Timing Diagrams: Formalization and Algorithmic Verification

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

Abstract. Timing diagrams are popular in hardware design. They have been formalized for use in reasoning tasks, such as computer-aided verification. These efforts have largely treated timing diagrams as interfaces to established notations for which verification is decidable; this has restricted timing diagrams to expressing only regular language properties. This paper presents a timing diagram logic capable of expressing certain context-free and context-sensitive properties. It shows that verification is decidable for properties expressible in this logic. More specifically, it shows that containment of ω-regular languages generated by Büchi automata in timing diagram languages is decidable. The result relies on a correlation between timing diagram and reversal bounded counter machine languages.

Keywords: computer-aided verification, counter machines, language containment, model checking, timing diagrams.

1. Introduction

Hardware designers use diagrammatic representations extensively and pervasively. During design, they depict architectural and control-flow ideas with circuit diagrams and state machines. Upon completion, they document design features, such as timing requirements and component layouts, with timing diagrams and block diagrams. Given the variety of both diagrammatic notations and the situations in which they appear, diagrams are clearly a fundamental tool for reasoning about designs.

Unfortunately, tool support for hardware reasoning provides only limited support for diagrams. Diagrams often provide interfaces to textual representations (such as second-order or temporal logic) or illustrate counter-examples; support for reasoning directly with diagrammatic representations is rare. Furthermore, these uses often lack a formal basis, rendering them potentially unsound and precluding theoretical analyses of the diagrammatic representations.

* This research was conducted at Indiana and Rice Universities, and was partially supported by Bell Laboratories and the National Science Foundation.

In contrast, diagrammatic logics provide a solid foundation for both reasoning and theoretical analyses. This paper presents a timing diagram logic and a corresponding decidability result. The logic, which has been presented before [8], is unique because it expresses certain context-free and context-sensitive languages. The decidability result shows that timing diagram specifications can be verified algorithmically against designs with infinite behaviors. This is significant because timing diagrams express different properties than the formalisms currently used in automated verification. A preliminary version of this result, covering only designs with finite behaviors, appeared previously [8, 10]; the extension to infinite behaviors is new.

Our results also provide preliminary insight into the potential role of diagrammatic representations in algorithmic verification. Although human factors in design practice inspired our research into diagrammatic representations, we are interested in efficacy from a computational perspective. We seek to understand how data structures and algorithms based on diagrammatic representations affect computational processes along a variety of axes, including time and space efficiency, decidability, and the shapes of computations. Our decidability results provide a significant datapoint in this effort.

Section 2 provides a brief overview of computer-aided verification. The timing diagram formalization appears in Section 3; readers familiar with the logic can skip this section. Section 4 discusses timing diagram expressibility and presents the decidability result. Section 5 discusses other questions relevant to timing diagrams and verification. Concluding remarks appear in Section 6. The appendix contains the proofs of several theorems used in the decidability results.

2. Computer-Aided Verification

Computer-aided verification provides a rigorous approach to design validation. It requires three basic elements: (1) a language for expressing models of a design, (2) a language for stating properties that a design should satisfy, and (3) a method for checking whether the design model satisfies the stated properties. Properties are often behavioral in nature (as opposed to structural). A traffic light controller, for example, should display green lights to only one street at a time; in addition, each street should periodically get a green light.

One approach to computer-aided verification expresses both designs and properties as automata [18]. Automata may be viewed either as language *generators* or as language *acceptors*. Viewing a design as a generator and a property as an acceptor, this method asks whether the property automaton accepts the language generated by the design automaton. This is simply language containment: is the language of the design contained in that of the property?

When language containment is decidable, verification can be performed algorithmically.[1] Algorithmic verification methods are desirable because they are fully automatable and hence easier to apply than interactive methods, such as theorem-proving. In an automata-based framework, finite state machines capture designs; these machines correspond to *regular* languages. Such machines may generate finite or infinite words; in the latter case, the languages are called ω-*regular* [22]. Regular languages express a large class of properties (including those expressible with propositional linear temporal logic [20]) [23]. As language containment is decidable for (ω-)regular languages [22], these properties can be verified algorithmically. However, containment of a regular language (design) in a context-free language (property) is undecidable [11]. Accordingly, existing algorithmic verification tools support only regular language properties. As we show in Section 4, timing diagrams express a subset of context-free and context-sensitive languages in which containment of a regular language is decidable. This result therefore expands the set of algorithmically verifiable properties.

3. Timing Diagrams

Designers have traditionally used timing diagrams to document timing constraints on designs. Recognizing their appeal, researchers began formalizing timing diagram notations for use in hardware reasoning tasks. Dillon *et al.* designed a graphical, real-time, interval temporal logic [7].[2] Brzozowski, Gahlinger, and Mavaddat provide algorithms for testing consistency and satisfiability of timing specifications given as timing diagrams in the context of interfacing components [4]; Cerny and Khordoc have done similar work [5]. Several researchers propose writing behavioral specifications with algebras of timing diagrams annotated with programming constructs [17, 19]. Damm, Josko, and Schlör [6], as well as Berkane, Gandrabur, and Cerny [3], have formalized timing diagram notations for use in verification.

Although their syntax differs between formalizations, all timing diagrams express some common information: patterns of value changes on signals, precedence and synchronization relationships between changes, and timing constraints between changes. A sample timing diagram in

[1] Unfortunately, verification is intractable for many realistic designs due to the exponential growth of the state space; this problem is called *state explosion* [18].

[2] A *real-time* temporal logic accounts for timing constraints in one of many ways.

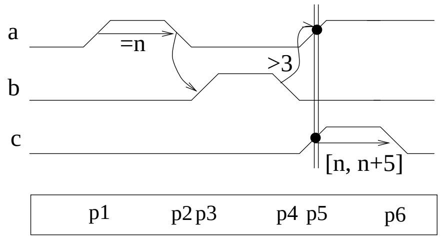


Figure 1. A TDL timing diagram. The labels in the box are not part of the diagram. They indicate *time points*, a component of timing diagrams which we derive from the rest of the diagram.

our notation appears in Figure 1. 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, a transitions from low to high, then returns 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.[3]

Vertical parallel lines indicate synchronization of the levels or transitions anchored with circles. 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; annotation "$= n$" (shorthand for $[n, n]$) indicates that the lower and upper bounds are the same, while "> 3" (shorthand for $[3, \infty]$) indicates a lower bound of 3 but no upper bound. Unannotated arrows have the time-bound $[1, \infty]$. *Valid bound expressions* consist of natural numbers, variables, and addition and subtraction expressions over them, as well as the symbol ∞. The labels in the box are for explanatory purposes and are not part of the timing diagram. The rest of this section formally presents the syntax and semantics of our timing diagram logic, TDL.

3.1. Syntax

Formally, a timing diagram contains three components: (1) an ordered sequence P of *time points*, which are abstract moments of time at which events occur; (2) a function N from names to *waveforms*, defined as functions from P to voltage level designators of type H (for high level), L (for low level), F (for falling transition), and R (for rising transition); and (3) a ternary relation O on $E \times E \times B$ capturing temporal ordering, synchronization, and time-bounds, where $E \subseteq N \times P$ captures time points on waveforms and B consists of time-bound expressions of the form $[l, u]$, such that l is a valid bound expression other than ∞ and

[3] Some engineering conventions reverse these interpretations.

u is any valid bound expression. In a well-formed timing diagram, all time-bounds expressions must be satisfiable. The time points may be taken from any partially-ordered domain.

As an example, we construct tuple $\langle P, N, O \rangle$ to capture the timing diagram in Figure 1. P contains time points $\{p_1, p_2, p_3, p_4, p_5, p_6\}$, as shown in the box. N is constructed from the levels on each waveform at each time point.

$$\begin{aligned} N = \{ & (a, \{(p_1, R), (p_2, F), (p_3, L), (p_4, L), (p_5, R), (p_6, H)\}), \\ & (b, \{(p_1, L), (p_2, L), (p_3, R), (p_4, F), (p_5, L), (p_6, L)\}), \\ & (c, \{(p_1, L), (p_2, L), (p_3, L), (p_4, L), (p_5, R), (p_6, F)\}) \end{aligned}$$

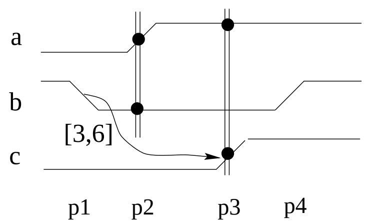
Relation O contains five elements, the first four corresponding to the arrows and the fifth to the synchronization line:

$$\begin{aligned} O = \{ & ((a, p_1), (a, p_2), [n, n]), ((a, p_2), (b, p_3), [1, \infty]), \\ & ((b, p_4), (a, p_5), [3, \infty]), ((c, p_5), (c, p_6), [n, n + 5]), \\ & ((a, p_5), (c, p_5), [0, 0])\} \end{aligned}$$

Although we have provided only an example here, building up sets P, N, and O as a timing diagram is constructed (with a structure editor, for example) is straightforward. The resulting tuple representation is unique up to isomorphism on the time point names. In the remainder of the paper, the term "timing diagram" refers to the tuple form, rather than to the original picture. As the tuples capture all of the information that is relevant for our purposes, we consider the tuples and the original diagrams to contain the same information. Therefore, while the semantics given in the next section are presented in terms of the tuples, we view them as providing semantics for the original diagrams.

3.2. Semantics

Timing diagrams are a form of linear-time temporal logic. Linear temporal logic is given semantics with respect to traces of variable values over time. These traces can be viewed as strings in which each "letter" corresponds to a valuation over all variables. The string view of traces suggests a correlation between linear temporal logic and formal languages. This is borne out in practice; verification tools support linear temporal logics by translating expressions into Büchi automata and performing language containment as discussed in Section 2 [23]. Formal languages therefore appear to provide an appropriate semantic model for timing diagrams. TDL takes this approach, but requires a more



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						
								p_1			p_2	p_3, p_4	

$$
\begin{aligned}
P &= \{p_1, p_2, p_3, p_4\} \\
N &= \{(a, \{(p_1, L), (p_2, R), (p_3, H), (p_4, H)\}), \\
&\qquad (b, \{(p_1, F), (p_2, L), (p_3, L), (p_4, R)\}), \\
&\qquad (c, \{(p_1, L), (p_2, L), (p_3, R), (p_4, H)\})\} \\
O &= \{((b, p_1), (c, p_3), [3, 6]), ((a, p_2), (b, p_2), [0, 0]), ((a, p_3), (c, p_3), [0, 0])\}
\end{aligned}
$$

Figure 2. Timing diagram and word used to illustrate index assignments and the semantics. The tuple form of the timing diagram is provided for reference.

general class of languages than the regular languages due to the use of variables in time-bound annotations.

TDL models timing diagrams by (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. Formalizing timing diagrams is difficult partly because their intended meanings differ widely between contexts and among users. Rather than fix one interpretation, TDL allows customization of the semantics through several parameters. This provides a uniform treatment of the common variations on timing diagram. We illustrate both the parameters and the semantics using the timing diagram and word in Figure 2.

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 three lines directly beneath the table indicate three separate assignments of indices to time points, as explained shortly.

The semantics operates by walking along a word looking for indices that satisfy each time point. Intuitively, 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.

Setting up the search for satisfying indices raises a question: under what conditions do we look for these indices? For example, we may want to satisfy the whole diagram only when we satisfy some prefix of it. TDL therefore allows a timing diagram to specify an assume-guarantee relationship among its events. A parameter captures the prefix to be assumed as a subset of time points. We call the portion of a timing diagram defined over these time points the *assume portion* of the diagram; the rest is the *guarantee portion.* We refer to the largest time point in the assume portion as p_{split}; $p_{\max}$ denotes the largest time point in the entire timing diagram. For the purposes of this example, we take the assume portion to contain only p_1, indicating that we only care about satisfying the timing diagram when we find a falling transition on signal b.

For the word and timing diagram in Figure 2, 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* and have the form (n, p, p'), where n is a waveform name and p and p' are time points such that $p < p'$ and the waveform corresponding to n shows a single level between p and p'. For this example, assume we have only one such constraint: (a, p_2, 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.

TDL requires timing diagrams to be matched repeatedly in a word. Two notions of repetition appear useful in practice: one in which the next repetition starts after the previous one has been completed (common in chip documentation), and the other in which a new repetition starts in every index (useful for verification). We therefore define two semantic relationships: $\models_{\mathrm{Iter}}$ (for iterative) and $\models_{\mathrm{Inv}}$ (for invariant), respectively. Under the invariant semantics, the next walk would begin in index 1, while under the iterative semantics, the next walk would begin in index 5; the corresponding assignments are shown in the next two lines beneath the table. Note that the assignment in the second line is not valid since the assigned indices violate the time-bounds on the arrow. This word therefore models the diagram under the iterative semantics, but not under the invariant semantics.

