CLOVER: A TIMING CONSTRAINTS VERIFICATION SYSTEM

Dimitris Doukas
Andrea S. LaPaugh

CS-TR-274-90

July 1990

Appendix

Clover: A Timing Constraints Verification System

Dimitris Doukas
Andrea S. LaPaugh

CS-TR-274-90

July 1990

CLOVER: A Timing Constraints Verification System

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

July 23, 1990

♣

Abstract

Correct timing is a critical issue in hardware design, especially in the case of bus interfaces. In this paper we present the design and implementation of a timing verification tool, CLOVER. CLOVER provides the designer of a digital circuit with an integrated environment where he can describe his design, formally specify the timing constraints governing the implementation, obtain the timing behavior of the implemented design and verify it against the stated timing constraints.

Our timing constraints specification language, which is used to formally express timing constraints, is a general language but has been designed particularly for the description of asynchronous designs. The language is based on dependency graphs and vectors of signal values over the time. We give specifications for well known timing interface problems to illustrate the expressive power of our language. Finally, we specify and verify an implementation of the multibus design frame.

1 Introduction

CLOVER is a computer-aided design (CAD) tool, which provides the designer of a digital circuit with an integrated environment where he can describe his design, state the timing constraints governing the design, obtain the timing behavior of the described design and verify it against the stated timing constraints, being informed of possible violations. It is a system oriented towards the description and timing analysis of medium size asynchronous systems, where the basic components can be as simple as a logic gate or as complicated as a complex functional module. This does not mean that synchronous designs cannot be analyzed. However, parts of the tool, for example the timing constraints specification language, are designed to best suit the description of asynchronous designs. Much more expressive power is needed to describe timing constraints for an asynchronous design than for a synchronous one. Of particular interest are the description and analysis of interface circuits ([18], [17]) where more complicated timing relations exist between signals.

The main contributions of this research are:

- A new specification model to express both simple and complicated timing constraints. The model is particularly suited to the description of the temporal behavior of interface circuits.

- A verification methodology of timing constraints based on the concept of *event graphs*. Event graphs are derived from an implemented design using event-driven time simulating techniques.

The basic component of CLOVER is the verifier. It compares the intended timing behavior of a design against its implemented behavior, and identifies possible violations. The timing behavior is described in our specification language ATCSL, which is based on dependency graphs and vectors of signal values over time. It provides the expressive power needed to describe complex timing relations between signals. We have built an event driven timing simulator based on an extended value system, which takes as input the netlist of the implemented circuit and derives the timing behavior of the implementation, in the form of an event graph. To derive the netlist of the circuit, the design is described with PDL-e, a hardware description language made for ease of use and to support hierarchical design. PDL-e is based on PDL [23], a generator language developed at Princeton for register transfer design.

The paper is organized as follows. In Section 2 a framework for characterizing a verification system is presented and existing verification systems are discussed. In Section 3 we present our formal specification model and the constraints language ATCSL, used to express timing constraints within the model. Section 4 describes the event driven timing simulator and in Section 5 the verifier is presented. In Section 6 we study an example, where we specify and verify an interface circuit using the methodology introduced in Sections 3 through 5. Finally Section 7 presents conclusions and thoughts for future research.

2 Circuit Verification and Verification Systems

Timing verification is viewed as a subcase of a more general problem, that of hardware design verification. During verification both the functional and the temporal behavior of the hardware design is verified. It is very possible for a temporal violation to cause a functional error; the inverse is also true. Consequently the verification systems presented here perform timing and/or functional verification.

2.1 A Framework for Characterizing a Verification System

We have been talking about verification and timing verification assuming that a well established notion for these concepts exists. However our experience suggests the contrary. A common question asked when someone is talking about verification is: what do you mean by verification? In this section, we present a framework for the characterization of a verification system. This framework will be used as a guide to reason about research work in hardware design verification.

Designs are built based on specifications which determine the designs' intended behaviors. Informally we can define verification as follows:

Definition 2.1 Verification *is the process of validating that an implemented design behaves according to its specifications.*

It follows from the definition that a verification system should provide a specification model to specify the intended behavior of a design. The specification model is the heart of the verification system. The way we specify the behavior of a design bounds the outcome of the verification system: we cannot verify more than we specify. The specification model we use should be formally structured and able to capture and express our whole understanding of the design's behavior.

The specification of a design should be compared with the behavior of its implementation. That brings us to the second major requirement of a verification system: to obtain the behavior of a design's implementation. The behavior of the implementation is dependent on the design's inputs and states. Assuming a discrete set of input values, a complete description of it should account for every possible combination of input values and states. Accounting for all the possible combinations, we say that a *complete behavior* of the implementation is obtained. Since both the space to store the complete behavior of a large design and the time to verify it can be very large, it may be reasonable for a verification system to characterize the behavior of a design's implementation using a subset of its complete behavior. We say then, that an *incomplete behavior* of the implementation is obtained. Depending on the verification methodology, a complete or incomplete behavior of a design can be used for the verification process. We define the concepts of complete and incomplete verification methodologies as follows:

Definition 2.2 *A complete verification method will verify a design's specification against its complete behavior.*

Definition 2.3 *An incomplete verification method will verify a design's specification against an incomplete behavior of it.*

The next issue we have to address is how we model the behavior of the components which constitute the design and the interactions between them. These components can range in complexity from individual sub-systems to simple gates or transistors. Every transaction of the design is at the very end a physical process. How closely the behavioral model of the verification method matches the physical behavioral characteristics of the design measures the accuracy of the method. For purpose of efficiency, we often use behavioral models with less accuracy, e.g. digital models. A complete verification method does not imply a certain accuracy nor does a given accuracy imply completeness. Since we use approximations, the verifier may generate spurious errors.

Definition 2.4 *A verification system which permits the report of erroneous errors but never fails to report existing violations is called* **pessimistic**.

Finally for a verification system to be really useful it should not only identify a violation but also help the designer locate the possible causes of it.

Our framework for the characterization of a verification system, is summarized as follows:

Specification model: The model used by the verification system to specify the intended behavior of a design.

Completeness of the Methodology: Characterizing the verification methodology of the system as being complete or incomplete.

Accuracy of the Methodology: How accurate the behavioral model is.

Error report: How elaborate the error reporting is, for identifying and locating a violation.

In choosing a verification methodology there are certain trade-offs to consider. Most of them have to do with completeness and accuracy of the method versus efficiency.

2.2 Verification Systems for Hardware Designs

We classify existing verification systems into five classes: simulators, timing analyzers, formal verifiers, design audit and incomplete verifiers (CLOVER is a member of this class). Among the five classes there are systems which share a common characteristic: they are concerned only with the verification of the temporal behavior of a design. We refer to such systems as *timing verifiers.*

Simulators ([32], [8]) are not verifiers in the real sense. The behavior of the implementation is obtained using an incomplete methodology but there is no comparison with some specified behavior. There is no specification model. In order for a simulator to completely verify a set of timing constraints, the circuit should be exhaustively tested, which usually means an exponential number of circuit inputs.

Timing Analyzers like simulators, are not considered true verifiers since they do not support a specification model. The verification methodology remains incomplete. The difference between analyzers and simulators is that analyzers work on a circuit without relying on specific signal values ([16]). Constraints related to the critical path of a circuit and the optimum cycle time for synchronous implementations can be satisfied without use of specific signal values or knowledge about the functionality of the circuit. This absence of knowledge may lead in some cases to pessimistic or incorrect answers (false-paths [26]).

Formal verification is complete. The behavior and the constraints for each individual component of a circuit are formally specified. Verification of the circuit is then accomplished by composing the components' specifications under formal rules, without relying on specific input values. Verification proofs can be quite tedious, unless the design is small. On the other hand, they have the mathematical power to completely verify a design. VERIFY [3] and Silica Pithicus [40] are two well known MOS formal functional verifiers.

We are particularly interested in a subclass of formal verifiers, the formal timing verifiers. Formal timing verifiers, like the ones presented in [4], [13] and [33], use the concept of the *state graph* as the foundation upon which designs are formally verified. A state graph covers transitions between states, for all the possible combinations of input values. No assumptions about initial inputs have to be made. As designs increase in size, their state graphs increase exponentially. A formal verification framework is presented in [14]. The framework defines an axiomatic theory that describes relevant properties of arithmetic, time, waveforms and structures. The use of hierarchy greatly facilitates the verification of complex systems. A bottom-up hierarchical formal verifier is presented in [29]. The timing properties of complex circuit modules are established by formally composing the timing properties of their constituent parts.

Design audit is an attempt to make circuit designers follow some standard design guidelines in order to minimize the chances of making a design error. These guidelines are expressed in some form of constraints. The idea can be applied not only to timing constraints but to functional and physical constraints as well. What differentiates design audit systems from other verifiers is the effort to design systems according to predetermined rules rather than verifying the rules after the systems have been built ([37], [22]). These rules are either user specified or are collected by the system as experienced knowledge after a verification process. Some design audit systems, starting from a set of well accepted design rules, derive a set of constraints under which a given circuit follows these rules ([21]). The verification methodology employed by design audit systems can be either complete ([21]) or incomplete ([22]). As a result of the structured, hierarchical way the design and the specifications are expressed, the error reporting of design audit systems is elaborate and sophisticated.

Incomplete verifiers are characterized by two main properties:

1. They provide a specification model (formal or not), and the specified intended behavior of a design is compared against its implemented behavior. This distinguishes them from simulators and timing analyzers. The ability to specify the intended behavior of a design results in error reporting that is more sophisticated than that provided by simulators or timing analyzers.

2. The verification methodology is incomplete.

Incomplete verifiers are differentiated according to their specification model. In Section 3 we discuss the importance of the specification model and the expressive power necessary to specify complex temporal behaviors for certain designs. At the end of Section 3 the specification model of CLOVER is compared with the specification models of different formal and incomplete verifiers.

The system by Bryant in [9] is an incomplete verifier. The user is able to specify relations between state transitions in the circuit. Simulating techniques are then used to prove the correctness of the specification. Incomplete verifiers like SCALD [27], TDS [20], HDTV [24] and CLOVER share the common property that they are timing verifiers. The verification techniques of SCALD are based on timing simulation while TDS incorporates expert system heuristics. In HDTV, verification is accomplished by composing incomplete behaviors of the design's components. CLOVER incorporates time simulation techniques to derive the behavior of the implementation under some given set of input signals supplied to the implemented circuit. An incomplete behavior is derived in the form of what we call an *event graph* of the implementation. The event graph is used by the verifier.

