. *Name* must be unique to a given graph. If it matches an existing node name or nickname in *graph*, the previously existing node is returned, unless **:dup-err** is specified and is non-nil, in which case an error is signaled. Any other keyword parameters are passed to the node-creator function.

(create-arc *fnode tnode* **&rest r &key :arc-creator)** *[Method]*

This function creates an arc between *fnode* and *tnode*, which must both be nodes belonging to the same graph. The arc-creator function of that graph is invoked, and passed any remaining arguments. (Note: the default arc-creator, make-arc, will create a new name for the arc unless one is specified with a :name keyword pair). The new arc is checked for legality (based on **multiple-arcs, dup-arc-action, self-arcs-allowed,** and **default-dir** of the graph), and appropriate corrective action is taken if it is not legal.

(create-subgraph *graph nodelist* **)** *[Method]*

Create-subgraph creates a subgraph of abstract-graph *graph*, consisting of the nodes in *nodelist*, which must be a list of nodes of *graph* (duplicates are allowed).

6.3. Instance Variables and Pseudo-Instance Variables

Most of the methods described here represent true instance variables. They are described with their default values. A pseudo-instance variable is a method that acts as a read-only instance variable, and which may be specialized to a true instance variable in subclass implementations. The only pseudo-instance variable described here is **display-name.**

(**name** *basic-graph*) *[GVar]*

(**name** *node*) *[GVar]*

(**name** *arc*) *[GVar]*

Name, although it can be used in a setf expression, should be treated as read-only once an instance is initially created. It serves two functions: (1): it is used to generate the print name of the instance, and (2): for nodes and arcs, it is used as a handle to look up the associated node or arc in a graph. The default value for graphs and nodes is **nil**; for arcs, a new string of the form "ARCnn" will be generated if no name is specified when the arc is created.

(**default-dir** *graph*) *[GVar]*

Default-dir establishes the default interpretations of arcs for a given basic-graph, *graph*, and all of its subgraphs. Any method that uses the **:dir** keyword uses **default-dir** of the (possibly implied) basic-graph as its default value if **:dir** is not specified. The legal values are: **:forw**, for directed graphs, which interprets the arc (fnode tnode) as a directed arc from fnode to tnode; **:both**, for undirected graphs, which interprets the arc (fnode tnode) as an undirected edge connecting fnode and tnode; and **:back**, which is unlikely as a default, but which would interpret the arc (fnode tnode) as a directed edge from tnode to fnode. The default value of **default-dir** is **:forw**.

(**multiple-arcs** *basic-graph*) *[GVar]*

This is a boolean variable which indicates whether it is legal for more than one arc to connect the same two points. For directed graphs (as determined by **default-dir**), an arc from A to B is considered distinct from an arc from B to A. The default is **nil**.

(**dup-arc-action** *basic-graph*) *[GVar]*

If multiple-arcs are not legal (as determined by **multiple-arcs**), this variable indicates the action to be taken when an attempt is made to create a duplicate arc. The legal values are: **:new**, which replaces the old arc with the new one silently, **:old**, which discards the new arc silently, and **:notify**, which notifies the user and asks for the desired action. The default is **:notify**.

(**self-arcs-allowed** *basic-graph*) *[GVar]*

This boolean variable indicates whether it is legal to have an arc from a node to itself. The default is **nil**.

(**node-creator** *basic-graph*) *[GVar]*

(**arc-creator** *basic-graph*) *[GVar]*

These variables establish the functions to be used to create new nodes and arcs, respectively, for the given graph. They provide hooks for these functions to be specialized either for an entire subclass of basic-graph (by establishing new defaults for the instance variables), or for a particular instance of basic-graph (by rebinding the variables). The defaults are **#'make-node** and **#'make-arc**, respectively.

(**graph** *node*) *[GVar]*

This instance variable should be considered read-only. Its value is the basic-graph that *node* belongs to.

(**base-graph** *subgraph*) *[GVar]*

This instance variable should be considered read-only. Its value is the graph a given subgraph is based on.

(**display-name** *node*) *[Method]*

(**display-name** *arc*) *[Method]*

This pseudo-instance variable defines the name by which a given node or arc will be displayed (e.g., for sungrab format). For instances of the classes **node** or **arc**, this is the same as the name of the instance. But subclasses can specialize this, and make it a true instance variable.

6.4. Object Modification Methods

(**modify-arc** *graph arc* **&key :name :from-node :to-node** *[Method]*
:display-name &allow-other-keys)

Arc *arc* in graph *graph* has the specified instance variables modified, with all the internal data structures updated appropriately.

(**modify-node** *graph node* **&key :name :display-name** *[Method]*
&allow-other-keys)

Node *node* in graph *graph* has the specified instance variables modified, with all the internal data structures updated appropriately.

6.5. Object Deletion Methods

(**delete-arc** *arc*) *[Method]*

Arc is deleted from its containing graph.

(delete-node *node***)** *[Method]*

Node, and any attached arcs, is deleted from its containing basic-graph.

