ATV: An Abstract Timing Verifier

David E. Wallace and Carlo H. Séquin

Computer Science Division / Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

ABSTRACT

We discuss implementation and extensions of the Abstract Timing Verifier proposed in [WaS86], in particular, several new operations and a new algorithm for analyzing critical paths that extend through transparent latches and stretch over multiple machine cycles. By placing events in different *reference frames* that can be translated relative to one another, the program can be used either to check a design for timing errors when the clock schedule is fixed and known, or to derive spacing constraints between clock edges when only the relative ordering of the clock edges is known. The algorithms are designed to operate on a wide variety of representations of time and delay.

1. Introduction

At the 1986 Design Automation Conference, we demonstrated that it is possible, in principle, to build an Abstract Timing Verifier (ATV) that does not have fixed, built-in representations of time and delays [WaS86]. Instead, different models for event times and delays can be plugged into the framework provided by ATV, depending on the type of circuit and the goal of the analysis. We compared several different timing models and discussed the abstract operations that such a program has to carry out which are common to each model. In reducing this concept to practice, the timing model has been refined and some new operations added. Here, we report on our prototype implementation and on some early applications of our Abstract Timing Verifier, called ATV.

2. ATV Concepts

The Abstract Timing Verifier 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 [Ous85], 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 later rerun using ranges (min-max) or statistical descriptions (mean, standard deviation) to represent signal arrival times. It is possible for the sophisticated user to extend the program by developing new models to meet special needs.

2.1. Dependency Graph

Timing verification is primarily concerned with analyzing the stable/changing behavior of most signals, rather than looking at specific data values. Although similar to simulation, in that both use a scheduler to propagate events through a network, timing verification focuses on *when* nodes may change, rather than *what* specific values they take on, making generally conservative assumptions about when specific changes may occur. ATV does not run directly on a low-level description of the circuit (such as an extracted transistor list), but takes its input in the form of a dependency graph, where each individual delay is attached to an arc. At present, there are two possible sources for such a graph. The user can generate such a graph by hand, which may be the preferred mode when ATV is used as a tool in planning a microarchitecture. Alternatively, the graph can be generated automatically from an OCT [HMS86] symbolic description of a circuit that includes timing descriptions for its low-level blocks. A dependency graph and the circuit it represents are shown in Figure 1. This is a directed graph, where *nodes* correspond to terminals and other points where signals can be observed, and *arcs* correspond to circuit elements and interconnects with their associated delays.

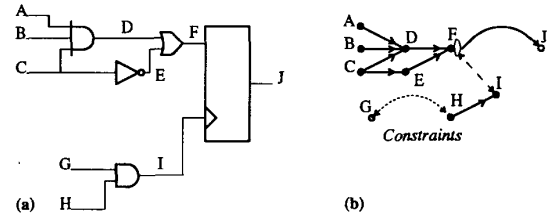

Figure 1: (a) Logic diagram. (b) Corresponding dependency graph.

Events occur at the nodes of the graph. An event includes 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: ST-CH (stable->changing), CH-ST (changing->stable), RISE (rising), and FALL (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.

Arcs run from a source node to all the destination nodes that are directly affected by an event at the source node. Arcs come in two flavors: *ordinary arcs*, that correspond to pure combinational elements and always transmit data, and *gated arcs*. The latter correspond to sequential elements, controlled by some *clock node*, and only transmit data when the clock node is in the appropriate state (see Figure 2). Transparent latches are modeled with gated arcs that open and close on two different edges at the clock node (e.g., open on RISE, close on FALL); edge-triggered registers are modeled with gated arcs that open and close on the same edge (e.g., the rising

edge). All types of arcs have *delays* attached to them, which specify how an event-time at the source node is transformed into the resulting event-time at the destination node. The format used to represent delay is defined by the particular timing model used. 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 possess an *arc type*, which can take one of three values: *inverting, non-inverting*, or *unknown*. For boolean gates, AND and OR gates would use non-inverting arcs, NOT, NAND and NOR would use inverting arcs, and XOR would use unknown arcs.