Why would one use an incomplete verification methodology to verify a design, especially when it is tempting to verify that the temporal behavior of the design perfectly matches its specified behavior, under all operating conditions? Most of the formal timing verification methods use a state graph to capture the behavior of the implemented design. A state graph contains information about signals' timing and transitions for every possible combination of input signals. The main disadvantage of this method is that the size of the state graph is exponential in the size of the design and thus the verification process tends to be time consuming. That limits the applicability of the approach to the verification of small size circuits. On the other hand, theorem-proving verification methodologies, like the one in [14], are quite tedious even for small size circuits. Furthermore the extended value system used by our simulator can represent several behavioral cases by one value. CLOVER is intended to support, like formal verification systems, hierarchical verification. Verified components can be substituted by their verified timing behavior. As a result our verification methodology although incomplete provides a much more efficient way to verify a circuit.

3 A Timing Constraints Specification System

This section is organized in three main subsections. In section one, we present the semantics of our system, and we define the concepts upon which it is built. In section two, ATCSL, our specification language, is presented along with specification examples. Finally in section three, related work in the timing constraints specification area is presented and compared with our specification system.

3.1 The Semantics of the Specification System

3.1.1 What constitutes a Good Specification System?

One of the motivations for this research was the lack of a good way to express timing constraints; this is especially important for designs where accurate timing is critical for the operation of the entire system (for example interface circuits). Part of the reason for this absence is that the models used in the past (like temporal logic) although formal and concise, have failed to capture all the necessary ingredients of the specification problem (see discussion in Section 3.3). As a result, the specification process was incomplete. We believe that in order to design a good specification system, we should look at what happens in the "real word". We should study how the designers receive and express their information about the timing constraints of the system they design. We should find which are the concepts upon which they build their information. Having done these, we can design a formal model for the specification system, based on our gained knowledge and experience. This is the way we started thinking about designing our specification system.

Timing diagrams and specification manuals are the ways designers and engineers think of and specify their designs. To get an insight on the kind of timing constraints assumed between bus interface signals, we present in Figure 1 the timing diagram specifying the timing relations between signals during the Multibus Read cycle [18]. The diagram presents the waveforms of the signals, annotated with timing information and directed arcs between them. We distinguish two kinds of timing constraints specified in this diagram. First, we have constraints which determine the relative timing between events of the same or different signals. For example, after $MRDC*$ signal goes from high to low, $XACK*$ signal should make the transition from high to low no later than $8\mu s$ ($txack$) and no sooner than $0\mu s$. Second, dependencies between signals are described with the use of directed arcs. We refer to this kind of constraints as dependency constraints. For example, a transition for $XACK*$ from high to low will cause some time later a transition from low to high for the signal $MRDC*$. Keep in mind that signals that appear only on special occasions, like bus error signals, may have separate timing diagrams describing their relations and behaviors.

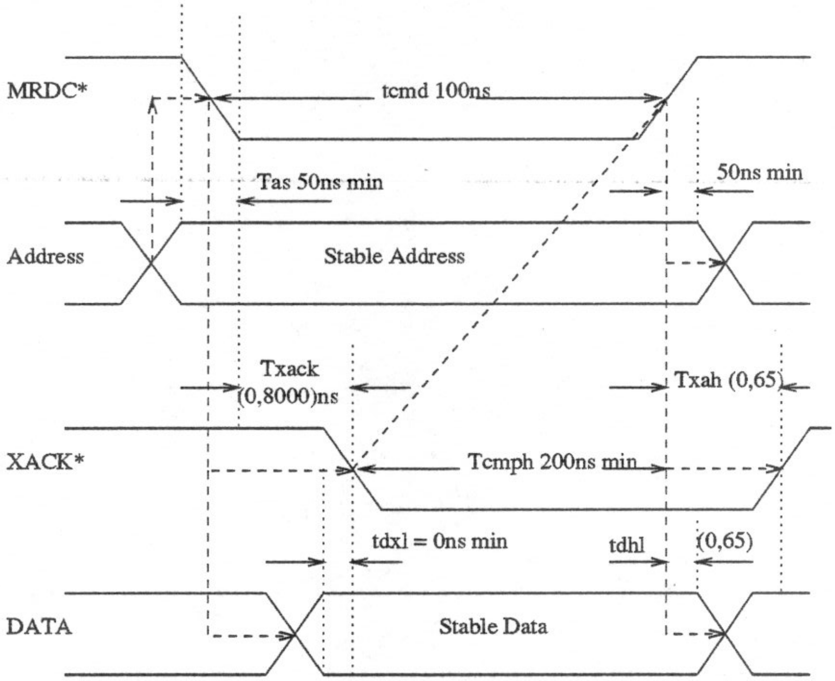

Figure 1: Multibus Read Cycle

Our specification system has been designed around the concepts and the relations obtained from a timing diagram or a specification manual. They are informally summarized as follows:

- A key concept is the concept of a signal which at some point in time undergoes a transition to a different value.
- Timing bounds exist which specify the relative timing between these transitions.
- Dependency information may be specified between transitions and determines a certain sequencing between them (directed arcs in Figure 1).
- The timing diagrams, being a two-dimensional interface, can describe the sequences of all signal transitions over the time. As a consequence the constraint relations can refer to present transitions as well as to past ones. We refer to such a sequence of transitions over time as a transition *history*.
- The idea of having constraints which apply under certain conditions introduces the necessity of conditional specification.

Next, a formal description of the semantics of our specification system is presented.

3.1.2 Defining the Model

In the following discussion, we assume a discrete time model with a time unit being the smallest quantum of time. There is the implicit concept of the time a circuit starts to operate. We refer to it as the *starting* time. All time variables, defined next, assume values relative to this time. Each magnitude or constant of dimension time is expressed in terms of a number of time units. The concepts of an *event* and a *signal* are the main concepts defined by our model.

Definition 3.1 *A* **functional module** *is a hardware system with certain inputs and outputs. The behavior of the module is a function of its inputs and internal state. Its hardware complexity can range from that of a simple gate to that of a complex hardware system.*

We consider functional modules to be the basic building blocks of a circuit.

Definition 3.2 *A* **circuit** *is a set of functional modules and wires which interconnect the input and output ports of the functional modules.*

Definition 3.3 *A* **value system** *is a finite set of discrete values.*

Definition 3.4 *Given a value system and a circuit, an* **event E** *is defined as the occurrence of a new value assumed by a port at some point in time during the operation of the circuit, and it is characterized by a 5-tuple $\prec P, V, I, T, W \succ$, where:*

P: The port of the module which assumed the new value.
V: The value assumed by the port, which is a member of the value system.
I: The index of the event. This is the $I + 1$th time (we count indices starting from zero), during the operation of the circuit where the port P assumed the value V.
T: The starting time of the event, called also **initial time** *of the event.*
W: The time during which the event retains the value assumed at time T, called also **width** *of the event. We say that the event* **holds** *during that time.*

The particular value system used by CLOVER is presented in Section 4.

Observation 3.1 *For two events $E_i = \prec P, V, I, T_i, W_i \succ$ and $E_j = \prec P, V, J, T_j, W_j \succ$ with the same P, V and such that $I < J$, it is true that: $T_i + W_i < T_j$.*

The use of indexing in the characterization of an event E provides us with the capability to refer to the past and present instances of events with the same value V and appearing at the same port P. We say that by indexing we gain access to the *history* of the events characterized by $\prec P, V, i, T_i, W_i \succ$.

Definition 3.5 *We define a* **V-signal SV** *on a port P having the value V, as the set of events E_i over time, which appear at port P and assume the value V. Given that n events with value V appear at port P, then V-signal SV is the union over i of the E_i: $\bigcup_{i=0}^{i=n} \prec P, V, i, T_i, W_i \succ$ and is characterized by the 5-tuple $\prec P, V, n, \vec{T}, \vec{W} \succ$, where:*

P: The port of the module which assumes the value V.
V: The value assumed by the port.
n: The number of events with value V appeared on port P.
$\vec{T}$: The initial time of the V-signal SV which is the vector: $\prec T_0, T_1, \ldots, T_n \succ$
$\vec{W}$: The width of the V-signal SV which is the vector: $\prec W_0, W_1, \ldots, W_n \succ$

The number of events **n** *is also called the* **size** *of the V-signal SV. A* **signal S** *on a port P is the set of V-signals SV_j which appear on port P and have distinct values V_j.*

We can see from definition 3.5 that a signal S on a port P is constituted from all the events occurring on port P. If we sort all these event instances in ascending order over their initial time, we will obtain the continuous history over time of transitions that occur on port P and that constitute the signal S. This is the equivalent of the *waveform* of signal S as would appear in a timing diagram describing S.

Our model of timing constraints consists of two relation classes: *timing relations* and *dependency relations*. The timing relations refer to the timing characteristics of an event, that is the initial time and the width. Dependency relations refer to another characteristic of an event: its value. With a dependency relation we want to specify the cause for the creation of a new event.

Definition 3.6 *A* **timing arithmetic expression** *over a set of events $\{E_i\}$ is any arithmetic expression constituted from the initial times T_i or the widths W_i of the events E_i. It returns as a result a number which is the arithmetic result of the expression over the T_i and W_i of the events E_i. It is uncomputed if any event is missing.*

Definition 3.7 *A* **timing arithmetic expression** *over a set of events $\{E_i\}$ and V-signals $\{SV_j\}$ is any arithmetic expression constituted from the initial times T_i or the widths W_i of the events E_i and the initial times $\vec{T_j}$ or the widths $\vec{W_j}$ of the V-signals SV_j. It returns as a result a vector with size equal to that of the smallest sized V-signal SV_j, such that:*

$$\begin{aligned} &kth\ element\ of\ arith\_expression(\prec P_i, V_i, I_i, T_i, W_i \succ, \prec P_j, V_j, n_j, \vec{T_j}, \vec{W_j} \succ) = \\ &\qquad arith\_expression(\prec P_i, V_i, I_i, T_i, W_i \succ, \prec P_j, V_j, k, T_j[k], W_j[k] \succ). \end{aligned}$$

Where $T_j[k]$ denotes the $k+1st$ element of vector $\vec{T_j}$ $(k \geq 0)$. If any signal or event is missing the result is uncomputed.