(erase-graph *graph***)** *[Method]*

If *graph* is a basic-graph, all of its nodes and arcs are deleted. If it is a subgraph, it becomes empty (but its previous nodes are still members of their basic-graph).

6.6. Basic Graph Operations

All the operations in this section should work equally well for basic-graphs and subgraphs.

(neighbors *graph node* **&key :dir)** *[Method]*

Neighbors returns a list of all nodes adjacent to *node* in *graph*. If **:dir** is **:forw**, only outgoing arcs from *node* are examined; if it is **:back**, only incoming arcs are examined. *Node* need not be in *graph*: if *graph* is a subgraph, and *node* is a node outside the subgraph, the result will be only those neighboring nodes that belong to the subgraph.

(arcs-of *graph node* **&key :dir)** *[Method]*

Arcs-of returns a list of all arcs connected to *node* in *graph*. If **:dir** is **:forw**, only outgoing arcs from *node* are examined; if it is **:back**, only incoming arcs are examined.

(nodes-of *graph arc* **&key :dir)** *[Method]*

Nodes-of returns a list of all nodes connected to *arc* in *graph*. If **:dir** is **:forw**, only the to-node of *arc* is in the list; if it is **:back**, only the from-node is in the list. Only those nodes that are actually in *graph* will be in the list, though *arc* need not be entirely in *graph* (if, for example, *graph* is a subgraph,

and arc goes between it and the rest of the graph).

(**all-arc-names** *graph*) *[Method]*

This function returns a list of all the names and nicknames of arcs in *graph*.

(**all-node-names** *graph*) *[Method]*

This function returns a list of all the names and nicknames of nodes in *graph*.

(**all-arcs** *graph*) *[Method]*

This function returns a list of all the arcs in *graph*.

(**all-nodes** *graph*) *[Method]*

This function returns a list of all the nodes in *graph*.

(**is-arc-p** *graph name*) *[Method]*

(**find-arc** *graph name*) *[Function]*

These two functions look up *name* in *graph*, returning the associated arc if *name* is the name or nickname of a arc of *graph*, and **nil** otherwise. The choice between them is merely stylistic: is-arc-p may be used as a predicate, while find-arc may be used when the actual arc must be looked up.

(**is-node-p** *graph name*) *[Method]*

(**find-node** *graph name*) *[Function]*

These two functions look up *name* in *graph*, returning the associated node if *name* is the name or nickname of a node of *graph*, and **nil** otherwise. The choice between them is merely stylistic: is-node-p may be used as a predicate, while find-node may be used when the actual node must be looked up.

(**arc-of-p** *graph arc*) *[Method]*

(**node-of-p** *graph node*) *[Method]*

These predicates verify that *arc* or *node* is indeed part of *graph*, returning **t** if this is true, **nil** otherwise.

(**end-vertices** *graph* **&key :dir**) *[Method]*

This function returns a list of all nodes of *graph* that have no attached arcs in the specified direction. Thus it can be used to find sources and sinks of a digraph, or isolated nodes in any graph.

6.7. Nickname Methods

Nicknames are alternate names by which a node or arc can be looked up in a graph. A given node or arc can only have one name, but it can have an unlimited number of nicknames. These may be used whenever the full name of the node or arc would be unwieldy, or where it may be known by another name to the user (see **display-name** above). Nicknames, like names, must be unique within a graph. A nickname of an arc cannot conflict with another arc name or nickname, though it may be the same as a node name or nickname.

(**add-nickname** *node nickname*) *[Method]*

(**add-nickname** *arc nickname*) *[Method]*

(**delete-nickname** *node nickname*) *[Method]*

(**delete-nickname** *arc nickname*) *[Method]*

The given nickname is added to or deleted from the list of nicknames for the given node or arc.

6.8. Subgraph Methods

These methods are those relating to the generation and manipulation of subgraphs.

(**add-to-subgraph** *subgraph node*) *[Method]*

Node, which must already be a node of the graph *subgraph* is based on, is added to the list of nodes in *subgraph*.

(**add-to-subgraph** *subgraph arc*) *[Method]*

Arc, which must already be an arc of the graph *subgraph* is based on, is added to the list of arcs in *subgraph*. The endpoints of *arc* are added to the list of nodes in *subgraph*.

(**delete-from-subgraph** *subgraph node*) *[Method]*

Node, which must already be a node of *subgraph*, is deleted from the list of nodes in *subgraph*. Any arcs attached to *node* are also deleted from *subgraph*.

(**delete-from-subgraph** *subgraph arc*) *[Method]*

Arc, which must already be an arc of *subgraph*, is deleted from the list of arcs in *subgraph*.

(**root-graph** *graph*) *[Method]*

Root-graph is the basic-graph at the bottom of the chain of subgraphs from abstract-graph *graph*.

(**subgraph-of-p** *graph super*) *[Method]*

This predicate is true of abstract-graphs *graph* and *super* only if they have the same **root-graph** and if every node of *graph* is also a node of *super*.

