On Tableau Constructions for Timing Diagrams

Kathi Fisler
Department of Computer Science
Rice University
6100 S. Main, MS 132
Houston, TX 77005-1892
kfisler@cs.rice.edu

Abstract

Designers often cite unfamiliar notation as one obstacle to wider acceptance of formal methods. Formalizations of design notations, such as timing diagrams, promise to bridge the gap between design practice and formal methods. How to use such formalizations effectively, however, remains an open question. Developing new tools around design notations might provide better support for reasoning at the level of the preferred notations. On the other hand, translating the formalizations into established notations enables leveraging off of existing tools. This decision of whether to treat design notations as interfaces depends largely on computational tradeoffs. This paper explores this issue in the context of specifying properties for automata-theoretic verification using timing diagrams. Automata-theoretic algorithms perform a tableau construction to convert properties into Büchi automata. We contrast direct compilation of timing diagrams into Büchi automata with an approach that uses linear-time temporal logic (LTL) as an intermediate language during translation. Direct compilation generally produces much smaller automata and scales significantly better with variations in key timing diagram parameters. We attribute this to combination of a correspondence between timing diagrams and weak automata and certain shortcomings in current LTL-to-Büchi algorithms.

1 Introduction

Computer-aided verification uses techniques from logic and mathematics to prove whether design models satisfy certain properties. Although these techniques have been used successfully on several sizable examples, many designers are reluctant to adopt them. One frequently cited problem is the notation that verification tools employ [9]. Verification technologies are grounded in formal logic. Accordingly, most tools use their underlying logics as property specification languages. For example, model checkers employ temporal logics, while theorem provers use various flavors of higher-order logic. In contrast, designers use a wide array of notations, including circuit diagrams, timing diagrams, state machines, VHDL and Verilog. This rich array of representations, some of them diagrammatic, stands in stark contrast to the monolithic textual logics of verification tools.

Bridging this gap requires verification tools that support notations that are more familiar to designers. One approach is to develop new tools and algorithms which support design notations directly [3]. Another is to create interfaces from design notations to existing languages [1, 8]; this leverages off existing tool development efforts.[1] Which approach yields more efficient algorithms is an open question. There may exist algorithms for model checking timing diagrams, for example, that outperform those for the temporal

[1] Many efforts (other than those cited) are ad-hoc, however, because they do not formalize the design notations.

logics into which we might translate timing diagrams. Understanding these tradeoffs requires studies of the logical nature of design notations and their role in verification algorithms.

This paper explores these tradeoffs in the context of compiling timing diagrams to Büchi automata. Automata-theoretic verification tools, which support linear-time logics such as LTL, operate at the level of automata. Using such tools on timing diagrams requires algorithms for compiling timing diagrams to Büchi automata. We compare two compilation methods, one which compiles timing diagrams directly into Büchi automata and one which translates timing diagrams into LTL and then uses existing algorithms for compiling LTL into Büchi automata. Our results show that the direct approach produces far smaller machines even on simple examples. This appears due to a combination of structural properties of the automata that capture timing diagrams and shortcomings in existing LTL-to-Büchi translation algorithms.

Section 2 presents an overview of automata-theoretic verification. Section 3 describes timing diagrams and linear-time temporal logic, the two notations used in this paper. Section 4 presents our algorithms for compiling timing diagrams into LTL and Büchi automata. Section 5 presents an experimental comparison of the two approaches to obtaining Büchi automata from timing diagrams. Section 6 discusses the experimental results and their implications for verification research.

2 Automata-Based Verification

Automata-theoretic verification views both systems and properties as finite-state automata [12, 14]. Verifying whether a system satisfies a property is analogous to asking whether the property automaton accepts the language generated by the system. In other words, for a system S and a property P, verification reduces to a language containment question of the form $\mathcal{L}(S) \subseteq \mathcal{L}(P)$, where $\mathcal{L}$ denotes the language of an automaton. This is equivalent to asking whether $\mathcal{L}(S) \cap \mathcal{L}(\overline{P}) = \emptyset$. In practice, automata-theoretic verification tools implement the latter; they intersect the automaton for the negation of the property with the automaton for the system and check whether the language of the product automaton is empty.

Many other verification problems can be expressed in terms of operations on languages. Property decomposition is one example. Properties often prove intractable to verify because they require too many computational resources, such as time or memory. One can approach such cases by decomposing the original property into a set of simpler properties, each of which is tractable to verify. If the simpler properties collectively imply the original property, then verifying each simple property independently is sufficient to verify the original property. To support decomposition, verification tools must check whether one set of properties implies another. If a property P is decomposed into properties $P_1, \ldots, P_k$, this check reduces to $\mathcal{L}(P) \subseteq \mathcal{L}(P_1) \cap \ldots \cap \mathcal{L}(P_k)$.