For example, given a V-signal SV_1 with size 3 and a V-signal SV_2 with size 2 then the arithmetic expression: $\vec{T_1} + \vec{T_2}$ is equal to a vector $\vec{v}$ where: $\vec{v} = \prec T_1[0] + T_2[0], T_1[1] + T_2[1] \succ$.

Definition 3.8 *A* **timing relative relation** *is any relative order relation over two timing arithmetic expressions.*

According to definitions 3.6 and 3.7, a timing arithmetic expression can be a number or a vector respectively. When the arithmetic expressions are two numbers the meaning of the relative relation is obvious. When the expressions are two vectors $\vec{v_1}$ and $\vec{v_2}$ with sizes n and m respectively, then the relative relation is applied to each one of the n pair of numbers: $(v_1[i], v_2[i])$, where $0 \leq i < n$ (assuming $n \leq m$). Finally when the expressions are a vector $\vec{v}$ with size n and a number b, then the relative relation is applied to each one of the n pair of numbers: $(v[i], b)$, where $0 \leq i < n$.

Definition 3.9 *A* **timing relation** *can be any timing relative relation or any boolean combination of timing relations. Any* **universal** *or* **existential** *quantification of a timing relation over the indices of the events which appear in the relation is also a timing relation.*

A timing relation can be conditionally evaluated and the conditional predicate can be any timing or dependency (to be defined) relation. As an example of universal and existential quantification consider a V-signal $SV = \prec P, V, n, \vec{T}, \vec{W} \succ$, and the timing relation $\mathcal{R} : T[i] \geq 100$. The relation $\mathcal{R}$ is universally quantified over i as: $\forall i, 0 \leq i < n, T[i] \geq 100$. An existential quantification for $\mathcal{R}$ is expressed as: $\exists i$, where $0 \leq i < n$ such that $T[i] \geq 100$.

With definition 3.9 the formal description of a timing relation is complete. Next, the concept of a dependency relation is formally defined.

Definition 3.10 (Causality and weak dependency definition) *Consider an operating circuit at a timing instance t and a functional module of it $\mathcal{F}$ with N inputs and M outputs. We assume that at time t all value transitions of the events at the input and output ports of $\mathcal{F}$ have been stabilized and that a set of N events I_i $(0 \leq i < N)$ appear at the input ports of $\mathcal{F}$ and another set of M events O_j $(0 \leq j < M)$ appear at the output ports of $\mathcal{F}$. Lets t' be the time when one or more new events I'_k $(0 \leq k < N)$ appear at the input ports of $\mathcal{F}$. After a time d, equal to the delay through module $\mathcal{F}$, $\mathcal{F}$ computes a set of output events according to its functional specification. If some of the events on the output ports are different than the old events appeared on the same ports, then new events O'_l $(0 \leq l < M)$ are created on these output ports. We say that an input event* **causes** *a new output event O'_l, if it is one of the new input events I'_k. We also say that all events O'_l $(0 \leq i < M)$ are* **weakly** *dependent on all the input events which appear or hold at time t'. Furthermore the causality and weak dependency relations are transitive.*

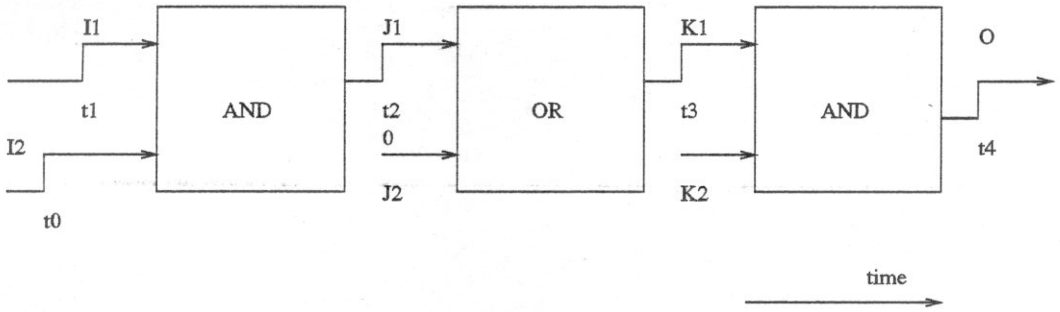

Figure 2: Dependency Relations

$$(E_i \textit{ causes } E_j) \bigwedge (E_j \textit{ causes } E_k) \Longrightarrow E_i \textit{ causes } E_k$$
$$(E_i \textit{ weakly depends on } E_j) \bigwedge (E_j \textit{ weakly depends on } E_k) \Longrightarrow E_i \textit{ weakly depends on } E_k$$

In other words, from all the input events of $\mathcal{F}$, the ones which *cause* the new output events are the ones which happen temporally last. Using the causality relation we can specify the sequencing of events through the circuit during its operation.

Definition 3.11 *We call a sequence of signals (or events) S_1, S_2, ..., S_n a* **causality sequence** *if:*

$$S_1 \textit{ causes } S_2,\ S_2 \textit{ causes } S_3,\ \ldots,\ S_{n-1} \textit{ causes } S_n.$$

Definition 3.12 (Strong dependency definition) *Given a functional module $\mathcal{F}$ and an output event of it O, we say that O is* **strongly** *dependent on an event I, if I is either an event on an input port of $\mathcal{F}$ to which O is weakly dependent, or if I causes an input event $I^{'}$ of module $\mathcal{F}$, and O is weakly dependent on $I^{'}$.*

Observation 3.2 *Two events which are causally related are strongly and weakly related as well. Similarly two events which are strongly related are also weakly related.*

Next we define the three dependency relations as they are applied to signals.

Definition 3.13 *Given two V-signals, $S_1 = \prec P_1, V_1, n_1, \vec{T_1}, \vec{W_1} \succ$ and $S_2 = \prec P_2, V_2, n_2, \vec{T_2}, \vec{W_2}, \succ$ (assume $n_1 \leq n_2$), we say that S_1* **causes** *S_2 if:*

$$\forall i, 0 \leq i < n_1, \prec P_1, V_1, i, T_1[i], W_1[i] \succ \textbf{ causes } \prec P_2, V_2, i, T_2[i], W_2[i] \succ$$

Similarly we can define strong and weak dependencies between signals, substituting the word "causes" in the previous definition by "strongly" depends on and "weakly" depends on respectively.

A causality relation is not a functional relation. An event I may cause an event O, but the creation of the event O may depend on the presence of other events in some input ports of the circuit modules. With the strong and weak dependency relations we provide our model with the capability to express dependency relations, not necessarily causal, between events.

In Figure 2 we have a sequence of three functional modules: an AND gate, an OR gate, and another AND gate in series. The waveforms of the input and output signals are represented at the input and output ports of the gates. We assume event K_2 to hold forever at value one and event J_2 to hold at value zero. At time t_0, t_1, t_2, t_3 and t_4 the I_2, I_1, J_1, K_1 and O events appear respectively. Assuming a common delay d for all three gates, we have the relations: $t_4 = t_3 + d$, $t_3 = t_2 + d$ and $t_2 = t_1 + d$. The following dependency relations hold:

1. *I_1 causes J_1, J_1 causes K_1, K_1 causes O.*

2. From the transitive property of the causality relation it is also true that: I_1 *causes* K_1, I_1 *causes* O and J_1 *causes* O.
3. O is ***strongly*** dependent on K_2, K_1 is ***strongly*** dependent on J_2 and J_1 is ***strongly*** dependent on I_2.
4. Finally O is ***weakly*** dependent on J_2, O is ***weakly*** dependent on I_2 and K_1 is ***weakly*** dependent on I_2.

Note that if $t_0 = t_1$, then both events I_1 and I_2 cause event O. As an example that dependency relations are not strictly functional, note that event O is ***weakly*** dependent on event J_2 even though event J_2 has the value zero and has no effect in the creation of the output event K_1 with value one, at the output port of the OR gate.

Observation 3.3 *Strong dependency is not a transitive relation.*

For example, in Figure 2 event O is caused by event K_1, and therefore also ***strongly*** dependent on K_1. Event K_1 is ***strongly*** dependent on event J_2. But event O is only weakly dependent on J_2.

Definition 3.14 *A* **dependency relation** *is any causality, strong dependency or weak dependency relation, or any boolean combination of dependency relations. Any* **universal** *or* **existential** *quantification of a dependency relation over the indices of the events which appear in the relation is also a dependency relation.*

A dependency relation can be conditionally evaluated and the conditional predicate can be any timing or dependency relation. In our specification system the dependency relations are represented by a *dependency graph* with nodes representing events and labeled edges between related nodes. The labels characterize the kind of dependency relation (causal, strong or weak) which exists between events.

Definition 3.15 *A* **timing constraint** *is any set of timing or dependency relations, as well as any conditional evaluation of each of them, with the conditional predicate being any timing or dependency relation.*

In the next section we present ATCSL, the language we developed to represent our specification model, along with some specification examples which will show the expressive power of the model.

3.2 ATCSL: A Timing Constraints Specification Language

ATCSL is a specification language developed for the representation of the specification model introduced in the previous section. In ATCSL timing and dependency relations are specified separately. An informal presentation of the basic ATCSL constructs is given next:

A signal characterized by the 5-tuple $\prec P, V, n, \vec{T}, \vec{W} \succ$ is represented as $P.V$ (in ATCSL a port is described by its name and it can be any alphanumeric character string). For example, to refer to a V-signal on a port S with value rising(r), we say: $S.r$. For simplicity we will usually refer to a signal $P.V$ rather than a V-signal. Any event appearing on S with value r will be one of the n events which comprise signal $S.r$. We will refer to them as: *S.r[i]* where $0 \leq i < n$. The initial times of signals and events are represented by appending a circumflex (^) in front of the signals' or events' representations. For example, the initial time of event *S.r[i]* is represented as: *^S.r[i]*. Similarly to represent their widths an underscore (_) is appended in front of their representations. For example, the width of signal $S.r$ is represented as: *_S.r*. Finally, the number n of events *S.r[i]* which comprise signal $S.r$ is denoted by *#S.r*.