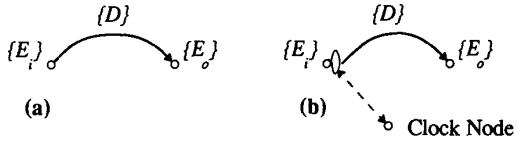

Figure 2: (a) Ordinary arc. (b) Gated arc.

Clocks are distributed over ordinary arcs with known inversion properties (i.e., no *unknown* arcs) from user-specified *clock origin nodes*. Clock enable signals are not connected to their respective clocks with arcs, but rather add constraints that will require the enable signal to be stable when the clock is active (e.g., signal G in Figure 1). When all clock gating is done at the latches, as is true for designs coming from the Berkely Synthesis System, this handling of enable signals is done when developing the library latch models. A typical timing graph is shown in Figure 3.

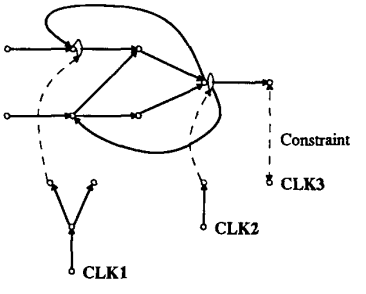

Figure 3: Typical timing graph.

2.2. Reference Frames

There are several types of analysis possible with ATV. The piece of the timing graph being analyzed may involve purely combinational logic, or it may include sequential elements; in the latter case, the clock schedule may be known or unknown initially. To support analyses where the clock schedule is not known in advance, ATV supports multiple *reference frames*. Every input event to the network is specified in some reference frame. 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. If the clock schedule and all input events are fixed, all events will be specified in the same reference frame, and the program will report any resulting timing violations. If only some events are known relative to each other, e.g., the widths of the clock phases are known but not their phase relationships, then the rising and falling edge of a clock are specified in the same reference frame, but other clocks will be specified in different reference frames. If only the relative clock ordering is known, then every clock edge will be specified in its own reference frame, with any convenient origin. Whenever there are multiple reference frames, the program reports any violations between events occurring in the same reference frame, and a set of spacing constraints between the origins of different reference frames required to avoid violations between events in those frames.

The sequence of clock edges is initially specified in the form of a *clock graph*, a (generally cyclic) directed graph with a node for each clock edge in some user-defined *machine cycle* and directed arcs from each clock edge to its direct successors. Multiple instances of the same clock edge are allowed in this graph (to represent clocks with a higher frequency than the basic machine cycle), and clock edges can be defined as *aliases* for other clock edges to represent clock edges that ideally occur simultaneously. In setting up for the analysis, this machine cycle is repeated a user-specified number of times to form the *unrolled graph*, a directed acyclic graph with a distinct node for every clock edge in every machine cycle. This graph defines a partial order on individual clock events which will be observed by the program during the analysis. Figure 4 shows the clock graph and unrolled graph for a simple 2-phase non-overlapped clock sequence. Because the clock graph is a directed graph, it can represent more complex clocking schemes involving overlapping clocks and partially asynchronous clock edges.

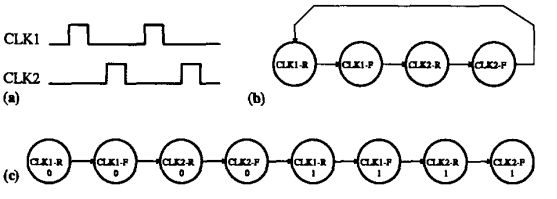

Figure 4: (a) Clock Schedule; (b) Corresponding Clock Graph; (c) Unrolled Graph for Two Machine Cycles.

2.3. 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. We described several different timing models in a previous paper [WaS86]. The timing models supplied with the current version of ATV are:

(1): Single number (delays and event-times are single numbers),

(2): Min-max (delays and event-times are (min, max) pairs),

(3): Simple mean, standard-deviation (delays and event-times are expressed as means and standard deviations),

(4): Asymmetric rise-fall.