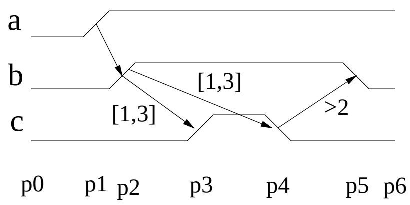
Both of these checks are decidable for a large class of verification problems. Implementing them requires procedures to obtain two kinds of automata: those that accept the language of a given property and those that accept the language of the negation of a given property. This project investigates both problems in the context of timing diagrams.

3 Timing Diagrams and LTL

3.1 Timing Diagrams

Timing diagrams express patterns of value changes on signals. In addition, they express precedence and synchronization relationships between changes, and timing constraints between changes. As part of our overall research program, we have developed a logic of timing diagrams [5]. This section describes the portion of the logic that is relevant to this paper.

Figure 1 provides a sample timing diagram that will serve as our running example. 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 low to high, then later returns to low. We interpret low as logical false and high as logical true. Arrows indicate temporal ordering between transitions; for this paper, we assume that timing diagrams spec-



a	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1
b	0	0	1	1	1	1	0	1	1	1	1	1	1	1	0
c	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	p_1	p_2	p_3	p_4		p_5									
					p_1		p_2		p_3		p_4			p_5	

Figure 1: A timing diagram and an illustration of its semantics.

ify a total ordering on the transitions through arrows and ordering within waveforms. Annotations of the form $[l, u]$ on the arrows indicate lower and upper bounds on the time between the related transitions; l is a natural number and u is a natural number or the symbol ∞.[2] The labels at the bottom, referred to as *time points*, are for explanatory purposes and are not part of the timing diagram; intuitively, there is one time point for each transition in the diagram, plus one for each of the endpoints of the diagram. The portion of the diagram between each pair of time points is an *interval*; interval I_j spans from time point p_j to p_{j+1}.

Since timing diagrams express sequences of values of 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 such a 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. A *timing diagram language* is any set of words such that every word in the set models the timing diagram. This paper provides an intuitive description of the semantics; the full details appear elsewhere [5].

Consider the timing diagram and word in Figure 1. 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 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 variable correspond to those required by the transitions at the time point; satisfaction relies on both the current index and its immediate successor. For example, in Figure 1, time point p_1 contains a rising transition on signal a; index d satisfies p_1 if d assigns value 0 to a and index $d+1$ assigns value 1 to a.

For the word and timing diagram in Figure 1, index 0 satisfies the rising transition on a. The walk now searches for an index containing a rising transition on b; index 1 meets this criterion. When the walk locates the rising transition on c in index 2, the semantics checks whether the located indices respect the timing constraint between the transitions on b and c. The two transitions occurred one index apart, which is valid. Continuing the walk locates time point p_4 at index 3 and time point p_5 at index 5. The first row below the table shows this assignment of time

[2]The full logic supports richer bounds with variables [5].

points to indices. The second row shows another assignment, starting from index 4. This walk fails, because the distance between the indices satisfying p_2 and p_4 is larger than 3, the maximum allowed by the time bound on the arrow from the rising transition on b to the falling transition on c. The semantics always checks the first occurrence of a transition that it finds once it begins searching for it. The formal semantics [5] defines this precisely.

Three other aspects of our semantics are relevant:

- Timing diagrams express assume-guarantee relationships; we specify some prefix of the time points as the *assume portion*, and only check the entire diagram when we locate indices satisfying the assume portion. In our example, taking the assume portion to be time points p_0 and p_1, we would search for the entire diagram only if an index reflects a rising transition on a.
- We view timing diagrams as invariants, meaning that we attempt to satisfy the timing diagram from every index which satisfies the assume portion. In our example, we would search from every index containing a rising transition on a, namely indices 0 and 4, as in our demonstration.
- A parameter over the timing diagram indicates which segments of waveforms should be matched exactly within words; the rest are treated as don't-cares. Segments to be matched exactly are called *fixed-level constraints*. For example, we could require a to remain high until the rising transition on c by putting a fixed-level constraint on a between time points p_1 and p_4.

Index satisfaction and fixed-level constraints are simply constraints on the values of particular variables; each constraint is a conjunction of literals capturing the values required on each variable. A fixed-level constraint requiring a to be low and c to be high would be the conjunction $\neg a \wedge c$. The actual conjunctions are irrelevant to the algorithms in the rest of the paper. We therefore describe our algorithms in terms of the following symbols:

- A_i: the fixed-level constraint in interval I_i.
- $AP_{i\text{init}}$: the first index required to satisfy the transition at time point i.
- AP_i: the second index required to satisfy the transition at time point i.
- T_i: The conjunction $AP_{i\text{init}} \wedge \mathsf{X} AP_i$, which uses the temporal logic next-time operator to capture the requirements for satisfying a transition.