The standard arithmetic, relative and logical operators of ATCSL are the same as the corresponding operators of the *C* language. Using the *FOR* iterative construct of ATCSL and ranging

```
       _MRDC*.0 ≥ 100;
_MRDC*.0 ⟶ [0 8000] XACK*.0;
 _MRDC*.1 ⟶ [0 65] XACK*.1;
  _MRDC*.1 ⟶ [0 65] Data.c;
 ^Address.s + 50 ≤ ^MRDC*.0;
 ^MRDC*.1 + 50 ≤ ^Address.c;
 ^XACK*.0 + 20 ≤ ^MRDC*.r;
    ^Data.s ≤ ^XACK*.0;
```

Table 1: Timing Relations for the Read Cycle of Multibus

```
       MRDC*.0 ⟹ Data.s;
MRDC*.0 ⟹ XACK*.0 ⟹ MRDC*.r;
      MRDC*.r ⟹ Address.c;
      MRDC*.r ⟹ XACK*.1;
      MRDC*.r ⟹ Data.c;
```

Table 2: Dependency Relations for the Read Cycle of Multibus

the index over a set of events, references to event instances of signals can be made. Any subrange of indices can be specified. To conditionally evaluate an expression the *IF* operator is used. ATCSL provides four operators, *WHILE*, *OVERL*, *BEFORE* and *SYNC* which are shorthand for common relations between durations of events. The *WHILE* operator is used to express that an event appears and holds while another event holds. The operator *OVERL* (for overlap) specifies that an event that starts earlier than another ceases to hold while the other one is still holding its value. To state the relation that an event should start and cease before the appearance of some other event we use the operator *BEFORE*. The *SYNC* operator is used to specify that a signal appears in synchrony with another signal. We existentially quantify an expression with the *THERIS* construct by also specifying a list of indices and their range over which the expression is quantified. Finally we express the causality relation that signal *A.1* causes signal *B.1* by: *A.1* ⟹ *B.1*.

Tables 1, 2 and 3 give examples of the use of ATCSL. Tables 1 and 2 present the complete set of the timing and dependency relation specifications, respectively, of the Multibus read cycle [18]. Both sets of relations are as given in Figure 1, where the dependency relations are denoted by dashed directed arcs. Table 3 is a timing specification for reading and writing RAM from an expansion card in Macintosh SE [17] (Figure 3). The rule being represented is:

Rule: AS falling must occur not later than 20ns into S_3. If AS has not fallen by that time, AS must not fall until after the first 20ns of S_4 (data will be read or written in the next RAM access).

The specification in Table 3 is an exact interpretation of the rule. The increment step in the *FOR* loop is equal to the number of states (eight) specified in Figure 3.

In the next section we compare our model with previous and present approaches by other researchers.

3.3 Related Work in Timing Constraints Specification

Logic formalism enriched with the concept of time has been used for the specification of timing constraints. *Temporal logic* ([7]) is one representative of logic formalism. The motivation for the

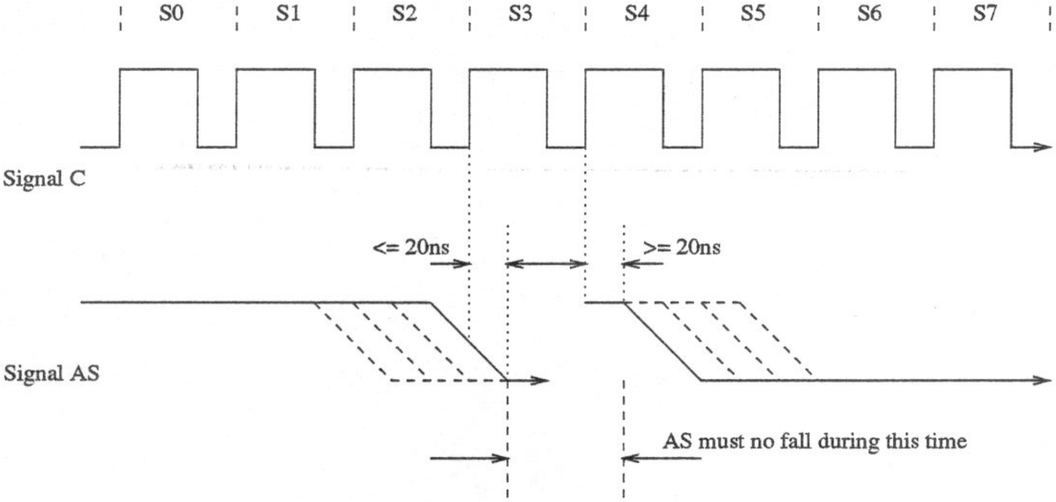

Figure 3: Part of RAM timing in a Macintosh SE

```
FOR (i; 0; #C.1; 8) {
        IF ($AS.0[i]$ OVERL $C.1[i+3]$ {
            $\hat{}AS.0[i] \leq \hat{}C.1[i+3] + 20$
        }
        ELSE { $\hat{}AS.0[i] \geq \hat{}C.1[i+4] + 20$ }
}
```

Table 3: Accessing a RAM in Macintosh SE

temporal logic formalism was to specify temporal behavior with a precise, mathematical semantics and verify the behavior by proving or disproving the specified constraints [4]. However proving the correctness of designs with reasonable size using the logic formalism turns out to be impractical. Furthermore the specification model is inadequate to express complicated timing relations. One can state relations about the relative ordering of events but not about their absolute timing. One can specify as a constraint that a signal should make a certain transition at some point in time, but not when. There have been extensions to handle specific timing (ITL [31] for example) but they share the following disadvantages: First, they retain the logic formalism but cannot use the proof methods for verification; second, they are not capable of specifying the information contained in a timing diagram or a specification manual. The latter is true because none of the existing temporal logic models supports reference to transitions' history (indexing) or conditional evaluation. As a result, it would be difficult for a designer to make good use of them. (See also the discussion in Section 3.1.1 on what constitutes a good specification system.)

Higher-order predicate logic has also been used for the specification and verification of digital systems [15]. By describing the input and output signals as functions of time, using the notion of a signal's waveform, timing relations can be specified between signals. The functional and timing behavior of separate modules of a circuit can be described as waveform relations. No way is provided to reference specific time instances of the waveforms (done with indexing in ATCSL). None of the logic formalisms provides a way for conditional evaluation.

Trace Theory [34] is another representative of a behavioral specification. A *trace* at some port of a circuit module can be viewed as a series of signal transitions occurring at this port during the operation of the circuit. It is analogous to what we called transitions' *history*. Relations between input and output sequences can be expressed, but no specific time or timing relation can be specified between them. Work in [12] deals with the problem of defining a formal framework where time can be specified in trace theory specifications. The approach is based on a *continuous* model of time (times are real numbers, not integers). The complete behavior of the implementation in the form of a state-graph is temporally verified by associating with each state of the graph a convex linear region, describing the states of individual timers, which keep track of the possible times at which events can occur. Sequences that violate the timing assumptions are identified by this automaton.

Petri nets have been used for the specification of timing constraints [30]. Petri nets are much like dependency graphs, in the sense that they provide the same dependency information between signals as dependency graphs do, but they cannot handle the specification of the timing relations. Petri nets are better suited to describe self-timed circuits [36]. There are two extensions to Petri Nets that try to address the time issue. *Timed Petri Nets* [28] specify explicit time intervals, (with the use of min's and max's) during which a transition may fire. These are very similar to dependency graphs. We can use timed petri nets to express the class of timing relations covered by ATCSL. This can be done by using the petri net structure to specify all the possible transitions(with timing information) between events, obtaining the equivalent of a state graph. However these representations can be huge and the specification becomes awkward and impractical. *Temporal Petri Nets* are a combination of temporal logic operators and the petri net structure [38]. Timed and temporal Petri Nets do not cover the same class of specifications (this is proved by a counter example in [38]).

Problems similar to that of timing specifications in digital circuits have been addressed for the specification of systems such as communication protocols, parallel machines and distributed systems. The IC* [10] model of computation provides an environment where temporal and structural constraints of a design can be specified and verified. The system operation is described as a sequence of states over time and the constraints are invariants that should hold during each state of the system. Causal expressions determine the state transitions and the invariants can also be conditional.

In parallel with and independently from our work, the CPA framework has been developed by Michael McFarland [25]. The concept of an event is defined very much like we do in our model. There is however a difference in how an event is indexed. In our model, events occurring on a particular port are characterized by different set of indices according to their value. Given a V-signal, the combination of its port and its value determines its set of indices (its set of events). In CPA, events occurring on a particular port are characterized by one set of indices. Given an event, the combination of its port and its index determines its value. As a result, it is not obvious how to express in CPA specifications referring to different event instances of the same V-signal. On the other hand, CPA can express sequences of values on a port over time and therefore provides a better way to express the transformations of a data sequence between an input and output port.

Gaetano Borriello [6] has developed a specification method based on what he calls formalized timing diagrams, and has applied it to transducer synthesis. His idea is that the most natural way to describe a timing diagram is a timing diagram, but problems arise when trying to express complex constructs like conditionals and loops. Formalized timing diagrams provide just a timing diagram, and it is hard to extend their expressive power. A different specification approach is presented by the same researcher in [1]. The specification exhibits a symmetry between structure and temporal behavior and can be used for synthesis and simulation tools. An event, much like in our model, is characterized by its value, port, start time (initial time) and completion time (initial time + width). Indexing is not provided. Instead the concept of an event's *ancestry* is introduced. An ancestor of an event is any previous occurring event that led to the generation of its descendant. Certain relations between events are determined over some (if it exists) common ancestor of them.

The timing verifiers HDTV [24], SCALD [27] and TDS [20] incorporate specification models which are consistent but rather ad hoc and restricted. Interestingly, the HDTV specification system has some similarities with the specification model of CLOVER. This is because HDTV, like CLOVER, aims towards the specification and verification of interface circuits. In SCALD, the designer can only specify set-up and hold times between signal transitions as well as constraints on the pulse width of certain signals (the system was designed to verify clocked designs). The TDS system is an expert system approach to automate the timing design of interfaces between VLSI chips in microcomputer systems. Constraints are expressed as minimum and maximum time between two events.

The roots of our representation can be found in the work described above. However, unlike the logic formalisms, it provides a formal framework whose concepts are much closer to and based upon the way people design and specify systems. The combination of our timing and dependency relations make our framework general and powerful for describing timing constraints of system components.

4 An Event Driven Timing Simulator

Our constraints specification language is based on events and signals. In order to verify the constraints we need to obtain information about the signals and the dependencies between events from the implemented design. The information is derived in the form of an *event graph*. Every event corresponds to a node of the event graph. The dependency relations between events are represented by the arcs of the event graph. An arc representing a causal relation is labeled as a *one-edge*, other edges are labeled as *zero-edges*. We call a path of the event graph consisting of one-edges a *one-path*.

