An Event Sequence Language and its Relationship to Weak Automata.[*†]

Kathi Fisler
Department of Computer Science
WPI (Worcester, MA, USA)
kfisler@cs.wpi.edu

August 18, 2003

Abstract

Many industrial verification teams are developing suitable event sequence languages for hardware verification. Such languages must be expressive, designer friendly, and hardware specific, as well as efficient to verify. While the formal verification community has formal models for assessing the efficiency of an event sequence language, none of these models also accounts for designer friendliness. We propose an intermediate language for event sequences that addresses both concerns. The language achieves usability through a correlation to timing diagrams; its efficiency arises from its mapping into deterministic weak automata. We present the language, relate it to existing event sequence languages, and prove its relationship to deterministic weak automata. These results indicate that timing diagrams can become more expressive while remaining more efficient for symbolic model checking than LTL.

1 Introduction

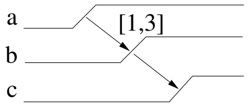
The increasing adoption of formal verification has led to a flurry of research into property specification languages for hardware verification. Large-scale efforts include Accellera's standardization of Sugar [1], Synopsys' OVA [13], and Intel's FTL [4]. Generally speaking, these are *event sequence languages*: they allow designers to express sequences of events to monitor and check during verification. The proliferation of work from industry on event sequence languages emphasizes that they must be designer friendly, expressive, and specific to the hardware domain in addition to efficient to verify. Although practical experience and theoretical results give insights into how to achieve these goals individually, few formal models attempt to address usability and efficiency simultaneously.

In the space of event sequence languages, timing diagrams provide an appealing combination of usability and efficiency. Designers have established their utility by regularly employing them as an informal design tool. Mappings from formalized timing diagrams to deterministic weak automata [8] provide effectively linear symbolic verification algorithms [5]. That timing diagrams are not more widely used as event sequence languages suggests that they lack the expressiveness needed in industrial verification [3]. Their combination of utility and efficiency, however, raises an interesting question: how expressive can we make an event sequence language while retaining both diagrammability and efficiency?

This paper explores this question by proposing a (textual) intermediate language for capturing event sequence languages. To target diagrammability, we design the core of the language around timing diagrams. To target expressiveness, we extend the core language to capture constructs from other event sequence languages. To target efficiency, we syntactically characterize which expressions in this language map to deterministic weak automata. The results of this work are twofold: first, our language provides a framework in which to assess both usability and efficiency of other event sequence languages; second, our characterization

[*] A shorter version of this paper appears in the proceedings of CHARME 2003, published by Springer-Verlag.

[†] This research is supported through NSF grant CCR-0132659.



$$\text{LTL: } \neg a \wedge \mathsf{X}(a \wedge ((\neg b \wedge \mathsf{X}(b \wedge \mathsf{F}(\neg c \wedge \mathsf{X}c)))\vee \\ \mathsf{X}(\neg b \wedge \mathsf{X}(b \wedge \mathsf{F}(\neg c \wedge \mathsf{X}c)))\vee \\ \mathsf{XX}(\neg b \wedge \mathsf{X}(b \wedge \mathsf{F}(\neg c \wedge \mathsf{X}c))))$$

Sugar: $\neg a$ & next!(a & next_e![1,3]($\neg b$ & next! (b & eventually! ($\neg c$ & next! c))))

Figure 1: Expressing an event sequence in three languages.

proves that timing diagrams can be extended with several new features—such as partial orders between events, interleaved environmental assumptions, escaping conditions, and event clocks—without losing their mapping to deterministic weak automata. Our long-term goal is to develop formal models that simultaneously characterize both usability and efficiency in event sequence languages. This paper focuses on the efficiency of verifying our proposed language; future papers will treat formal models of diagrammability as a measure of usability.

2 Preliminaries

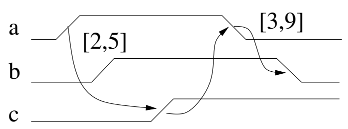
2.1 Event Sequences and Timing Diagrams

Event sequences, as their name implies, capture sequences of events on signals in a design; they express properties for verification or simulation. Regular expressions and linear temporal logic have similar goals, but also some subtle differences. Event sequences often monitor *transitions* on signals in the design, rather than just boolean values of propositions. In addition, event sequences generally capture timing constraints between events. While both regular expressions and linear temporal logic can capture these features, the resulting expressions can be rather cumbersome, especially in contrast to event sequences and timing diagrams. Figure 1 shows a simple example of the same event sequence expressed as a timing diagram, in linear temporal logic (LTL), and in Sugar.

Although timing diagrams present event sequences somewhat intuitively, they are not as expressive as some other event sequence languages. For example, textual event sequence languages easily express disjunctions, while diagrams in general capture disjunctive information poorly. The mapping from timing diagrams to weak automata, which does not hold for full LTL, demonstrates benefits to this limited expressive power. The question, then, is how far we can push timing diagrams while retaining this mapping. The timing diagram shown in Figure 2, for example, expresses some disjunction as the order of events is left unspecified (a partial order rather than a total one). This extension adds expressive power without sacrificing diagrammability or weakness. We are interested in similar extensions based on constructs from modern event sequence languages.