In the asymmetric rise-fall model, event-times are specified as a rising time and a falling time, while delays are specified as a rising delay and a falling delay plus a delay type (which is like the arc-type defined above). These component event-times and delays may be selected by the user from any other timing model, whether this model is built-in (single-number, min-max, or simple mean-standard deviation) or is a user-supplied extension. Any new timing model

created is automatically supported in an asymmetric rise-fall version. Other possible timing models include: mean & standard-deviation models with more realistic merge operations or variable correlations between delays, the weighted two-point distribution proposed by Steck *et. al.* [GSS86], arbitrary probability distributions, and time-slope models. These are discussed in [Wal88].

2.4. Fundamental Operations and Critical Paths

Every timing model defines a minimum of four operations that apply to event-times in that model. These are:

(1) *translating* an event-time forwards or backwards in time by some real number,

(2) *delaying* an event-time by some delay along an arc of the timing graph,

(3) *merging* multiple event-times at a node to produce a "worst-case" event-time at that node (when there are multiple inputs to the node),

(4) *comparing* two event-times.

The last three operations were defined in [WaS86], the translation operation is new. It takes an event-time $\{E\}$ and shifts it in time by some real number t, producing a new event-time $\{E\}+t$. Unlike the delay operation, it does not transform the internal structure of the event-time, and is always invertable (translating by $-t$). Operations (2-4) are all time-invariant with respect to the translation operation: if the inputs to a series of delays, merges, and comparisons are all translated by the same amount, t, the result will be the same as performing the operations on the untranslated inputs and then translating any output results by t. No such assertions can be made about the generic delay operation: there exist reasonable examples of timing models (e.g., most probabilistic models) for which these properties do not hold for the delay operation.

These properties make the translation operation the key to generating spacing constraints between events whose exact relationship is unknown. Input events can be grouped together with all their consequent events into reference frames of unspecified origins, which are to be translated relative to one another. Because of the time-invariance of the delay, merge, and comparison operations, all events in a reference frame will be rigidly translated together, so that any local constraints on translations between two events in different reference frames will apply equally to all the events in those respective reference frames (see Figure 5). The translation operation is also essential to defining critical paths and slacks in the general case.

2.5. Model Coercions

The general rule for timing models is that the type of the event-times entered by the user determines the timing model to be used. Thus the delay operation for a given type of event-time will attempt to coerce whatever delay information is available into the appropriate form. There may be more than one delay format specified for a given arc. The general idea is to allow the delay to be specified in its natural form, and to use coercions to obtain the format needed by a particular timing model.

Although this general rule applies for almost all delays, it is also possible to define special types of delay that override the normal delay operation for any event time. In ATV, one such special delay is called a *null-delay*, which always passes the input event time through unchanged, bypassing the normal delay operation for the

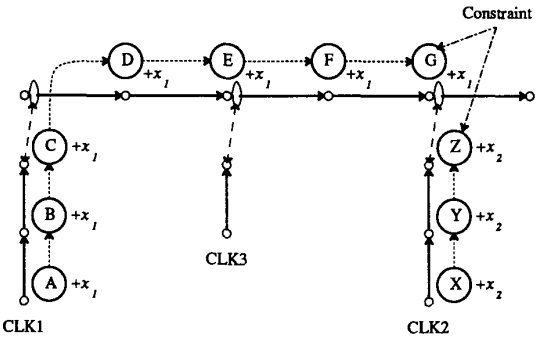

Figure 5: Constraint analysis. A constraint exists between events G and Z, which are in two different reference frames with translations x_1 and x_2, respectively. Any bound on the difference of these two translations applies to the translations of the original input events, A and X.

event time. This idea can be extended to allow for coercion between different types of event-times, by defining special delays that will perform the coercions when they are involved in a delay operation. This could be used to verify a sub-circuit using a different timing model from the one used in the surrounding circuit, by making all ingoing arcs to the sub-circuit coerce the input event-times to the model used in the sub-circuit, and having all outgoing arcs from the sub-circuit perform the inverse coercion. For example, a designer might want to verify a MOS chip using a different timing model from the one used for the surrounding TTL circuitry on a board.

2.6. Compound Operations

In addition to the basic operations defined above, compound operations have been implemented in each model to improve efficiency. Some of these are necessary operations that could, in principle, be synthesized out of the basic operations. Others represent optional optimizations that do not affect the correctness of the results, but can improve the efficiency of the program.