We have built an event driven timing simulator which takes as input the description of the implemented design in the form of a netlist (derived from the PDL-e description of the design) and the input signals, and produces as output an event graph, where the dependency relations of the events are represented, and with a detailed description of all the circuit signals. We use an extended value system (similar to the one used by McWilliams in SCALD [27]), to permit the analysis of a circuit without referring to specific values. The table below defines the values and their meaning.

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

The use of the *rising* and *falling* values will become apparent in the subsection where our delay model is discussed. The *stable* value provides an abstraction over the more specific *zero* and *one* values, and it is used for the analysis of circuits where exhaustive case analysis on the input signal values is not necessary. The same abstraction provides the *changing* value over all possible values of our system. To understand how useful the abstraction of stable and changing values can be, consider the arbiter problem [11]. In order to avoid metastability, certain signals should remain stable while others are changing. We really do not care if signals remain stable at zero or one, nor do we care how signals change. The model provided by stable and changing is exactly what we need in this case to specify the intended behavior of the arbiter. Finally, *undefined* is the initial value for every signal in the circuit. It provides us with the same amount of information as the changing value does, but it is used only for initialization purposes. Depending on the input values, the performance of our event driven timing simulator ranges from a traditional simulator to a static timing analyzer, where case analysis for only certain inputs lies in between.

The timing simulator works like a standard event driven simulator. Initial values of primary inputs are given by the user and can be periodic (to facilitate specifying clocks). Events come from a priority queue (implemented as a heap), in increasing chronological order. If the values for the output signals are different from their old ones, the output events are inserted as new events into the queue. The simulator can be reinitialized to test the design for more than one input vector (case

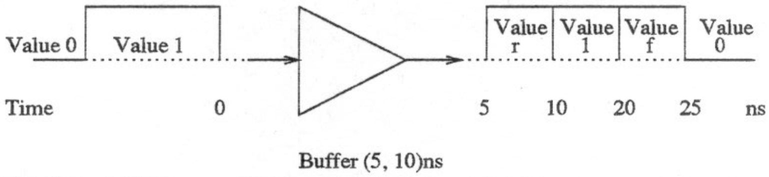

Figure 4: Signal Propagation

analysis). Case analysis may be necessary in order to obtain more accurate results and to avoid problems like false-paths. The user can code in *C* the behavioral description for any component of the design. For standard components like gates, latches, multiplexers etc, a built-in library may also be used.

Knowing the set of input events on which an output event depends and knowing their timing gives us a straightforward way to construct the event graph for the implemented circuit. Since the timing of the events is known, we can identify the events which cause the output to occur, according to the definition of causality in Section 3.2. At the same time events are processed, the signals are calculated. After the simulator stops running, it outputs the constructed event graph; when case analysis is done an event graph is output for each case. For a more complete and accurate description of the behavior of the circuit components, there are built-in functions which give the user the capability to obtain information about the event graph during the simulation. Thus information about the initial time and the width of events as well as the dependency relation between events can be obtained. This information can be used as conditional qualification in the behavioral description of the circuit.

Standard modules like flip-flops and latches have specific set-up and hold time constraints determined by their specification sheets. The simulator reports in a report file any violation of these constraints. The same standard constraints can be expressed in ATCSL and verified by the verifier; however, it would be inconvenient for the user to specify the constraints for each latch or flip-flop of a design's implementation. The user can specify his or her own set-up or hold time constraints using ATCSL.

Our event-driven timing simulator supports both the inertial and transport timing models (see discussion in [2]). These models characterize how devices will respond to rapid input changes (rapid in comparison with the devices' delays). There is a mechanism to detect oscillation, based on predetermined time-out periods. The time-out time can be either user specified or specified by the simulator. In two-level logic (low and high), an oscillation is defined as an indefinite back and forth transition of a signal between values in different logic levels. For our value system, we can say that the one and rising values belong to the high logic level and that the zero and falling values belong to the low level. Since an oscillation is most likely to appear in a feedback loop, we calculate the strongly connected components of the circuit graph and topologically sort them. After the time-out period, we examine each signal coming out of the queue and we check if it has made a transition back and forth between the two logic levels. If that is the case and in addition the event is an output from a module which is a member of a strongly connected component with more than one members, then an oscillation is suspected. The highest topologically numbered component (which is the suspected feedback loop) is then printed with the suspected signal in a report file.

4.1 Delay Model

The delay of a device is not a constant parameter. It is rather a random variable which depends on the device's physical parameters: the temperature, the power supply, the inputs' slopes and the outputs' load. To capture the delay uncertainty we adopt the minimum-maximum delay model, where the switching time of a device is bounded below by a minimum time and above by a maximum time. This

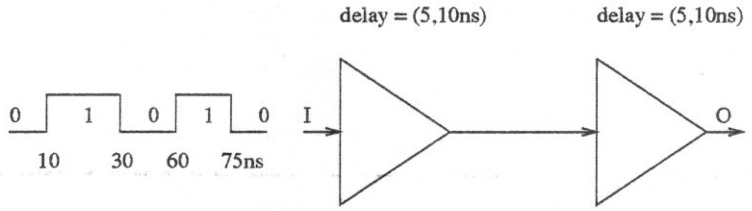

Figure 5: Two buffers connected in series

is the way most of the parts' books define the timing characteristics of their components. In [39], Wallace and Sequin compare the min-max model with the slope model and a probabilistic model where the delay is modeled as a random variable following some distribution. For the slope model to work, load information should be available and the delay calculations become more complicated. Probabilistic analysis may give more optimistic results than the min-max model does. However it is difficult to propagate the delay information through the circuit, unless certain assumptions are made (e.g., distributions, correlations etc). In addition it is preferable to know that an implementation is definitely error free, than to know that there is a 99% probability assuring this.

In Figure 4, we present an example of how the min-max delay model can be used in conjunction with our value system, to propagate information through a functional module. The module is a buffer with minimum delay time 5ns and maximum delay time 10ns. Given as input the waveform "in", the output waveform is a combination of the rising, one and falling values. Our knowledge about when the output is one has been diminished compared with that of the input, instead during the rising and falling intervals, the output may be or may not be one (stable). This is a disadvantage of the min-max model since it leads to pessimistic results.

The min-max model is a good compromise between accuracy and efficiency. It may lead to pessimistic results but it will never derive a behavior which will be verified as correct when it is not.

5 The Verifier

In Section 3 we introduced the specification model of CLOVER, used to formally describe the temporal behavior of a design. In Section 4 we discussed the derivation of an event graph to capture the behavior of a design's implementation. Here we will focus on the verification process. During verification, the tool matches the intended temporal behavior of the design against the event graph of its implementation.

5.1 The Verification Process

A very simple circuit of two buffers connected is series will be used as an example to clarify our verification procedure (Figure 5). The minimum and maximum propagation delays for the two buffers are 5 and $10ns$ respectively. The initial input signal (its waveform appears on the input port of the first buffer) consists of two one-pulses. The intended temporal behavior of the design is specified with the following timing and dependency relations:

Timing Relation $\_O.1 \geq 10$
Dependency Relation $I.1 \Longrightarrow O.1$

Parsing these ATCSL expressions, we produce their corresponding parse graphs. The parse graphs of the ATCSL expressions and the event graph derived from the circuit's analysis are the two inputs of the verifier. When case analysis is incorporated, more than one event graphs are fed to the verifier and they are verified under the same set of specifications.

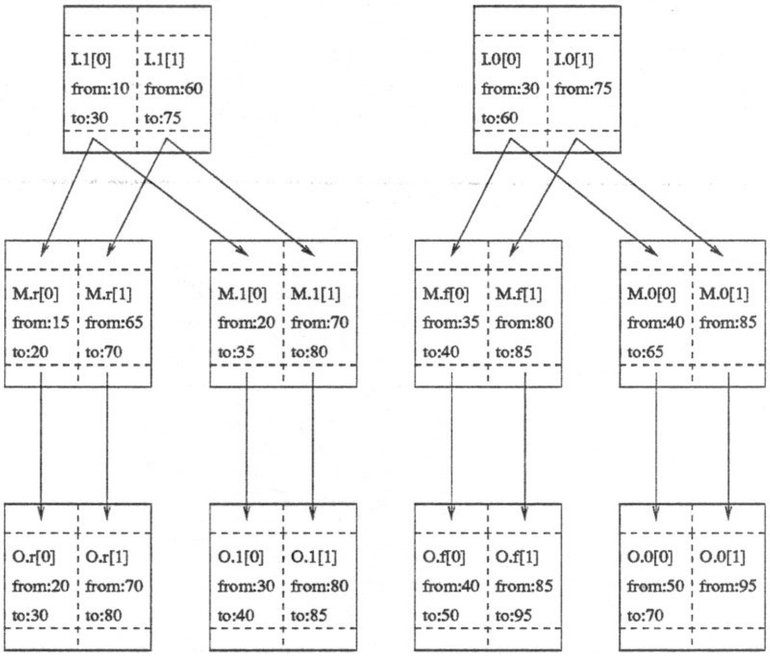

Figure 6: Event Graph

5.1.1 Verification of Timing Relations

The algorithm for the verification of timing relations is straightforward. The timing relations are expressions with variables that are the timing characteristics (initial time or width) of events or V-signals. In the event graph we have recorded for every signal the exact timing information (initial time and width) for all the events which constitute the signal. For each variable in a timing relation, we substitute the corresponding timing information for the event or V-signal. To verify the timing relations we simply have to evaluate them using the parse graphs. It is possible for a V-signal or event specified in a timing relation to be absent from the event graph. The reason could be some anomalous behavior of the circuit. In that case we cannot evaluate the timing relation and we report the signal or event which is missing.

Let's apply the verification procedure to our example. The output signal *O.1* consists of two events: *O.1[0]* and *O.1[1]*. The specified timing relation is a relative timing relation over a signal *O.1* ($\_O.1 \geq 10$). Being a relation over the width of a V-signal, it has to be evaluated for all the events which constitute signal *O.1*. Evaluating the expression, we find out that the event *O.1[1]* of signal *O.1* with width $5ns$ violates the specified timing relation.