2.2 Weak Automata

A Büchi automaton $\langle Q, \Sigma, q_0, R, L, \mathcal{F}\rangle$ is *weak* if it has only one fair set and each of its strongly connected components has either all states fair or no states fair [9]. Weak automata are attractive in verification because symbolic cycle detection is effectively linear for weak automata, as opposed to quadratic for full LTL [5]. Deterministic weak automata are particularly interesting for their properties under complementation. Automata-based verification approaches complement automata that capture properties. In the general case, complementing a Büchi automaton can blowup the number of states exponentially. Complementing a deterministic weak automaton, however, requires only complementing the fair set; the structure of an automaton and its complement are otherwise identical. This represents a substantial savings in construction time, and more importantly, in the size of automata used to represent complemented properties.



$$\begin{aligned} C &= \langle \{a\uparrow, b\uparrow, c\uparrow, a\downarrow\}; b\downarrow\rangle \\ T &= \{\langle a\uparrow, c\uparrow, 2, 5, \mathsf{true}\rangle, \\ &\qquad \langle c\uparrow, a\downarrow, 1, \infty, \mathsf{true}\rangle, \\ &\qquad \langle a\downarrow, b\downarrow, 3, 9, \mathsf{true}\rangle\} \end{aligned}$$

Figure 2: A timing diagram with partial orders and its mapping into an event sequence.

3 An Intermediate Language for Event Sequences

This section presents a regular-expression-like notation for event sequences. We motivate the development of the language using the example timing diagram shown in Figure 2. We explain the semantics of the diagram informally; the formal details appear elsewhere [7].

To capture the diagram, the language must express transitions on signals and constraints (timing and ordering) between these transitions. Let propositional literals (p, $\neg q$) denote boolean values and propositional variables annotated with arrows ($p\downarrow$, $p\uparrow$) denote falling and rising transitions, respectively. Let semicolons denote concatenation (temporal sequencing) of events. Using these notations and reading off the timing diagram from left to right suggests the expression $\langle a\uparrow; b\uparrow; c\uparrow; a\downarrow; b\downarrow\rangle$. If we interpret semicolons as implying order between events (a common interpretation of concatenation), this expression is inconsistent with the semantics of the timing diagram. The rising transitions on a and b may occur in any order since no constraint orders them (the falling events on a and b, in contrast, must occur in order). The event sequence language must therefore support partial, rather than only total, orders between events.

Timing diagrams consist of totally-ordered regions within which individual events are partially ordered. For sake of generality, our event sequence language supports hierarchical combinations of ordered, unordered and iterated groups of events. In the formal syntax and semantics that follows, we refer to these groups of events as *clusters*. We capture partial orders within unordered clusters using a separate annotation for transition (timing) constraints between events; a timing constraint specifies the events covered, lower and upper bounds on the time between the events, and the clock against which the bounds are measured (true specifies the system clock). This approach treats constraints between events uniformly, whether they occur in ordered or unordered clusters. Figure 2 shows the resulting event sequence for our example timing diagram.

3.1 Syntax

The timing diagram example suggests the following syntax for event sequences:

Definition 1 Clusters are defined hierarchically as follows:

- An *event* is a conjunction of values of and transitions on variables that contains at least one transition. Propositional literals (p, $\neg q$) denote boolean values; propositions with arrows ($p\downarrow$, $p\uparrow$) denote transitions.
- A *cluster* is either:
  - a single event, or
  - an *unordered cluster* $\{C_1, \ldots, C_k\}$ where each C_i is a cluster, or
  - an *ordered cluster* $\langle C_1; \ldots; C_k\rangle$ where each C_i is a cluster, or
  - a *repeating cluster* C^M where C is a non-repeating cluster and M is a positive number, $*$, or $+$ (called a *repetition marker*; markers $*$ and $+$ are called *unbounded*).

An event sequence consists of a (top level) cluster and three kinds of modifiers. Temporal constraints, already motivated, may be relative to a designer-specified *event clock*, as captured by a boolean expression

$C = \langle a \uparrow^{+}; \{b \uparrow, c \uparrow\}; d \uparrow \rangle$
$H = \{b \uparrow, c \uparrow\} \rightarrow a$
$T = \{\langle c \uparrow, d \uparrow, 2, 5, \mathsf{true} \rangle\}$
$S = \{\text{accept-if-don't-complete}(a \uparrow^{+})\}$

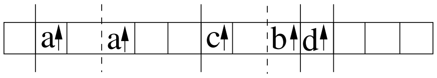


Figure 3: A sample event sequence and an example of its semantics.

(this is a common feature in event sequence languages). To indicate that certain variables hold value during regions (between events) in a diagram, *holding patterns* constrain variable values within clusters. To allow portions of diagrams to serve as assumptions rather than requirements, *escape conditions* capture circumstances under which the sequence should be immediately rejected or accepted.

