Diagrams and Computational Efficacy

Kathi Fisler
Department of Computer Science
Rice University
6100 S. Main, MS 132
Houston, TX 77005-1892 USA

Studies of diagrammatic reasoning have tended to focus on cognitive or logical issues, but rarely on computational ones. A computational view of diagrams studies how diagrammatic representations affect solutions to computational problems. This paper presents a computer scientist's view on diagrams research. As a case study, it considers the role of diagrams in hardware verification, an area with a long tradition of using diagrams in informal reasoning. We discuss several ongoing projects and some emerging results on the benefits of diagrams in this problem domain. We also identify features of diagrams that might explain our findings.

1 Introduction

Diagrams are an important tool in problem solving. In domains from mathematics to architecture to political science to engineering, people develop and communicate ideas through diagrams. This pervasive use in such widespread areas has inspired much research into the nature of reasoning with diagrams. This research generally takes one of two perspectives: the cognitive or the logical. The cognitive perspective explores diagrams' role in human thought processes. The logical perspective explores the mathematical nature of diagrammatic representations. These perspectives are complementary, and help paint a broad picture of diagrammatic reasoning.

Relatively little work, however, has presented a *computational* perspective of diagrammatic reasoning. Such a view is sorely lacking. A plethora of computer-based tools support problem solving in a variety of domains. Recognizing the cognitive advantages of visual information, many of these tools employ visual interfaces in order to attract

Submitted for
Proceedings of LLC8
D. Barker-Plummer, D. Beaver, J. van Benthem, P. Scotto di Luzio (eds.)

and support users. Beneath the interface, however, these tools generally translate the diagrams into existing textual representations before running the problem solving algorithms. This approach is sound if the translation is sound[1], but could the computer-based algorithms operate more effectively on the diagrammatic representations themselves? Diagrams suggest certain data structures; how do these data structures compare computationally to those that traditionally capture textual representations? The computational view on diagrammatic reasoning addresses these questions by exploring the potential computational benefits, or *computational efficacy*, of diagrammatic representations.

This paper presents both a primer to computational efficacy in diagrammatic reasoning and some results arising from our own research in this area. Section 2 describes computational efficacy and motivates our choice of hardware verification as a case study. Section 3 provides an introduction to hardware design and verification and their diagrammatic representations. Section 4 describes our ongoing research projects and some results. Sections 5 and 6 summarize our perspectives on diagrammatic reasoning as gained from these projects and offer concluding remarks, respectively.

2 Computational Efficacy and a Case Study

Computational efficacy studies the interplay between representations and the computational resources, such as time and memory, required to process them. This focus on resource requirements distinguishes a computational perspective on diagrammatic reasoning from both the cognitive and logical ones. Unlike the cognitive view, the computational perspective studies machine-based, rather than human, information processing. Computational studies therefore yield more mathematically grounded analyses than cognitive ones.

The differences between the computational and logical perspectives are more subtle. In essence, logic studies representation, while computer science studies how representation affects computation. For example, a logician might study completeness and compactness, whereas decidability and descriptive complexity are more important to a computer scientist. The two views are by no means independent: both study expressibility, certain algorithms require complete logics, and logicians study proof theory, which has a computational flavor. Nonetheless, these areas emphasize different criteria of logics and representations, and those differences should be significant to the diagrammatic

[1] Unfortunately, developers rarely address this issue.

community. Whereas a logician might study how diagrams affect our notions of information and inference, a computer scientist might ask whether diagrams enable machines to solve particular problems more effectively. Lemon and Pratt [Lemon and Pratt (1997)] illustrate both logical and computational views on diagrams in spatial reasoning.

Exploring the computational efficacy of diagrams requires a set of problems to study in a domain for which diagrams provide suitable representations. Candidate domains and problems should meet three criteria. First, the domain should use diagrams in interesting ways: if the diagrams are essentially isomorphic to existing textual notations, then the diagrams are unlikely to offer computational benefits. Second, the problems should be computationally difficult, so that the relative merits of different representations are more readily apparent. Finally, both the domain and the problems should be important in order to justify the search for new and improved representations.

Hardware design and verification satisfy all three criteria. Hardware designers have a long history of using diagrams, albeit in informal contexts. This practice has persisted, despite the largely textual nature of early computer-aided design tools. This suggests that, at least on a cognitive level, diagrams play roles unmatched by existing textual representations. Whether they play similar roles at a computational level is an interesting and open question for computer science. Hardware design also contains many important and computationally challenging problems. Verification, the process of proving that designs satisfy certain behavioral requirements, is one example. Verification is important because it helps guarantee that the designed circuits will not malfunction. It is also extremely difficult, as its computational requirements often surpass the available resources of modern computers.

Our study of the computational efficacy of diagrams in hardware verification uses a logic of design diagrams that we describe in Section 3. Currently, our project asks the following questions:

- How expressive is our logic relative to other verification logics?
- How concise are statements in our logic relative to other logics?
- Is verification decidable in our logic?
- Does translating our diagrams to text yield larger instances of verification problems than if we processed the diagrams directly?
- How do proofs in our logic compare to ones of the same properties in other logics with respect to proof-theoretic properties?

Section 4 describes these projects in more detail.