5.1.2 Verification of Dependency Relations

To verify a timing relation we use the timing information contained in the event graph. In order to verify a dependency relation we use the structural information (nodes and edges) of the graph. Given an event graph G, we say that an event A is *reachable* from another event B in G if there is a path of G which starts from event B and passes through event A. If A is reachable from B then B is not reachable from A because graph G is acyclic. The verification of a dependency relation is a reachability test between the events which constitute the relation. The labeled edges of the path will help determine if the dependency relation is causal, strong or weak.

An algorithm similar to the one computing the transitive closure of a graph, but modified to characterize the dependency relations between every pair of the graph's nodes, will provide the answer

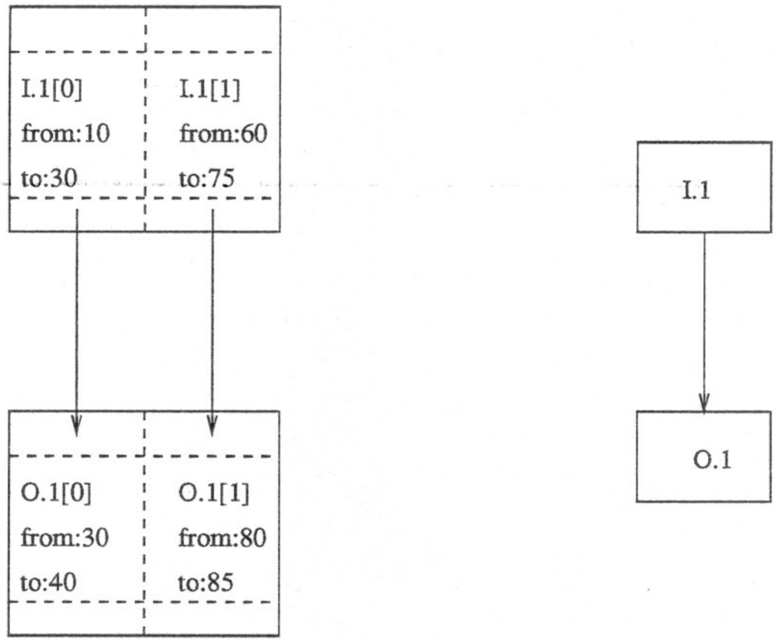

Figure 7: Match Graph and the corresponding dependency relation

to the reachability question. For a graph G with V nodes and E edges, Warshall's algorithm [35] finds the transitive closure of G in $O(VE + V^2)$ steps. For a large event graph the overhead of the algorithm will be large. Furthermore, since the majority of the signals appearing in the event graph are not part of a dependency relation, the algorithm will generate a lot of useless information. In order to improve the algorithm's efficiency we apply the following modification.

Initially we traverse the parse graph of the dependency relations and we identify the events and signals which constitute the relations. Then we traverse (using depth first search) the event graph and for every dependency relation we find its *match*. The match of a dependency relation is a collapsed version of the event graph. The nodes of the match are the events of the relation which appear in the event graph, and the edges are determined by the dependencies between these events in the event graph. The edges of the match are labeled according to the dependency relations existing between the interconnected nodes, using the dependency rules.

To verify a relation, we find the transitive closure of its match. A path of one-edges yields a one-edge between its source and destination. Any other path yields a zero-edge. Since the size of a match is comparable with the size of its corresponding dependency relation, the saving in computational time with this approach is usually considerable.

For the two buffer example, the *match* of the dependency relation consists of the nodes of the event graph (Figure 6) with signals *I.1* and *O.1* (Figure 7). It is easy to check by inspection that the dependency relation between the two signals is verified for both of the signals' event instances.

5.2 Error Reporting

CLOVER uses three files to report the outcome of the verification. The first file is the error report file. Each violated relation is reported to this file along with a description of the events which constitute the violated relation. For the two-buffer example, CLOVER reports the following error message:

```
**** _O.1 >= 10
```

```
Signal O.1 index: 1 from: 80 to: 85
Violate(s) rel_operator: '>=' against the number: 10
```

To help the designer identify or better understand the reason for a violation, the verifier reports a description for every specified signal of the design. The description includes the events which constitute the signal, their timing characteristics and their dependency relations. Since the description contains information about the dependency relations of the signals, the set of causality sequences that caused the events appearing in the specification of the failed constraint can be identified. This information may help the designer to locate the problem that caused the violation in some earlier stage of the circuit. We call these signal descriptions *traces* and they are reported in the trace file. Going back to our example, we list below the trace information for the output signal *O.1*:

```
Signal:  O.1
Index: 0 Initial time: 30 Width: 10
In-Dependencies: I.1[0](cause)
Out-Dependencies:

Index: 1 Initial time: 80 Width: 5
In-Dependencies: I.1[1](cause)
Out-Dependencies:
```

An empty error file indicates that the design under test successfully verified its specifications. However during the verification of a timing constraint we may violate part of a relation without causing a constraint violation. Consider for example a timing relation $\mathcal{R}$ which is composed by the logical OR of two timing relations $\mathcal{R}_1$ and $\mathcal{R}_2$. Relation $\mathcal{R}$ is verified to be true in the case when one of $\mathcal{R}_1$ or $\mathcal{R}_2$ is violated. In a situation like this, CLOVER outputs a warning violation message for either $\mathcal{R}_1$ or $\mathcal{R}_2$ in a third file, the warning file.

We believe that the error reporting of CLOVER gives enough feedback to the designer to identify the reason for a violation. An interesting but harder problem is the design of a more elaborate and sophisticated error report. This is highly related to design audit (see Section 2): the tool tries to give hints to the designer as to where in the circuit a problem may be found after a constraint is violated.

5.3 Hierarchical Verification

The verification process for large systems can be sped up considerably by hierarchical analysis. Hierarchy is used to describe large systems in a structured way. Our hardware description language (PDL-e) supports hierarchical design. If we partition the given design into well-defined submodules with timing constraints specified for each one of them, then after analyzing and verifying each submodule independently, the entire submodule description can be substituted by its verified behavior. The functional and timing specifications of the submodules are used, and the parts of the given circuits that are evaluated are those that have not been verified. On the other hand, we can follow the opposite direction and assume that a module follows its intended behavior, then use it to abstract the module behavior and verify the rest of the circuit.

In our current prototype, the user has to code the verified behavior of a submodule in *C* for use by the event driven timing simulator. Since the verification was done against the module behavior described in ATCSL, it would also be nice to use the verified ATCSL specifications as a behavioral specification of the module. Even though information can be lost doing this (ATCSL is a timing specification language not a functional one), the system becomes self-contained, since no user intervention is needed. Furthermore since the same specification language is used for the specification of the constraints and the behavior, we do not need to transform from one specification to another, something which would be prone to consistency errors.

Event though this feature is not yet implemented we will briefly present some of the issues here. A description of a module within the framework of the event-driven time simulator should conform to the following basic requirement: during the simulation process, for a given set of input events, a set of output events should be returned within a specified time period. Dependency relations of ATCSL can be used to represent the dependencies between input and output signals. For example, assume a module $\mathcal{M}$ with the following behavioral specification: $A.1 \Longrightarrow B.1$, where $A.1$ is an input signal and $B.1$ is an output signal. Each time module $\mathcal{M}$ is evaluated, an output event *B.1[i]* will be created as a response to an input event *A.1[i]*. The timing relations of ATCSL can be used to determine the timing of the output signals specified with dependency relations. Timing relative relations are the basic components of a timing relation. We view them as a set of inequalities over the initial time and widths of signals. To determine the timing for the output events we have to solve these inequalities over the initial time of the output events. Upper and lower bounds corresponding to minimum and maximum delays can be determined this way.

Hierarchical verification can be used to support a *partially complete* verification methodology. In a partially complete verification methodology we analyze and verify some of the small sized submodules of the design using a complete verification method (for example state graphs). Then we substitute the submodule's behavior by their verified behavior.

Hierarchical verification also supports *incremental* verification. In incremental verification, the circuit is analyzed in stages, with more information about the operation of the circuit added in each stage. Since the user has the ability to determine the behavior of modules, the user can question the functionality of the implementation at different abstraction levels.

5.4 Reasoning About the Verification Methodology

According to the discussion in Section 2, there are two criteria which characterize a verification methodology: completeness and accuracy.

The accuracy of our verification methodology is directly related to the accuracy of the analysis model, which produces the event graph. The shortcomings of the model (min-max delay model, propagation of uncertainty regions, etc.) as well as alternative models have been discussed in Section 4.1.

CLOVER, belonging to the class of incomplete verifiers, shares the shortcomings of an incomplete verification methodology. We have incorporated into the system three alternative ways of improving the completeness of our methodology:

Value System The extended value system incorporated provides an abstraction which permits one to reason about signals without referring to specific signal values like *zero* or *one*. This is the equivalent of reducing the number of states of the design's state graph. For example, substituting the *zero* and *one* values of a set S of signals by the value *stable* will cause the states of the state graph constituted from signals belonged to S to collapse into one state. In addition, we can still reason about the temporal behavior of the design where the behavior is not dependent on the *zero* or *one* values of signals in S.

Case Analysis The timing simulator provides the user with the ability to create more than one instances of event graphs, which will correspond to different input sequences.

Hierarchical Verification Hierarchical verification both speeds up the verification process and facilitates an incremental verification of a design. We saw in the previous section how hierarchical verification is intended within CLOVER and how it is related to a partially complete verification methodology.

In a number of cases, using a combination of abstract signal values and case analysis we can make our specification methodology complete. Consider the example of a simple bus interface circuit. We assume that the interface supports two transactions: read and write. They are initiated by a read or write command signal respectively. The address of the data to be read or written is placed on the address lines of the interface. Data are placed on the data lines. The command, address and data signals are the only external signals of the interface (through them the interface communicates with the outside world). We can completely verify the interface by doing case analysis based on the type of the transaction cycle (read or write) and abstracting the address and data signal values of the interface using stable values (for data or address signals which are driven) and changing values (for data or address signals which are undriven).

6 Case Study: The Multibus Design Frame

Design frames support a new methodology for VLSI system construction. Like operating systems, they provide standard interfaces to system components. The Multibus Design Frame(MDF) [5] connects a simple synchronous interface to the Intel Multibus [18]. While the Multibus has an asynchronous transaction protocol and a synchronous arbitration protocol, the MDF has a synchronous interface to its internal circuit and support four basic operations: slave read, write and master read, write. We have chosen this example because it is a good representative of the kind of designs we are interested in verifying. It is an interface circuit of medium size (around 80 modules) with a number of timing constraints characterizing its operation. Furthermore it is interfaced to a very well known bus: the Multibus. We analyzed and verified the MDF for the operations of slave and master read. The write operations are very similar. Due to space limitations, we will present here the specification and verification results only for the master read operations. However the statistics at the end of this section characterize the results after verifying the whole read operation. A description of the MDF operations can also be found in [6].