Definition 2 An *event sequence* is a tuple $\langle C, H, T, S \rangle$ where C is a cluster, H (the holding patterns) is a partial function from C to propositional formulas, T is a set of temporal constraints with no clock overlaps and S is a set of escape conditions.

- A *temporal constraint* is a tuple $\langle e_1, e_2, l, u, clk \rangle$ where e_1 and e_2 are (uniquely identified[1]) events in C, l is a positive integer, u is either an integer at least as large as l or the symbol ∞, and clk is a boolean expression (the clock for the constraint; true indicates the system clock). Events e_1 and e_2 may lie in different clusters, but then they must lie in the same repeated clusters.
- An *escape condition* has one of three types, where X is a boolean expression over events (the events need not be in C) and C' is a cluster within C:
  - "accept if don't complete C'"
  - "reject if see X in C'"
  - "accept if see X in C'"

Figure 3 illustrates an event sequence of some number of rising transitions on a, followed by rising transitions on b and c (in either order), followed by a rising transition on d. The transition on d must occur between 2 and 5 ticks (inclusive) after the transition on c (the timing constraint), signal a must remain true until the transition on d occurs (the holding pattern), and the rest of the sequence is only checked if the transition on a occurs (the escape condition).

The language contains some redundancy for sake of clarity: ordered clusters, for example, can be viewed as unordered clusters plus timing constraints. To simplify the semantics and proofs, we assume that all sequences are in *reduced form*, in which all clusters C^{+} are replaced with $\langle C; C^{*} \rangle$, all C^{M} for a concrete number M are replaced with an ordered cluster of M copies of C, and all ordered clusters $\langle C_1; \ldots; C_k \rangle$ are replaced with unordered clusters and timing constraints that require an event from each C_i to occur before an event from each C_{i+1}.

3.2 Semantics

The semantics of event sequences is defined in terms of languages over infinite words, where each character in a word is an assignment of boolean values to variables. An infinite word models an event sequence if there exists a mapping from the clusters in the sequence to ranges of indices into the word (herein called *windows*) such that the windows assigned to each cluster preserve the cluster's constraints; these mappings are called *index assignments*.

As an example, consider the event sequence and word shown in Figure 3. The word is divided into windows per cluster (demarcated by solid lines), and subwindows as necessary for nested clusters (demarcated

[1] A numbering scheme could distinguish syntactically similar events.

by dashed lines). While an event sequence may contain unordered clusters, each such cluster appears in a particular order within a given word. An ordering of an unordered cluster is called a *schedule*.

Definition 3 Given an unordered cluster $C = \{C_1, \ldots, C_k\}$, a *schedule of* C is a sequence $CO_1, \ldots, CO_j$ of non-empty subsets of C such that

- $CO_1, \ldots, CO_j$ partition C,
- In every CO_i that contains multiple elements of C, all elements of CO_i are single events (rather than other complex clusters), and
- For each timing constraint $\langle e_1, e_2, l, u, clk \rangle$ such that $e_1 \in CO_i$ and $e_2 \in CO_j$, $i < j$.

We begin with definitions about windows and the mappings from clusters to windows.

Definition 4 Given a word W, a *window of* W is a subword of W; a pair of indices into W, denoted $[i, j]$ where $i \leq j$, defines a window. Furthermore,

- An individual index i defines a trivial window $[i, i]$.
- Window $[i_1, i_2]$ *contains* window $[i_3, i_4]$ iff $i_1 \leq i_3$ and $i_4 \leq i_2$.
- Window $[i_1, i_2]$ is *earlier* than window $[i_3, i_4]$ iff $i_1 < i_3$ or $i_1 = i_3$ and $i_2 < i_4$.
- Given a window $w = [start, end]$, a sequence $[s_1, e_1], \ldots, [s_k, e_k]$ forms a *non-overlapping covering sequence of windows for* w if $s_1 = start$, $e_k = end$, and for all $1 \leq j < k$, $e_j < s_{j+1}$.

Definition 5 A (partial) *index assignment* for event sequence V and word W is a (partial) function from the clusters in and nested within V to non-empty sets of windows of W.

A window must meet certain requirements in order to capture the constraints of a cluster: for example, windows assigned to single events must reflect the variable values within those events, each window assigned to a cluster must contain subwindows satisfying the elements of the cluster, and the distance between windows assigned to events must respect timing constraints. Two characteristics of index assignments capture these constraints: *structural validity* requires that windows mapped from events satisfy those events and that the index assignment respects structural features (such as cluster nesting); *constraint validity* requires that index assignments respect timing constraints and holding patterns. The following three definitions formalize these notions of validity.

Definition 6 Let $E = v_1 \wedge \ldots \wedge v_k$ where each v_i is a proposition, its negation, or a rising or falling transition on a proposition.[2] Let W be a word and i an index into W. Let $W_i(q)$ denote the value of proposition q at index i into W. Index i *satisfies* E if for every v_i, $W_i(p) = 0$ if $v_i = \neg p$, $W_i(p) = 1$ if $v_i = p$, $W_{i-1}(p) = 0$ and $W_i(p) = 1$ if $v_i = p \uparrow$, and $W_{i-1}(p) = 1$ and $W_i(p) = 0$ if $v_i = p \downarrow$.