Satisfy_constraint takes two event-times, $\{E_1\}$ and $\{E_2\}$, and returns the minimum translation amount t such that $\{E_1\} \le \{E_2\}+t$ (see Figure 6a). The semantics of $\le$ then guarantee that $\{E_1\} \le \{E_2\}+s$, for any $s \ge t$. *Merge_with_slacks* performs a merge on its input event-times, and also returns the *local slack* for each input. The local slack of an input is the amount by which it could be translated in the direction of the merge (positively for long-path merges, negatively for short-path merges) before it becomes critical to its successor (see Figure 6b). In principle, satisfy_constraint could be synthesized out of the translation and comparison operations, and merge_with_slacks out of the merge, translation, and comparison operations.

2.7. Graph Simplification

Because the time required for most of the analysis depends on the size of the graph, simplifying the graph can result in a significant speedup. Graph simplification is an optional step based on an operation, *sum_delays*, that takes two delays and tries to compute a delay that will have the same effect as applying each of the input delays in series, based on an assumed default timing model for the particular delay format. Graph simplification looks for non-essential nodes with a single, unclocked fan-out (Fig. 7a). If such a node is found,

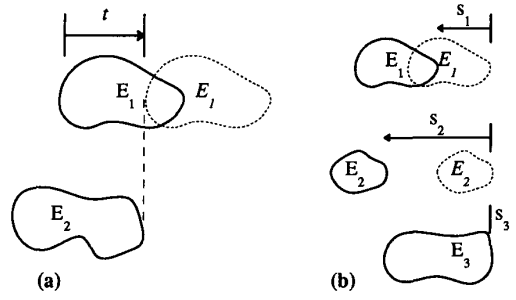

Figure 6: (a): Satisfy_constraint; (b): Merge_with_slacks

and the fan-out delay can be summed with all the input delays (via sum_delays), the node and its fan-out arc can be removed from the graph, with the fan-in arcs extended to meet the destination of the original fan-out arc. These extended arcs are renamed to reflect the arcs that have been incorporated into them, so that reported paths continue to be meaningful in the original input domain. Iterating over all nodes of the original graph can produce some dramatic reductions in the size of the graph and in the length of the reported critical paths.

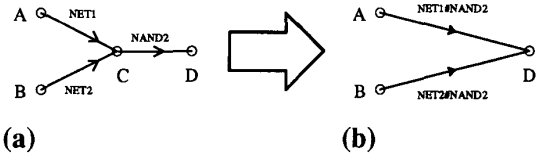

Figure 7: Graph simplification: (a): Before; (b): After.

This form of graph simplification has its limitations: only certain models (such as single-number and min-max) can define sum_delays in a manner guaranteed to produce the same results as the original graph in all cases. Two necessary conditions for this are that the combined delay must always exist in closed form, and the delay operation must distribute over the merge operation. The mean-standard-deviation model and the [time, rise-time] model described in [WaS86] fail this second condition, and hence do not define sum_delays. Care must also be taken when simplifying a graph to be used with timing models other than the assumed default.

3. Timing Verification and Transparent Latches

Transparent latches complicate the timing verification of designs that include them. Edge-triggered or master-slave storage elements interrupt the flow of data through them such that critical paths will never extend through them. If all the storage elements in a design are of this type, we only need to look at paths through the combinational logic between storage elements. But with transparent latches, critical paths can potentially extend over multiple cycles, passing through several intermediate latches at any time during their respective open periods.

Previous timing verifiers have attempted to address this problem in various ways. Crystal [Ous85] operates one clock phase at a time, relying on the user to specify how available time should be split between phases. Jouppi's TV program [Jou84] relies on "cycle borrowing" to shift available time between consecutive clock phases. Glesner *et al.*[GSS86] describe an algebraic approach that can analyze paths extending over multiple machine cycles, assuming that all clock phases have the same width. LEADOUT [Szy86] creates a series of equations relating all events in the networks by their respective delays derived from a dependency graph between events (not nodes, as in ATV); no procedure is given for solving these equations when the initial clock schedule is unknown.