6.1 Implementation of the design

Figures 8 and 9 present a manually designed MDF interface[5] (showed in two parts for clarity of presentation). The interface signals to the Multibus are headed towards the right side of the circuit, and the interface signals to the MDF are coming from the left side. The only external input signals are the signals: *MRD*, *MWR*, *DATO*, *ADRO*, *BUSYIN* and the clocks. All the other input signals appearing in the two figures are internally created signals of the MDF. The data signals are 16-bit wide and the address signals are 19-bit wide. Our analysis is not dependent on specific data and address values. We assume the values for the data and address signals to be either stable, when they are driven by the bus, or changing when they are not. We group the 16-bit wide data signals and we represent them as a single data signal. Similarly for the address signals. The dotted boxes in the figures represent user-defined modules.

To give you an idea of how PDL-e can be used to structurally describe a design, we present below the PDL-e description of the sub-circuit of MDF with input signals *MRD*, *MACK* and output signal *MRDC*∗ (Figure 8).

```
/* Defining modules NOT_buf, NOR and R_S with */
/* their specific input and output ports */
#define NOT_buf(i1, i2) create_fun("%s %w %w %O", "not_buf", i1, i2, 1)[0]
#define NOR(i1, i2) create_fun("%s %w %w %O", "nor", i1, i2, 1)[0]
#define R_S(s, r) create_fun("%s %w %w %O", "r_s", s, r, 2)

main ()
{
wire *mrd, *mack, *mrdco, *mrdc, *cmden;

/* Creating named signals MRD, MACK and CMDEN */
mrd = name_wire("MRD");
mack = name_wire("MACK");
```

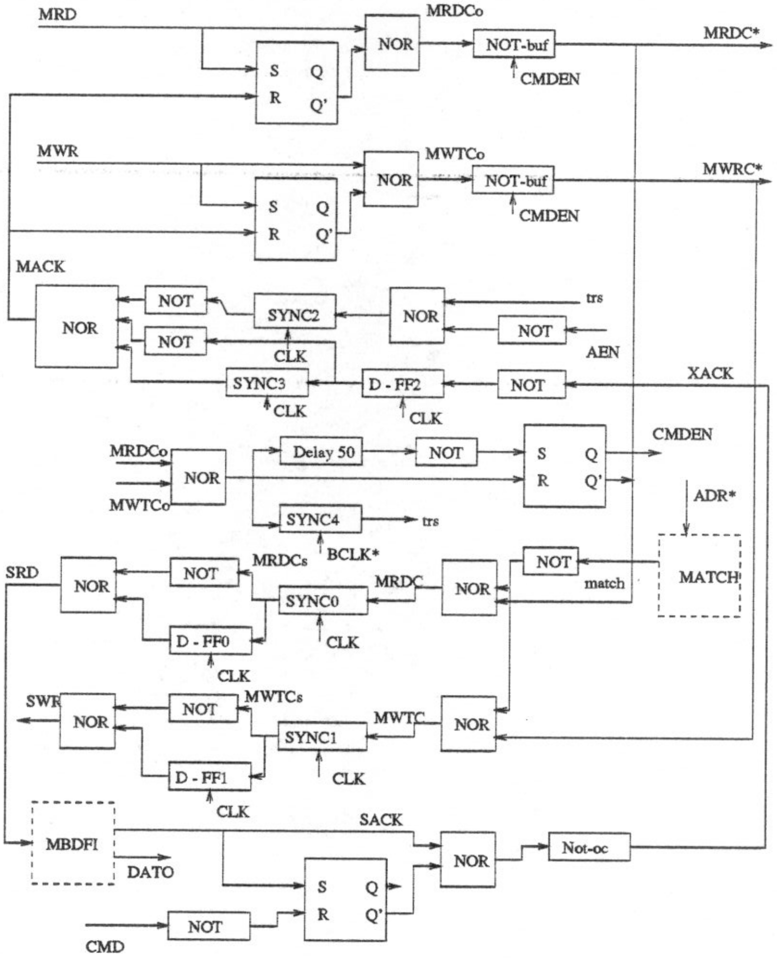

Figure 8: MDF Interface Circuit (part one)

```
cmden = name_wire("CMDEN");

/* Create signal MRDCo, the output of the module NOR, with inputs: */
/* signal MRD and the second output signal of the module R_S */
mrdco = change_name(NOR(mrd, R_S(mrd, mack)[1]), "MRDCo");

/* Create signal MRDC* as the output of the module NOT_buf */
/* with input the signal MRDCo, and the enable signal CMDEN */
mrdc = change_name(NOT_buf(mrdco, cmden), "MRDC*");
}
```

6.1.1 Clock Specification

The MDF allows precise control of the duty-cycle and the amount of non-overlap of the system clock. The two clock phases are generated from the input clock; phase one will occur on a rising transition of the input clock and phase2 on a falling transition. Adjusting the frequency of the input clock, we can regulate the amount of non-overlap between the two phases. For the analysis, we assume the following timing for the two phases:

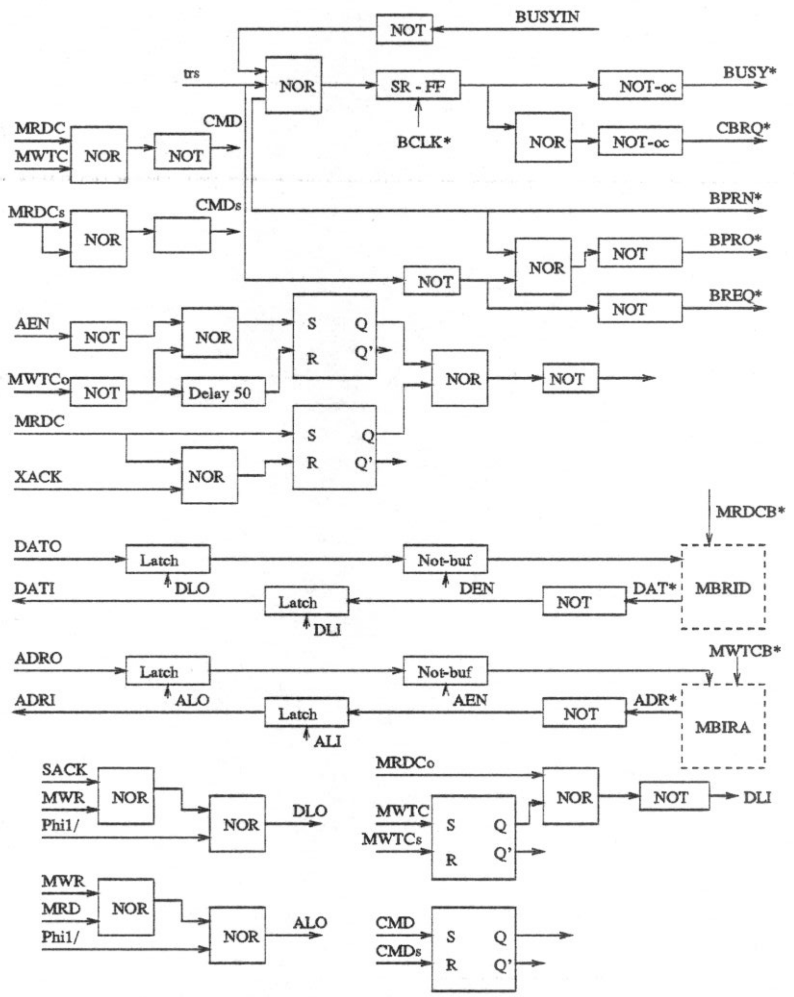

Figure 9: MDF Interface Circuit (part two)

Phase1 Clock period = $100ns$. It is one from 0 to $20ns$.

Phase2 Clock period = $100ns$. It is one from 40 to $90ns$.

The internal clock of the Multibus (signal $BCLK*$) has a period of $100ns$ and we assume that $BCLK*$ stays high (value one) between 0 and $50ns$.

6.1.2 Master Read

The timing diagram in Figure 10 specifies the temporal behavior of the operation. The master generates a pulse of at least one clock cycle wide (signal *MRD*) to initiate a read transaction. The interface circuit will transfer this request to the Multibus interface and will initiate a Multibus arbitration and read transaction cycle. An acknowledge pulse (signal *MACK*) exactly one cycle wide and the data read (signal *DATI*) will be returned as a response. The complete temporal specification of the operation in ATCSL, based on Figure 10, follows:

Timing Relations for a Master Read Operation of MDF

FOR (i; 0; #DATI.s; 1) {

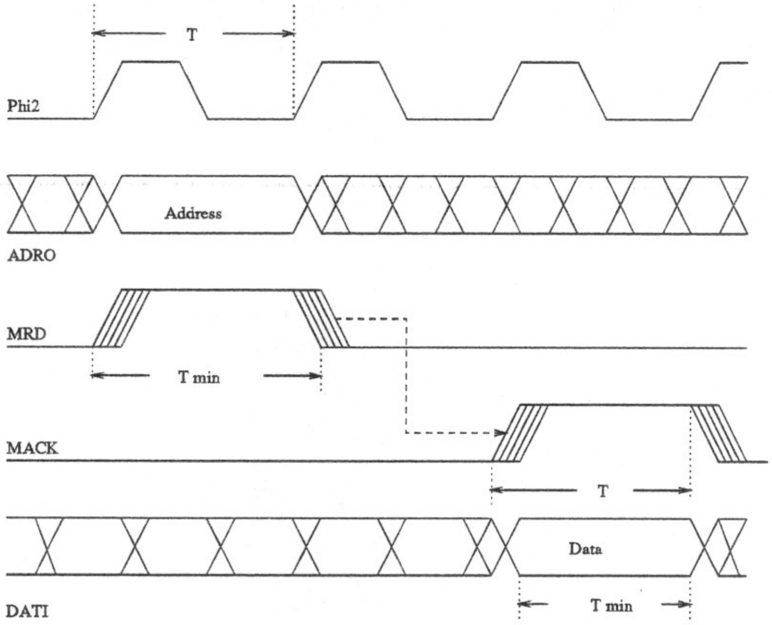

Figure 10: Master Read Timing Diagram

```
        DATI[i].s SYNC Phi2.1;
        DATI[i+1].c SYNC Phi2.1;
}
_DATI.s ≥ 100;
FOR (i; 0; #MACK.1; 1) {
        THERIS j [0 : #Phi2.1]
        (MACK.1[i] SYNC Phi2.1[j] && MACK.0[i] SYNC Phi2.1[j+1])
}
```

Dependency Relations for a Master Read operation of MDF

$MRD.f \Longrightarrow MACK.1$

6.1.3 Analysis and Verification

We should point out that the actual MDF design was implemented in VLSI [5]. The design we verify here is a TTL-based implementation. As a result, the timing violations found during the verification process do not necessarily apply to the actual implementation.