(**formal-subgraph-of-p** *graph super*) *[Method]*

This is like **subgraph-of-p**, but is true only if a subgraph relationship between *graph* and *super* can be inferred strictly from the formal chain of inclusions between them: it is t if and only if *super* is part of the chain of base-graphs descending from *graph* to its root-graph. It should be a lot faster to check than **subgraph-of-p**.

(**complement-of** *graph base*) *[Method]*

This function returns a new subgraph, based on abstract-graph *base*, that is the complement of abstract-graph *graph*. In order to apply it, (subgraph-of-p *graph base*) must be t: *graph* must be a subgraph of *base*. The subgraph returned contains all arcs of *base* that do not connect to nodes of *graph*, and all nodes of *base* that are not in *graph*.

(**neighborhood** *graph base radius* **&key :dir**) *[Method]*

Graph and *base* are abstract-graphs; *radius* is an integer. *Graph* must be a subgraph of *base* (as determined by subgraph-of-p). The function returns a subgraph, based on *base*, that consists of *graph* and all nodes within *radius* of *graph* in *base*. All the arcs of *base* induced on this node set are included in the neighborhood graph.

(**induce-subgraph** *graph base*) *[Method]*

This function adds all arcs of *base* to *graph* that connect two nodes in *graph* and are in the base graph of *graph*. It returns *graph* as the result. In order to apply it, (subgraph-of-p *graph base*) must be t: *graph* must be a subgraph of *base*.

6.9. Output Methods

(**sungrab-form** *graph destination*) *[Method]*

This function outputs a sungrab-format description of abstract-graph *graph* to *destination*, which may be any valid destination of the **format** function. If the output is to a file, sungrab may be run on this file to browse the graph. The **display-names** of nodes and arcs are used for the sungrab labels, so if you want these labels to be other than the names of the nodes and arcs, you have to use subclasses of node and arc that have display-name defined as a true instance variable.

(**dump-graph** *graph destination* **&optional** *graphname*) *[Method]*

This function outputs a series of commands to *destination* which, when evaluated, will rebuild abstract-graph *graph* as a basic-graph, bound to symbol *graphname* (which defaults to a gensym if not supplied). *Destination* may be any valid destination of the **format** function. It makes use of the methods **dump-node** and **dump-arc**, which will probably need to be specialized to work on subclasses of node and arc. In general, this is only likely to work if the names of nodes and arcs are self-identifying objects (like strings). Dump-graph returns the value of *graphname*.

(**dump-node** *node destination graphname*) *[Method]*

This function outputs a series of commands to *destination* which, when evaluated, will rebuild *node* (with its nicknames) as part of the graph that is assumed to have been bound to *graphname*.

(**dump-arc** *arc destination graphname*) *[Method]*

This function outputs a series of commands to *destination* which, when evaluated, will rebuild *arc* (with its nicknames) as part of the graph that is assumed to have been bound to *graphname*.

(**dump-subgraph** *subgraph destination graphname* *[Method]*
&optional *subname*)

This function outputs a series of commands to *destination* which, when evaluated, will rebuild *subgraph* as part of the graph that is assumed to have been bound to *graphname*, binding it to *subname*

(which defaults to a gensym if unspecified). It assumes that the base graph has already been bound to *graphname*. Dump-subgraph returns the value of *subname*.

6.10. Possible Extensions and Future Ideas

Subgraphs should be able to select individual arcs from the base graph, rather than automatically inheriting all the arcs connecting nodes of the subgraph. This would be useful for displaying spanning trees or critical paths without extraneous arcs, for example. This feature has been added, though some functions continue to use node-only semantics, as noted above.

Traversal functions (such as **neighbors**) should be able to take a general predicate function as a keyword parameter, and only traverse those arcs for which the predicate is true. For example, one might want only those neighbors connected via "BLUE" arcs.

Likewise, one might want to mix nodes and arcs of different classes within the same graph. **Create-node-in-graph** and **create-arc** could take optional parameters to override the default arc-creator or node-creator of the graph.

It might be useful to allow the user to define relationships between different basic-graphs, based on the idea that nodes with the same name are the "same" node in the two different graphs.

Add some way of "clustering" nodes and arcs so that they act as a single node or arc.

Add standard graph algorithms such as minimum-spanning-tree, shortest-path, etc.

7. DISPLAYING DIRECTED GRAPHS USING VEM

A program called *digraph-maker* has been written to display directed graphs, based on the *sungrab* program, written by Eli Messinger, *et al.* (UC Berkeley), and the *slide-maker* program, written by Rick Spickelmier, *et al.* (UC Berkeley). Its manual page is reproduced below.

NAME

digraph-maker – remote application for displaying directed graphs.

DESCRIPTION

digraph-maker is a program for displaying directed graphs. It is run as a remote application from VEM. The following sections describe the *digraph-maker* commands.

CREATION

"file name" : **read-graph** Read the specified file, and lay out the graph as an internal data structure. This must be the first command executed from within the digraph-maker. The file specifies the graph in sungrab input format, which is:

Format of file is as follows. All lines start at leftmost column; all fields are separated by tabs. Optional fields are indicated in brackets []. If a field is included, all previous fields on the line must also be included. If no name is given, the *NAME* line can be omitted.