3.2 Linear-time Temporal Logic

Like timing diagrams, linear-time temporal logic describes patterns of changes in variables over sequences of assignments. LTL is a propositional temporal logic [13], 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. Formally, LTL formulas are given semantics relative to sequences of assignments to $\mathcal{P}$. An infinite word $\xi = x_0 x_1 \ldots$ is a sequence of elements of $2^{\mathcal{P}}$. ξ_i denotes the suffix of ξ starting at x_i. A word ξ models formulas according to the following definition:

- $\xi \models q$ iff $q \in x_0$, for $q \in \mathcal{P}$,
- $\xi \models \neg\varphi$ iff not $\xi \models \varphi$,
- $\xi \models \varphi \vee \psi$ iff $\xi \models \varphi$ or $\xi \models \psi$,
- $\xi \models \mathsf{X}\varphi$ iff $\xi_1 \models \varphi$,
- $\xi \models \varphi \mathsf{U} \psi$ iff there is an $i \geq 0$ such that $\xi_1 \models \psi$ and $\xi_j \models \varphi$ for all $0 \leq j < i$.

A language models a formula iff every word in the language models the formula.

4 Tableau Constructions

As discussed in Section 2, automata-theoretic verification tools compile formulas into Büchi automata. As LTL model checking uses the automata-theoretic framework, several algorithms exist for compiling

LTL formulas into Büchi automata [2, 7]; these algorithms use a technique called tableau construction. The timing diagram semantics effectively define a Büchi automaton accepting a timing diagram language. Thus, we have two possible routes to compiling timing diagrams into Büchi automata, as shown in the diagram below: compile the timing diagram directly to a Büchi automaton which corresponds to the semantics, or translate the timing diagram into LTL and use existing LTL-to-Büchi algorithms. The second approach reflects the view of timing diagrams as visual interfaces for temporal logics [1].

$$\begin{tikzcd} \text{TD} \arrow[r, "\text{our translation}"] \arrow[rr, bend right, "\text{semantics}"'] & \text{LTL} \arrow[r, "\text{existing algorithm}"] & \text{Büchi} \end{tikzcd}$$

We would like to compare the Büchi automata arising from these two approaches. Is one substantially larger than the other? Size is important because this form of verification computes the cross-product of the automata representing the design and the property. Does one approach yield a Büchi 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.

Our translations from timing diagrams to each of LTL and automata rely on the same intermediate representation, a form of abstract state machine. States in this machine record which interval they correspond to, their transitions to other abstract states, and a set of labels which provide information to the backend tools. The abstract machine captures one pass or walk of the timing diagram semantics, leaving the backend tools to support repetitions as necessary.

4.1 Generating the Abstract Machine

Generating an abstract machine from a given timing diagram proceeds in two steps. First, we need to

I_1
1+

I_2	I_3
1	1
1	2
2	1

I_4
2+

Figure 2: Step distribution tables for the example timing diagram.

calculate the possible numbers of steps that a valid word can spend in each interval. We partition the time points into cells such that time points i and j are in the same cell iff there is an arrow spanning intervals i and j: for our example timing diagram, the cells are $\{0\}$, $\{1\}$, $\{2,3\}$, $\{4\}$, and $\{5\}$. For each cell, we generate a table showing the possible combinations of steps allowed in each interval. Each row of the table provides one distribution of the time allowed by the bounds across the corresponding intervals; if the total amount of time is a lower bound, the value in the last column of the table is marked with a +. Figure 2 shows the tables for our example diagram. They say that a valid word must contain at least one letter in the first interval (1+ in the first table), some combination of 2 or 3 letters in the interval between time points 2 and 4 (the middle table), and at least two letters in interval I_4. We generate the tables using a straightforward procedure for calculating distributions across variables. We then eliminate distributions that violate some timing constraint; the example diagram, for example, allows the arrow from the rising transition on b to the rising transition on c to last 3 steps, but doing so would violate the constraints of the edge from the rising transition on b to the falling transition on c. The tables in Figure 2 contain no row allowing 3 steps in interval I_2.