The input vector for the analysis of a read cycle consists of a read command signal and an address:

MRD This signal initiates a read cycle. It is synchronous to the *Phi2* clock and should be active for at least one cycle of the clock.

ADRO Along with the read command an address should be provided for the value to be read. As we said, the address lines are grouped into one signal and they assume the stable value as long as the read signal *MRD* is active.

At the initiation of the read cycle, signals *XACK** and *MRDC** are in their inactive state (value one). Data and address lines are undriven. Furthermore we assume that the requesting master has the highest priority (during the arbitration).

The simulator runs for sixteen clock cycles before its event queue becomes empty. As we said in Section 4, standard set-up and hold time constraints can be verified during the simulation process. Four set-up time constraint violations were reported at the end of the simulator run. As an example, we present below the report for the set-up constraint violation of the synchronizer (an ALR technology flip-flop) *_sync0*.

```
_sync0: SET_UP time Violation -- set_up_time = 10ns
The 0 instance of the input event assumes value: c at time: 1205
The clock signal assumes the value one(1) at time: 1240
```

The derived event graph is used for the verification process and the results are reported below, starting with violations of the timing relations.

```
**** Signal MACK.1 never appears
```

The *MACK.1* signal is missing because of the set-up time violations on synchronizer SYNC0 and consequently on the flip-flop D_FF0.

```
**** DATI.c[1] SYNC Phi2.1
Index: 1 for Signal DATI.c does not appear: from: 0 to: 0
```

The violated relation shows that the data lines are not properly released at the end of the read cycle. That happens because the latch with input signal *DATO* is not latched correctly when its inputs are undriven.

The report for the dependency relations follows:

```
**** Signal MACK.1 never appears
```

Again *MACK.1* signal is missing as a result of the set-up time constraints we detected earlier.

In the clock specification section we mentioned that the widths of the clock phases are controllable by the user. Using the case analysis feature of CLOVER, we were able to analyze and verify the MDF design for various clock pulse widths. By doing case analysis the designer can better understand the temporal characteristics of his or her design. One of the interesting outcome of the case analysis was the following: we modified the pulse width of the clock phase *Phi2* from $50ns$ to $40ns$; after analyzing and verifying the design we came up with the same set of violations except in one case. The following constraint of the master read operation was verified:

```
FOR (i; 0; #DATI.s; 1) {
    DATI[i+1].c SYNC Phi2.1;
}
```

In this case, the circuit at the end of a read transaction cycle correctly releases the data lines. That happens because the latch with input signal *DATO* properly latches its input signal.

The table below, presents verification statistics and execution time information for the different phases of the analysis and verification of a complete slave and master read operation of MDF.

MDF Interface	
Number of Modules	79
PDL-e Description to Netlist	4.1s + 3.1u CPU seconds
Simulator	1.62s + 1.02u CPU seconds
Size of Event Graph	1540 signals
ATCSL parsing	0.3s + 0.4u CPU seconds
Verifier	0.9s + 0.8u CPU seconds
Set-up Constraints Violations	4
Missing Signals Due to Set-up Violations	5
Timing Relation Violations	4
Dependency Relation Violations	1

7 Conclusions

A prototype for CLOVER has been built and it consists of about 11,000 lines of C code. It includes all features except hierarchical verification. The specification language, written with *yacc* [19], is easily extentable in case we decide that a new operator will be particularly useful. We believe that CLOVER provides a powerful automated environment, which permits the designer to describe his design in a well-structured way, hierarchically and at different abstraction levels. The specification model provides the expressive power for the user to specify timing constraints at the event and signal level and thus understand the design more deeply. The verification time is linearly dependent on the event graph size (nodes and edges) times the number of the specified timing constraints. Of course, if case analysis is done, the running time becomes proportional to the number of cases. A lot of timing constraints can be verified using the more abstract values of our value system.

Right now to run the simulator we provide the initial events and their starting times at the circuit inputs. An interesting problem is to give as inputs to the simulator only partially defined values: for example, the signal A has the value one from time zero to time x, where x is a variable. Given the timing constraints and the input events, the verifier should compute (if this is possible), a solution for x, probably in the form of an inequality. A (possibly empty) set of solutions will determine under which inputs the implementation meets the specification constraints. Another interesting but even harder problem, is the design of a more elaborate and sophisticated error report. This is highly related to design audit (see Section 2): the tool tries to give hints back to the designer as to where in the circuit a problem may be found, after a constraint is violated.

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

References

[1] T. Amon, G. Boriello, A. Sequin, and W. Winder. A Unified Behavioral/Structural Representation for Simulation and Synthesis. In *4th International High-Level Synthesis Workshop (ACM/IEEE)*, Kennebunkport, ME, October 1989.

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

[3] H. G. Barrow. Proving the Correctness of Digital Hardware Designs. *VLSI Design*, July 1984.

[4] G. V. Bochmann. Hardware Specification with Temporal Logic: An Example. *IEEE Transactions on Computers*, 31, March 1982.

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

[6] Gaetano Boriello. *A New Interface Specification Methodology and its Application to Transducer Synthesis*. PhD thesis, University of California, Berkeley, May 1988.

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

[8] R. E. Bryant. A Switch-Level Model and Simulator for MOS Digital Systems. *IEEE Transactions on Computers*, 34, February 1984.

[9] R. E. Bryant. Can a Simulator Verify a Circuit. In G. J. Milne and P. A. Subramanyan, editors, *Formal Aspects of VLSI Design*. Elsevier Science Publishers, 1986.

[10] E. J. Cameron et al. The IC* Model of Parallel Computation and Programming Environment. *IEEE Transaction on Software Engineering*, 14, March 1988.

[11] T. J. Chaney and C. Molnar. Anomalous Behavior of Synchronizer and Arbiter Circuits. *IEEE Transactions on Computers*, 23, April 1973.

[12] D. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Workshop on Verification of Finite State Systems*, Granoble, France, 1989.

[13] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1988.

[14] M. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. A. Subramanyan, editors, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1988.

[15] F. K. Hanna and N. Daeche. Specification and Verification Using Higher-Order Logic: A Case Study. In G. J. Milne and P. A. Subramanyan, editors, *Formal Aspects of VLSI Design*. Elsevier Science Publishers, 1986.

[16] R. B. Hitchcock. Timing Verification and the Timing Analysis Program. In *19th Design Automation Conference*, 1982.

[17] Apple Computer Inc. *Designing Cards and Drivers for MacII and Mac SE*. Inside Macintosh Library. Addison Wesley, 1987.

[18] Intel Corporation. *Intel Multibus Specification*, 1982.

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

[20] A. Kara, R. Rastogi, and K. Kawamura. An Expert System to Automate Timing Design. *IEEE Design & Test of Computers*, October 1988.

[21] K. Karplus. Exclusion Constraints for Digital MOS Circuits. In *MIT Conference in Advanced Research in VLSI*, 1986.

[22] Kolodny, Friedman, and Ben-Tzur. Rule-Based Static Debugger and Simulation Compiler. In *International Conference on Computer-Aided Design*, pages 150–152, 1985.

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

[24] A. Martello, S. Levitan, and D. Chiarulli. Timing Verification Using HDTV. In *27th Design Automation Conference*, 1990.

[25] M. C. McFarland. CPA: Giving an Account of Timed System Behavior. In *TAU 1990 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, Vancouver, Canada, August 1990.

[26] P. McGeer and R. K. Brayton. Provably Correct Critical Paths. In *CalTech Decennial Conference on VLSI*, April 1989.

[27] T. McWilliams. *Verification of Timing Constraints on Large Digital Systems*. PhD thesis, Lawrence Livermore Laboratory, May 1980.

[28] J. A. Merlin and D. J. Farber. Recoverability of Communication Protocol-Implications of a Theoretical Study. *IEEE Transaction on Communications*, 24, September 1976.

[29] G. J. Milne. Timing Constraints: Formalizing their Description and Verification. In *Computer Hardware Description Languages and their Applications, Proceedings of the 9th IFIP Symposium, June 1989*, June 1989.

[30] D. Misunas. Petri Nets and Speed Independent Design. *Communication of the ACM*, 16, 1973.

[31] B. Moszkowski. A Temporal Logic for Multilevel Reasoning About Hardware. *IEEE Computer*, February 1985.

[32] L. W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, University of California, Berkeley, May 1975.

[33] S. Nowick and D. Dill. Practicality of State-Machine Verification of Speed-Independent Circuits. In *International Conference on Computer-Aided Design*, 1989.

[34] M. Rem et al. Trace Theory and the Definition of Hierarchical Components. In R. Bryant, editor, *Proceedings of the Third Caltech Conference on VLSI*, pages 225–239. Computer Science Press, 1983.

[35] Robert Sedgewick. *Algorithms*. Addison Wesley, 2nd edition, 1988.

[36] Charles L. Seitz. *Introduction to VLSI Systems*, chapter System Timing (7). Addison Wesley, 1980.

[37] Spickelmier and Newton. Critic: A knowledge-Based Program for Critiquing Circuit Designs. In *International Conference on Computer Design*, pages 324–327, 1988.

[38] I. Suzuki and H. Lu. Temporal Petri Nets and Their Application to Modeling and Analysis of a Handshake Daisy Chain Arbiter. *IEEE Transaction on Computers*, 38, May 1989.

[39] D. Wallace and C. Sequin. Plug-In Timing Models for an Abstract Timing Verifier. In *23rd Design Automation Conference*, 1986.

[40] D. W. Weise. *Formal Multilevel Hierarchical Verification of Synchronous MOS VLSI Circuits*. PhD thesis, MIT, June 1987.