Definition 7 Let V be an event sequence, W a word, and I a partial index assignment for V and W. I is *structurally valid* iff for every cluster C in V:

- If C is an event, then for every $[i, i] \in I(c)$, i satisfies C (Defn 6).
- If C is a repeating cluster C'^*, then for every wp in $I(C'^*)$ there exists a natural number m and some sequence $wp_1, \ldots, wp_m$ of non-overlapping covering windows for wp such that each $wp_i \in I(C')$.
- If C is an unordered cluster $\{C_1, \ldots, C_k\}$, then for every window $w \in I(C)$ there exists a schedule $CO_1, \ldots, CO_j$ for C and a sequence $w_1, \ldots, w_j$ of non-overlapping covering windows for w such that for all $i \leq h \leq j$ and all $e \in CP_h$, $w_h \in I(e)$.

Definition 8 Let $V = \langle C, T, H, S \rangle$ be an event sequence, let W be a word, and let I be an index assignment for V and W. I is *constraint valid* for V and W iff

[2] E is more general than an event, which requires at least one transition.

1. I satisfies the holding patterns, in that for all clusters C', every $x \in H(C')$ and every window $[w_1, w_2] \in I(C')$, every index $w_1 \leq i \leq w_2$ satisfies x, and
2. I satisfies the timing constraints, in that for every $\langle e_1, e_2, l, u, clk \rangle \in T$ and every $t_1 \in I(e_1)$ and $t_2 \in I(t_2)$ such that t_1 and t_2 fall in a common window for the smallest cluster containing both e_1 and e_2, the number of indices satisfying clk between t_1 and t_2 (inclusive) is within the range $[l, u]$.

The semantics generally requires index assignments to be both structurally and constraint valid. For sake of conciseness, we use the single term *valid* for index assignments that satisfy both sets of criteria.

Constraint validity handles timing constraints and holding patterns, but not escape conditions. The next two definitions handle escape conditions. Definition 12 relates words and event sequences based on the existence of index assignments that may or may not invoke escape conditions. Given index assignment I, let $\overline{I}$ be the inverse of I (mapping windows to sets of clusters).

Definition 9 Let V be an event sequence, W a word, and I a valid index assignment for V and W. Let E be an escape condition of type "accept/reject if see X in C". Index i into W *invokes* E *under* I if $i \in I(C)$, i satisfies X, and I is defined for all clusters in the images of $\overline{I}$ for windows occurring before i. We also say that I *invokes an escape condition of* V.

Definition 10 Let V be an event sequence, W a word, and I a valid index assignment for V and W. I *loops under escape condition* E if E is of the form "accept if don't complete C" and I is defined for all clusters in the images of $\overline{I}$ for windows occurring before i, but not for a window containing i.

For the semantics to yield a deterministic procedure for checking whether a word satisfies an event sequence, index assignments must assign the fewest and earliest possible windows to clusters (in particular, this renders both * and scheduling of unordered clusters deterministic for a given word). We formally define this notion of minimality by constructing a partial order over valid index assignments: intuitively, index assignment I_1 occurs before I_2 in this order if I_1 assigns events to earlier windows than does I_2.

Definition 11 Let V be an event sequence and let W be a word. Let I and I' be non-equivalent index assignments for V and W. Let $RgW(I)$ denote the union of the sets in the range of I (in other words, $RgW(I)$ yields the set of all windows that I assigns to events). $I \prec I'$ iff

1. the earliest window in one but not both of $RgW(I)$ and $RgW(I')$ is in $RgW(I)$, or
2. $RgW(I) = RgW(I')$ but for w, the earliest window such that $\overline{I}(w) \neq \overline{I'}(w)$, $\overline{I'}(w) \subset \overline{I}(w)$.

An index assignment I is *locally minimal* if there does not exist an index assignment I' such that $I' \prec I$. Given a set Σ of index assignments, $I \in \Sigma$ is *minimal in* Σ iff for all $I' \in \Sigma$, $I \prec^* I'$, where $\prec^*$ is the transitive closure of $\prec$.

Lemma 1 *Let I and I' be different index assignments for the same event sequence V and word W. At most one of $I \prec I'$ or $I' \prec I$ holds.*

Proof: If $RgW(I) \neq RgW(I')$, then the first case of the definition applies. There must exist an earliest window that is not in both $RgW(I)$ and $RgW(I')$, so at most one of $I \prec I'$ or $I' \prec I$ will hold. If $RgW(I) = RgW(I')$ but $I \neq I'$ (by assumption), then I and I' must distribute the windows in their equivalent ranges differently across events. Let w be the smallest window that I and I' assign to different events. By choice of w, $\overline{I}(w) \neq \overline{I'}(w)$, so at most one of $I \prec I'$ and $I' \prec I$, holds by the second case of the definition; if neither $\overline{I}(w) \subset \overline{I'}(w)$ nor $\overline{I'}(w) \subset \overline{I}(w)$, then $\prec$ does not order I and I'.