Next, we generate abstract states from the cells and tables. Each abstract state contains the time point it corresponds to, a set of transitions to other abstract states, and a set of labels (which we describe shortly). We generate a final state (labeled final) with a self-transition; this corresponds to the maximal time point. We also generate two abstract states with transition to the final state for each time point in the assume portion: one labeled PM for pat-

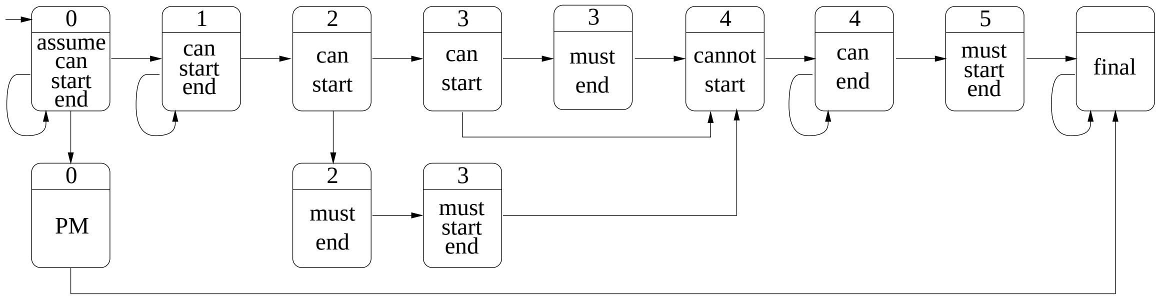


Figure 3: The abstract machine for the example timing diagram.

tern mismatches and one labeled CV for constraint violations; these capture violations of the timing diagram patterns in the assume portion. The generation method processes the cells in reverse order. For each cell, we generate a set of states, designating one as the initial state for the cell, as follows. If there is no table for the cell, we generate one abstract state with two transitions: one to itself and one to the initial state for the cell containing the next time point. If the time point is in the assume portion, the abstract state also contains a transition to the pattern-mismatch state for the corresponding time point.

If there is a table for a cell, we must generate sequences of states that count steps in the intervals as indicated in the tables. Rather than generate these sequences independently, however, we share states at the prefixes of the sequences when possible. All sequences will share at least one common prefix state; this is the initial state for the cell. For the example timing diagram, all rows for cell $\{2, 3\}$ require at least one state in interval 2. Each state contains a transition to the next state in the sequence; states in common prefixes may have transitions to multiple suffixes. In addition, if the last entry in a row is annotated with $+$, the final state in the sequence for the row contains a self-loop. If the cell is in the assume portion, each state also contains transitions to the pattern-mismatch and constraint-violation states for the corresponding time points. Figure 3 shows the abstract machine corresponding to our example timing diagram. We have explained the structure of this machine; we now describe the labels.

Each state corresponding to a time point in the assume portion receives the label assume. For each state other than the final, pattern-mismatch, and constraint-violation states, we add all labels from the following list for which the state satisfies the indicated constraints relative to the structure of the transition system; let B be a state at time point p_i:

- start: no other state for time point p_i reaches B;
- end: B reaches no other state for time point p_i;
- can: B has successors for time points p_i and p_{i+1};
- cannot: all successors are for time point p_i;
- must: no successor is for time point p_i.

The labels start and end indicate the first and last states for each corresponding time point; can, cannot, and must indicate whether a word can, cannot, or must advance to the next time point from this state. While some of these labels have overlapping meaning (all must states are end states, for example), no two labels are equivalent.

4.2 Generating LTL

This section generates an LTL formula corresponding to one pass of the timing diagram semantics. Wrapping the formula in LTL operator G yields the invariant formula. The procedure follows the structure of the abstract machine. There are two steps in generating the LTL for a given abstract state: generating the propositional expression that captures the fixed-level constraints for the state and connecting this expression with those for other states using temporal

$$
\begin{array}{l}
([(A_0 \wedge \neg T_0) \mathsf{U} (A_0 \wedge T_0)] \rightarrow \\
\;[(A_0 \wedge \neg T_0) \mathsf{U} \\
\;\;(A_0 \wedge T_0 \wedge \\
\;\;\;\mathsf{X}[(A_1 \wedge \neg T_1) \mathsf{U} \\
\qquad (A_1 \wedge T_1 \wedge \mathsf{X}((A_2 \wedge T_2 \wedge \mathsf{X}((A_3 \wedge T_3 \wedge \mathsf{X}(A_4 \wedge \neg T_4 \wedge \mathsf{X}[(A_4 \wedge \neg T_4) \mathsf{U} (A_4 \wedge T_4)])) \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad (A_3 \wedge \neg T_3 \wedge \mathsf{X}(A_3 \wedge T_3 \wedge \mathsf{X}(A_4 \wedge \neg T_4 \wedge \mathsf{X}[(A_4 \wedge \neg T_4) \mathsf{U} (A_4 \wedge T_4)]))))) \vee \\
\qquad\qquad\qquad (A_2 \wedge \neg T_2 \wedge \mathsf{X}(A_2 \wedge T_2 \wedge \mathsf{X}(A_3 \wedge T_3 \wedge \mathsf{X}(A_4 \wedge \neg T_4 \wedge \mathsf{X}[(A_4 \wedge \neg T_4) \mathsf{U} (A_4 \wedge T_4)]))))))])])
\end{array}
$$

Figure 4: LTL generated for example timing diagram

operators. The expression for a state is the fixed-level constraint A_i; if the state is the first or last in a time point, we conjoin A_i with AP_i or $AP_{i+1\text{init}}$, respectively. The temporal operators are based on the transition structure of the abstract machine.