3 Hardware Design and Verification

Hardware design is the process of creating an electronic circuit to process information for a particular task. A typical hardware design involves hundreds or thousands of interacting components that collectively perform the intended task. The interactions between these components can be complex and subtle. As a result, they often yield unexpected, and erroneous, system behavior. Given our increasing reliance on computer technology, techniques for locating and eliminating these errors are of increasing importance. Unfortunately, the sheer size of modern designs makes this task extremely difficult. A modern design can contain millions of transistors, which leads to a system with far more states than there are atoms in the universe. Developing so-called validation techniques to analyze and debug designs on this scale is one of the most challenging problems facing modern hardware design.

Verification is a validation technique that brings logic and proof to bear on error detection [Clarke et al. (2000); Manna and Pnueli (1995)]. In verification, designers state properties that a design should satisfy, then formally prove that a model of the design satisfies those properties. For example, a design for a traffic-light controller should satisfy at least the following two properties:

- Green lights are never on in both directions simultaneously.
- If a car is waiting at a red light, that light eventually turns green.

This example presents two flavors of properties: they can express either invariants about the design or various forms of progress guarantees. For verification, both the design model M and the desired property φ must be expressed in some formal notation. Verification then entails proving that $M \models \varphi$. Specific verification techniques vary from model-theoretic to proof-theoretic, and employ a range of heuristics to control the size of the model to be checked. A typical verification task can take from several seconds to several months to complete, depending upon the complexity of the design, the complexity of the property, and the particular techniques that apply to the problem at hand.

Most modern verification tools capture models and properties using textual logics. Hardware design, in stark contrast, uses a wide array of representations, many of them diagrammatic. A design encompasses a vast array of information of varying sorts. Each design representation excels at capturing some aspect or set of aspects of a design. Figure 1.1 shows examples of three common diagrammatic design representations: state machines (left) capture flow of control; circuit diagrams (center) capture layout and implementation details; and timing diagrams (right)

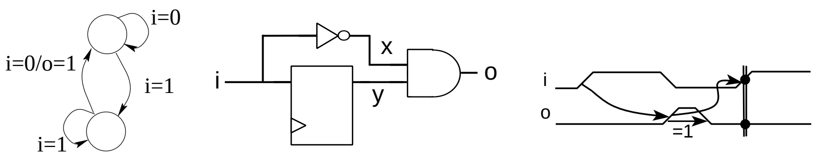


Figure 1.1: A collection of design representations.

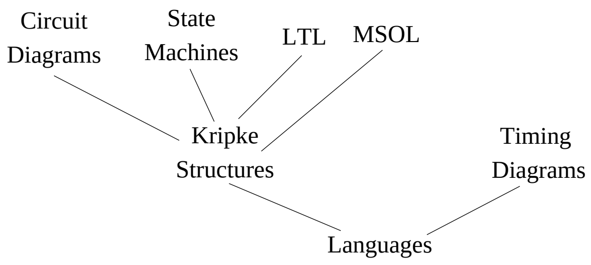


Figure 1.2: A heterogeneous hardware logic.

capture temporal relationships. A single design generally involves information given in several of these, as well as other, representations.

This representation gap between design and verification complicates the designer-controlled aspects of verification. Designers are more comfortable with their highly evolved design notations than with verification logics. Furthermore, designers develop intuitions about their designs in terms of their original notations; once a design has been translated to another notation, designers find reasoning about it more difficult. This is problematic both in guiding the verification effort, and in locating the sources of errors found during verification.

In an effort to bridge this gap, many verification tool developers now provide diagrammatic interfaces to their tools. While these interfaces don't support the full suite of design notations, they provide some support, such as a timing diagram interface for specifying properties [Damm et al. (1995)]; this frees the designer from part of the task of manually translating their ideas into verification notations. At a minimum, creating the interfaces requires developing a well-defined syntax and semantics for every supported design representation, as well as sound translations between representations. While this approach allows developers to leverage off of existing tools, it bypasses any potential computational efficacy of hardware diagrams. Our work asks whether we lose any computational benefits by taking this approach.

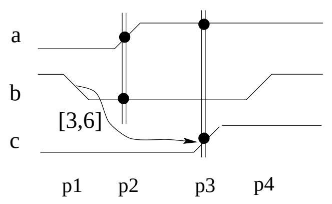
We have developed a logic of hardware representations in order to study this question [Fisler (1996)]. Figure 1.2 provides an overview of the logic. It includes five syntactic representations: the three diagrammatic notations from Figure 1.1, monadic second-order logic (MSOL), and a linear-time temporal logic (LTL). The logic uses two levels of semantic models: Kripke structures for four of the representations, and a general language model to encompass timing diagrams, which are more expressive than Kripke structures. Inference rules in the logic are proven sound relative to the greatest common semantic model between the representations involved. The projects described in this paper primarily use timing diagrams and LTL; we describe each in more detail in the following sections.

3.1 Timing Diagrams

Timing diagrams express patterns of value changes on signals. In addition, they express precedence, synchronization, and timing constraints between changes. Figure 1.3 shows a sample timing diagram in our logic. Variables a, b, and c name boolean-valued signals. To the right of each name is a *waveform* depicting how the variable's value changes over time. For example, b transitions from high to low, then later returns to high. The terms *low* and *high* correspond to voltage levels; in our logic, we interpret low as logical false and high as logical true.