...

...

[*NAME*

one line description of the graph]

NODES

node-name [node-label] [shape] [x y] [status] [level position]

...

...

EDGES

from-node-name to-node-name [edge-label]

...

...

All lines before the *NAME* line are ignored.

See "BUGS" below for caution on file names.

: **write-gremlin** Write the graph out as a gremlin file. The output file name is derived from the file name used in the read-graph command, by either changing the suffix to .grn, if a period was used in the original file name, or else by appending .grn to the whole file name.

: **display-graph** Display the graph in the current oct cell. If the properties of this cell are not properly set up to use the slide technology, the program will adjust them and attempt to re-read the cell from disk. You will be prompted to confirm this action, and should do so unless there are other changes to the cell that you want to save.

"file name" : **select-subgraph** Read the specified file, which should be a subgraph of the original graph, and highlight all the nodes and arcs listed in it. The file is in the same format as the original input file. See "BUGS" below for cautions on file names and edge labels.

MISCELLANEOUS

: **fonts** Show a list of the known fonts. The list of known fonts is: cyrilic, gothgbt, gothgrt, gothitt, greekc, greekcs, greeks, italicc, italiccs, italict, romanc, romancs, romand, romanp, romant, scriptc, scripts.

: **exit** Exit *digraph-maker*. Remember to **save-window** the slide before exiting **VEM**.

PRINTING

Use *grn* with *ditroff* to print the generated gremlin file, or incorporate it in a document. Use *hp-slide* or *oct2ps* to generate plots of slides. *hp-slide* can generate color plots, on paper or transparencies. *oct2ps* can generate black and white plots on PostScript printers. See the man pages for more information.

BUGS

All file locations are relative to the home directory of the remote user, not relative to the working directory of VEM (this will be fixed eventually).

All colors, fonts, etc., are currently hardwired into the program.

Selection of arcs is done by edge-labels, so this will fail unless all such arcs have unique labels.

FILES

~cad/lib/technology/slide

SEE ALSO

hp-slide(1), oct2ps(1), vem(1), grn(1)

TRADEMARKS

PostScript is a trademark of Adobe.

AUTHOR

Dave Wallace (UC Berkeley)

ACKNOWLEDGEMENTS

Graph layout code is based on the sungrab program, written by Eli Messinger, *et al.* (UC Berkeley)

Display code is based on the slide-maker program, written by Rick Spickelmier, *et al.* (UC Berkeley), though the digraph-maker uses a different oct format to save disk space, so the two are not compatible.

8. DEFINING YOUR OWN TIMING MODELS

This section describes how to define your own timing models. To do anything at all fancy requires knowledge of Common Lisp [Ste84] and some knowledge of Portable Common Loops.* It may be helpful to read Chapter 4, Section 3, "IMPLEMENTATION IN PORTABLE COMMON LOOPS," and Chapter 6, Section 1.2, "Coercion Management," of the companion to this report, *Abstract Timing Verification for Synchronous Digital Systems*. These sections discuss how features of PCL are used in ATV. It may be helpful to look at how the standard models supplied with ATV are implemented: most delay classes are defined in the file delays.lsp, and most event-time classes are defined in the file models.lsp. The *class* is the basic type unit in PCL; a class is defined in ATV for each delay format and for each type of event time. New classes may either be defined from scratch, or based on existing classes. A class that is based on one or more other classes is called a *subclass* of those classes; they are *superclasses* of the new class. For example, you could define a new mean and standard-deviation event time class that uses the delay and translation operations of the standard model, but implements new versions of the merge and comparison operations. The lisp forms to create new classes and methods are called **defclass** and **defmethod**, respectively.

The syntax of the defclass form is:

(**defclass** *classname* (*superclass-list*) (*slot-list*) *class-options*)

Classname is the name of the class being defined; it should not conflict with the name of any existing class (unless that class is being redefined). *Superclass-list* is a list of names of the immediate ancestors of this class in the class inheritance hierarchy; objects in the new class are automatically members of all the ancestor classes, and inherit their slots and behavior. The list may be the empty list (), in which case the new class is only a descendent of the universal class, **t**. *Slot-list* is a list of the new instance

* Up-to-date documentation on PCL is difficult to come by, in part because it is a rapidly evolving system. The basic published reference is [BKK86], which discusses the basic principles behind PCL; however, much of the specific syntax in the examples of this paper is now obsolete. A more recent reference was the *Common Lisp Object System Specification*, by Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya Keene, Gregor Kiczales, and David A. Moon, which was distributed electronically from Xerox (anonymous ftp from parcvax.xerox.com, over the ARPANET); this, however, appears to have been withdrawn pending the publication of a revised edition.

variables of this class (like member names of a structure). Every object in this class has a slot for each such instance variable, including any slots inherited from ancestor classes. Each element of the slot-list can either be the name of a slot, or a parenthesized list headed by the name of the slot that includes options that apply to that slot. *Class-options* is a set of 0 or more options that apply to the whole class. Examples of the use of defclass are presented below.