Thus, in order to interpret a timing diagram T, we need tuples of the form $\langle T, S, X\rangle$, where S is the set of assumed time points and X is a set of fixed-level constraints. We henceforth overload the term "timing diagram" to also refer to one of these tuples. The sets S and X as described are external to the diagram, in that the notation described in this paper does not provide diagrammatic representations for them. In this sense, one could view our timing diagrams $\langle P, N, O\rangle$ as ambiguous. In an implementation of our logic, we would overcome this problem by using colors on events to visually capture S and X. However, we treat them as external in this paper in hopes of providing a general framework that captures the various interpretations of timing diagrams that appear in practice.

We now define the TDL semantics formally. Given a timing diagram, we can derive a partial order on its time points from the ordering of events within individual waveforms and the temporal ordering relationships. For a time point p, its *enabling points* are those time points p' such that p' precedes p in this partial order. A timing diagram is *temporally unambiguous* when the order is total. Without loss of generality, we assume time points in a temporally unambiguous timing diagram to be numbered sequentially.

An *index assignment* is a partial function I from time points to natural numbers indexing a word (as shown in Figure 2); if I is not total, the time points on which it is defined must form a prefix of the partial order on time points. Given an index assignment, we can check whether the assigned indices satisfy each time point while respecting the fixed-level constraints and time-bounds. Definition 1 formalizes these conditions. A time-bound expression $((a, p), (a', p'), [l, u])$ is *relevant to time point* p_i iff $p' \leq p_i$. For index i into a word W, $W_i(n)$ denotes the value of W on the signal named n at index i.

DEFINITION 1. Let $D = \langle\langle P, N, O\rangle, S, X\rangle$ be a timing diagram. Let U be a prefix of the partial order on P and let I be an index assignment for U over a word W. I *satisfies the constraints of* U *relative to* W iff

1. For each time point $p \in U$, $I(p)$ *satisfies* p; *i.e.*, for every waveform name n in N, $W_{I(p)}(n) = 0$ (resp. 1) and $W_{I(p)+1}(n) = 1$ (resp. 0) if n has a rising (resp. falling) transition at p and $W_{I(p)}(n) = 0$ (resp. 1) if n has a low (resp. high) level at p and there is a synchronization line through n at p.

2. Each variable in a time-bound expression in O can be replaced by a natural number such that for each $((a, p), (a', p'), [l, u])$ in O that is relevant to a time point in U, $l \leq I(p') - I(p) \leq u$.

3. For each fixed-level constraint (n, p, p') in X, if p and p' are both in U, then for each index i such that $I(p) < i \leq I(p')$, $W_i(n) = 0$ (resp. 1) if n has a low (resp. high) level between p and p'.

By condition 2, variables in time-bound expressions are treated as existentially quantified within a single index assignment. However, variables may be instantiated differently across the many index assignments produced while repeatedly matching a diagram against a word.

Starting from a given index, multiple index assignments may satisfy Definition 1. The semantics uses the one that assigns to each time point the smallest index that both satisfies it and respects the partial order among the time points; this index assignment is called *minimal*.

DEFINITION 2. Let D be a timing diagram with time points P. Let W be a word, i be an index into W, and U be a prefix of the partial order on P. Index assignment I is *minimal* for D, W, i, and U iff I satisfies the constraints of U relative to W and for each time point $p \in U$, $I(p)$ is the smallest index of W larger than i that both satisfies p and is larger than all indices assigned to the enabling points of p.

Index assignments must also be defined on the largest possible prefix of the time point partial order. An index assignment is *fully minimal* if it is minimal and cannot be extended to a minimal index assignment for the same timing diagram, word, and starting index, but with a larger prefix of time points. Fully minimal index assignments are unique for temporally unambiguous timing diagrams.

Given a timing diagram $D = \langle T, S, X\rangle$, a word W, and an index i into W, the fully minimal index assignment I for D, W, and i is a *match* if I being defined for all time points in S implies that I is defined for all time points in T. Matches are the foundation of both semantic definitions. The invariant semantics simply requires that the assignment located from each index in the word is a match.

DEFINITION 3. Let D be a timing diagram with time points P. Let W be a word. D *invariantly describes* W (denoted $W \models_{\mathrm{Inv}} D$) iff for all indices i into W, the fully minimal index assignment for D, W, i, and P is a match. D *invariantly describes* a language L (denoted $L \models_{\mathrm{Inv}} D$) iff for every word W in L, $W \models_{\mathrm{Inv}} D$. The set of all words D invariantly describes is denoted $\mathcal{L}(D)^{\mathrm{Inv}}$.

The iterative semantics is more selective regarding the indices from which matches are attempted. Whenever a fully minimal index assignment is a match, an *end index* is also defined to indicate the index at which the next search should start. The end index is left undefined if the fully minimal index assignment is not a match.

As an example, consider again the timing diagram and word in Figure 2. The first index assignment is a match, so the end index is 5; this is the same index in which the next search began under the iterative semantics. Had we started a search in index 4, the end index would have been undefined because the time-bound violation occurred in the guarantee portion of the diagram. Had S contained both p_1 and p_2, and had we had an additional fixed-level constraint (b, p_1, p_2), then the attempt starting in index 7 would detect a fixed-level constraint violation in index 9, yielding the end index 10.

Formally, the end index is defined over three cases: (1) if the fully minimal index assignment I is defined for all time points in T, the end index is one larger than the last index in the range of I; (2) if a fixed-level constraint is violated while searching for an index that satisfies a time point in S, the end index is one larger than the index in which the constraint was violated; (3) if a time-bound constraint is violated by assigning index d to a time point in S, the end index is $d + 1$; the end index is left undefined in all other cases.

DEFINITION 4. Let D be a timing diagram with time points P. Let W be a word. Let i be an index into W, I be the fully minimal index assignment for D, W, i, and P, and e be the end index from i. D *iteratively describes* W *from* i, written $W, i \models D$, iff i is greater than the length of W or I is a match and $W, e \models D$. D *iteratively describes* a language L, written $L \models_{\mathrm{Iter}} D$, iff for all words W in L, $W, 0 \models D$. The set of all words D iteratively describes is denoted $\mathcal{L}(D)^{\mathrm{Iter}}$.

A *timing diagram language* is any language that some TDL timing diagram describes, either iteratively or invariantly.

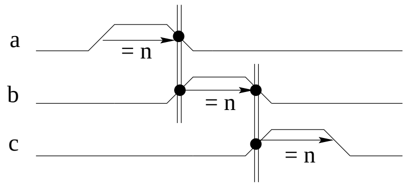


Figure 3. A timing diagram that captures a context-sensitive language property.

4. Expressibility and Decidability

Although TDL provides a linear-time temporal logic, it is incomparable with textual linear temporal logics such as LTL [20] with regard to expressiveness. TDL cannot express the LTL formula $\mathsf{F}p$ ("eventually p", same as $\Diamond p$) because there is no way to stop the repeated walks at a particular point. TDL is also unable to express LTL formula $\mathsf{G}(p \rightarrow q)$ ("always $(p \rightarrow q)$", same as $\Box(p \rightarrow q)$) because propositional disjunctions cannot be expressed in a TDL timing diagram.

In contrast, LTL corresponds to a subset of the regular languages. TDL timing diagrams can describe some non-regular languages by virtue of the variables allowed in time-bound expressions. Consider the context-sensitive language $(a^n b^n c^n)^*$ (where a is $\langle a = 1, b = 0, c = 0 \rangle$, with similar interpretations for b and c). The timing diagram in Figure 3, interpreted with no assume portion and fixed-level constraints on all segments, describes this language. However, TDL is unable to express non-regular languages with arbitrary counting, such as "the number of a's equals the number of b's".

A natural question arises: is verification decidable for TDL? Recalling the overview of verification presented in Section 2, this question asks whether testing containment of a regular language in a timing diagram language is decidable. It is a challenging question because TDL can express certain context-free and context-sensitive languages, the general classes for which containment problems are known to be undecidable.

Others have investigated the decidability of verification problems involving timing constraints. Ramakrishna *et al.* have proven decidability for a graphical, real-time, interval temporal logic without parametric timing constraints [21]. Jahanian, Mok, and Stuart have a textual logic of real-time timing constraints (RTL), that is undecidable [16]. Alur, Henzinger, and Vardi studied parametric timing constraints for real-time systems using a theory of parametric timed automata with multiple clocks for tracking parametric values [2]. They showed language emptiness is decidable for one parametric clock, undecidable for three parametric clocks, and an open problem for two parametric clocks. TDL timing diagrams yield certain properties requiring multiple parameterized clocks; the sets of problems addressed by their work and ours are therefore inequivalent.

Despite TDL's expressive power, containment of an ω-regular language in a temporally unambiguous timing diagram language is decidable. Our decision procedures are based on a correspondence between temporally unambiguous timing diagram languages and the languages accepted by deterministic, two-way, 1-counter machines (1-2DCM). We determine whether the language of a Büchi automaton models a timing diagram by computing relations on the states of the automaton using a series of counter machines. Formally, $\langle Q, \Sigma, \delta, q_0, Q_F\rangle$ represents a *Büchi automaton* where Q is a set of states, Σ is the input alphabet, δ is a subset of $Q \times \Sigma \times Q$ indicating the transition relation, $q_0 \in Q$ is the initial state, and $Q_F \subseteq Q$ are the accepting states. A Büchi automaton accepts an infinite word iff the set of states visited infinitely often while processing ω intersects Q_F. A (nondeterministic) finite automaton (NFA) is also represented by a tuple $\langle Q, \Sigma, \delta, q_0, Q_F\rangle$, where all components other than Q_F have the same interpretation as for a Büchi automaton. Here, Q_F is a set of final states; an NFA A accepts finite word w iff A is in some state in Q_F after processing w.

A 1-2DCM has a finite-state control, a two-way, read-only head over a finite, bounded-length input tape, and one counter which can store any natural number. Transitions are based on the current state, the letter being read, and whether the counter contains zero; the transition indicates a next state, which direction (if any) to move the input head, and whether to increment, decrement, or hold the value of the counter. The following formal definition is adapted from a paper by Ibarra [14].

DEFINITION 5.

1. A 1-2DCM M is a tuple $\langle K, \Sigma, \lhd, \rhd, \delta, q_0, F\rangle$ where K, Σ, $\lhd$, $\rhd$, q_0, and F are the states, inputs, left and right endmarkers, initial state, and accepting states, respectively. δ is a mapping from $K \times (\Sigma \cup \{\lhd, \rhd\}) \times \{0, 1\}$ into $K \times \{-1, 0, 1\} \times \{-1, 0, 1\}$.

2. A *configuration* of M is a tuple $(q, \lhd x \rhd, i, c)$ denoting that M is in state q with the input head reading the i^{th} symbol of $\lhd x \rhd$, and value c in the counter.

3. Relation $\Rightarrow$ on configurations maps $(q, \lhd x \rhd, i, c)$ to $(p, \lhd x \rhd, i + d, c + d_c)$ if a is the i^{th} symbol of $\lhd x \rhd$ and $\delta(q, a, \lambda(c))$ contains (p, d, d_c) where $\lambda(c) = 0$ if $c = 0$ and $\lambda(c) = 1$ if $c \neq 0$. The transitive closure of $\Rightarrow$ is denoted $\Rightarrow^*$.

4. M *accepts* a string $x \in \Sigma^*$ if $(q_0, \lhd x \rhd, 1, 0) \Rightarrow^* (q, \lhd x \rhd, i, c)$ for some $q \in F$, $1 \leq i \leq |\lhd x \rhd|$, and natural number c. The set of strings that M accepts form the *language* of M, denoted $\mathcal{L}(M)$.

Languages accepted by 1-2DCM can be characterized by the number of times the counter changes between incrementing and decrementing while reading the input tape. Denoting this parameter by r, the following results about 1-2DCM(r) are due to Ibarra *et al.* [15]:

THEOREM 1. (Ibarra *et al.*, 1993).

– *For every $r \geq 1$, emptiness is decidable for 1-2DCM(r).*

– $\bigcup_r$ *1-2DCM(r) is effectively closed under complementation, intersection, and union.*

– *Containment and equivalence are decidable for* $\bigcup_r$ *1-2DCM(r).*

By this result, we can use 1-2DCM in decision procedures if we can bound the number of counter reversals made while processing any input. Such machines are called *reversal bounded.*

The word-walking style of TDL semantics closely resembles the transition structure of automata models such as 1-2DCM. Intuitively, a 1-2DCM can mimic the semantics by walking a word looking for indices that satisfy each time point; while walking, the machine looks for fixed-level constraint violations. As each time point is satisfied, the 1-2DCM walks back over the word to check for time-bound violations using its counter. Given the restriction to reversal boundedness, however, our 1-2DCM can only process a bounded number of match attempts before accepting or rejecting the word. Each machine used in the decision procedures attempts at most one match. This is sufficient for reversal boundedness, as argued in the appendix.