In general, the definition of $\prec$ does not guarantee a unique minimal index assignment for a given V and W. Consider the event sequence $\{\{a, b\}, \{c, d\}\}$ and a word that satisfies both a and c in indices 0 and 2, and both b and d in indices 1 and 3. Two possible index assignments I_1 and I_2 for this sequence and word would be $I_1(a) = 0, I_1(b) = 1, I_1(c) = 2, I_1(d) = 3$, and $I_2(a) = 2, I_2(b) = 3, I_2(c) = 0, I_2(d) = 1$. Schedules of unordered clusters may only overlap individual events (but not more complex clusters); I_1 and I_2 would both be locally minimal, but incomparable to one another. The restrictions outlined in Section 5.2 for event sequences to yield deterministic automata are sufficient to yield unique minimal index assignments, as we prove in that section.

We now define when a word models an event sequence:

Definition 12 Let V be an event sequence and let W be a word. $W \models V$ if there exists a locally minimal and valid index assignment I for V and W such that I is either defined for all clusters in V, or I loops under some escape condition in V, or I invokes some escape condition in V.

The semantics captures one occurrence of an event sequence, rather than the multiple occurrences needed to treat an event sequence as an invariant. The one-occurrence semantics offers two benefits: it provides a foundation for defining different multiple occurrence semantics [7], and it enables the mapping to weak automata. Two multiple occurrence semantics appear common in practice: an invariant semantics in which a word must match an event sequence starting from every index, and an iterative semantics in which the event sequence must be matched repeatedly, as if it were enclosed in a repetition cluster. Although the syntax and semantics here directly capture the iteration semantics, Section 5.3 shows that iteration generally breaks weakness; for this reason, we find it advantageous to define the semantics such that iteration at the outermost level can be handled separately from the core sequence being iterated over. In addition, separating the definition of sequences from their behavior over multiple occurrences lends a certain cleanliness to the definitions and proofs.

Constructing automata for a single occurrence would seem fairly limiting, however, as verification algorithms usually require an automaton or monitor for the entire property, including multiple-occurrences; for verification, algorithms actually need the negation of automaton for multiple occurrences. Fortunately, the restriction to single occurrences is not as limiting as it might seem: for those sequences that map to deterministic weak automata, we can construct the automaton for the negation of the sequence as an invariant by complementing the fair states and adding self-loops at the initial states [8]. We can handle iterative semantics directly in the logic as just described. This restriction therefore supports our theoretical results without much cost in utility or expressiveness.

4 Relationship to Existing Event Sequence Languages

To motivate the intersection between our simultaneous goals of diagrammability and efficiency, this section shows how several features of existing event sequence languages do or do not map into the proposed intermediate language.

4.1 Timing Diagrams

Section 3 illustrated the connection between timing diagrams and our proposed event sequence language. The language presented here extends our previous results on the relationship between timing diagrams and weak automata [8] in two ways. The previous result held for timing diagrams with a total order on their transitions and a prefix of the diagram as an environmental assumption (as in, "if the rising transition on a occurs, then match the whole diagram"). As a corollary to the results in this paper, timing diagrams with partial event orders and multiple non-contiguous assumptions on the environment also map to deterministic weak automata. We view environment assumptions as events that are only constrained if they occur [6]; unlike other events, their failure to occur does not violate the diagram's requirements. For the diagram in Figure 2, we could treat the two transitions on a as environment assumptions by rewriting the event chain using nested clusters (as $\langle\{a\uparrow, b\uparrow, c\uparrow, a\downarrow\}; b\downarrow\rangle$ and adding "accept-if-don't-complete" escape conditions on the two clusters for a.

The proposed language is more expressive than our current timing diagram formalization. Consider the cluster $\langle a\uparrow^*; b\uparrow\rangle$. The timing diagram semantics requires all depicted transitions to occur unless an escape condition matches, so this expression (without escape conditions) is currently not expressible as a timing diagram (since $a\uparrow$ might not occur). Similar examples involving repetitions also exist. Furthermore, timing diagrams cannot handle the arbitrary cluster nesting allowed in the current language. Enriching the timing diagram notation, as Amla et al. have done [3], could resolve some of these issues; this remains an issue for future work.

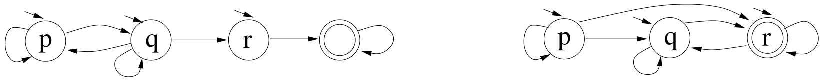


Figure 4: Automata for two LTL formulas: the one on the left is weak while the one on the right is not.

4.2 LTL, Sugar, and FTL

Sugar and FTL are similar in that each extends conventional LTL. Since there exist LTL formulas that cannot be captured by weak automata, certain FTL and Sugar formulas will not map into our intermediate language. Weakness primarily characterizes the location of fair sets in automata. In LTL, fairness constraints arise from combinations of eventualities and cycles (the operators U and G). Figure 4 shows automata that capture two formulas: $(p \;\mathsf{U}\; q) \;\mathsf{U}\; r$ and $p \;\mathsf{U}(\mathsf{G}(q \;\mathsf{U}\; r))$. The first example yields a weak automaton and corresponds to cluster $\langle (p^*; q)^*; r\rangle$. The second corresponds to cluster $\langle p^*; \langle q^*; r^+\rangle^*\rangle$ with escape condition "accept if don't complete r^+"; this expression violates our syntactic restrictions for weakness presented in Theorem 5 (Section 5.3).