Vertical parallel lines indicate synchronization. Figure 1.3 requires b to be low when a's value rises. A transition or a synchronization is called an *event*. Arrows indicate temporal ordering between events. Annotations of the form $[l, u]$ on the arrows indicate discrete lower and upper bounds on the time between the related events; l and u may consist of natural numbers, variables, and addition and subtraction expressions over them, as well as the symbol ∞. The annotation "$=n$" abbreviates $[n, n]$. The labels at the bottom, referred to as *time points*, are for explanatory purposes and are not part of the timing diagram.

Since timing diagrams express sequences of values on variables over time, an appropriate semantic model for them must do the same. Formal languages, which are sets of sequences over a given alphabet, suggest an appropriate semantic model. Our semantics considers finite or infinite words over an alphabet consisting of all possible assignments of boolean values to the names labeling waveforms. Intuitively, a word models a timing diagram when the transition patterns in the diagram reflect the changes in values assigned to names in the word. This paper provides an intuitive description of the semantics; the full details appear elsewhere [Fisler (1999)].



a	0	0	0	1	1	0	1	1	1	1	0	1	1
b	1	0	0	0	1	0	0	1	0	1	0	0	1
c	0	0	0	0	0	1	0	1	0	0	0	0	1
	0	1	2	3	4	5	6	7	8	9	10	11	12
	p_1		p_2	p_4	p_3								
					p_1	p_2	p_3, p_4						

Figure 1.3: A timing diagram and an illustration of its semantics.

Consider the timing diagram and (random) word in Figure 1.3. The word appears in tabular form: the waveform names label the rows and the indices into the word label the columns. Each cell in the table indicates the value on the corresponding signal at the corresponding index. Symbols 0 and 1 denote logical false and true, respectively. The two lines directly beneath the table indicate two separate assignments of indices to time points, as explained shortly.

Intuitively, the semantics walks along a word looking for indices that satisfy each time point. An index satisfies a time point if the values assigned to each name in the index correspond to those required by the events at the time point; for transition events, satisfaction relies on both the current index and its immediate successor. For example, if a time point p_i contains a rising transition on signal a, index d satisfies p_i if d assigns value 0 to a and index $d + 1$ assigns value 1 to a.

Verification properties often have the form $\varphi \rightarrow \psi$; we would therefore like our timing diagrams to express some notion of implication. For example, we might want the behavior shown in the timing diagram in Figure 1.3 to occur only when the first event, the falling transition on b, occurs. We use a parameter that is external to the diagram to indicate which prefix, if any, of the diagram should be treated as a trigger for satisfying the entire diagram. We call the portion of a timing diagram defined over these time points the *assume portion* of the diagram. Taking the assume portion in Figure 1.3 to contain only p_1 treats a falling transition on b as the trigger.

For the word and timing diagram in Figure 1.3, index 0 satisfies the falling transition on b. The walk continues, now searching for an index satisfying the synchronization line at p_2. According to this line, b must be low when a rises. However, is b required to be low *until a* rises? The desirability of the "until" interpretation is heavily context-dependent. A second parameter therefore indicates which segments of waveforms should be matched exactly within words. Such segments are called *fixed-level constraints.* For this example, assume we have only one such constraint: that a must remain high between p_2 and p_3. Index 2 satisfies the first synchronization line.

Now, we search for indices satisfying either p_3 or p_4. Although p_3 appears before p_4 in the diagram, the semantics allows them to occur in either order because no relationship is explicitly stated between their events. Index 3 satisfies p_4 and index 4 satisfies p_3. Once finding an index that satisfies p_3, the semantics must also check whether the indices assigned to p_1 and p_3 respect the bounds given on the successor edge. If not, the walk fails to match the timing diagram. The first row below the table contains this first assignment of indices to time points. Verification properties are often invariant over a system; thus, we need to check for the timing diagram pattern starting from each index containing a rising transition on b (indices 4, 7, and 9). The second line beneath the table shows a walk on which the semantics fails (from index 4); thus, this word does not satisfy the timing diagram.

Thus, our semantics defines when a timing diagram describes the sequences in a given language (*i.e.*, a set of words). A *timing diagram language* is any language that some timing diagram describes. One of our projects in Section 4 discusses characteristics of timing diagram languages in more detail.

3.2 Linear-time Temporal Logic (LTL)

Like timing diagrams, linear-time temporal logic describes patterns of changes in variables over sequences of assignments. LTL is a propositional temporal logic. Its syntax and semantics are defined relative to a finite set of propositions $\mathcal{P}$. The formulas of LTL include $\mathcal{P}$ and are closed under unary operators $\neg$ and X (next), and binary operators $\vee$ and U (until). Intuitively, $\mathsf{X}\varphi$ says that φ holds in the next state, while $\varphi\mathsf{U}\psi$ says that φ holds in every state until ψ holds, and ψ eventually holds. Other temporal operators, such as G (something holds in all states) are defined in terms of U.

Verification of LTL formulas relies on a well-known correlation between LTL and Büchi automata [Vardi and Wolper (1986)]. A Büchi automaton is a tuple $\langle Q, \Sigma, q_0, \delta, \mathcal{F}\rangle$ where Q is a set of states, Σ is an input alphabet, $q_0 \in Q$ is the initial state, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $\mathcal{F} \subseteq Q$ is a set of states (called the *fair* states) used to determine acceptance. Given a design represented as a Büchi automaton A and an LTL formula F, verification consists of checking whether the language generated by A is contained in that accepted by F. In other words, LTL verification reduces to testing a language containment of the form $\mathcal{L}(A) \subseteq \mathcal{L}(F)$; verification tools implement this check as $\mathcal{L}(A) \cap \overline{\mathcal{L}(F)} = \emptyset$. This check is decidable for a large class of verification problems. The first step of this check involves compiling the negation of formula F into a Büchi automaton. One of our projects looks at this compilation step as a means of comparison between LTL and timing diagrams.