Our decision procedures construct several 1-2DCM for a given timing diagram. This section describes each machine by characterizing the language that it accepts. This is sufficient for establishing the decidability results. For the sake of thoroughness, however, the appendix provides the algorithm for constructing the actual machines and proves that those constructions satisfy the needed language characterizations. The first 1-2DCM, M_{FAIL}, accepts exactly those words for which the fully minimal index assignment starting from index 0 is not a match. To maintain reversal boundedness, it does not accept words in which a match succeeds from index 0, but one fails from a later index.

The decision procedure for the invariant semantics is simple when the regular language to be tested contains only finite words: the language generated by NFA A is contained in the language of a timing diagram iff no reachable state of A generates a word accepted by M_{FAIL} [8]. Recall Figure 2, which illustrates the index assignments detected by the semantics. The subword starting from index 4 fails to match the given timing diagram. If some state of an NFA A could generate that subword, A could not invariantly model that timing diagram. M_{FAIL} would therefore accept the subword beginning at index 4.

The problem is more difficult for ω-regular languages. Consider a word W in which no index satisfies some time point in the guarantee portion. If W is finite, M_{FAIL} can determine that no satisfying index exists by reaching the right endmarker on the input tape, which occurs at the end of W. If W is infinite, however, M_{FAIL} is insufficient. As M_{FAIL} is restricted to bounded-length input tapes, it can examine only a finite prefix of W. The lack of a satisfying index in a finite prefix does not imply that no such index exists beyond the given prefix.

A two-stage search is the key to both ω-regular decision procedures. For each time point, the first stage queries whether the time point is eventually satisfied; the second checks the fixed-level and time-bound constraints. Only the first stage examines the infinite behavior of the word; the second is finite because a satisfying index is known to exist. The property tested by the first stage, existence of a pattern, is ω-regular; only the second requires the counter. We can therefore implement the first stage using language containment on Büchi automata and the second using containment of NFA languages in reversal bounded 1-2DCM languages, both of which are decidable. Alpern and Schneider's result on splitting an ω-regular language into an infinite component (liveness property) and a finite component (safety property) is similar in spirit [1].

We now describe the remaining machines needed to implement these stages for timing diagram $D = \langle T, S, X \rangle$; the actual constructions appear in the appendix. Let $p_1 \ldots p_{\max}$ be the time points of T, and let $S = \{p_1, \ldots, p_{\mathrm{split}}\}$. For each split $< j \leq$ max, we construct Büchi automaton B_j to accept a word iff it eventually satisfies p_j. For each split $\leq j <$ max, we also construct 1-2DCM M_j accepting finite words that match D up to time point p_j and end in the index immediately following that which satisfied p_j. Words that match D up to p_j but have extra letters are not accepted; we justify this restriction shortly.

Let $A = \langle Q, \Sigma, \delta, q_0, Q_F \rangle$ be the Büchi automaton for which containment in the language of D is to be tested. $A|_q$ denotes Büchi automaton $\langle Q, \Sigma, \delta, q, Q_F \rangle$, which changes the start state of A to q. $A|_{(q,q')}$ denotes

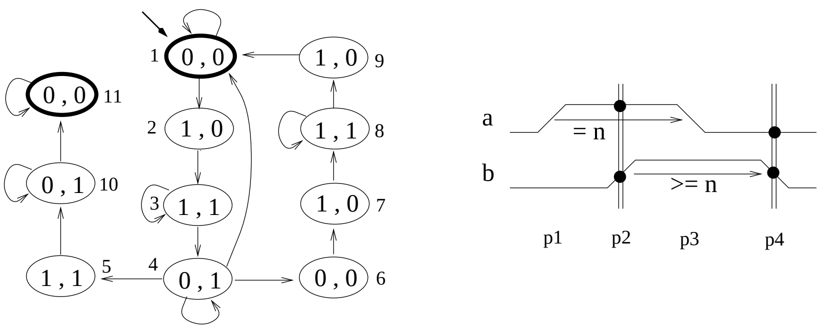


Figure 4. Automaton and timing diagram to demonstrate the decision procedures.

NFA $\langle Q, \Sigma, \delta, q, \{q'\}\rangle$, a finite-word version of A with start state q and final state q'.

Using M_{FAIL} and all machines B_j and M_j, we define several subsets of Q. We motivate and explain these definitions using the example timing diagram and Büchi automaton shown in Figure 4. Each automaton state contains an assignment a, b to variables a and b. Accepting states are in bold and the state marked with the dark arrow (the state labeled 1) is the start state. In the timing diagram, let the assume portion S contain only time point p_1, and assume fixed-level constraints (a, p_1, p_2) and (b, p_2, p_4).

For all time points p_j that are not in S, $\mathsf{Progress}_j$ contains exactly those states from which we are guaranteed to satisfy time point p_j:

$$\mathsf{Progress}_j = \{q \in Q \mid \mathcal{L}(A|_q) \subseteq \mathcal{L}(B_j)\}.$$

For Figure 4, $\mathsf{Progress}_2 = \{2, 6, 7\}$, $\mathsf{Progress}_3 = \{2, 3, 5, 6, 7, 8, 9\}$, and $\mathsf{Progress}_4 = \{2, 3, 4, 5, 6, 7, 8, 10\}$. State 4 is not in $\mathsf{Progress}_2$ because the word generated by following states 4, 5, 10, and an infinite loop at 11 never exhibits a rising transition on b.

Using $\mathsf{Progress}_{j+1}$ and M_j, Stuck_j contains states from which a walk can satisfy up to time point p_j, but fail to locate time point p_{j+1}. The need to concatenate results from M_j and results from $\mathsf{Progress}_{j+1}$ in this definition justifies the strict requirements on the end index of words accepted by M_j.

$$\mathsf{Stuck}_j = \{q \in Q \mid \exists q' \in Q : \mathcal{L}(A|_{(q,q')}) \cap \mathcal{L}(M_j) \neq \emptyset \wedge q' \notin \mathsf{Progress}_{j+1}\}$$

To compute each Stuck_j for Figure 4, we must first compute those pairs of states that can generate words accepted exactly by M_j. M_1, for example, accepts words that end exactly by satisfying a rising transition on a and contain only one such rising transition. The pairs of states generating words in M_1 are $\{(1, 2)$, $(4, 5)$, $(6, 7)$, $(9, 2)$, $(8, 2)$, $(7, 2)$, $(3, 5)$, $(2, 5)$, $(4, 7)$, $(3, 7)$, $(2, 7)\}$. For each pair (q, q') in this list, q

is in Stuck_1 iff q' is not in $\mathsf{Progress}_2$. Stuck_1 therefore contains states $\{2,3,4\}$. This corresponds to intuition; for each of these states, we can generate a word from which the match gets "stuck" trying to satisfy time point p_2. Using a similar technique, both Stuck_2 and Stuck_3 turn out empty.

Set Avoid captures those states from which the first match attempt can fail. Elements of Avoid are either elements of some Stuck_j (failure in the first stage), or can generate a word accepted by M_{FAIL} (failure in the second stage). Recall that max is the largest time point number and split is the largest time point number in S.

$$\mathsf{Avoid} = \bigcup_{j=split}^{max-1} \mathsf{Stuck}_j \cup \{q \in Q \mid \exists q' \in Q : \mathcal{L}(A|_{(q,q')}) \cap \mathcal{L}(M_{\text{FAIL}}) \neq \emptyset\}.$$

Computing Avoid for Figure 4 requires the states that can generate words accepted by M_{FAIL}. $A|_{(6,9)}$ can generate such a word because the fixed-level constraint on b is violated in state 9. Since M_{FAIL} does not have an exact ending requirement, $A|_{(6,2)}$ also generates such a word. Note that $A|_{(4,11)}$ does not generate such a word, even though the match attempt on a word generated between these states would fail; as this pair of states violates a progress condition, Stuck captures it. M_{FAIL} catches only fixed-level and time-bound constraint violations. The set of states generating words accepted by M_{FAIL} is $\{2,3,4,6\}$ (4 is here from $A|_{(4,9)}$, for example). Avoid is therefore $\{2,3,4,6\}$. Since at least one of these states is reachable from the start state (in fact, all of them are), the language of the automaton fails to model the timing diagram in Figure 4.

The sets $\mathsf{Progress}$, Stuck, and Avoid are sufficient to decide containment under the invariant semantics. Before discussing the iterative semantics, we prove our claim that verification is decidable in the invariant case. This requires one lemma and one theorem, as follows.

LEMMA 1. *State q is in* Avoid *iff A generates some word generated from q that fails to match D from index 0.*

Proof. If q is in Avoid, then by definition either A generates a word from q accepted by M_{FAIL} or A with start state q and some final state q' that is not in $\mathsf{Progress}_{j+1}$ generates a word accepted by some M_j. In the first case, the lemma holds because M_{FAIL} only accepts words that fail to match D from index 0 by construction. In the second case, some word w generated from state q to state q' matches D up to and including time point p_j by the construction of M_j. However, since q' is not in $\mathsf{Progress}_{j+1}$, by definition A must generate some word w' from q' that does not contain an index satisfying time point p_{j+1}. The concatenation of w and w' is therefore a word generated from q that fails to match D from index 0.

For the other direction, assume A generates a word w from q that fails to match D from index 0. By the definition of match, the fully minimal index assignment over w must be defined over all time points in S, but not over all time points in T. Let j be the smallest time point for which the index assignment is undefined. Let q_{j-1} be the state that A was in when it generated the index i_{j-1} assigned to time point p_{j-1}. There are now three cases:

1. No index larger than i_{j-1} satisfies p_j. In this case, by definition q_{j-1} is not in $\mathsf{Progress}_j$ and by construction, $A|_{(q,q_{j-1})}$ generates a word in M_{j-1}. By definition, q must be in Avoid.

2. A fixed-level constraint is violated before locating an index satisfying p_j. If A generated the violation in state q', then $A|_{(q,q')}$ generates a word accepted by M_{FAIL}; q is therefore in Avoid.

3. The smallest index i_j larger than i_{j-1} that satisfies p_j violates a time-bound constraint. As in the previous case, there must exist state q' such that $A|_{(q,q')}$ generates a word accepted by M_{FAIL}, which places q in Avoid.

As a corollary to Lemma 1, a state q is in Avoid iff a word can be generated from q that does not model the timing diagram under either the invariant or the iterative semantics.

THEOREM 2. *Let A be a Büchi automaton with start state q_0 and let D be a timing diagram. $\mathcal{L}(A) \subseteq \mathcal{L}(D)^{\mathrm{Inv}}$ iff no state of A reachable from q_0 is in* Avoid.

Proof. Assume $\mathcal{L}(A) \subseteq \mathcal{L}(D)^{\mathrm{Inv}}$. Then by definition, D invariantly describes every suffix of every word generated by A. Assume there exists a reachable state q that is in Avoid. By the corollary, A generates some word from state q that D does not invariantly describe. Since q is reachable, this word is extendible to one that is contained in $\mathcal{L}(A)$ but not in $\mathcal{L}(D)^{\mathrm{Inv}}$. Contradiction.

Assume $\mathcal{L}(A) \not\subseteq \mathcal{L}(D)^{\mathrm{Inv}}$. Then there exists infinite word ω that is in $\mathcal{L}(A)$ but not in $\mathcal{L}(D)^{\mathrm{Inv}}$. By definition of invariant description, there must exist an index i into ω such that the fully minimal index assignment for W, ω, and i is not a match. A was in some state q when it generated index i. By the corollary, q must be in Avoid. The existence of ω proves that q is reachable from q_0 in A.

Theorem 2 provides the decision procedure for the invariant semantics. The sets $\mathsf{Progress}$, Stuck, and Avoid are all constructible, the set of reachable states is constructible for Büchi automata, and the intersection of these sets is computable. The language containment tests in the definitions of Stuck and Avoid require converting an NFA to a 1-2DCM, which is done by determinizing the NFA to a DFA and augmenting each DFA transition with a counter whose value is never changed. The tests themselves are decidable by Theorem 1.