One key difference between these two formulas is that the second contains a repetition within its last cluster, while the first does not. This same difference characterizes the automata for the regular expressions $(aa)^*$ and $(aa)^*b$, the first of which cannot be captured by a deterministic weak automaton while the second one can. An automaton can recognize a nonrepeating final pattern without creating a fair set. This motivates our characterization of weakness: the final cluster cannot end with an unbounded repetition marker.

Certain other features of Sugar and FTL do not adversely impact weakness. FTL's *change_on* and *reject_on* constructs indicate when a sequence should be immediately accepted or rejected; escape conditions capture such scenarios in the proposed intermediate language. For example, augmenting $(p \;\mathsf{U}\; q) \;\mathsf{U}\; r$ with escape condition "accept if see *reset* in q" would introduce a new state labeled *reset* with an incoming edge from the state for q; this automaton is also weak.

4.3 OVA

Of the recent event sequence languages discussed in this paper, OVA most closely matches the proposed language. Unlike Sugar and FTL, OVA does not explicitly support LTL or CTL operators. The OVA *istrue* construct maps into holding patterns, and their non-overlapping event clocks map into ours. Unlike the proposed language, however, OVA can express disjunction among sequences and negation of sequences. Our language does not support negation because negated sequences generally cannot be realized diagrammatically. Our language does, however, still support constructing deterministic weak automata for the negations of event sequences, as outlined at the end of Section 3.

5 Relationship to Deterministic Weak Automata

This section characterizes which sequences in our language map to deterministic weak automata; almost all do, with the exception of those with particular interactions between escape conditions and repeated clusters. We construct an automaton corresponding to the semantics, prove the construction sound, then characterize when the resulting machine is both weak and deterministic.

Given an event sequence V, we construct a Büchi automaton that accepts all words with a prefix that models V; this captures the "single occurrence" nature of the semantics, as discussed in Section 3.2. Figure 5 illustrates the intuition behind the expansion. The construction recursively expands states corresponding to clusters until all states correspond to individual events. Holding patterns, escape conditions, and the ordering aspects of timing constraints are incorporated as this expansion proceeds. The durational aspects of timing constraints are handled in a final phase once all states correspond to individual events. Each intermediate machine during the computation abstracts the final machine, in that if there is no path from one abstract state to another, then there is no path from any state in the expansion of the first to the expansion of the second in the final machine.

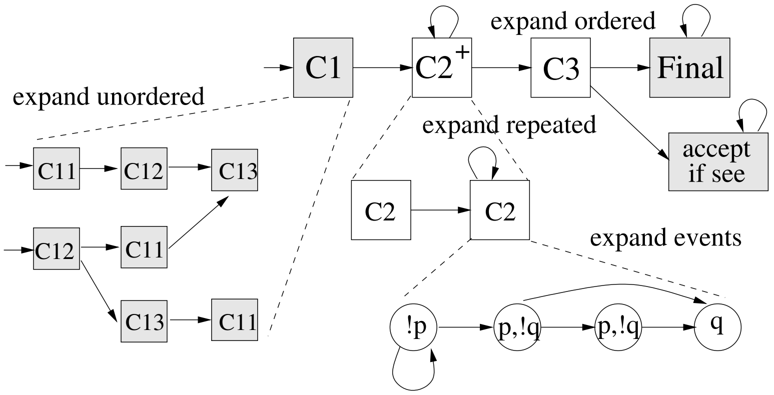


Figure 5: Overview of the automaton construction algorithm.

The construction creates edges between abstract states based on which clusters can precede or follow other clusters; it also relies on notions of the first and last subclusters that could be encountered in a cluster. These concepts match intuition and examples of all four notions follow the definition. The theorem in Section 5.2 also refers to first and last *events*, which are obtained by iterating the first and last computations on clusters until they contain only events. We present examples first, followed by the formal definitions.

Examples: Given event sequence $\langle C_1; C_2; C_3^*; C_4^+\rangle$, next($C_2$)=next($C_3$)=$\{C_3, C_4\}$ and prev(C_3)=$\{C_2, C_3\}$. Given an event sequence $\langle C_1; \{C_{21}, C_{22}^*\}; \{C_{31}, C_{32}\}^*; C_4\rangle$ that has a timing constraint from C_{21} to C_{22}, next(C_{21})=$\{C_{22}, C_{31}, C_{32}, C_4\}$. For the first and last sets, first($\{C_{21}, C_{22}^*\}$)=$\{C_{21}\}$, last($\{C_{21}, C_{22}^*\}$)=$\{C_{22}^*\}$, and first($\{C_{31}, C_{32}^*\}$) = last($\{C_{31}, C_{32}^*\}$) = $\{C_{31}, C_{32}^*\}$.