In contrast, ATV examines all paths up to a user-specified number of machine cycles in length, allows clock phases to have non-uniform widths, allows for more complex clocking schemes than just p non-overlapped clock phases per machine cycle, and allows all, some, or none of the original clock schedule to be prespecified by the user.

Events in ATV include:

1) A signal_edge (RISE, FALL, CH-ST, or ST-CH),
2) an event_time (with model dependent values),
3) a reference_frame, and
4) the last_clock_edge known to have occurred.

The basic idea of the clock-phase length analysis is that the user will initially specify each clock edge whose position is to be calculated in its own reference frame (with arbitrary origin). Clock edges whose positions are known relative to each other will share a reference frame. There is some set of translations:

$$\mathbf{X}=(x_1, x_2, \cdots, x_n)$$

from each of these reference frames to the underlying absolute reference frame. ATV computes bounds on the differences between pairs of translations:

$$x_i - x_j \geq b_{ij}$$

Such bounds could subsequently be solved as a linear program together with any other required constraints, to solve for the actual set of translations.

Consider the circuit shown in Figure 8, where L is a transparent latch, and the other storage elements are edge-triggered registers. For simplicity, assume we are using the single-number model with delays as indicated, and that all setup and hold times are zero. Initially we assume that each clock edge occurs in its own reference frame at time 0. In preprocessing, the program traces forward from each clock origin node to determine which clock origin controls each gated arc and which phase of that clock opens the arc (by counting inversions along the clock distribution network). Thus it will mark arc B-C as controlled by CLK1, opening on its falling edge. Because it is important to be able to identify the controlling phase of each gated arc, special rules apply to the clock distribution network. Clocks must be distributed over pure fan-out trees with known inversion properties (i.e., no *unknown* arcs) from the clock origin nodes. Also, clock enable signals are handled specially as described earlier. These requirements are reasonable for most design styles, except for clock generation circuitry itself, which must be handled separately.

ATV proceeds in a series of phases, one for each clock edge in the unrolled clock graph, in an order compatible with the partial order defined by this graph. In each phase it initiates events starting in this phase and propagates them through the network together with any other events waiting for this phase to occur before they can proceed. Each phase begins with Jouppi's predecessor counting [Jou84], a quick depth-first search to determine how many predeces-

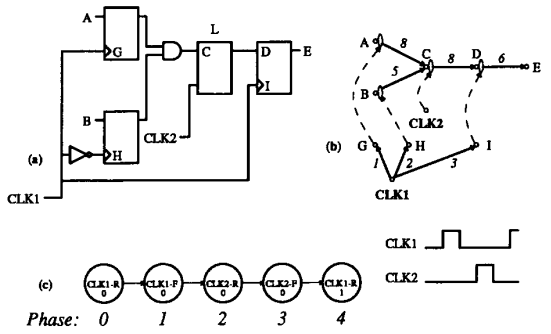

Figure 8: Transparent latch example. (a): Logic Diagram (L is the transparent latch), (b): Timing Graph, (c): Clock Schedule.

sors of each node are active in the current phase. Thus in phase 0 (CLK1 rising), predecessor counting will note that node C has only one active predecessor (node G); thus it can be evaluated as soon as node G has been evaluated. Each node is then evaluated in a breadth-first manner as soon as all of its predecessors have been evaluated. The last_clock_edge of each event is equal to the phase in which it is calculated.

When a clock event reaches a clock node that gates some arc, it will initiate both a ST-CH and a CH-ST event on that arc with the same event-time as the arrival at the clock node. This represents the possibility of a (brief) transition at the opening clock edge if the input of the arc has previously reached a value different from the output. Clock events after the first machine cycle are typically disabled from initiating such events, as no new violations or constraints will be discovered unless the user is performing *case analysis* (selectively disabling impossible paths) in the first cycle. Events with the same last_clock_edge, edge_type, and reference_frame at a given node are merged together (CH-ST events merge for longest paths, ST-CH events merge for shortest paths). When events arrive at the input to a transparent-latch arc (i.e., with different opening and closing edges) that is open in the current phase, they propagate through the arc. If the arc is not currently open, they wait for the next phase that opens this arc, and update their last_clock_edges as they pass through. Relevant events calculated for the first five phases of the example from Figure 8 are shown in Table I below.