For the iterative semantics, the existence of a reachable state in Avoid is insufficient because matches are not attempted from all indices. The iterative semantics only attempts a match from an index if the previous test ended in the preceding index. The decision procedure must therefore track those states in Avoid from which matches can be attempted. To this end, we define 1-2DCM M_{SKIP} accepting exactly those finite words that are undefined for some time point in S and have the end index as the last index of the word. Words for which the entire timing diagram is matched and the end index is the end of the word are already accepted by $M_{\max}$ (where max is the largest time point number). Using M_{SKIP} and $M_{\max}$, we define set Avoid* to be the smallest set closed under the following conditions:

1. Avoid $\subseteq$ Avoid*.
2. $q \in Q$ is in Avoid* if there exists a state $q' \in$ Avoid* such that:
   a) $\mathcal{L}(A|_{(q,q')}) \cap \mathcal{L}(M_{\max}) \neq \emptyset$ or
   b) $\mathcal{L}(A|_{(q,q')}) \cap \mathcal{L}(M_{\mathrm{SKIP}}) \neq \emptyset$.

Essentially, Avoid* builds up chains of states that can be followed under the iterative semantics to states in Avoid. Since Avoid captures states from which a match from index 0 fails, Avoid* should provide an accurate measure of iterative description.

Returning to the example in Figure 4, to compute Avoid*, we need to consider M_{SKIP} and M_4 (recall max$=4$). Since S contains a single time point, M_{SKIP} does not accept anything (recall it accepts only words that violate a constraint within time points in S). The following pairs of states can generate words that M_4 accepts: $\{(1,1)$, $(1,6)$, $(1,10)$, $(9,6)$, $(8,6)$, $(7,6)$, $(9,10)$, $(8,10)$, $(7,10)\}$. By definition, Avoid* is $\{1,2,3,4,6,7,8,9\}$. Since the initial state (state 1) is in Avoid*, the language of the automaton does not model the timing diagram under the iterative semantics either. However, if we remove states 6 through 9 from the automaton, Avoid becomes $\{2,3,4\}$ and Avoid* becomes empty. The modified language would therefore model the timing diagram iteratively, but not invariantly.

The following theorem confirms that having an initial state in Avoid* characterizes containment under the iterative semantics.

THEOREM 3. *Let A be a Büchi automaton with start state q_0 and let D be a timing diagram. $\mathcal{L}(A) \subseteq \mathcal{L}(D)^{\text{Iter}}$ iff $q_0 \notin \textsf{Avoid}^*$.*

Proof. Assume $\mathcal{L}(A) \subseteq \mathcal{L}(D)^{\text{Iter}}$. If q_0 is in $\textsf{Avoid}^*$, there exists a sequence of states $q_0, q_1, \ldots, q_m$ such that each q_i is in $\textsf{Avoid}^*$, q_m is in $\textsf{Avoid}$, and for all $0 \leq i < m$, $A|_{(q_i,q_{i+1})}$ generates a word accepted by either $M_{\max}$ or M_{SKIP}. By the corollary, A generates some word from q_m that D does not iteratively describe. It therefore suffices to show that a match attempt could start in state q_m. This follows from the relationships between each pair (q_i, q_{i+1}) in the sequence. Each pair yields a word accepted by one of $M_{\max}$ or M_{SKIP}. By construction, these machines accept finite words that match D from index 0. Concatenating the words arising from each (q_i, q_{i+1}) pair produces a word not in $\mathcal{L}(D)^{\text{Iter}}$. Contradiction.

If $\mathcal{L}(A) \nsubseteq \mathcal{L}(D)^{\text{Iter}}$, there exists an infinite word ω in $\mathcal{L}(A)$ but not in $\mathcal{L}(D)^{\text{Iter}}$. Let $i_1, i_2, \ldots$ denote the indices from which matches are attempted on ω. Since ω is not in $\mathcal{L}(D)^{\text{Iter}}$, there is a smallest index i_m such that the attempted match fails. Let q be the state A was in when it generated i_m. By Lemma 1, q must be in $\textsf{Avoid}$. The subwords between each pair of consecutive indices up to i_m must match D from their starting index. Since all such words are accepted by either $M_{\max}$ or M_{SKIP}, the states corresponding to each index must all lie in $\textsf{Avoid}^*$. State q_0 is one such state. This completes the proof.

Theorem 3 provides the decision procedure for $\mathcal{L}(A) \subseteq \mathcal{L}(D)^{\text{Iter}}$ by an argument similar to that for Theorem 2. Unfortunately, both decision procedures are computationally expensive. Testing containment of a regular language in a 1-2DCM language lies in PSPACE. Our current procedures require quadratically many such tests relative to the number of states in the Büchi automaton. Clearly, improvement on these naive algorithms is essential for using these results in practice. Relevant upper and lower bounds on this problem remain an area for future work.

5. Related Timing Diagram Decidability Questions

By Theorems 2 and 3, properties expressible as timing diagrams are amenable to algorithmic verification. Verification researchers are also interested in other questions related to properties: for example, when do properties $\varphi_1, \ldots, \varphi_k$ entail property φ? This is useful for decomposing verification problems. If $\varphi_1, \ldots, \varphi_k$ entails φ, we can verify φ by verifying each φ_i; these may be simpler properties requiring fewer computational resources [18].

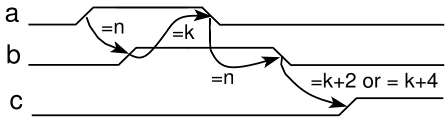


Figure 5. A pseudo-timing diagram, showing a language that well-formed timing diagrams cannot express.

Performing such reductions requires the ability to test containment between timing diagram languages. Timing diagrams without parametric timing constraints correspond to regular languages [10]; containment, and hence decomposition, is therefore decidable for such diagrams. Unfortunately, our current decision procedures are insufficient for addressing containment between arbitrary timing diagram languages, finite or infinite. Assume we want to test containment between the languages of timing diagrams T_1 and T_2; *i.e.*, we want to check whether $\mathcal{L}(T_1) \subseteq \mathcal{L}(T_2)$. Using similar techniques to those for constructing each 1-2DCM M_j, we could construct a single 1-2DCM accepting those finite words modeling T_1 under either semantics; call this machine M_{T_1}. Even in this simpler (finite) case, we cannot necessarily test whether $\mathcal{L}(M_{T_1}) \subseteq \mathcal{L}(T_2)$. Our previous decision procedures relied on the ability to enumerate the states of the Büchi automaton in order to chain results together; analogously, we would need to be able to enumerate the states of M_{T_1}. However, since the counter in a 1-2DCM is unbounded, finite enumeration of the states is not possible.

Whether containment between 1-2DCM representing timing diagram languages is decidable is an open problem. The single 1-2DCM accepting a finite timing diagram language is not reversal bounded, so Ibarra's decidability results do not apply. Containment between arbitrary 1-2DCM languages is known to be undecidable [14]. However, timing diagrams cannot express all 1-2DCM languages. The "timing diagram" in Figure 5 is not well-formed in TDL due to the disjunction in the time-bound expression. A 1-2DCM could be constructed to accept the language of this diagram under either semantics. Therefore, the undecidability of containment for 1-2DCM languages does not prove the undecidability of containment for timing diagram languages. Furthermore, it is not clear whether the expressive limitations of timing diagram languages, such as the inability to express disjunctions and many kinds of negation, affect decidability. We are interested in exploring these problem as well as possible syntactic characterizations of timing diagram language containment.

6. Conclusions

TDL is novel among timing diagram formalizations for two reasons: it supports variables in timing constraints (*parametric* timing constraints), and it has a customizable semantics. TDL also differs in its design goals. With one exception [7], other formalizations are used as interfaces to existing sentential representations for which verification algorithms already exist. In contrast, TDL is part of a larger project investigating logical features of diagrammatic design notations [10].

Our work on decidability for TDL begins to explore the computational efficacy of diagrammatic representations. Based on previous undecidability results regarding timing constraints [2, 16], we expected containment of a regular language in a timing diagram language to be undecidable. In addition, our timing diagrams clearly describe context-free and context-sensitive languages, and containment for each is known to be undecidable [13]. A paper by Harel and Raz [12], which proved that non-regular extensions of propositional dynamic logic expressions are decidable under certain structural restrictions, challenged this hypothesis.

Timing diagrams depict non-regular languages that meet certain structural restrictions. First, the non-regular aspects are concerned only with counting; the full power of stacks, necessary for general context-free languages, is not required. This suggests counter machines as a model for timing diagram languages. Second, the use of counting is localized and pre-determined, rather than arbitrary, for a given word. This allows the counter machines to be reversal bounded. Both features are crucial to the problem being decidable.

Consequently, the diagrammatic notation played an important role in developing these decision procedures. Initially, the diagrams only suggested how to design the corresponding counter machines; based on the structural restrictions, however, we suspect there is a deeper, perhaps more formal, connection between timing diagram-like notations and reversal bounded counter machines. We intend to explore this connection further in the context of more general decidability questions related to counter machine languages.

References

1. Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, October 1985.
2. Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *Proceedings of the 25th ACM Symposium on the Theory of Computing*, pages 592–601, 1993.

3. Bachi Berkane, Simona Gandrabur, and Eduard Cerny. Timing diagrams: Semantics and timing analysis. LASSO, University of Montreal, 1996.
4. J.A. Brzozowski, T. Gahlinger, and F. Mavaddat. Consistency and satisfiability of waveform timing specifications. *Networks*, 21:91–107, 1991.
5. E. Cerny and K. Khordoc. Interface specifications with conjunctive timing constraints: Realizability and compatibility. In *Second AMAST Workshop on Real-Time Systems*, June 1995.
6. Werner Damm, Bernhard Josko, and Rainer 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.
7. L.K. Dillon, G. Kutty, L.E. Moser, P.M. Melliar-Smith, and Y.S. Ramakrishna. A graphical interval logic for specifying concurrent systems. Technical report, UCSB, 1993.
8. Kathi Fisler. Containment of regular languages in non-regular timing diagram languages is decidable. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, 1997.
9. Kathi Fisler. Counter machine constructions for timing diagrams. Technical report, Department of Computer Science, Rice University, 1998. In preparation.
10. Kathryn Fisler. *A Unified Approach to Hardware Verification Through a Heterogeneous Logic of Design Diagrams*. PhD thesis, Indiana University, 1996.
11. Seymour Ginsburg. *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, 1966.
12. David Harel and Danny Raz. Deciding properties of nonregular programs. *SIAM Journal of Computing*, 22(4):857–874, August 1993.
13. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
14. Oscar H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM*, 25(1):116–133, January 1978.
15. Oscar H. Ibarra, Tao Jiang, Nicholas Tran, and Hui Wang. New decidability results concerning two-way counter machines and applications. In *Proceedings of the 20th International Colloquium on Automata, Languages, and Programming (ICALP)*, 1993. Lecture Notes in Computer Science, 700.
16. Farnam Jahanian, Aloysius K. Mok, and Douglas A. Stuart. Formal specification of real-time systems. Technical Report CS-TR-88-25, Department of Computer Science, University of Texas at Austin, 1988.
17. K. Khordoc, M. Dufresne, E. Cerny, P.A. Babkine, and A. Silburt. Integrating behavior and timing in executable specifications. In *Proceedings of Computer Hardware Description Languages and their Applications*, pages 385–402, 1993.
18. Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.
19. Philippe Mooeschler, Hans Peter Amann, and Pausto Pellandini. High-level modeling using extended timing diagrams. In *Proceedings of the European Design Automation Conference*, pages 494–499, 1993.
20. Amir Pnueli. The temporal logic of programs. In *Proceedings of the Eighth Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.
21. Y.S. Ramakrishna, L.K. Dillon, L.E. Moser, P.M. Melliar-Smith, and G. Kutty. A real-time interval logic and its decision procedure. In *Proceedings of the Thirteenth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 173–192, December 1993. Springer-Verlag Lecture Notes in Computer Science volume 761.

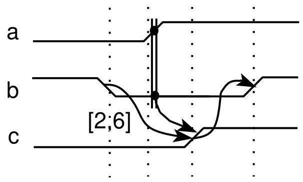


Figure 6. A timing diagram with slices demarcated by dotted lines.

22. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 133–191. Elsevier Science Publishers, 1990.
23. Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic and Computer Science (LICS)*, 1986.

Appendix

A. Machine Definitions and Theorems

Section 4 established the decidability results using a collection of automata: the Büchi automata B_j and 1-2DCM M_j, M_{SKIP}, and M_{FAIL}. It characterized each automaton in terms of its language; these characterizations were sufficient for the decidability proofs. This section completes the picture. It provides construction algorithms for creating these machines for a given timing diagram and proves that the generated machines satisfy the necessary characterizations. Before presenting the algorithms, we discuss the representations used to capture timing constraints in Section A.1 and those capturing time point satisfaction and fixed-level constraints in Section A.2. Section A.3 discusses the Büchi automata B_j. Section A.4 treats the counter machines.

A.1. Capturing Timing Constraints

A timing diagram supports timing constraints through the time-bound expressions. The counter must be able to test whether a set of time-bound expressions is satisfiable. To facilitate this, we first convert the time-bound expressions into a form that the machines can process easily. This section describes the form and provides an algorithm for converting the time-bound expressions for a given timing diagram into this form.