Formally, procedure GenLTL(B) produces the LTL for abstract state B as follows, where R is the transition relation of the abstract machine. For abstract states B without self-loops, GenLTL(B) produces

$$A_i \wedge T_i \wedge \bigvee_{B' \in R(B)} \mathsf{X}(\text{GenLTL}(B')).$$

For abstract states B with self loops, GenLTL(B) is

$$[(A_i \wedge \neg T_i) \mathsf{U} (A_i \wedge T_i \wedge \bigvee_{B' \in R(B)} \mathsf{X}(\text{GenLTL}(B')))].$$

The T_i's require the expression to match the first available transition to the next time point. To handle the assume portion, the algorithm generates LTL for the restriction of the abstract machine to the assume portion and forms an implication from this formula to the LTL for the entire diagram. This follows the intuitive semantics of timing diagrams. Figure 4 shows the resulting LTL for our running example. The contrast between the formula and the original timing diagram motivates designers' frustrations with common verification notations.

4.3 Generating Büchi Automata

A Büchi automaton is a tuple $\langle Q, q_0, R, L, \mathcal{F} \rangle$ where Q is a set of states, q_0 is the initial state, $R \subseteq Q \times Q$ is the transition relation, L indicates propositions that are true in each state, and $\mathcal{F} \subseteq Q$ is a set of fair states. The abstract machine resembles a Büchi automaton; however, it does not capture a timing diagram because it does not enforce matching the first occurrences of transitions. The Büchi automaton states enforce this by examining propositions $AP_{i+1\text{init}}$ and AP_{i+1}, which indicate when transitions should occur. These states also refer to the fixed-level constraint A_i.

Monitoring $AP_{i+1\text{init}}$ and AP_{i+1} implies that an abstract state can expand into four Büchi states (A_i must hold in each; the pattern-mismatch states account for when A_i does not hold). The number may be more or less depending on the abstract state's labels. Regardless of the labels, only a few combinations of propositions arise in practice. Table 1 (left) lists templates of the generated Büchi states. For each state, we list the propositions that are true in that state and a set of labels. These labels are not part of the Büchi automaton; the algorithm uses them to create transitions between states. The labels can be divided into two sets, depending upon whether they contain `this`; we explain the distinction shortly.

The Büchi automaton generator converts abstract state B into Büchi automaton states $b_1, \ldots, b_m$ in two steps. First, it creates the template states indicated in Table 1 (right). Second, it adds the outgoing transitions for each b_k. These outgoing transitions depend on B's labels and whether b_k outputs proposition $AP_{i+1\text{init}}$. This proposition matters because it indicates that b_k could recognize the start of the next time-point. Any transitions from b_k to states outputting proposition AP_{i+1} must be to states corresponding to the next time-point.

	Propositions	Incoming Labels
S_1	A_i, $AP_{i+1\text{init}}$, AP_{i+1}	this-tp, this-tp-trans
S_2	A_i, $\neg AP_{i+1\text{init}}$, AP_{i+1}	this-tp, this-tp-trans
S_3	A_i, $AP_{i+1\text{init}}$, $\neg AP_{i+1}$	this-tp, this-tp-no-trans
S_4	A_i, $\neg AP_{i+1\text{init}}$, $\neg AP_{i+1}$	this-tp, this-tp-no-trans
S_5	A_i, $AP_{i+1\text{init}}$, AP_i	tp-start
S_6	A_i, $\neg AP_{i+1\text{init}}$, AP_i	tp-start
S_7	$\neg AP_i$	cv-no-trans
S_8	A_i, AP_i	cv-trans
S_9	$\neg A_i$	pv-this
S_{10}	$\neg A_i$, AP_i	pv-on-trans
S_{11}	$\neg A_i$, $\neg AP_i$	pv-this-no-trans
S_{12}		final

Type	Start?	States
cannot	yes	S_5, S_6
cannot	no	S_1, S_2, S_3, S_4
must	yes	S_5 plus this-tp label
must	no	S_1, S_3
can	yes (ex. p_0)	S_1, S_2, S_3, S_4, S_5, S_6
can	no or p_0	S_1, S_2, S_3, S_4
CV		S_7, S_8
PM		S_9, S_{10}, S_{11}
final		S_{12}

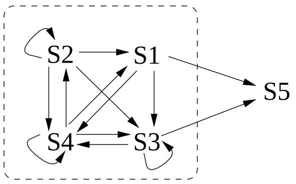
Table 1: Tables defining translation of abstract states to Büchi states.