4 Exploring Computational Efficacy

This section describes four specific projects that explore the computational efficacy of hardware diagrams. We discuss the role of hardware diagrams in interactive theorem proving, the expressiveness of timing diagrams relative to LTL, the decidability of verification of timing diagram properties, and two methods for compiling timing diagrams to Büchi automata for purposes of verification. For each problem, we identify the features that distinguish the diagrammatic representations from their textual counterparts.

4.1 Interactive Theorem Proving

In interactive theorem proving, users guide tools in developing proofs (using natural deduction or the sequent calculus, for example) that models satisfy properties. Designers use theorem proving when a verification problem is either undecidable or too complex for automated techniques to handle effectively. We are interested in comparing natural deduction proofs developed in our heterogeneous logic with those developed in textual logics. In particular, we are interested in whether the heterogeneous proofs are longer or shorter, or whether they involve more or less branching into subcases than purely textual proofs.

This section presents an example verification of a simple device known as a single pulser. A single pulser has one input and one output. For each arbitrary but finite length pulse on its input wire, the single pulser produces a unit-duration pulse on its output wire. Such a device is useful, for example, to convert long depressions of a call button on an elevator to single requests in the underlying hardware. The defining property of a single pulser is simple: the device should produce exactly one unit duration output pulse between each pair of consecutive input pulses. The timing diagram in Figure 1.1 captures this property.

The circuit diagram in Figure 1.1 implements a single pulser. In the diagram, the triangular icon represents logical negation, the rounded icon represents conjunction, and the rectangular icon represents a unit delay. The latter device updates its value in regular intervals (defined by a system clock). In each interval, the value on the output is the value that was on the input in the previous interval. The lines connect inputs and outputs of these smaller devices to one another; the inputs of each device enter on the left side, while the outputs are on the right.

We wish to contrast two proofs that the circuit diagram satisfies the property expressed in the timing diagram. We perform one proof in our heterogeneous logic (the inference rules and their soundness proofs are in the style of those in Hammer [Hammer (1996)] and appear elsewhere [Fisler (1996)]). For the second proof, we translate both diagrams into first-order logic formulas and use first-order logic inference rules to prove that the circuit diagram formula implies the timing diagram formula. Converting the two diagrams into first-order logic formulas is straightforward. For the circuit diagram, we write a formula that captures each gate in the diagram, then name the inner wires via existentially quantified variables.

$$\begin{array}{rcl}
delay(i,o) & \equiv & \forall t : o(t+1) = i(t) \\
inverter(i,o) & \equiv & \forall t : o(t) = \neg i(t) \\
and(i_1,i_2,o) & \equiv & \forall t : o(t) = i_1(t) \wedge i_2(t) \\
SP(i,o) & \equiv & \exists x \exists y : delay(i,y) \wedge inverter(i,x) \wedge and(x,y,o)
\end{array}$$

For the timing diagram, we introduce an existentially-quantified time variable for each event in the diagram and capture the relationships between the signals around each time variable. In the following formula, t' captures the falling transition on i, t'' captures the second rising transition on i, and t_1 captures the rising transition on o.

$$\begin{array}{l}
\forall t : (\neg i(t) \wedge i(t+1) \wedge \exists t' : t' > t \wedge i(t') \wedge \neg i(t'+1) \wedge \\
\qquad\qquad\qquad\qquad \forall t_h : t < t_h \wedge t_h < t' : i(t_h) \wedge \\
\qquad\qquad\qquad\qquad \exists t'' : (t'' > t' \wedge \neg i(t'') \wedge i(t''+1)) \wedge \\
\qquad\qquad\qquad\qquad\quad \forall t_l : t' > t_l \wedge t_l \leq t'' : \neg i(t_l)) \rightarrow \\
\quad \exists t_1 : t_1 > t \wedge \underline{\neg o(t_1)} \wedge \underline{o(t_1+1)} \wedge \underline{\neg o(t_1+2)} \wedge \\
\qquad (\forall t_2 : (t_2 \geq t \wedge t_2 \leq t_1) \rightarrow \underline{\neg o(t_2)}) \wedge \\
\qquad \forall t_3 : (t_3 > t_1 + 1 \wedge t_3 \leq t'' \rightarrow \underline{\neg o(t_3)})
\end{array}$$

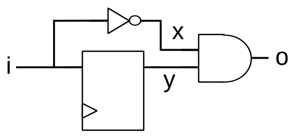
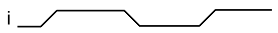
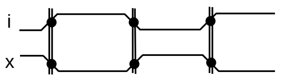
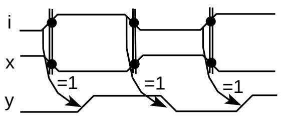
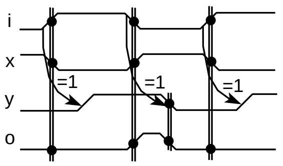
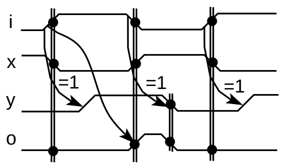
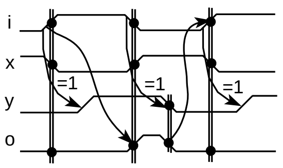
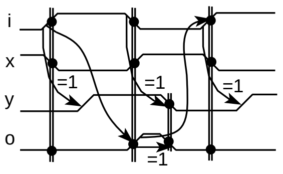
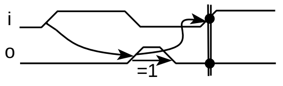
1.		Given
2.		Given
3.		Inverter Rules: 1, 2
4.		Delay Rules: 1, 3
5.		And Rules: 1, 4
6.		Bounds Slide: 5
7.		Bounds Slide: 6
8.		Bounds Slide: 7
9.		Simplification: 8

Figure 1.4: The diagrammatic single pulser proof.