Methods are defined by the **defmethod** form. Methods are a lot like functions, except that there may be many different definitions of a method distinguished by the classes of the required arguments, and the actual code invoked is selected at run-time by matching the actual run-time types of the arguments to the method against the different method definitions. The most specific method found is invoked.

The syntax of defmethod is just like the function definition form **defun** [Ste84, p. 67], except that for any required parameter in the lambda-list, you may substitute the expression `(parm class)` for `parm`, which defines this method as matching only *parm*s of type *class*. If a required parameter appears without such a class restriction, it is assumed to be the same as specifying `(parm t)`, matching parms of type t (the universal class) - i.e., match any parm whatever in this position. All methods with the same name must have the same number of required parameters. If a method appears with exactly the same class specifications for all its parameters as a previous method of the same name, it replaces the previous method.

The body of any method may include a call to the special function **pcl::call-next-method**, a function with no arguments. This function invokes the next-most specific form of the method in which it is enclosed. Thus it can be used to invoke default behavior for a method that has been partially redefined. Note that replacing a method with a new method who's body consists of the single line (**pcl::call-next-method**) effectively deletes that particular method, except for the added run-time penalty of invoking the (now null) method.

8.1. Defining New Delay Classes

Figure A-7 shows the class hierarchy for the built-in delay classes. In general, a separate class should exist for each unique delay format. Most delay classes should be a sub-class, directly or indirectly, of the class abstract-delay, which establishes default behavior for all delay objects. The only exception are delay classes that are intended to have special side-effects when involved in a delay operation; these should be subclasses of some higher-level class, such as coerceable-object. Any such classes should have special delay methods defined for it; the only example of such a class currently part of ATV is the null-delay class, which passes any event-time through unchanged.

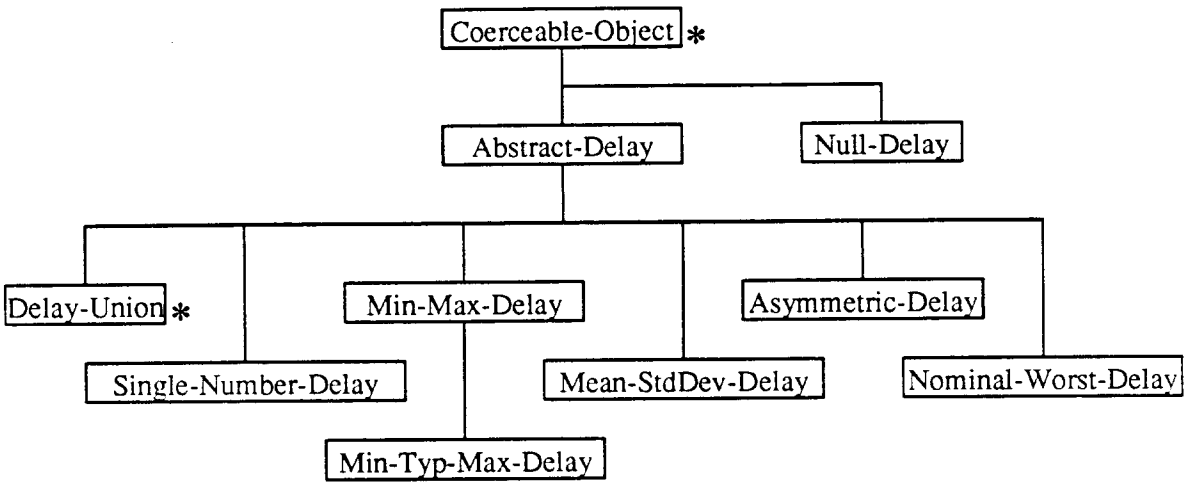

Figure A-7: Class Hierarchy for Delay Classes. Coercion methods are initially specified only for the coerceable-object and delay-union classes; other coercions must be explicitly loaded by the user. When a coercion is attempted on a delay, PCL looks up the class hierarchy from the class of the delay, looking for the first class that has a coercion method defined. A method can either succeed (by returning a delay of the desired class), fail (by returning a failure code), or invoke the next higher-level method. The default coercion method supplied by coerceable-object fails unless the source delay is already a member of the desired class. Thus when a min_typ_max delay is coerced, PCL looks first for a coercion method defined for min-typ-max-delay, then for min-max-delay, then for abstract-delay, and finally will invoke the coerceable-object method (which fails unless the target is one of the above classes) if no more specific method was found.