Type	Next Init?	Outgoing Labels
can	yes	tp-start, this-tp-no-trans pv-this-no-trans, pv-on-trans
can	no	this-tp pv-this
cannot	yes	this-tp-no-trans, pv-this, cv-trans
cannot	no	this-tp, pv-this, cv-trans
must		tp-start, pv-on-trans, cv-no-trans

Table 2: Determining transitions between states.

More specifically, we connect the transitions for b_k, generated from abstract state B, according to the following algorithm: Let $c_1, \ldots, c_n$ be the states that expand all successors of B in the abstract machine. Let h_k be the set of labels for b_k according to Table 2. For each c_j, add a transition from b_k to c_j iff c_j comes from the same (resp. a different) time point as b_k and the incoming labels for c_j contain some this (resp. non-this label) label from h_k. The fair states consist of the state labeled final and all states expanding abstract states labeled assume.

As an example, let B be the rightmost abstract state for time point 4 from Figure 3. The following diagram shows the expansion. The four states in the dashed box correspond to B. Table 1 (right) tells us to create these states because B matches the second can line. State S_5 expands the abstract state for time point 5; we include it to illustrate the transition connection procedure.



Tables 1 and 2 determine the outgoing transitions for each state in the dashed box. For example, S_3 matches the first row in Table 2 because B has label can and S_3 outputs $AP_{i+1\text{init}}$. Thus, it needs a transition to each state in the dashed box with incoming label this-tp-no-trans (states S_3 and S_4 by Table 1 (left)) and each state outside the box with label tp-start (state S_5). We ignore the pv labels since there are no PM states for time points 4 or 5. A similar process yields the transitions for the remaining states.

Having presented algorithms for translating timing diagrams to both LTL formulas and Büchi automata, we need to check whether the derived formulas and automata correspond on a logical level. Given a timing diagram D, let D_{LTL} and D_{BA} be the formula and automaton derived for D, respectively. We have proven that $\mathcal{L}(D_{\text{BA}})$ models D_{LTL} according to LTL's semantics. As a sanity check on this result, we constructed an LTL formula capturing the structure of D_{BA} and compared it to D_{LTL} using an LTL equivalence checker [10]. These formulas are equivalent for a large test suite of timing diagrams, including those used in our experiments. Thus, we have high confidence in the correctness of our translations.

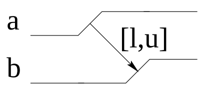
5 Experimental Results

This section compares our D_{BA} automata to those derived from D_{LTL} using an existing LTL-to-Büchi translation algorithm [2] with respect to their numbers of states. We do not report running times because the algorithms have been implemented in different paradigms, which reduces the value of such figures; in practice, the direct translations were substantially faster than the LTL-based translations. We report two groups of experiments. In the first, we generate automata for one pass of the timing diagram semantics. In the second, we generate automata for the negation of timing diagrams when treated as an invariant. The latter is required to model check timing diagrams using an automata-theoretic approach.

When comparing how each approach scales with respect to a given timing diagram, there are two classes of parameters to consider: the values of the lower and upper time bounds on the edges and the size of the assume portion. While the bounds certainly affect the size of the resulting automata, we conjecture that the size of the assume portion will be more significant. Consider the structure of D_{LTL}. As Figure 4 shows, the subexpression for the assume portion appears on both sides of the implication in the LTL formula. LTL-to-Büchi algorithms normalize formulas before 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 algorithm, which yields more compact automata than other algorithms [2].

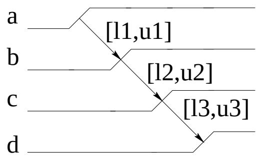
5.1 Accepting Timing Diagrams

As an initial experiment, consider a very simple diagram with an empty (trivial) assume portion. The table shows the number of states in the D_{BA} automaton (column "D_{BA}") and the number of states obtained compiling D_{LTL} to an automaton (column "via D_{LTL}"). The first two columns vary the bounds. Each automaton sees 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.



l	u	D_{BA}	via D_{LTL}
1	1	7	9
2	2	10	12
3	3	14	16
4	4	18	20
1	∞	12	17
2	∞	12	16
3	∞	16	20
4	∞	20	24