The semantics walk along a given word, looking for indices that satisfy each time point. The time-bounds constrain the number of steps that pass between each consecutive pair of time points. We call each pair of consecutive time points a *time slice*. Figure 6 uses dotted lines to demarcate the time slices of a timing diagram. As the figure shows, time-bound expressions may span multiple time slices. We want to convert all time-bound expressions into a form that explicitly identifies the role of each slice in each timing constraint. This supports the incremental search for each time point in the semantics.

We associate a *slice variable*, v_i, with each time slice (p_i, p_{i+1}). A slice variable captures the duration of a slice, which varies across each index assignment; given an actual index assignment I, v_i corresponds to the value $I(p_{i+1}) - I(p_i)$. We want to convert all timing requirements into inequalities in one of the following forms

$$v_{a_1} + \ldots + v_{a_z} < v_{b_1} + \ldots + v_{b_z} + m$$

$$v_{a_1} + \ldots + v_{a_z} > v_{b_1} + \ldots + v_{b_z} + m$$

where m is a natural number constant and each v_i is a slice variable. Variables and constants are classified as being on the *greater-than side* or the *less-than side* of an expression, under the obvious interpretation. The following algorithm converts time-bound objects to this representation.

ALGORITHM 1. Let $D = \langle\langle P, N, O\rangle, S, X\rangle$ be a temporally unambiguous timing diagram. The *timing inequalities for D* is a set S of linear inequalities constructed as follows.

1. For each $((w_1, p_i), (w_2, p_j), [l, u])$ in O for which u is neither zero nor ∞, add inequalities $l \leq v_i + \ldots + v_{j-1}$ and $v_i + \ldots + v_{j-1} \leq u$ to S. If u is ∞, add inequality $l \leq v_i + \ldots + v_{j-1}$ to S.

2. For each non-slice variable x that appears in some inequality

   a) For each pair of inequalities e_1 and e_2 such that x is on the less-than side of e_1 and on the greater-than side of e_2, add inequality $e'_2 \leq e'_1$ to S, where e'_1 and e'_2 solve e_1 and e_2 for x, respectively.

   b) Remove all inequalities involving x from S.

3. Rewrite each inequality in S such that

   a) it uses $<$ and $>$, instead of $\leq$ and $\geq$,

   b) it contains at most one constant, which must be a natural number, and

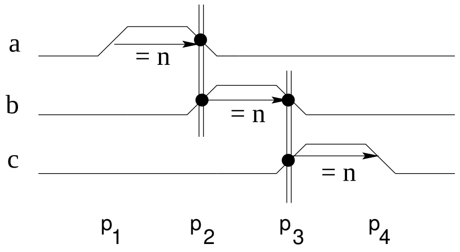


Figure 7. Example timing diagram for timing inequalities generation.

c) every slice variable appears in positive form.

Step 3 converts the constraints into the desired form. A timing inequality is *relevant to* p_i if it refers only to slice variables $v_0, \ldots, v_{i-1}$. The construction algorithm will assume some ordering on the time inequalities relevant to each time point; $C(i,k)$ denotes the k^{th} timing inequality relevant to time point p_i.

Mathematically, the algorithm is conservative, in that it may generate more inequalities than are strictly necessary. Consider the timing diagram in Figure 7. After step 1, S contains the following expressions:

$$\begin{array}{ll} v_1 \leq n & n \leq v_1 \\ v_2 \leq n & n \leq v_2 \\ v_3 \leq n & n \leq v_3 \end{array}$$

At the end of the algorithm, S contains all of the following expressions, even though the boxed ones would suffice mathematically.

$$\begin{array}{lll} v_1 \leq v_1 & \boxed{v_2 \leq v_1} & \boxed{v_3 \leq v_1} \\ \boxed{v_1 \leq v_2} & v_2 \leq v_2 & v_3 \leq v_2 \\ \boxed{v_1 \leq v_3} & v_2 \leq v_3 & v_3 \leq v_3 \end{array}$$

The tautological statements are technically unnecessary and could be eliminated. Otherwise, however, the machines rely on the conservation generation as seen in the proof of the following lemma which establishes the algorithm's correctness.

LEMMA 2. *Let I be an index assignment and p_i be a time point on which I is defined. The time-bound expressions relevant to p_i are satisfiable iff all the timing inequalities relevant to p_i hold when each v_j is replaced by $I(p_{j+1}) - I(p_j)$.*

Proof. Assume the time-bound expressions relevant to p_i are satisfiable. Then the timing inequalities must be satisfiable since Algorithm 1 replaced $I(p_{j+1}) - I(p_j)$ with v_j and applied only mathematically sound transformations to the resulting expressions. The lemma therefore holds in this direction.

For the other direction, assume the timing inequalities relevant to p_i hold when each v_j is replaced with $I(p_{j+1}) - I(p_j)$. Algorithm 1 defines a series of sets of inequalities $S_0, \ldots, S_k$, where S_0 contains the original time-bound expression inequalities, there are k variables appearing in the time-bound expressions, and each S_j results from eliminating one more variable from the inequalities (in step 2). By the assumption in this direction, set S_k is satisfiable. We complete the proof using an inductive argument, showing that if set S_j is satisfiable, then set S_{j-1} is also satisfiable.

If S_j is satisfiable, there exists an assignment A_j of values to all variables appearing in S_j that makes each expression in S_j hold. Let x be the variable that appears in S_{j-1} but not in S_j. Convert each expression in S_{j-1} into a form that solves for x. These expressions define upper and lower bounds on x. Let E_{lub} and E_{glb} represent the least upper bound and greatest lower bound on the value of x. Clearly, x is satisfiable iff $E_{\mathrm{glb}} \leq E_{\mathrm{lub}}$.

Set S_j contains such an equation for each combination of upper and lower bound expressions on x (after step 2a); this guarantees that x will be constrained by its most restrictive bounds regardless of which actual expressions provide the tightest bounds for x under a given assignment to the remaining variables. Since S_j is satisfiable by assumption, the tightest bound on x, namely $E_{\mathrm{glb}} \leq E_{\mathrm{lub}}$, is satisfiable. Set S_{j-1} is therefore satisfiable by extending A_j with a value for x that falls within the bounds $E_{\mathrm{glb}} \leq x \leq E_{\mathrm{lub}}$. The induction holds, so the time-bound expressions relevant to p_i (set S_0) must be satisfiable.

A.2. Capturing Time Points and Fixed-Level Constraints

As the machines walk the input word searching for indices that satisfy each time point, they also check that the fixed-level constraints are satisfied. These checks require letters that capture these restrictions. We define three letters for each time point: a_i captures the fixed-level constraints for time slice (p_i, p_{i+1}), ap_{iinit} and ap_i capture the requirements for an index to satisfy a time point.

DEFINITION 6. Let $D = \langle\langle P, N, O\rangle, S, X\rangle$ be a temporally unambiguous timing diagram. For each time point p_i,

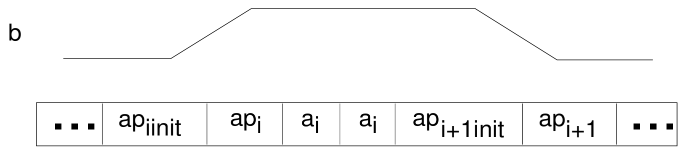


Figure 8. An example of how the letters would need to appear in a word in order to satisfy waveform b. If there were a fixed-level constraint on b, a_i would require b to equal 1.

1. partial assignment a_i captures the fixed-level constraints. If there is a fixed-level constraint (p_a, p_b, n) in X such that $p_a \leq p_i$ and $p_{i+1} \leq p_b$, $a_i(n) = 0$ (resp. 1) if n contains a low-level (resp. high-level) object between p_i and p_{i+1}.

2. partial assignment ap_{iinit} extends a_{i-1} as follows:

   - If p_i contains a concurrent edge object, ap_{iinit} assigns 0 to each signal in the concurrent edge object that contains either a rising transition object or a low-level object and ap_{iinit} assigns 1 to each signal in the concurrent edge object that contains either a falling transition object or a high-level object.

   - If p_i does not contain a concurrent edge object, ap_{iinit} assigns 0 (resp. 1) to each signal that contains a rising (resp. falling) transition object at p_i.

3. partial assignment ap_i extends a_i by assigning 0 (resp. 1) to each signal containing a falling (resp. rising) transition object at p_i.

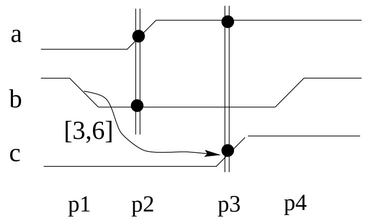
Figure 8 illustrates these letters and how they correspond to a waveform fragment. The machine correctness theorems in Section A.4 require lemmas stating that the letters capture when indices satisfy time points and fixed level constraints.

LEMMA 3. *A letter is inconsistent with a_i (and hence $ap_{i+1init}$) iff it violates a fixed-level constraint that spans time slice (p_i, p_{i+1}).*

Proof. By definition of a_i and $ap_{i+1init}$ (Definition 6).

LEMMA 4. *If time point p_i contains a transition object, an index d satisfies p_i iff it is consistent with letter ap_{iinit} and index $d+1$ is consistent with letter ap_i. If p_i contains no transition object, index d satisfies p_i iff it is consistent with letter ap_{iinit}.*

Proof. By definition of ap_{iinit} and ap_i.



$a_0 : \langle\rangle$	$ap_{2init} : \langle a=0, b=0\rangle$	$ap_3 : \langle c=1\rangle$
$ap_{1init} : \langle b=1\rangle$	$ap_2 : \langle a=1\rangle$	$a_3 : \langle\rangle$
$ap_1 : \langle b=0\rangle$	$a_2 : \langle a=1\rangle$	$ap_{4init} : \langle b=0\rangle$
$a_1 : \langle\rangle$	$ap_{3init} : \langle a=1, c=0\rangle$	$ap_4 : \langle b=1\rangle$
		$a_4 : \langle\rangle$

Figure 9. Example of the letters used by machines assuming a single fixed-level constraint of (a, p_2, p_3).

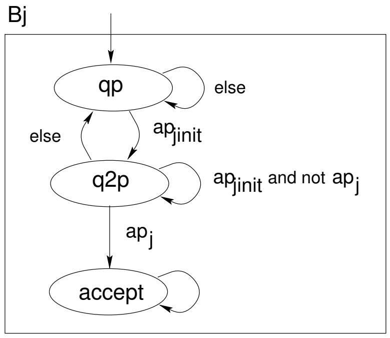


Figure 10. Machine diagram for machine B_j. B_j accepts any word on which it reaches the "accept" state.

A.3. Generating the Büchi Automata

The decidability results depend on a Büchi automaton B_j for each time point p_j larger than p_{split}. By definition, B_j should accept a word iff it eventually reads an index that satisfies time point p_j. Figure 10 shows the structure of B_j. The sole accepting state is the one labeled "accept". By the definition of Büchi acceptance, B_j will accept a word iff it reaches the accept state. Therefore, we need a theorem stating that B_j reaches the accept state iff it eventually finds an index that satisfies time point p_j.

THEOREM 4. *B_j reaches the accept state while reading word W iff W contains an index that satisfies time point p_j.*

Proof. First consider the forward direction. Looking at Figure 10, B_j only reaches state accept if it reads an index consistent with ap_{jinit} followed by one consistent with ap_j. By Lemma 4, this is precisely the condition for an index to satisfy time point p_j.

For the reverse direction, assume W contains an index that satisfies time point p_j. In each state other than accept, B_j checks whether the index is consistent with ap_{jinit}. If so, B_j moves to state $q2p$. From state $q2p$, it is clear that B_j moves to the accept state iff the index satisfies letter ap_j. Therefore, B_j cannot fail to reach the accept state on an index satisfying time point p_j.

A.4. Generating the Counter Machines

The language characterizations for each M_j, M_{FAIL}, and M_{SKIP} are relative to particular subsets of time points and various error conditions. However, these characterizations share a common foundation of fully minimal index assignments. It should not be surprising, therefore, that the machines are very similar structurally; they differ only in the prefix of time points they examine and in how they treat violations of timing diagram requirements. Accordingly, we initially take an abstract view of these machines that parameterizes over their differences. The needed theorems regarding their languages rely on a combination of lemmas about the abstract machine and lemmas about the specific details that complete each machine.

Figures 11 and 12 present an abstract view of the counter machines. Given a timing diagram, the abstract machine serves as a generator for the actual counter machines, based on the parameter instantiations provided in Table I, the letters corresponding to each time point in the diagram (Section A.2), and the constraints $C(i, k)$ corresponding to the time-bound expressions in the diagram (Section A.1).