Definition 13 Let C be a cluster immediately contained in a cluster C^P (if C has no enclosing cluster, next(C) is empty). If $C^P = \langle C_1; \ldots; C_k\rangle$ and $C = C_i$ for $i < k$, then next(C) is C_{i+1} if C_{i+1} is not an repeating-* cluster and $\{C_{i+1}\}\cup$ next(C_{i+1}) if C_{i+1} is an repeating-* cluster. If $C = C_k$, then next(C) is next(C^P). If C is an repeating-* cluster, next(C) also includes C. The case for unordered clusters unions similar results over all possible schedules, and repeated clusters C have next(C) as $\{C\}\cup$ next(C^P).

Definition 14 Let C be a cluster immediately contained in a cluster C^P (if C has no enclosing cluster, prev(C) is empty). If $C^P = \langle C_1; \ldots; C_k\rangle$ and $C = C_i$ for $i < k$, then prev(C) is C_{i-1} if C_{i-1} is not an repeating-* cluster and $\{C_{i-1}\}\cup$ prev(C_{i-1}) if C_{i-1} is an repeating-* cluster. If $C = C_0$, then prev(C) is prev(C^P). If C is an repeating-* cluster, prev(C) also includes C. The case for unordered clusters unions similar results over all possible schedules, and repeated clusters C have prev(C) as $\{C\}\cup$ prev(C^P).

Definition 15 Let C be a cluster. If C is a repeating cluster C'^M, then first(C) is first(C'). If C is an unordered cluster, then first(C) is the union of all C_0 in valid schedules $C_0, \ldots, C_k$ of C. If C is a single event, then first(C)=$\{C\}$.

Definition 16 Let C be a cluster. If C is a repeating cluster C'^M, then last(C) is last(C'). If C is an unordered cluster, then last(C) is the union of all C_k in valid schedules $C_0, \ldots, C_k$ of C. If C is a single event, then last(C)=$\{C\}$.

Definition 17 Let e be an event with parent cluster C. The *next events of* e, denoted next-e(e), is the set of first events of all clusters in next(C). The *previous events of* e, denoted prev-e(e), is the set of last events of all clusters in prev(C).

The construction algorithm appears in Figures 6 through 8. We capture timing constraints with additional automata that run concurrently (via a cross product) with the automaton for the core of the sequence. In order to preserve weakness, our construction must create one copy of the constraint automaton for each

Algorithm 1 To construct an automaton for event sequence $\langle C, T, H, S\rangle$:

1. Create a state *Final* with a self loop and mark it fair.
2. Create a state for C and mark it initial, final, and unexpanded.
3. Repeatedly select an unexpanded state N for some non-event cluster C and
   - Add holding patterns and edges for the escape conditions for C to N (Figure 7).
   - Expand N according to the type of C and remove N.
   - If N was marked initial (resp. final), mark the new states for all first (resp. last) clusters of C initial (resp. final). Copy all other propositional annotations (including fair) from N to the new states from the expansion.
   - Handle the timing constraints (Figure 8).
   - Define the fair states of the automaton to be all those in the cross product of the core automaton and the timing constraints such that either (1) the core machine is in a fair state and all timing constraints are in their initial or final states, or (2) the core machine is in a fair state arising from an accepting escape condition and none of the timing constraint automata are in the error state.
4. Add an edge from each state marked final to the state *Final.*

Expand Repeated Clusters: For a state N for repeated cluster C^*, add an edge from the state for each previous cluster of C^* to that for each next cluster of C^* and mark N as capturing C instead of C^*.

Expand Unordered Clusters: For a state N for unordered cluster $C = \{C_1, \ldots, C_k\}$:

- For every schedule $CO_1, \ldots, CO_h$ of C, create a chain of abstract states $CON_1, \ldots, CON_h$. For every non-self-loop edge coming into N, add an edge from the same source to CON_1. For every non-self-loop edge leaving N, add an edge from CON_k to the target of the original edge.
- Eliminate unnecessary nondeterminism by merging states with the same incoming transitions and labels into single states (this shares common prefix states across the various permutations).
- If N had an edge to itself, add an edge from each sink state in the subgraph that expands N to each source state in the subgraph that expands N.

Expand Events: For a state N corresponding to an event E, replace N with two states, one that searches for the first half of the event and one that searches for the second half of the event. Figure 9 shows the transitions to generate, where $e1 - 1$ and $e1 - 2$ are the two states that expand the state for e_1; $e1 - 1$ captures having seen the first half of e_1, while $e1 - 2$ captures having seen the second half of e_1. The labels $e1/1$ and $e1/2$ on the transitions refer to the two sets of propositional values required to satisfy e_1; for a rising transition on a, $e1/1$ would require $a = 0$, while $e1/2$ would require $a = 1$.

Figure 6: The algorithm for constructing an automaton from an event sequence.