After all events have propagated through the network, ATV examines all events that are involved in a possible constraint. Constraints specify two events to be checked, an optional translation to be applied to the second event, and a condition ($\leq$, $\geq$) that is expected to hold between them. In the above example, there are two relevant constraints: the data signals at C must stabilize before the closing edge of CLK2 occurs (CLK2 falling), and the data at node D must stabilize before the next closing event at node I (I rising). The last_clock_edges (phases) of the events at the two nodes are used to screen out incorrect comparisons: the default rule is that the last_clock_edges of events in a valid comparison must be in the same relation as the condition being checked. Thus the rising event at node I in phase 0 is not checked against the CH-ST events at node D in phase 2, because $0 < 2$. If the constraint is specified as:

$$E_i \leq E_j + \Delta,$$

Table I: Selected Events from Transparent Latch Example

Phase	Node	Edge	Event-Time	Reference Frame	From
0	CLK1	R	0	0	-
	I	R	3	0	CLK1
	H	F	2	0	CLK1
	G	R	1	0	CLK1
	C	CH-ST	9	0	G
	C	ST-CH	9	0	G
1	CLK1	F	0	1	-
	I	F	3	1	CLK1
	H	R	2	1	CLK1
	G	F	1	1	CLK1
	C	CH-ST	7	1	H
	C	ST-CH	7	1	H
2	CLK2	R	0	2	-
	D	CH-ST	8	2	CLK2
	D	ST-CH	8	2	CLK2
	D	CH-ST	17	0	C
	D	ST-CH	17	0	C
	D	ST-CH	15	1	C
	D	CH-ST	15	1	C
3	CLK2	F	0	3	-
4	CLK1	R	0	4	-
	I	R	3	4	CLK1
	H	F	2	4	CLK1
	G	R	1	4	CLK1

for some translation Δ, then E_i and $E_j+\Delta$ are passed as arguments to the *satisfy_constraint* routine. This will provide the minimum t such that

$$E_i \leq (E_j+\Delta)+t,$$

which produces a bound on the translation difference between the two reference frames:

$$x_j - x_i \geq t. \qquad \text{(I)}$$

Recall that because of the time-invariance of the delay, merge, and compare operations, this same bound applies to the allowable translation of the source events in each reference frame (see Figure 5).

From the two events involved in a constraint, we can derive the two critical paths (one short, one long) that produced this bound, by tracing backwards and looking for the preceding events with smallest slacks. After generating all the $x_j - x_i$ bounds, the largest t is the overall bound. If $i = j$, the left hand side of (I) is necessarily zero, so $t > 0$ indicates an error.

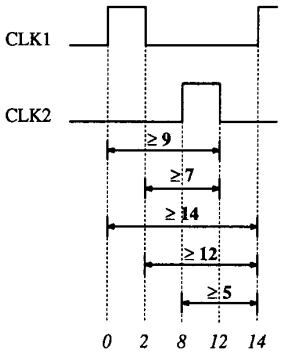

Figure 9: Constraints between reference frames for the transparent latch example. Sample translation times obeying constraints are shown.

Constraints for the reference frames in our example are shown in Figure 9, along with possible translations obeying these constraints with a minimum cycle time of 14. If the user desires, these translations could be applied to the original input event times (yielding event times equal to the translations, since the original event times were all zero), and the analysis could be rerun with all events now in the same (absolute) reference frame. This would yield results similar to those shown in Table I with the appropriate translations, but now many of the events in the same phase can be merged together, since they are now in the same reference frame. This would allow the user to calculate times for all events in the absolute reference frame.

4. Evaluation

Our ATV prototype is implemented in Portable Common Loops (PCL) [BKK86], an object-oriented extension to Common Lisp [Ste84]. We used PCL because its object-oriented facilities are useful in defining generic models such as the asymmetric-rise-fall model, which can be used with components that are user-defined timing models, not even defined at the time when the asymmetric-rise-fall code was written. Writing in PCL allows the user to create and link in additional timing models without changing the existing ATV code; PCL offers the additional advantage over some other object-oriented systems of supporting methods (functions) that discriminate on multiple arguments, useful for enabling delays with special properties to be created and linked with the existing system.