The abstract machine contains a hierarchy of states. Each rounded rectangle abstracts over a block of states. Rounded rectangles drawn with dashed lines must be customized for the individual machines according to Table I. Rectangles enclose named blocks of states. Any parameterizations for a block appear above the upper left-hand corner of the enclosing rectangle. Ovals represent actual states. When generating the actual machines, each instantiation of a block requires fresh copies of its actual states. Furthermore, some blocks, such as those for the timing constraints, may not need to be generated for all time points; these blocks become transparent in the actual machine. The sole final state of the machine is named "final". Symbols $\triangleleft$ and $\triangleright$ denote the left and right endmarkers of the input tape, respectively.

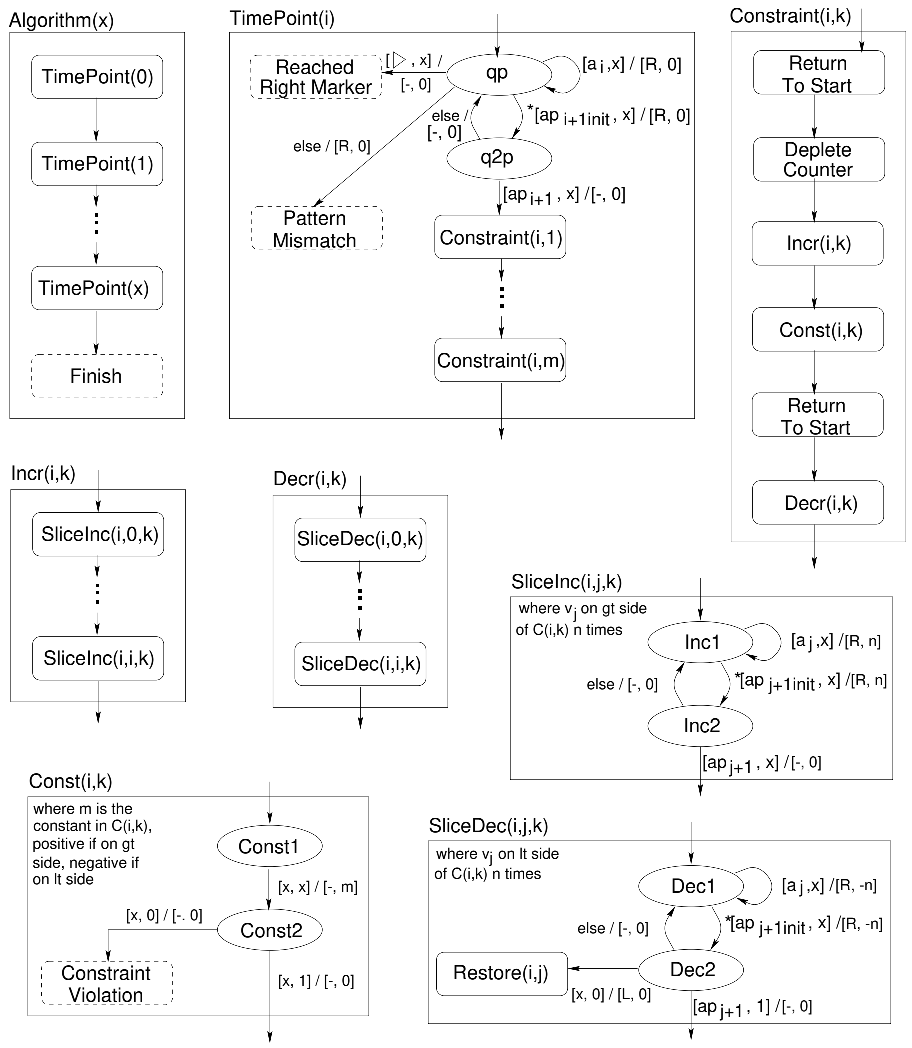


Figure 11. Diagrammatic specification of the counter machine generator, part 1.

Recalling from Definition 5, transitions in a 1-2DCM are based on the current state, the current input letter, and whether the counter's value is zero or non-zero. A transition indicates the new state, which direction (if any) to move the input head, and whether to increment, decrement, or hold the counter. The abstract machine diagram contains two types of transitions: annotated and unannotated. Unannotated transitions occur between blocks and are always enabled. Taking such a transition affects neither the position of the input head nor the value

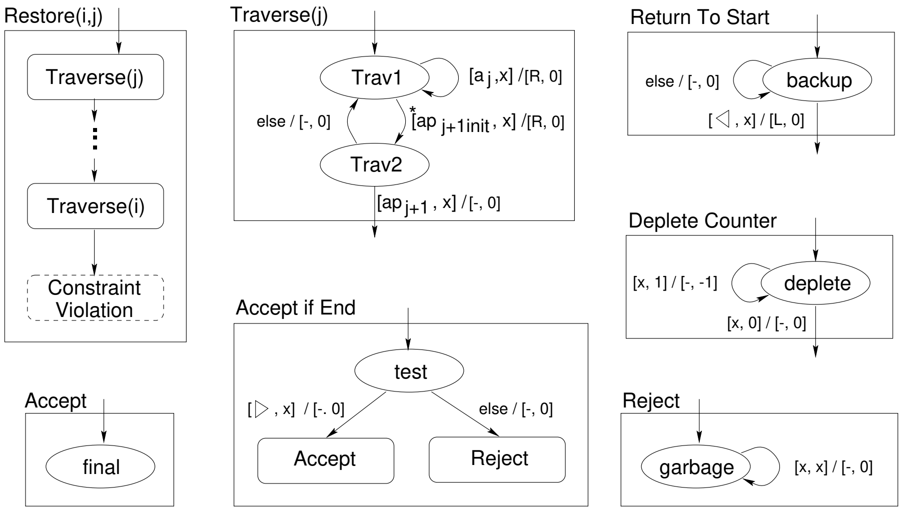


Figure 12. Diagrammatic specification of the counter machine generator, part 2.

Table I. The instantiations of the parameters for each machine generated from the diagrams in Figures 11 and 12. For M_{FAIL}, max denotes the largest time point in the timing diagram.

	M_j	M_{SKIP}	M_{FAIL}
x	$j-1$	split-1	max - 1
Finish	Accept if End	Reject	Reject
Reached Right Marker	Reject	Accept	Accept if $i >$ split Reject otherwise
Pattern Mismatch	Reject	Accept if End	Accept if $i >$ split Reject otherwise
Constraint Violation	Reject	Accept if End	Accept if $i >$ split Reject otherwise

in the counter. A machine is said to *exit from a block* if it takes a transition leading out of the rectangle enclosing a block of states.

The remaining transitions have annotations of the form $[a, c]$ / $[dir, \Delta]$. In general, such transitions are enabled if the letter being read is consistent with a and the counter's value is zero if c is 0 and non-zero if c is 1. However, we want our machines to be deterministic, and there are some states that appear to have multiple enabled transitions (since some of the machine letters defined in Section A.2 extend others). Certain transitions are therefore marked with a $*$ to indicate that they take precedence in the actual machines. The generated machines would hardwire these precedences. If a or c appears as x, any value for that position enables the transition.

Upon taking a transition $[a,\ c]$ / $[dir,\ \Delta]$, the input head moves one cell in the direction dir, where dir is one of R (right), L (left), or - (don't move). The value Δ is added to the counter. Technically, transition definitions in 1-2DCM may only contain Δ values of 0, 1, or -1. For sake of clarity, we let the abstract machine transitions contain any constant. For a specific value v of Δ, the actual machine would have v states, each making a unit adjustment to the counter's value. The counter will not decrement beyond zero, however; if it is at zero and asked to decrement, it will hold its value.

The structure of the abstract machine mimics the semantics. A machine (the Algorithm block) seeks each time point in turn (the TimePoint blocks). After locating an index matching a time point, the machine checks all constraints relative to that time point (the Constraint blocks). Each constraint is in the inequality form presented in Section A.1. The machine checks a constraint by sweeping the underlying word twice, first incrementing the counter according to the greater-than side of the constraint (the Incr block; each SliceInc block in Incr processes one time slice), then adjusting it for the constant (the Const block), then decrementing it according to the less-than side of the constraint (the Decr block; each SliceDec block in Decr processes one time slice).

The remaining blocks handle small details, error cases, and final acceptance or rejection of a word. Block Return To Start moves the input head back to the left endmarker. The Deplete Counter block decrements the counter to zero without moving the input head. The Accept block contains only the final state. The Reject block sends the machine into an infinite loop from which it will never accept the input word. Block Accept If End accepts the word if it is reading the right endmarker, otherwise it rejects the word; this block is used to enforce end index requirements. The remaining blocks, Restore and Traverse, return a machine to the index that satisfies time point p_{i+1} from an earlier time slice (p_j, p_{j+1}). Restore contains one Traverse block for each intermediate time slice.

The decidability results require each counter machine to be reversal bounded; *i.e.* there must be a finite bound on the number of times the counter changes between incrementing and decrementing. This is a structural property of the machines; we prove it based on the machine diagrams.

THEOREM 5. *Machine M generated from the diagrams in Figures 11 and 12 is reversal bounded.*

Proof. Each timing inequality $C(i, k)$ has a corresponding block Constraint(i, k). In each Constraint(i, k), M decrements the counter to set its value to zero, increments the counter while processing the greater-than side, and decrements the counter while processing the less-than side. Since the Constraint blocks are chained together, the decrement at the beginning of block Constraint(i, k) continues the decrement at the end of block Constraint$(i, k-1)$. Within block TimePoint(i), the counter therefore reverses direction $2k$ times. There are a finite number of time points. Therefore, the total number of counter reversals is bounded iff there number of timing inequalities is also finite.

A timing diagram contains a finite number of time-bound expressions. Algorithm 1, which converts time-bound expressions into timing inequalities, iterates a finite number of times and adds a finite number of expressions to the set S on each iteration. The number of time inequalities is therefore finite, so the theorem holds.

Correctness

This section establishes that the machines, as specified by Figures 11 and 12 and Table I, satisfy the language characterizations required by the decidability results in Section 4. The language characterizations are all based on index assignments satisfying the constraints of sets of time points. The proofs therefore require a correlation between the machines and the fully minimal index assignments corresponding to the input word.

Each machine implicitly defines an index assignment as it processes a word. Intuitively, a machine moves from block TimePoint(i) to block TimePoint$(i+1)$ when it locates an index that satisfies time point p_{i+1}. We can define this index assignment more precisely.

DEFINITION 7. Let W be a word and let M be a machine derived according to the diagrams in Figures 11 and 12 and Table I. Define index assignment I_M such that for each time point p_i, if M was reading index d when it exited block TimePoint$(i-1)$, then $I_M(p_i) = d - 1$.

For a given machine M, our desired language characterization theorems require that I_M and I, the fully minimal index assignment used by the semantics, agree on the values assigned to all time points. Furthermore, M must respect the necessary end indices when accepting words. The lemmas in this section work towards these two results.

Following the semantics, M first locates an index satisfying a time point, then checks whether that index also respects the time-bound expressions. After checking the constraints, M must return to the satisfying index to begin searching for an index satisfying the next time point. This first group of lemmas establishes that M returns to the index immediately after that which satisfied a time point after processing that time point's constraints.

LEMMA 5. *Assume machine M enters block TimePoint(j) while reading index d_1 and reaches block Constraint($j, 1$) while reading index d_2. If M enters block SliceDec(i, j, k) while reading index d_1, then if it exits that block, it does so while reading index d_2.*

Proof. Assume M enters block SliceDec(i, j, k) reading the same index as when it entered block TimePoint(j). Compare the transitions between states qp and $q2p$ in block TimePoint(j) with those between *Dec1* and *Dec2* in SliceDec(i, j, k). These transitions are identical minus the effects on the counter. The value in the counter only matters in state *Dec2*, in that a zero value in the counter raises a violation through the Restore block. The lemma is not concerned with the counter's value. Furthermore, as the lemma considers only the case when the computation exits SliceDec(i, j, k), we can ignore the violation case. Therefore, as the transitions are identical and deterministic, M must exit SliceDec(i, j, k) reading the same index as when it reached Constraint$(j, 1)$.

The next two lemmas establish similar results for the SliceInc and Traverse blocks.

LEMMA 6. *Assume machine M enters block TimePoint(j) while reading index d_1 and reaches block Constraint($j, 1$) while reading index d_2. If M enters block SliceInc(i, j, k) while reading index d_1, then it exits that block reading index d_2.*

Proof. Assume M enters block SliceInc(i, j, k) reading the same index as when it entered block TimePoint(j). Compare the transitions between states qp and $q2p$ in block TimePoint(j) with those between *Inc1* and *Inc2* in SliceInc(i, j, k). These transitions are identical minus the effects on the counter. The lemma is not concerned with the counter's value. Therefore, as the transitions are identical and deterministic, M must exit SliceInc(i, j, k) reading the same index as when it reached Constraint$(j, 1)$.