1.	$SP(i, o)$	Given
2.	$delay(i, y) \wedge inverter(i, x) \wedge and(x, y, o)$	$\exists$ Elim: 1
3.	$\neg i(t) \wedge i(t+1) \wedge$	Assume
	$\exists t' : t' > t \wedge i(t') \wedge \neg i(t'+1) \wedge$	
	$\forall t_h : t < t_h \wedge t_h < t' : i(t_h) \wedge$	
	$\exists t'' : (t'' > t' \wedge \neg i(t'') \wedge i(t''+1)) \wedge$	
	$\forall t_l : t' > t_l \wedge t_l \leq t'' : \neg i(t_l)$	
4.	$t' > t \wedge i(t') \wedge \neg i(t'+1)$	$\exists$ Elim: 3
5.	$y(t'+1)$	Def *delay*: 2, 4
6.	$x(t'+1)$	Def *invtr*: 2, 4
7.	$o(t'+1)$	Def *and*: 2, 5, 6
8.	$\neg x(t')$	Def *invtr*: 2, 4
9.	$\neg o(t')$	Def *and*: 2, 8
10.	$\neg y(t'+2)$	Def *delay*: 2, 4
11.	$\neg o(t'+2)$	Def *and*: 2, 10
12.	$t_2 \geq t \wedge t_2 \leq t'$	Assume
13.	$i(t_2)$	$\forall$ Elim: 3
14.	$\neg x(t_2)$	Def *invtr*: 2, 13
15.	$\neg o(t_2)$	Def *and*: 2, 14
16.	$t_2 \geq t \wedge t_2 \leq t' \rightarrow \neg o(t_2)$	$\rightarrow$ Intro: 12, 15
17.	$\forall t_2 : t_2 \geq t \wedge t_2 \leq t' \rightarrow \neg o(t_2)$	$\forall$ Intro: 16
18.	$t'' > t' \wedge \neg i(t'') \wedge i(t''+1)$	$\exists$ Elim: 3
19.	$t_3 > t' \wedge t_3 \leq t''$	Assume
20.	$\neg i(t_3)$	$\forall$ Elim: 3, 19
21.	$\neg y(t_3+1)$	Def *delay*: 2, 20
22.	$\neg o(t_3+1)$	Def *and*: 2, 21
23.	$t_3 > t' \wedge t_3 \leq t'' \rightarrow \neg o(t_3+1)$	$\rightarrow$ Intro: 19, 22
24.	$\forall t_3 : t_3 > t' \wedge t_3 \leq t'' \rightarrow \neg o(t_3+1)$	$\forall$ Intro: 23
25.	$t' > t \wedge \neg o(t') \wedge o(t'+1) \wedge \neg o(t'+2) \wedge$	$\wedge$ Intro: 4, 7,
	$\forall t_2 : t_2 \geq t \wedge t_2 \leq t' \rightarrow \neg o(t_2) \wedge$	9, 11, 17, 24
	$\forall t_3 : t_3 > t' \wedge t_3 \leq t'' \rightarrow \neg o(t_3+1)$	
26.	$\exists t_1 : t_1 > t \wedge \neg o(t_1) \wedge o(t_1+1) \wedge$	$\exists$ Intro: 25
	$\neg o(t_1+2) \wedge$	
	$\forall t_2 : t_2 \geq t \wedge t_2 \leq t_1 \rightarrow \neg o(t_2) \wedge$	
	$\forall t_3 : t_3 > t_1 \wedge t_3 \leq t'' \rightarrow \neg o(t_3+1)$	
27.	[line 3 $\rightarrow$ previous line]	$\rightarrow$ Intro: 3, 26
28.	$\forall t$[previous line]	$\forall$ Intro: 27

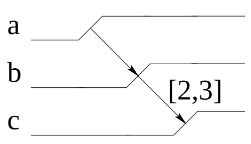
Figure 1.5: The textual single pulser proof. In the justifications, *invtr* abbreviates *inverter*.

Figures 1.4 and 1.5 show the two proofs. Each proof uses core inference rules of its respective logic, rather than derived lemmas, so the granularity of reasoning between the proofs is comparable. The diagrammatic proof is much shorter, requiring only eight steps as opposed to twenty-eight for the textual proof. The main difference lies in how each proof handles information about a single signal. The textual proof must establish several statements about the signal o; these statements, although derived from the same lemmas about the relationship between o and the other signals, are spread throughout the formula representing the timing diagram, as shown by the underlining in the formula. The diagrammatic proof, in contrast, handles o in one step because all information about o is localized within a single waveform; in this sense, the timing diagram is inherently conjunctive. The two-dimensional nature of timing diagrams controls the length of the diagrammatic proof: we can access information specific to a particular time or information about an entire waveform quickly and easily.