Here is the class definition for the min-max delay class:

```
(defclass min-max-delay (abstract-delay)
  ((min-delay :initform 0)
   (max-delay :initform 0))
  (:accessor-prefix ||)
  (:constructor make-min-max-delay)
  (:constructor mm-del (min-delay max-delay))
  (:constructor mmdelay (min-delay max-delay)))
```

The name of the class is **min-max-delay**; it has one immediate superclass, **abstract-delay**. This class is defined to have two slots, **min-delay** and **max-delay**. Each of these is defined with the **:initform** slot option that defines a default value for that slot. The class options for this class define *accessors*, functions that can be used to obtain or alter the value of a slot, and *constructors*, functions that are used to create a new object belonging to this class. The option (**:accessor-prefix ||**) requests an accessor function for each slot of the class with no prefix; this means that the name of each accessor function is just the name of the slot. (Note: the :accessor-prefix option is becoming obsolete in new versions of PCL, and will be replaced by the :accessor slot option.) The argument to :accessor-prefix serves the same purpose as the argument to :conc-name for structures [Ste84, p. 311]. Hence if d is a min-max delay, we can obtain the value of its min-delay slot by writing `(min-delay d)`, and set it to the value 5 by writing `(setf (min-delay d) 5)`. The above definition defines three constructor functions, called **make-min-max-delay**, **mm-del**, and **mmdelay**. Since make-min-max-delay is defined without an argument list, it takes keyword arguments to define initial slot values. Any slot who's value is not supplied gets the default :initform value. The other two constructors have positional argument lists. Each is a function of two arguments, with the first being the value of the min-delay slot, and the second being the value of the max-delay slot. (The two functions mm-del and mmdelay are completely equivalent; mmdelay is only defined for compatibility with old examples that used it to construct min-max delays.) The syntax of constructor functions is similar to that of constructors for structures [Ste84, pp. 311, 315-316], except that no constructors are defined by default for classes. As defined above, any of the following

three expressions would create a min-max delay with a min value of 3 and a max of 7.3:

```
(make-min-max-delay :min-delay 3 :max-delay 7.3)
(mm-del 3 7.3)
(mmdelay 3 7.3)
```

Here is the class definition for the min-typ-max delay class, which is based on min-max delay:

```
(defclass min-typ-max-delay (min-max-delay)
  ((typ-delay :initform 0))
  (:accessor-prefix ||)
  (:constructor make-min-typ-max-delay)
  (:constructor mtm-del (min-delay
                         typ-delay
                         max-delay)))
```

This class has min-max-delay as its immediate superclass, and thus inherits all the methods that are defined on min-max-delay. It defines one new slot, typ-delay. It also inherits the slots min-delay and max-delay from its superclass, along with their accessor functions. The :accessor-prefix form defines an accessor function for typ-delay; the :constructor forms define keyword constructor make-min-typ-max-delay and positional constructor mtm-del. Any min-typ-max delay is automatically a min-max delay, and thus any methods that specify a min-max delay automatically accept a min-typ-max delay in the same position (thus providing the inheritance behavior).

If you want to be able to simplify graphs with a given delay format, you need to define the **sum-delays** method on that delay class. Sum-delays should take two delays of a given class, and return a delay of the same class that has the same effect as applying the two delays in order (see Chapter 4, Section 4.5 "Graph Simplification," of *Abstract Timing Verification for Synchronous Digital Systems*, the companion to this report, for a full explanation and cautions about restrictions and necessary assumptions). Examples of sum-delays for the above classes:

```
(defmethod sum-delays ((d1 min-max-delay) (d2 min-max-delay))
  (mm-del (+ (min-delay d1) (min-delay d2))
          (+ (max-delay d1) (max-delay d2))))

(defmethod sum-delays ((d1 min-typ-max-delay) (d2 min-typ-max-delay))
  (mtm-del (+ (min-delay d1) (min-delay d2))
           (+ (typ-delay d1) (typ-delay d2))
           (+ (max-delay d1) (max-delay d2))))
```

If both arguments are min-max delays, the first method matches. If both arguments are min-typ-max delays, both methods match, but the second is more specific, so it is the one invoked. If one argument is a min-max delay and the other is a min-typ-max delay, then the first method automatically matches (and returns a min-max delay).

Note that defining sum-delays on a delay class is strictly *optional*; if you don't define it on some delay class, nothing bad happens except that simplifying any graph that includes such a delay will have no effect.

For versions of PCL after 3/21/87, you may want to define the **print-object** method on any new delay classes created so they will print out in a reasonable manner. See the examples of print-object in the file delays.lsp.

8.2. Defining Coercions Between Delay Classes

Although not required, you may wish to define coercions between one delay class and another, so that you can re-analyze the same input data with a different timing model. The method to be defined on the source delay class is **coerce-meth**, a method that takes a delay and the type it is to be coerced to. Eventually, PCL may let you say something like:

```
(defmethod coerce-meth ((d single-number-delay)(type 'min-max-delay))
```

to define a method to coerce from single-number-delays to min-max-delays. As of the 8/27/87 version of PCL, however, such methods selecting on specific items (the symbol 'min-max-delay) have not been implemented. Thus it is necessary to define a single method for each source delay class, and do case selection within the method based on the target type. Thus, a possible coercion method from single-number-delays is:

```
(defmethod coerce-meth ((d single-number-delay) type)
  ;; Coercions from single number format: min-max is +/- 30%; mean-stdDev
  ;; uses number for mean, 10% of num for std. dev. (thus min-max matches
  ;; 3-sigma points).
  (case type
    ((single-number-delay) d)
    ((min-max-delay) (mm-del (* 0.7 (num d))
                             (* 1.3 (num d))))
    ((mean-stdDev-delay) (ms-del (num d)
                                 (* 0.1 (num d))))
    (otherwise (pcl::call-next-method))))
```