LEMMA 7. *Assume machine M enters block TimePoint(j) while reading index d_1 and reaches block Constraint($j, 1$) while reading index d_2. If M enters block Traverse(j) while reading index d_1, then it exits that block reading index d_2.*

Proof. By a similar argument as for Lemma 6.

LEMMA 8. *Assume machine M enters block Constraint(i, k) while reading index d_1 and exits block Constraint(i, k) while reading index d_2. If the machine enters block Constraint(i, k') while reading index d_1, then if it exits that block, it does so while reading index d_2.*

Proof. Looking at the structure of Constraint(i, k), a machine leaves this block reading the same index as when it left block Decr(i, k). A machine enters block Decr(i, k) immediately after returning the input head to the start of the word. All Decr blocks therefore start reading the same index. This lemma therefore holds if M exits Decr(i, k) and Decr(i, k') reading the same index.

A Decr block contains a sequence of SliceDec blocks. M enters Decr(i, k) and Decr(i, k') reading the same index; it therefore enters both SliceDec$(i, 0, k)$ and SliceDec$(i, 0, k')$ reading the same index. If we can prove that M must exit each SliceDec(i, j, k) reading the same index as when it left SliceDec(i, j, k'), we will have established the lemma.

Looking at the structure of the SliceDec blocks, the third parameter, k, only appears in the definition of value n. Value n affects the counter, but does not affect either the transitions taken or the movement of the input head. Furthermore, unless a machine reaches the Restore block, the value in the counter is irrelevant. The lemma assumes that M exits the Constraint block, so it must exit each SliceDec block; this eliminates the possibility of reaching the Restore block. Therefore, under the constraints of this lemma, the actual parameter k of SliceDec is irrelevant to determining the index being read upon exiting the block. M must therefore exit each SliceDec(i, j, k) reading the same index as SliceDec(i, j, k').

COROLLARY 1. *Assume machine M enters block Constraint($i, 1$) while reading index d_1 and also exits Constraint($i, 1$) while reading index d_1. If M exits block Constraint(i, k), it does so while reading index d_1.*

Proof. By a simple inductive argument using Lemma 8, since the Constraint(i, k) blocks within TimePoint(i) occur in direct sequence.

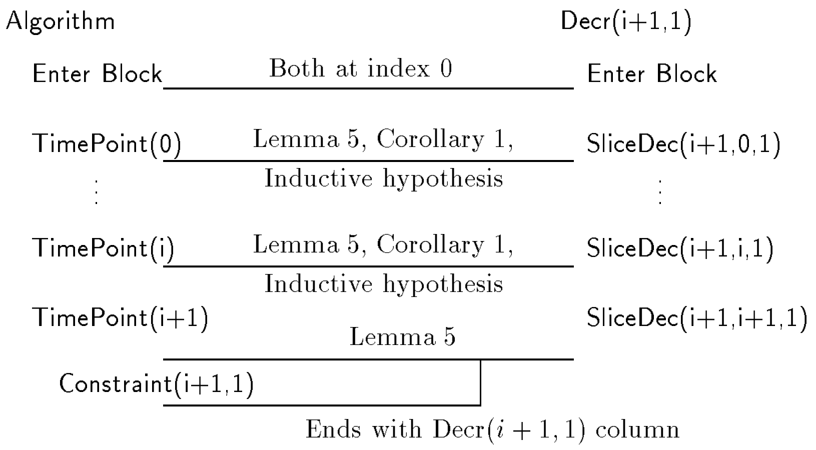


Figure 13. Diagrammatic view of the main argument used to prove Lemma 9.

LEMMA 9. *If machine M exits block TimePoint(i), then it does so reading the same index as it was when it reached block Constraint(i, 1).*

Proof. The proof is by induction on i. In the base case, i is zero. Block TimePoint(0) has no Constraint blocks because p_1 is the first time point in the timing diagram. M therefore exits block TimePoint(0) reading the same index at which it would have reached block Constraint(0,1), so the lemma holds trivially.

For the inductive step, assume the lemma holds for each time point up to and including p_i. Figure 13 outlines the proof; it shows a correlation between the indices being read at the end of each TimePoint block and those indices being read at the end of each SliceDec block. The horizontal lines indicate points at which M must be reading the same indices. For example, combining Lemma 5, Corollary 1, and the inductive hypothesis, for each i' up to and including i, M must exit block SliceDec(i', j, k) reading the same index as when it exited block TimePoint(i'). As Figure 13 shows, M leaves SliceDec($i+1, i+1, 1$) reading the same index as when it entered block Constraint($i+1, 1$). However, since SliceDec($i+1, i+1, 1$) is the last block in Constraint($i+1, 1$), M must be reading the same index when leaving Constraint($i+1, 1$) as when it entered Constraint($i+1, 1$). By Corollary 1, M is in the same index upon leaving Constraint($i+1, m$). As Constraint($i+1, m$) is the last block in block TimePoint($i+1$), the inductive case also holds.

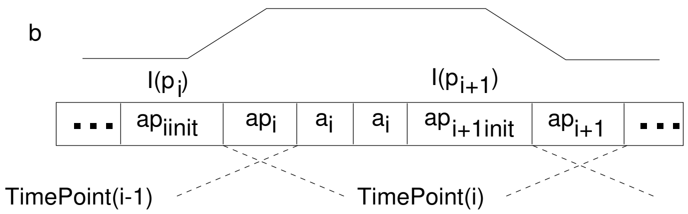


Figure 14. Augmenting the earlier letters example from Figure 8. A machine M visits the marked interval of input cells during block TimePoint(i); it processes all but the last index. The indices as assigned in an index assignment I are also noted.

COROLLARY 2. *If machine M enters block SliceInc(i, j, k) reading the same index as when it entered block TimePoint(j), then M exits SliceInc(i, j, k) reading the same index as when it exited TimePoint(j).*

Proof. By a straightforward combination of Lemmas 6 and 9.

Lemma 9 establishes that checking the timing inequalities does not affect the position of the input head. Lemmas that care only about the position of the input head can therefore ignore the Constraint blocks. We will use this observation while proving the next lemma.

We now begin to establish the correlation between I_M and I. By definition, I must assign to each time point the smallest index that satisfies it and is larger than the previously assigned indices. Lemma 10 argues that M properly locates this smallest index from a given starting point. Figure 14 depicts the relationship between the TimePoint blocks and the indices they read while processing words.

LEMMA 10. *Assume machine M enters block TimePoint(i) reading index d_1 and exits that block reading index d_2. Index $d_2 - 1$ is the smallest index at least as large as d_1 that satisfies time point p_{i+1}.*

Proof. By Lemma 9, M leaves block TimePoint(i) reading the same index as it was reading when it reached block Constraint($i, 1$). We can therefore ignore the Constraint blocks for the purposes of this lemma.

Block TimePoint(i) searches for an index satisfying time point p_{i+1}. From the time M enters the block, it checks each index against $ap_{i+1init}$. Upon finding one that is consistent, M advances one index and checks whether the next index satisfies ap_{i+1} (for time points with no transition objects, ap_{i+1} is equivalent to a_{i+1}, so the transition structure handles time points with or without transition objects).

If the current input is consistent with ap_{i+1}, M reaches the Constraint blocks, but does not move the input head. By Lemma 4, the index satisfying p_{i+1} is one earlier than the current index, as required by this lemma. If the current input is not consistent with ap_{i+1}, control returns to the top state in TimePoint(i), again without moving the input head. M therefore examines each input cell in turn, exiting TimePoint(i) while reading the index immediately following the first index that satisfies time point p_{i+1}.

Once M locates an index that satisfies a time point p_i, it must check all the time-bound constraints that are relevant to p_i. By construction, M contains a block Constraint(i, k) for each timing inequality $C(i, k)$ relevant to p_i. However, we still must prove that each Constraint block correctly checks whether its corresponding time inequality holds.

LEMMA 11. *Machine M exits block Constraint(i, k) iff I_M satisfies the corresponding inequality $C(i, k)$.*

Proof. By definition, $C(i, k)$ has one of the following two forms

$$v_{a_1} + \ldots + v_{a_z} < v_{b_1} + \ldots + v_{b_z} + m$$

$$v_{a_1} + \ldots + v_{a_z} > v_{b_1} + \ldots + v_{b_z} + m$$

where each v is a slice variable and m is a natural number constant. The algorithm that generates these inequalities (Algorithm 1) uses slice variable v_i to represent the value $I_M(p_{i+1}) - I_M(p_i)$. To establish this lemma, we therefore need to show that the states in block Constraint(i, k) correctly evaluate inequality $C(i, k)$ under this correspondence between I_M and the slice variables.

As discussed earlier, block Constraint(i, k) first returns to the beginning of the word and sets the counter to zero. Then, M increments the counter according to the greater-than side of $C(i, k)$ (block Incr(i, k)), adjusts for the constant (block Const(i, k)), and decrements the counter according to the less-than side of $C(i, k)$ (block Decr(i, k)). $C(i, k)$ is satisfied iff the counter's value is non-zero at the end of the Decr block. Therefore, a machine should exit the Decr block iff I_M satisfies the inequality.

Block SliceInc(i, j, k) tracks the number of indices lying in time slice (p_j, p_{j+1}) while processing inequality $C(i, k)$. SliceDec is similar. SliceDec and SliceInc have similar transition structures, minus the check for an empty counter in SliceDec. By Corollary 2, if M enters block SliceInc(i, j, k) reading the same index as when it entered block TimePoint(j), it leaves SliceInc(i, j, k) reading the same index as when it exited TimePoint(j). SliceInc(i, j, k) increments the counter for every index it reads except the last one; each increment increases the counter by the number of occurrences of the corresponding slice variable on the greater-than side of the inequality. If slice variable v_j does not appear on the greater-than side of $C(i, k)$, SliceInc(i, j, k) does not affect the counter. SliceInc(i, j, k) reads $I_M(p_{j+1}) - I_M(p_j) + 1$ indices (by the definition of I_M, and as shown in Figure 14), so slice variable v_j is calculated as $I_M(p_{j+1}) - I_M(p_j)$, as required.

Block SliceInc(i, j, k) therefore increments the counter to the sum of the slice variables on the greater-than side of $C(i, k)$, where each slice variable v_i corresponds to the value of $I_M(p_{i+1}) - I_M(p_i)$. Block Const(i, k) adjusts the counter with the value in the constant. The argument that the SliceDec blocks properly decrement the counter is similar to the argument for the SliceInc blocks. Therefore, the counter correctly tracks the value of the inequality over the values given by I_M for the slice variables.

Since M tallies everything on the greater-than side before considering the less-than side, $C(i, k)$ fails to hold iff the counter reaches zero before exiting block Constraint(i, k). The transition structures for blocks Const(i, k) and Decr(i, k) do not allow M to exit block Constraint(i, k) if the counter reaches zero. The lemma therefore holds.

COROLLARY 3. *I_M satisfies the time-bound expressions relative to time point p_i iff machine M exits block TimePoint($i - 1$).*

Proof. Machine M exits block TimePoint$(i - 1)$ iff it exits all blocks Constraint(i, k). The corollary follows from Lemmas 2 and 11.

The next two lemmas explore M's behavior when processing words that fail to satisfy the constraints on all the time points in the underlying timing diagram. The blocks that process such cases, Constraint Violation, Pattern Mismatch, and Reached Right Marker, are called the *error blocks*.

LEMMA 12. *Machine M reaches the Constraint Violation block iff some timing inequality $C(i, k)$ fails to hold. Furthermore, M reaches the Constraint Violation block reading the same index as when it entered block Constraint(i, 1).*

Proof. M reaches the Constraint Violation block in one of two ways: either the counter held value zero in block Const(i, k), or the counter held value zero in some block SliceDec(i, j, k) and M reached the Constraint Violation block through block Restore(i, j). In either case, M fails to exit block Constraint(i, k), so inequality $C(i, k)$ fails to hold by Lemma 11. This establishes the first part of this lemma.