ATV consists of approximately 4300 lines of Lisp code, including blanks and comments. The largest example it has been run on was a chip implementing the Data Encryption Standard, containing about 20,000 transistors. This chip was synthesized using standard cell components from the Mississippi State University CMOS double-layer metal library. The timing graph before simplification for this chip had 6982 nodes and 9533 arcs. After simplification, the graph had 3016 nodes and 5567 arcs. Component delays were taken from the data sheets supplied with the library. These delays were specified as nominal and worst case delays for both rising and falling outputs. The delays were entered in this form and then coerced to get single numbers (using the nominal delay values), min-max, and mean-standard-deviation delays. All three models identified the same set of constraints as critical for the length of the machine cycle. As expected, the single number model was most optimistic (yielding a critical constraint of 240ns), the min-max model was most pessimistic (530ns), and the mean-standard-deviation model was in between (320ns).

ATV in its current implementation is more flexible, but also slower and more memory intensive than other timing verifiers such as Crystal. Much of this overhead is associated with the underlying PCL system, and will improve as the underlying system matures. The current implementation of ATV is best viewed as a research prototype, good for developing new timing models and trying them out, and applications where flexibility and the ability to use different timing models in the same framework are more important than raw speed.

5. Summary

ATV generalizes timing verification in two respects. First, it decouples the mechanism of timing verification from any specific representations of time and delay; different models can be plugged into a single framework to suit the available data or the needs of the analysis. Second, the ability to handle more sophisticated clocking schemes has been greatly expanded.

We have implemented an abstract timing verifier in the Portable Common Loops extension to Common Lisp. Delays can be specified in the most natural format and coerced into the appropriate form for a particular analysis. The program can be used either to verify a proposed clock schedule, or to derive required clock-phase lengths from constraints that may extend over multiple machine cycles for designs that use transparent latches. The clock-phases need not all have uniform lengths. Critical paths can be computed for each constraint, identifying areas for potential speedups. ATV is also being used as a high-level analysis tool to study microarchitectural tradeoffs in microprocessor design.

6. Acknowledgements

This research was sponsored by the Defense Advance Research Projects Agency (DoD), monitored by Electronic Naval Systems Command, under Contract No. N00039-87-C-0182.

The reviewers and the Program Committee made many helpful comments on the original version of this paper. The example of Figure 8 was based on examples suggested by Tim Miller of Mentor Graphics Corporation.

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.

[GSS86] M. Glesner, J. Schuck and R. B. Steck, SCAT - A New Statistical Timing Verifier in a Silicon Compiler System, *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, Las Vegas, NV, June, 1986, 220-226.

[HMS86] D. S. Harrison, P. Moore, R. L. Spickelmier and A. R. Newton, Data Management and Graphics Editing in the Berkeley Design Environment, *ICCAD-86*, Santa Clara, CA, November 11-13, 1986, 24-27.

[Jou84] N. P. Jouppi, *Timing Verification and Performance Improvement of MOS VLSI Designs*, PhD. Thesis, Department of Electrical Engineering, Stanford University, Palo Alto, CA, June 1984.

[Ous85] J. K. Ousterhout, A Switch-Level Timing Verifier for Digital MOS VLSI, *ToCAD CAD-4*, 3 (July 1985), 336-349.

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

[Szy86] T. G. Szymanski, LEADOUT: A Static Timing Analyzer for MOS Circuits, *ICCAD-86*, Santa Clara, California, November, 1986, 130-133.

[WaS86] D. E. Wallace and C. H. Séquin, Plug-in Timing Models for an Abstract Timing Verifier, *Proceedings of the 23rd IEEE/ACM Design Automation Conference*, Las Vegas, NV, June, 1986, 683-689.

[Wal88] D. E. Wallace, *Abstract Timing Verification for Synchronous Digital Systems*, University of California, Berkeley, Berkeley, CA, 1988. PhD. Dissertation (in preparation).