Note that the default action (in the otherwise clause of the case) should be to call pcl::call-next-method, to let some superclass coercion method attempt the coercion when this method fails to be applicable. This will ultimately let the method for coerceable-object handle the case where no applicable coercion was found (since all delays are supposed to be direct or indirect subclasses of coerceable-object). That method handles this error condition based on the setting of the *silent-coerce* flag.

8.3. Defining New Event-Time Classes

Event-time classes define the various timing models in use. The general rule is that the class of the input event times determines the timing model to be used. Each timing model should define the following methods, or inherit them from some other model: **delay, translate, merge-meth, merge-with-slacks, satisfies-p,** and **satisfy-constraint.** For versions of PCL later than 3/21/87, you may also want to define the method **print-object** on the event-time class so that event times print in some readable form. The crucial methods that are currently used in the main ATV algorithm are delay, translate, merge-with-slacks, and satisfy-constraint. The other methods are for future use, or for direct use by the user; they can be omitted for now. The methods **compare** and **undelay**, defined for some of the built-in models, are now obsolete.

Every event-time class should be based on the class *abstract-event-time*, either directly or indirectly. That class defines default behavior for event times. To illustrate the definition of the various methods, I will use the min-max event time class.

The class definition is:

```
(defclass min-max-et (abstract-event-time)
  ((min-time :initform 0)
   (max-time :initform 0))
  (:accessor-prefix ||)
  (:constructor make-min-max-et)
  (:constructor mm-et (min-time max-time)))
```

A min-max-et has two slots, min-time and max-time, accessors with the same names as the slots, keyword constructor make-min-max-et, and a positional constructor mm-et.

The print-object method is defined as:

```
#-pcl-3-21
(defmethod print-object ((e min-max-et) stream depth)
  (declare (ignore depth))
  (format stream "(MM-ET ~A ~A)" (min-time e) (max-time e)))
```

The conditional read symbol at the beginning, #-pcl-3-21, makes sure that this definition is not read when using the 3/21/87 version of PCL (with which it is incompatible). Print-object takes three arguments: the object to be printed, the stream it is to be printed to, and the recursive depth at which it is being printed. Here I am ignoring the depth (to keep things simple), and using the Common Lisp **format** statement to print a min-max-et out in a form such as **(MM-ET 3 4.5)** (see [Ste84, pp. 385-407] for a thorough discussion of the format statement).

The delay method is defined as:

```
(defmethod delay ((e min-max-et) (d abstract-delay))
  (let ((d1 (coerce-meth d 'min-max-delay)))
       (make-min-max-et :min-time (+ (min-time e) (min-delay d1))
            :max-time (+ (max-time e) (max-delay d1)))))
```

The delay method takes an event time as its first argument, and a delay as its second argument, returning an event time that is the result of delaying the input event time by the delay. In general, delay methods should be defined to accept any abstract-delay as the second argument, and coerce it to the appropriate form, as shown here. The let form calls coerce-meth to coerce d to a min-max-delay, and binds the variable d1 to the result. The keyword constructor make-min-max-et is called to build the result event time, which is then returned as the result of the let form and of the method.

The translate method is defined as:

```
(defmethod translate ((e min-max-et) n)
  (declare (number n))
  (make-min-max-et :min-time (+ (min-time e) n)
                   :max-time (+ (max-time e) n)))
```

The translate operation takes an event-time as its first argument and a number n as the second, and returns the result of translating the input event time forwards or backwards in time by n.

The definitions of merge-meth is:

```
(defmethod merge-meth ((e min-max-et) l)
  ; Merge a list of several input events together to produce 1 event
  (make-min-max-et :min-time (apply #'min (min-time e) (mapcar #'min-time l))
                   :max-time (apply #'max (max-time e) (mapcar #'max-time l))))
```

The definition of merge-with-slacks is:

```
(defmethod merge-with-slacks ((e min-max-et) &rest l)
  ;; Merge a list of several input events together to produce 1 event; second
  ;; result is list of slacks.
  (let ((min-res (apply #'min (min-time e) (mapcar #'min-time l)))
    (max-res (apply #'max (max-time e) (mapcar #'max-time l))))
    (values (make-min-max-et :min-time min-res :max-time max-res)
        (if *long-path-search*
        (cons (- max-res (max-time e))
              (mapcar #'(lambda (e) (- max-res (max-time e)))
                  l))
          (cons (- (min-time e) min-res)
            (mapcar #'(lambda (e) (- (min-time e) min-res))
                l))))))
```

Both methods take an initial event-time that is used to select the method, and a (possibly empty) list of additional event-times that are to be merged with it. The additional event-times should be the same type as the initial event time, but this is not explicitly checked here. The calling sequence is a little different: merge-with-slacks takes all the input event-times as its arguments and collects all but the first into the &rest parameter l to form the list, while merge-meth is called by the function merge-et which has already collected all the additional event-times into a list. Merge-meth just returns the result of merging all the input event-times together; merge-with-slacks returns two values: the result of the merge, and an ordered list of slacks, one for each input event-time, that report the slack of each input event-time with respect to the merge. The **values** form is used to return multiple values; it returns one value for each of its arguments.