The two proofs also differ in their degree of branching. The diagrammatic proof never branches into subcases, while the textual proof involves multiple layers of subproofs. The textual proof requires a new subproof for each implication in the property formula. The implications capture requirements about particular regions of the original timing diagram. Since the timing diagram represents all regions simultaneously, it does not require additional subproofs to establish statements about individual regions. Thus the two-dimensional nature of timing diagrams affects the branching degree as well as the length of the diagrammatic proof. Multi-dimensionality therefore appears to be a key factor in the computational efficacy of timing diagrams.

4.2 Timing Diagram Expressiveness

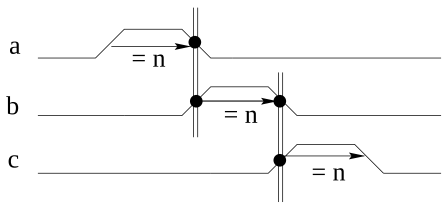
Section 3.1 defines a timing diagram language as a formal language that models a timing diagram. We would like to identify any interesting characteristics of timing diagram languages that might distinguish them from other classes of languages. The Chomsky hierarchy defines several layers of languages, such as regular, context-free, and context-sensitive, each of which corresponds to a different model of computational machines. Timing diagram languages defy characterization by the Chomsky hierarchy: they capture languages at each level of the hierarchy, but cannot fully capture any single layer. For example, timing diagram languages cannot capture simple disjunctions, which arise in some regular languages. They can capture context-free and context-sensitive languages through their use of parameters in timing



$$\begin{array}{l}
[(\mathrm{FL}_0 \land \neg \mathit{rising}(a))\mathsf{U}(\mathrm{FL}_0 \land \mathit{rising}(a))] \rightarrow \\
[(\mathrm{FL}_0 \land \neg \mathit{rising}(a))\mathsf{U} \\
\ (\mathrm{FL}_0 \land \mathit{rising}(a) \land \\
\ \ \mathsf{X}[(\mathrm{FL}_1 \land \neg \mathit{rising}(b))\mathsf{U} \\
\ \ \ \ (\mathrm{FL}_1 \land \mathit{rising}(b) \land \\
\ \ \ \ \ \mathsf{X}(\mathrm{FL}_2 \land \neg \mathit{rising}(c) \land \\
\ \ \ \ \ \ \ \mathsf{X}((\mathrm{FL}_2 \land \mathit{rising}(c)) \lor \\
\ \ \ \ \ \ \ \ \ (\mathrm{FL}_2 \land \neg \mathit{rising}(c) \land \\
\ \ \ \ \ \ \ \ \ \ \mathsf{X}(\mathrm{FL}_2 \land \mathit{rising}(c))))))])]
\end{array}$$

Figure 1.6: A timing diagram and its translation to LTL, assuming that the assume portion contains only the rising transition on a. The translation captures one walk of the timing diagram semantics.

constraints. The following diagram, for example, captures the context-sensitive language $A^nB^nC^n$, where A, B, and C are the assignments $\langle a = 1, b = 0, c = 0\rangle$, $\langle a = 0, b = 1, c = 0\rangle$, $\langle a = 0, b = 0, c = 1\rangle$.



To date, two-way, one counter machines provide the closest match we have found between timing diagram languages and a computational machine model. These machines also capture languages across the Chomsky hierarchy; in particular, they capture all timing diagram languages. However, we have examples of two-way, one counter machines for which there is no timing diagram representation, so the correspondence is not exact. Whether timing diagram languages define a class of languages with distinct, yet interesting, properties from two-way one counter machine languages remains an open problem.

As timing diagrams provide a form of linear temporal logic, a comparison between timing diagrams and LTL is natural. LTL is contained in the regular languages. Thus, it is clear that LTL does not subsume timing diagrams. Timing diagrams do not subsume LTL either, since LTL can express simple propositional disjunctions, such as $p \lor q$. Thus, the two logics are expressively incomparable. However, timing diagrams that are interpreted as invariants and do not include parameters in their time bounds can be translated into LTL. The LTL expression for a timing diagram follows the semantics for a single walk over the diagram: if a sequence satisfies the assume portion of the diagram, then it must satisfy the entire timing diagram.

Figure 1.6 shows a timing diagram and its corresponding LTL formula. In the formula, the notation $rising(s)$ abbreviates $\neg s \wedge \mathsf{X}s$. FL_i denotes the fixed-level constraints in a segment of the diagram: FL_0 is the constraint before the transition on a, FL_1 covers the segment between the transitions on a and b, and FL_2 spans the transitions on b and c. We leave these in symbolic format to highlight the general form of the translation. The assume portion forms the antecedent of the implication: it is true when a rising transition has been found on a and the fixed level constraint FL_0 has held in all states preceding that transition. The format of this until statement guarantees that the LTL semantics uses the first rising transition on a, as required by the timing diagram semantics. In the consequent of the implication, the LTL expression looks for the entire diagram: the first rising transition on a, followed by the first rising transition on b, followed by the first rising transition on c. The first rising transition on c must occur either two or three steps after the one on b. The nestings of next-time operators X towards the end of the formula express this requirement.