The picture changes dramatically as the assume portion grows beyond one transition. Consider a diagram with four transitions, as shown below. Each group of three experiments uses the same bounds and varies the assume portion size. The difference between assume portion sizes of one and two is substantial in each group. Furthermore, as the bounds in the assume portion grow, this difference appears to grow exponentially. Growth of each automaton still appears constant across experiments with the same assume portion size and varying bounds. This supports our hypothesis that the size of the assume portion is more important than the size of the bounds. The size of the bounds appear to matter more in the assume portion than in the non-assume portion. This makes sense, as the LTL-to-Büchi algorithm negates the assume portion to construct the automaton. This negation creates many disjunctions, which lead to branching and extra states in the LTL-to-Büchi translation. The larger the bounds, the more disjunctions result from the assume portion.



l_1	u_1	l_2	u_2	l_3	u_3	Split	D_{BA}	D_{LTL}
1	1	1	1	1	1	0	9	9
1	1	1	1	1	1	1	11	25
1	1	1	1	1	1	2	12	119
1	1	2	2	2	2	0	15	13
1	1	2	2	2	2	1	17	29
1	1	2	2	2	2	2	18	123
1	1	3	3	3	3	0	23	19
1	1	3	3	3	3	1	25	35
1	1	3	3	3	3	2	26	129
2	2	1	1	1	1	0	12	11
2	2	1	1	1	1	1	14	27
2	2	1	1	1	1	2	16	319
2	2	2	2	2	2	0	18	15
2	2	2	2	2	2	1	20	31
2	2	2	2	2	2	2	22	323
3	3	1	1	1	1	0	16	14
3	3	1	1	1	1	1	18	30
3	3	1	1	1	1	2	20	666

The LTL-to-Büchi approach produces smaller automata than our approach in some cases when the assume portion is empty. We believe this is due to a slight difference in how we handle relationships between the symbolic propositions (A_i, etc) in the two algorithms that would favor the LTL-based approach.

5.2 Rejecting Timing Diagrams

Model checkers require an automaton accepting the negation of a property. Even though we cannot draw the negation of a timing diagram as a timing diagram, we can still produce an automaton that accepts all words that fail to satisfy the timing diagram. This section compares these automata to those obtained for the expression $\neg \mathsf{G} D_{\mathrm{LTL}}$. We present two tables: the first summarizes experiments on the single transition diagram from the previous section and the second summarizes experiments on the two transition diagram. As an experiment in how the placement of temporal operators affects the construction of automata from LTL formulas, the first table includes an additional column, "Distrib", for which we distributed all X operations in D_{LTL} formula over boolean operators before compiling to an automaton.

l	u	Split	D_{BA}	via D_{LTL}	Distrib
1	1	0	7	112	199
2	2	0	10	310	588
3	3	0	14	654	1506
4	4	0	18	1307	3077
5	5	0	22	2613	6153
1	∞	0	12	295	295
2	∞	0	12	382	772
3	∞	0	16	705	1596
1	8	0	34	14599	24926
2	8	0	34	14632	25055
3	8	0	34	14728	25461
1	1	1	9	117	210
2	2	1	12	315	599
3	3	1	16	659	1519
1	∞	1	14	300	300
2	∞	1	14	387	781

l_1	u_1	l_2	u_2	Split	D_{BA}	via D_{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

In these tables, the difference between the two algorithms is striking. The direct translation still shows linear growth as we vary the bounds under a trivial assume portion. For the first section of the first table, the LTL-based algorithm shows exponential growth. The difference between zero and one transitions in the assume portion is not significant for either algorithm in the first table. Unfortunately, we were unable to generate the LTL-based automata for larger configurations than those shown within a reasonable amount of time (several hours per construction). However, the existing results are sufficient to demonstrate the drawbacks of the LTL approach to compiling timing diagrams into automata.

6 Discussion

The data in Section 5 suggest clear differences between our two approaches for compiling timing diagrams into Büchi automata. These differences could be due to the LTL-to-Büchi automaton translation, to our timing diagram to LTL translation, or to some property of timing diagrams that provides an inherent advantage over LTL.

LTL-to-Büchi algorithms are not canonical, in that they may produce different automata for logically equivalent LTL formulas; the Distrib experiments in the previous section show this. The Daniele *et al.* algorithm produces smaller automata than other algorithms because it uses some simple syntactic optimization techniques on propositional formulas [2]. More work should be done in this area; our timing diagrams research has yielded several formulas where simple manual transformations yielded much smaller automata from the Daniele *et al.* algorithm. Algorithms which perform optimizations across temporal operators are also needed, as our experiments show.

Currently, no known metrics indicate when one LTL formula will yield a smaller automaton than another. Therefore, it is possible that a different translation from timing diagrams to LTL would yield smaller automata. For several timing diagrams, we have tried to manually construct LTL formulas that yield our D_{BA} automata. We have been successful on occasion by translating the structure of D_{BA} into LTL. We are still working on such a translation procedure that acts as a fixpoint over Büchi automata, as a means of understanding the LTL-to-Büchi algorithms better. However, this approach is clearly redundant in practice, as it requires the construction of D_{BA}. We continue to experiment with other timing diagram to LTL translation algorithms, particularly ones which enable sharing of the assume portion.