For the second part of the lemma, we consider each case in turn. If M reaches the Constraint Violation block from Const(i, k), it is reading the same index as when it left block Incr(i, k). By a simple inductive argument using Corollary 2, M leaves block SliceInc(i, i, k), the last

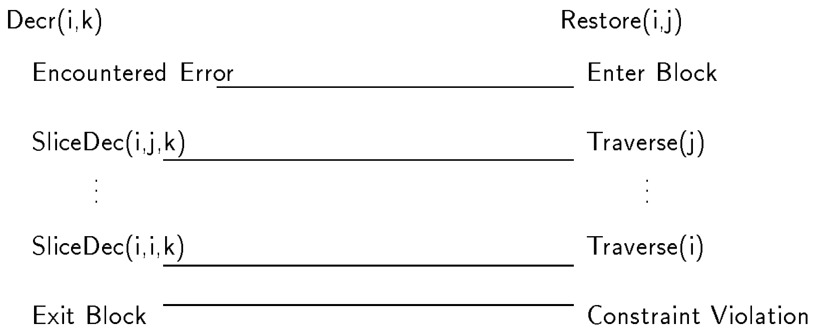


Figure 15. Diagrammatic view of an argument correlating the indices processed in block Decr with those processed in block Restore in Lemma 12.

block in block Incr(i, k), reading the same index as when it entered block Constraint$(i, 1)$, so the lemma holds in this case.

In the other case, M reaches the Constraint Violation block through some block SliceDec(i, j, k) via block Restore(i, j). Had M exited the Decr(i, k) block, it would have been reading the same index as when it entered block Constraint$(i, 1)$, as required. Hence, we must show that the path through Restore(i, j) leads to the same index as the path that leads to the exit from Decr(i, k).

Consider Figure 15. Restore(i, j) contains a series of Traverse blocks. Each block Traverse(j) has the same structure as the corresponding block SliceDec(i, j, k), minus the transition to the Restore state. When M transitions to Restore(i, j), it moves one index to the left. It therefore enters Restore(i, j) reading an index that SliceDec(i, j, k) processed from state *Dec1*. Given the transition structure similarities, M will leave Traverse(j) reading the same index as it would had it left SliceDec(i, j, k). The same correlation holds between the remaining blocks Traverse(j') and SliceDec(i, j', k) by Lemmas 5 and 7. Therefore, if M had passed through the remaining SliceDec blocks normally, it would follow the same transitions as it takes through the Traverse blocks. Accordingly, M must reach the Constraint Violation block reading the same index as it would have on exiting the Decr(i, k) block, so the lemma holds.

LEMMA 13. *A machine M reaches the Pattern Mismatch block iff it reads a letter that is inconsistent with a_i while in block TimePoint(i).*

Furthermore, M reaches the Pattern Mismatch block reading one index to the right of that which was inconsistent with a_i.

Proof. From the abstract machine diagrams, M can only reach the Pattern Mismatch block from within some TimePoint(i). Furthermore, the transition to this block is only enabled on a letter that is inconsistent with both a_i and $ap_{i+1init}$. Since $ap_{i+1init}$ extends a_i by definition, the iff statement in the lemma holds. When M takes the transition to the Pattern Mismatch block, it moves one index to the right. This establishes the second condition in the lemma.

Before proving that I_M and I agree on the indices assigned to all time points, we introduce two more lemmas. These lemmas establish that I_M is defined on a time point iff the index it assigns to that time point satisfies the constraints on that time point. These lemmas will play a crucial role in the proof that I_M and I agree on their values.

LEMMA 14. *If machine M exits block TimePoint(i) while processing word W, I_M satisfies the constraints of all time points up to and including p_{i+1} relative to W.*

Proof. In order to satisfy the constraints up to time point p_{i+1} (Definition 1), I_M must meet three requirements. First, it must satisfy every time point up to and including p_{i+1}. From M's structure, if it exits block TimePoint(i), it must also have exited blocks TimePoint(0) through TimePoint($i-1$). By Lemma 10 and the definition of I_M, I_M must therefore satisfy each time point up to and including p_{i+1}.

Second, I_M must satisfy all of the time-bound expressions relevant to p_{i+1}. Since M exits each block TimePoint(i') for i' ranging from zero to i, it must also exit each block Constraint(i', k). By Lemma 11, I_M therefore satisfies all the timing inequalities relevant to time point p_{i+1}. By Lemma 2, I_M must satisfy the time-bound constraints up to time point p_{i+1}.

Finally, I_M must respect the fixed-level constraints. By Lemma 3, letters a_i and $ap_{i+1init}$ enforce the fixed-level constraints required in each time slice. If neither of these letters is consistent with the current input, M moves to either the Reached Right Marker or Pattern Mismatch blocks. In either case, M fails to exit the respective TimePoint block, which violates the assumption on the lemma. Since M exits TimePoint(i), I_M must satisfy the fixed-level constraints.

LEMMA 15. *If machine M exits block TimePoint($i-1$) reading some index d, but contains and does not exit block TimePoint(i), then the smallest index at least as large as d that satisfies time point p_{i+1} either does not exist or fails to satisfy the constraints of time point p_{i+1}.*

Proof. Assume machine M exits block TimePoint$(i-1)$ reading some index d. Then by Lemma 14, I_M satisfies the constraints of all time points up to and including p_i. Looking at the structure of M (Figures 11 and 12), if M failed to exit block TimePoint(i), one of the following cases occurred:

1. M reached the Reached Right Marker block. This occurs if M encounters the right endmarker while looking for an index that satisfies time point p_{i+1}. Since M checks each index at least as large as d for such an index, no index at least as large as d satisfies time point p_{i+1}, so the lemma holds.

2. M reached the Pattern Mismatch block. By Lemma 13, M encountered a fixed-level constraint violation in TimePoint(i). By definition, M could not satisfy time point p_{i+1}.

3. M reached the Constraint Violation block. According to Lemma 12, M therefore fails to satisfy some $C(i,k)$, so the lemma holds.

We finally have enough intermediate results to prove the correspondence between I_M and I. This lemma provides the foundation for the language characterization theorems for the counter machines.

LEMMA 16. *I and I_M agree on the values assigned to all time points. Furthermore, when M reaches one of the Constraint Violation, Pattern Mismatch, Read Right Marker, or Finish blocks, it is reading an index that is consistent with the end index.*

Proof. We prove that I and I_M agree on all the values assigned to all time points by induction. Both the semantics and M start searching from the left endmarker for an index that satisfies time point p_1. By Lemma 10, M finds the smallest such index, if one exists, as does I by definition. There can be no constraints relevant to time point p_1 since p_1 is the smallest time point in the diagram, so the base case holds.

Assume M has entered block TimePoint(i) and I and I_M agree on the values assigned to all time points up to and including p_i. We must show they will agree on the value assigned to time point p_{i+1}. By definition, $I(p_{i+1})$ is the smallest index larger than $I(p_i)$ that satisfies the constraints of the given timing diagram up to time point p_{i+1}. By the definition of I_M, M enters block TimePoint(i) reading index $I_M(p_i)-1$ (which is the same as $I(p_i)-1$ by the inductive hypothesis). There are now two cases:

1. M reaches block TimePoint$(i+1)$. By Lemma 14, I_M must satisfy the constraints of all time points up to and including p_{i+1}. Assume

M reaches block TimePoint$(i+1)$ reading index d. Then by definition, $I_M(p_{i+1}) = d-1$. We must show that $I(p_{i+1})$ is also assigned to $d-1$. By Lemma 10, $d-1$ is the smallest index larger than $I(p_i)$ that satisfies time point p_{i+1}. Therefore, if $I(p_{i+1})$ is defined, it must have value $d-1$ (by Definition 1). $I(p_{i+1})$ can only be undefined if $d-1$ fails to satisfy the constraints of time point p_{i+1}. Since I_M satisfies these constraints and $I_M(p_{i+1}) = d-1$, however, index $d-1$ must satisfy them. Therefore, $I_M(p_{i+1}) = I(p_{i+1}) = d-1$, and the lemma holds.

2. M does not reach block TimePoint$(i+1)$. Lemma 15 indicates that a suitable index larger than $I(p_i)$ does not exist. Hence, $I(p_{i+1})$ must be undefined. By the definition of I_M, $I_M(p_{i+1})$ is also undefined since M did not reach block TimePoint$(i+1)$.

   Consider the end index requirement in the context of this case. Looking at the structure of M (Figures 11 and 12), if M failed to exit block TimePoint(i), one of the following cases occurred:

   a) M reached the Reached Right Marker block. The end index is undefined in this case; any position of the input head is consistent with this.

   b) M reached the Pattern Mismatch block. Lemma 13 covers this case.

   c) M reached the Constraint Violation block. This case occurs on a time-bound expression violation. In this case, the end index should be one larger than the index that potentially satisfied time point p_{i+1}. In other words, M should reach the Constraint Violation block reading the same index as when it entered block Constraint$(i, 1)$. Lemma 12 guarantees this.

The second case covers all the end index cases specified in the lemma with the exception of the Finish block. The machine reaches the Finish block only if it completes each TimePoint block in the Algorithm block. In this case, by the definition of I_M, M enters the Finish block reading index $I_M(p_x)+1$. As the end index for completed searches is one larger than the largest index assigned by I_M, the end index requirements hold in this case.

Having proven a series of lemmas about the abstract machine definition, we can now prove the necessary theorems regarding the behavior of each M_j, M_{FAIL}, and M_{SKIP}. These theorems will instantiate the existing lemmas with details about the individual machines according to Table I.

THEOREM 6. *A machine M_j accepts a word W iff the minimal index assignment I for W is defined for all time points up to and including time point p_j and the end index is last index into W.*

Proof. $I = I_{M_j}$ by Lemma 16. According to Table I, M_j accepts a word iff it leads to block Finish. Block TimePoint$(j-1)$ precedes block Finish. M_j reaches block Finish iff I_{M_j} satisfies the constraints, and is therefore defined, for all time points up to and including time point p_j by Lemmas 14 and 15. Thus the only remaining issue is whether M_j enforces the correct restrictions on the end index. By Lemma 16, M_j reaches block Finish reading the end index. For M_j, block Finish is really block Accept If End (by Table I). This block clearly accepts precisely those words for which the current index is the last index in the word. The theorem therefore holds.

THEOREM 7. *Machine M_{SKIP} accepts a word W iff the minimal index assignment I for W is undefined for some time point in the assume portion and the end index is the last index in W.*

Proof. $I = I_{M_{\text{SKIP}}}$ by Lemma 16. From Table I, M_{SKIP} rejects W iff it reaches the Finish block or if it reaches the Pattern Mismatch or Constraint Violation blocks reading something other than the last index into W. Therefore, M_{SKIP} should reach the Finish block or reach the Pattern Mismatch or Constraint Violation blocks reading something other than the last index iff it is defined for all time points in the assume portion, or if the end index is not the last index into W.

For the forward direction, if M_{SKIP} reaches the Finish block, it must have exited block TimePoint(split-1). $I_{M_{\text{SKIP}}}$ is therefore defined on all time points in the assume portion by Definition 7. If M_{SKIP} reaches the Pattern Mismatch or Constraint Violation blocks reading something other than the last index, then by Lemma 16, the end index is not the last index into W.

For the reverse direction, if $I_{M_{\text{SKIP}}}$ is defined for all time points in the assume portion, then by the definition of $I_{M_{\text{SKIP}}}$, M_{SKIP} must have exited block TimePoint(split-1), and hence reached block Finish. For the remaining case, assume $I_{M_{\text{SKIP}}}$ is undefined for some time point in the assume portion, but the end index is not the last index into W. Since $I_{M_{\text{SKIP}}}$ is undefined for some time point, it must have reached one of the error blocks. Since the end index is undefined in the Reached Right Marker case, M_{SKIP} must have reached either the Pattern Mismatch block or the Constraint Block. Since M_{SKIP} reaches these blocks reading the end index by Lemma 16 and the end index is not the last index into W, this case also holds.

THEOREM 8. *Machine M_{FAIL} accepts a word W iff the fully minimal index assignment I for W is defined for all time points in the assume portion, but not for some time point in the guarantee portion.*

Proof. $I = I_{M_{\mathrm{FAIL}}}$ by Lemma 16. From Table I, M_{FAIL} accepts a word iff it reaches one of Reached Right Marker, Pattern Mismatch, or Constraint Violation from some block TimePoint(i), where $i >$ split.

If M_{FAIL} reaches an error block from TimePoint(i), where $i >$ split, it must have exited block TimePoint(split-1). By Lemma 14, $I_{M_{\mathrm{FAIL}}}$ must be defined for all time points in the assume portion. By assumption, M_{FAIL} enters but does not exit TimePoint(i), so by Lemma 15, $I_{M_{\mathrm{FAIL}}}$ cannot be defined for time point p_{i+1}, which is in the guarantee portion.

For the reverse direction, assume $I_{M_{\mathrm{FAIL}}}$ is defined for all time points in the assume potion, but not for some time point p_i in the guarantee portion. By the definition of $I_{M_{\mathrm{FAIL}}}$ (Definition 7), M_{FAIL} could not have exited block TimePoint(i-1). M_{FAIL} therefore must have reached one of the error blocks. Since p_i is in the guarantee portion, $i - 1$ must be larger than split-1, so this direction also holds.