This example shows one shortcoming of LTL as compared to timing diagrams. LTL is unable to share the common subexpression

$$(\mathrm{FL}_2 \wedge \neg rising(c) \wedge \mathsf{X}(\mathrm{FL}_2 \wedge rising(c)))$$

which appears twice towards the end of the expression (once for each allowable duration between the transitions on b and c). The timing diagram, in contrast, needs only one instance of the rising transition on c. Thus, timing diagram expressions can be more concise than their LTL counterparts when timing constraints are involved. The LTL expression also duplicates the assume portion on both sides of the implication. These differences may not be significant computationally, however, unless they affect the resources required by a verification algorithm. Section 4.4 examines this issue. The relative merits of LTL and timing diagrams remains an open question for future research.

4.3 Decidability of Verification

Section 3.2 describes how verification of linear temporal logic properties reduces to language containment. In a similar vein, verification of timing diagrams also reduces to language containment, though of a different class of property languages. Therefore, we need to ask whether containment of a Büchi automaton language (which captures a design) in a timing diagram language is decidable. We might expect this problem to be undecidable, because containment of a regular language in a context-sensitive language is known to be undecidable [Hopcroft and Ullman (1979)]. However, we have proven that the problem is actually decidable, regardless of where the timing diagram language resides in the Chomsky hierarchy [Fisler (1999)]. This result holds because timing diagrams localize where they count (*i.e.*, their non-regular behavior) to areas between particular events. This property is sufficient to render verification of regular language design models decidable for properties expressed in timing diagram languages.

4.4 Compilation to Büchi Automata

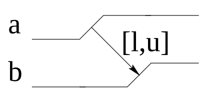
As discussed in Section 3.2, verification tools for LTL properties compile the negation of an LTL formula into a Büchi automaton. The timing diagram semantics effectively define a Büchi automaton accepting a timing diagram language. This suggests two routes to verifying timing diagram properties that are expressible in LTL: compile the timing diagram directly to an automaton, or translate the timing diagram into LTL and use the existing LTL compilation algorithms.[2] We would like to compare the automata arising from these two approaches. Is one substantially larger than the other? Size is important because verification computes the cross-product of the automata representing the design and the negated property. Does one approach yield an automaton that is more amenable to verification than the other? Some verification heuristics work only on property automata with particular structural features. Answers to these questions help determine whether verification tools can safely treat timing diagrams as interfaces to LTL expressions without having an adverse effect on the verification process.

When comparing how each approach scales with respect to a given timing diagram, there are two classes of parameters to consider: the value of the lower and upper time bounds on the edges, and the size of the assume portion. While the bounds on the edges certainly affect the size of the resulting automata, we conjuncture that the size of the assume portion will be more significant. Consider the structure of the LTL expressions resulting from the translation algorithm. As the example in Figure 1.6 shows, the subexpression for the assume portion appears on both sides of the implication in the LTL formula. Algorithms for converting Büchi automata into LTL normalize formulas be-

[2]New verification algorithms that operate directly on timing diagrams are another option; we are exploring this in separate research.

fore translation: the normalization process will destroy the similarities between the two copies of the assume portion. Our timing diagram to automaton algorithm, in contrast, translates the assume portion only once. Our experiments use Daniele, Giunchiglia, and Vardi's LTL-to-Büchi translation algorithm, which currently yields the most compact automata of all such translation algorithms [Daniele et al. (1999)].

As an initial experiment, consider a simple diagram with an empty (trivial) assume portion. The table shows the numbers of states in the semantics automaton (column "Direct") and the automaton obtained from the equivalent LTL formula (column "LTL").

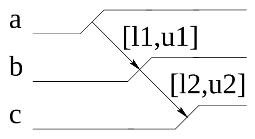


l	u	Direct	LTL
1	1	7	9
2	2	10	12
3	3	14	16
4	4	18	20

l	u	Direct	LTL
1	∞	12	17
2	∞	12	16
3	∞	16	20
4	∞	20	24

Each automaton has constant growth with respect to increases in the time bounds. This supports our hypothesis that the magnitude of the bounds does not yield significant differences between the two translation algorithms. Similar experiments on diagrams with more transitions show similar results: while the magnitude of the constant difference between the two machines increases slightly on these examples, the differences are still small constants when the assume portion is empty.

The picture changes dramatically once we produce the automata for the negated properties, as required in verification. Consider a diagram with three transitions. The table reports the number of states in the complemented semantics automaton and the automaton derived from the negation of the equivalent LTL formula.



l_1	u_1	l_2	u_2	Assume	Direct	LTL
1	1	1	1	0	8	650
2	2	2	2	0	14	5372
3	3	3	3	0	22	24174
1	∞	1	∞	0	18	4999
2	∞	1	∞	0	18	6369
2	∞	2	∞	0	18	8286
1	1	1	1	1	10	655
1	1	1	1	2	11	658
1	∞	1	∞	1	20	5004

The "Assume" column in the table indicates how many rising transitions (counting from the left) are in the assume portion; thus, an assume value of 1 indicates that only the rising transition on a is in the assume portion. The direct translation algorithm remains constant in its growth across the first three sets of experiments. The LTL-based translation algorithm, however, exhibits neither constant nor even linear growth. Thus, the LTL-based approach performs worse as size of the assume portions increase.