The global variable ***long-path-search*** sets the direction of the merge; it is t if the merge is for longest paths, nil if it is for shortest paths. The result of merging min-max event-times does not depend on the direction of the merge, but the slacks do. The slack of each input event time is the amount by

which it can be translated in the direction of the merge before the result of the merge changes "significantly." The definition of a significant change is model-dependent; for models such as min-max that have clear boundaries between no effect and some effect, any change should be considered significant. Other models may have to establish model-dependent thresholds to determine what change is significant. By definition, all slacks should be non-negative, and at least one should be zero.

The definition of the satisfies-p method is:

```
(defmethod satisfies-p ((e1 min-max-et)
                        test
                        (e2 min-max-et))
  (ecase test
    ((< <=) (funcall test (max-time e1) (min-time e2)))
    ((> >=) (funcall test (min-time e1) (max-time e2)))
    (= (and (= (min-time e1) (min-time e2))
            (= (max-time e1) (max-time e2))))
    ((INCOMP)
     ;; min-max-et's are only incomparable if there is no other condition
     ;; that covers both parts.  Leave out fuzz-eq for now; < should imply
     ;; <= and > should imply >=, so don't need to test for these.
     (notany #'(lambda (newtest) (satisfies-p e1 newtest e2))
             '(= <= >=)))        ; < and > covered by <= and >=

    ((fuzz-eq) (pcl::call-next-method))
    ;; fuzz-eq not implemented yet.
    ))
```

Satisfies-p takes two event-times and a condition, and asks if the condition holds between the two events. The possible conditions are: **<, <=, =, >=, >, incomp,** and **fuzz-eq**. The interpretations are:

= The equal relation means that the two event times are identical.

≃ The fuzzy equal relation means that the two event times are equivalent within some user-specified tolerance. It is used in the generic definition of critical inputs.

< $\{E_1\}<\{E_2\}$ means that an event occurring at event time $\{E_1\}$ is "guaranteed" to occur before an event occurring at event time $\{E_2\}$ (within the model-dependent interpretation of what "guaranteed" means). If $\{E_1<\{E_2\}$, then $\{E_1\}<\{E_2\}+t$, for all $t \geq 0$.

≤ $\{E_1\}\leq\{E_2\}$ means that an event occurring at event time $\{E_1\}$ is "guaranteed" to occur no later than an event occurring at event time $\{E_2\}$ (within the model-dependent interpretation of what "guaranteed" means). If $\{E_1\leq\{E_2\}$, then $\{E_1\}\leq\{E_2\}+t$, for all $t \geq 0$.

> $\{E_1\}>\{E_2\}$ means that an event occurring at event time $\{E_1\}$ is "guaranteed" to occur after an event occurring at event time $\{E_2\}$ (within the model-dependent interpretation of what "guaranteed" means). If $\{E_1>\{E_2\}$, then $\{E_1\}>\{E_2\}+t$, for all $t \leq 0$.

≥ $\{E_1\}\geq\{E_2\}$ means that an event occurring at event time $\{E_1\}$ is "guaranteed" to occur no sooner than an event occurring at event time $\{E_2\}$ (within the model-dependent interpretation of what "guaranteed" means). If $\{E_1\geq\{E_2\}$, then $\{E_1\}\geq\{E_2\}+t$, for all $t \leq 0$.

Incomparable

Two event times are *incomparable* if no more specific relation holds between them.

Note that the fuzz-eq relation is the one used for defining slacks; this relation has generally been left unimplemented for the standard models, as the program currently relies on each model providing merge-with-slacks rather than having to compute the slacks from scratch. Satisfies-p is the replacement for the compare operation, which returned a single value to indicate the relation between two event times.

The code for the satisfy-constraint method is:

```
(defmethod satisfy-constraint ((e1 min-max-et)
                               (e2 min-max-et))
  "Return smallest t such that e1 <= e2 + t"
  (- (max-time e1) (min-time e2)))
```

As the documentation string says, satisfy-constraint takes two event-times and returns the smallest number t such that e1 <= e2 translated by t.

REFERENCES

[BKK86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik and F. Zdybel, "CommonLoops: Merging Lisp and Object-Oriented Programming," *OOPSLA '86*, Portland, Oregon, Sept. 29 - Oct. 2, 1986, 17-29. Special Issue of SIGPLAN Notices Notices, Vol. 21, No. 11, November, 1986.

[Ste84] G. L. Steele, Jr., *Common LISP: the Language*, Digital Press, 1984.