This project is part of a larger investigation into whether timing diagrams offer any computational benefits over existing logics (including LTL) in verification contexts [4]. We have identified several differences between the two notations. Full timing diagrams and LTL have incomparable expressive powers [5] (this paper uses only a subset of timing diagrams). Timing diagrams enable sharing of common subexpressions to a greater extent than LTL. The LTL formula in Figure 4, for example, duplicates subexpressions across its disjuncts; these expressions correspond to entire suffixes of the timing diagram. LTL does not appear to provide a way to avoid this duplication. However, it is not yet clear whether these duplicated expressions contribute to the explosion in the generated Büchi automata.

The most interesting distinction that we've discovered between timing diagrams and LTL arises from the structure of the Büchi automata corresponding to each notation. Our timing diagram to Büchi translation always produces a particular structure of automaton known as a *weak* automaton [11]. An automaton with states Q and fair set $\mathcal{F}$ is weak if there exists a partition of Q into disjoint sets $Q_1, \ldots, Q_n$ such that (1) each Q_i is either contained in $\mathcal{F}$ or is disjoint from it, and (2) the Q_i's are partially ordered so that there is no transition from Q_i to Q_j unless $Q_i \leq Q_j$. Weak automata have several attractive features in the context of verification [11]; for example, symbolic cycle detection is effectively linear in weak automata, whereas existing algorithms for the general case are quadratic [6].

Another feature of weak automata is important to our study of timing diagrams: complementation of weak automata requires only complementation of the fair set $\mathcal{F}$; the structure of an automaton and its complement are otherwise identical. In Section 5, we explored translations of timing diagrams and their negations to Büchi automata. Our direct translation produces the same size automaton for a given timing diagram under each experiment because we exploit this feature of weak automata.[3] LTL-to-Büchi algorithms do not currently consider weak automata; this is an open problem as many LTL formulas do not cor-

[3] We require one extra transition to handle the invariant.

respond to weak automata. When we use LTL as an intermediate language, the Büchi automata for the negated timing diagrams are much larger than in the non-negated case. This is partly due to the structure of the LTL formulas corresponding to timing diagrams. As Figure 4 shows, LTL formulas corresponding to timing diagrams involve disjunctions of long sequences of conjunctions and temporal operators. The negation of such a formula contains many more disjunctions than the original formula. Disjunctions force branching and extra states in Büchi automata. It is therefore not surprising that the automata for the negated timing diagrams are so much larger than those for the one-pass timing diagrams.

In summary, many factors influence the size of the automata obtained when treating timing diagrams as an interface to LTL. These factors suggest a host of research problems in verification. We fully expect that improved LTL-to-Büchi algorithms would reduce the sizes of automata generated in our experiments. Until researchers develop such algorithms, however, direct compilation of timing diagrams to Büchi automata appears a better approach for verification applications.

References

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

[2] Daniele, M., F. Giunchiglia and M. Y. Vardi. Improved automata generation for linear temporal logic. In *Proc. 11th International Conference on Computer-Aided Verification*. 1999.

[3] Dillon, L., G. Kutty, L. Moser, P. Melliar-Smith and Y. Ramakrishna. A graphical interval logic for specifying concurrent systems. Technical report, UCSB, 1993.

[4] Fisler, K. Diagrams and computational efficacy. In review, Proc. of the CSLI Workshop on Logic, Language, and Information, October 1999.

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

[6] Fisler, K., R. Fraer, G. Kahmi, M. Y. Vardi and Z. Yang. A new symbolic cycle detection algorithm. In preparation, March 2000.

[7] Gerth, R., D. Peled, M. Y. Vardi and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of Protocol Specification, Testing, and Verification*, pages 3–18, August 1995.

[8] Grass, W. et al. Transformation of timing diagram specifications into VHDL code. In *Proceedings of Computer Hardware Description Languages and Their Applications*, pages 659–668, August 1995.

[9] Heitmeyer, C. On the need for 'practical' formal methods. In *Proc. 5th Intl. Symposium on Formal Techniques in Real-Time and Real-Time Fault-Tolerant Systems*, pages 18–26. Springer-Verlag, 1998.

[10] Janssen, G. PTL: A propositional logic tautology checker. Available online from `http://www.ics.ele.tue.nl/es/research/fv/research/research_index.shtml`.

[11] Kupferman, O. and M. Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *International Conference on Logic in Computer Science*, 1998.

[12] Kurshan, R. P. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[13] Pnueli, A. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

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