Although these figures suggest clear differences between our two approaches for compiling a timing diagram into a Büchi automaton, they do not indicate the source of the differences. Several factors influence the results. Most importantly, timing diagrams translate to a particular subset of Büchi automata that preserve their size under complementation [Fisler (2000)]; in the general case Büchi automata may grow exponentially under complementation. Second, the LTL to Büchi algorithms are not canonical, in that they may produce different automata for logically equivalent expressions. In practice, the current best algorithm produces larger automata than necessary on timing diagram formulas. Finally, this canonicity problem suggests that an alternative translation from timing diagrams to LTL might yield smaller automata. Using direct timing diagram translation algorithms and improving the LTL to automata algorithms seem the best options to pursue further.

5 Timing Diagrams and Computational Efficacy

The investigations into computational efficacy described in the paper have exposed several features of timing diagrams that may make them more computationally effective than similar textual representations in the context of hardware verification. In particular, we have cited their two-dimensional nature, their localization of non-regular behavior, their ability to capture certain forms of common subexpressions, and their relationship to a particular restricted class of automata models.

Two-dimensionality is useful because it provides two different perspectives on a set of signals simultaneously. Proofs about systems sometimes need to consider how the system looks at a particular point in time, and sometimes need to consider how a particular signal behaves over time. Timing diagrams provide both views, whereas common textual representations emphasize one over the other. LTL, for example, is well-suited to describing individual sequences of events, but not meeting points across sequences; this explains the duplication of the subexpression regarding the rising transition on c in the translation example in Figure 1.6. Accordingly, we believe the two-dimensionality issue and the common subexpression issue are closely related. Localization of non-regular behavior appears to be unrelated to these other attributes. This issue affects the machine models that capture timing diagram languages. The models, in turn, affect decidability for a range of questions, include several of interest in verification. We are still investigating connections between localization and the automata models that capture timing diagrams.

Researchers have proposed two explanations for why diagrams appear so useful in reasoning. One is their ability to capture spatial relationships [Glasgow et al. (1995)]. This explanation seems reasonable for contexts where the problem domain involves spatial reasoning, but we do not believe that to be the case for hardware verification. The second explanation is what Barwise and Etchemendy call the homomorphic relationship between diagrammatic representations and their models [Barwise and Etchemendy (1996)]. This appears more relevant to our work. Hardware designers can examine a design from many angles, which is similar to our multi-dimensional view of timing diagrams. However, timing diagrams are much more abstract than the devices that they describe, which seems inconsistent with existing examples of homomorphic representations. The root of computational efficacy in diagrams therefore remains a widely open question for future research.

6 Conclusion

Computer scientists have much to contribute to, and hopefully gain from, the growing interest in diagrammatic representations and reasoning. Computational analyses provide solid metrics for gauging how representations affect solutions to computational problems. A *computational* perspective on diagrams, as opposed to a cognitive or logical one, helps determine the boundary between diagrams as interfaces and diagrams as computational artifacts in their own right. This is increasingly important as more computer-based tools attempt to support human problem solving in various domains. In turn, computer science may benefit from studying diagrams as they may suggest new representations with better solutions to existing computational problems. The interplay of computer science and diagrammatic reasoning is thus a vibrant and promising research area.

This paper presents one computer scientist's approach to diagrams research. We are exploring the potential computational benefits, or *computational efficacy*, of diagrams in hardware verification. This area is well-suited to such research: it involves a wide array of diagrammatic representations, has many subproblems that require both reasoning and proof, and yields computationally intensive problems. Accordingly, results from this project should interest researchers in diagrams and hardware verification alike. This paper describes several of our ongoing projects in this area. Our results suggest that diagrams do have computational benefits over their textual counterparts for certain verification problems. Our current hypotheses as to the properties of diagram that yield these benefits, also discussed in this paper, suggest several open and exciting problems for future research.

References

Barwise, J. and J. Etchemendy. 1996. Visual information and valid reasoning. In Allwein, G. and J. Barwise, editors, *Logical Reasoning with Diagrams.* Oxford University Press.

Clarke, E., O. Grumberg, and D. Peled. 2000. *Model Checking.* MIT Press.

Damm, W., B. Josko, and R. Schlör. 1995. Specification and verification of VHDL-based system-level hardware designs. In Egon Börger, editor, *Specification and Validation Methods*, pages 331–409. Oxford Science Publications.

Daniele, M., F. Giunchiglia, and M. Y. Vardi. 1999. Improved automata generation for linear temporal logic. In *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV)*, number 1633 in Lecture Notes in Computer Science. Springer-Verlag.

Fisler, K. 1999. Timing diagrams: Formalization and algorithmic verification. *Journal of Logic, Language, and Information*, 8:323–361.

Fisler, K. 2000. On tableau constructions for timing diagrams. In *Proceedings of the NASA Langley Formal Methods Workshop.*

Fisler, K. 1996. *A Unified Approach to Hardware Verification Through a Heterogeneous Logic of Design Diagrams.* PhD thesis, Indiana University.

Glasgow, J., N. H. Narayanan, and B. Chandrasekaran. 1995. *Diagrammatic Reasoning: Cognitive and Computational Perspectives.* MIT Press.

Hammer, E. 1996. *Logic and Visual Information.* Studies in Logic, Language, and Information. Cambridge University Press.

Hopcroft, J. E. and J. D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley.

Lemon, O. and I. Pratt. 1997. Logical and diagrammatic reasoning. In *Proc. 19th annual conference of the Cognitive Science Society.*

Manna, Z. and A. Pnueli. 1995. *Temporal verification of reactive systems : safety.* Springer.

Vardi, M. Y. and P. Wolper. 1986. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic and Computer Science (LICS).*