Handle Escape Conditions and Holding Patterns:

- For each escape condition E of the form "reject if see X in C", create a new abstract state N_E for E, label N_E with X, add an edge from each state corresponding to C to N_E and add a self-loop at N_E.
- For each escape condition E of the form "accept if see X in C", create a new abstract state N_E for E, label N_E with X, add an edge from each state corresponding to C to N_E, add a self-loop at N_E, and mark N_E as fair (with a new fairness constraint).
- For each escape condition E of the form "accept if don't complete C", mark every abstract for C as fair (with a new fairness constraint).
- For each holding pattern h for cluster C and each abstract state N_C corresponding to or expanded from C, add h as a propositional label to N_C.

Figure 7: The algorithm for handling holding patterns and escape conditions.

appearance of the constraint's events in the reduced form of the original sequence; this is necessary because a single automaton would need to include an edge from the final state shown in the example (after the detection of e_2) back to the initial state, and the cycle that this back edge creates could break weakness. New propositional labels annotating occurrences of the events in the automaton from Algorithm 1 distinguish between machines. Having multiple machines will not lead to substantial state explosion, as the transitions from their initial states are enabled only when the events to be monitored occur.

The results on determinism and weakness that follow apply to those event sequences that end with a repetition-free cluster (for reasons motivated in Section 4.2). We call such sequences *event chains*.

Definition 18 An event sequence $\langle C, T, H, S\rangle$ is an *event chain* if the iterative expansion of $\mathrm{last}(C)$ contains no repeated clusters.

5.1 Soundness

The soundness theorem relates runs through the constructed automaton to the locally minimal index assignments by which a word satisfies an event sequence. In developing this relationship, we will find it useful to refer to the specific instances of event sequences embodied in index assignments and automaton runs. We begin by formalizing this concept.

Definition 19 Let V be an event sequence. An *instance of* V is an event sequence V' with the same hierarchical structure, timing constraints, holding patterns, and escape conditions as V, but in which every repetition operator $*$ is replaced with a natural number and each unordered cluster C is replaced with an ordered cluster C' that corresponds to a schedule of C.

Example: Event sequence $\langle C_1; \{C_{21}, C_{22}^*\}; \{C_{31}, C_{32}\}^*; C_4\rangle$, has sequence $\langle C_1; \langle C_{22}^2, C_{21}\rangle; C_4\rangle$ as an instance.

The constructions of both index assignments and the automata that capture event sequences use schedules of unordered clusters and concrete numbers of repetitions (the same specializations that yield instances of event sequences). We can therefore talk meaningfully about the instances that correspond to individual index assignments and paths through the automaton. We rely on both notions in the rest of this section. We can also prove that the automaton for a sequence contains paths corresponding to each of its instances.

Lemma 2 *Let V be an event sequence and M the automaton constructed to accept V. For every instance V' of V, M contains a path that expects events in the order specified in V'.*

Handle Timing Constraints: For each timing constraint $\langle e_1, e_2, l, u, clk \rangle$, construct an automaton template as follows:

1. (create core states) Create $n+1$ states, $s_0, \ldots, s_n$, where n is u if u is a number or l if $u = \infty$. Intuitively, these states track how many times *clk* has occured since e_1 occurred.
2. (create clock-counting transitions) For each state s_i for $0 \leq i \leq n-1$, add a self-loop labeled !see-$e_2 \wedge !clk$, and a transition labeled !see-$e_2 \wedge clk$ from s_i to s_{i+1}.
3. (create an initial state) Create a state s_{init} with a self-loop labeled !see-e_1, a transition to s_0 labeled see-$e_1 \wedge !clk$, and a transition to s_1 labeled see-$e_1 \wedge clk$. Mark s_{init} as the initial state of the automaton.
4. (create a violation state) Create a state s_{err} with a self-loop labeled `true`.
5. (account for lower bound) For each s_i, where $0 \leq i < l$, add a transition see-e_2 from s_i to s_{err}; for s_{l-1}, augment the label on this transition to also require $!clk$.
6. (create a final state) Create a state s_{done} with a self-loop labeled `true`.
7. (account for upper bound) Add a self-loop at s_n; if u is a number, label this transition !see-$e_2 \wedge !clk$, otherwise label it !see-e_2. Add a transition from s_n to s_{done} labeled with see-$e_2 \wedge !clk$. If u is a number, add a transition from s_n to s_{err} labeled *clk*.
8. (account for intermediate time values) For each state s_i where $l-1 \leq i < n$, add a transition from s_i to s_{done} labeled see-e_2; for s_{l-1}, augment this transition to also require *clk*.

For each expansion of the smallest cluster containing e_1 and e_2 in the core machine, create new labels for see-e_1 and see-e_2, add them to the states that satisfy e_1 and e_2 within that expansion, and create an instance of the template automaton that uses these labels for see-e_1 and see-e_2.

Figure 8: Algorithm for constructing automata for timing contraints. Figure 10 shows examples of the output from this algorithm.

Proof: This result is evident in the construction algorithm because each possible schedule of an unordered sequence is expanded into a path that follows that schedule; furthermore each repeated cluster yields a cycle in the automaton that can repeat any number of times.

Theorem 1 *Let $\langle e_1, e_2, l, u, clk \rangle$ be a timing constraint and M be the automaton template for that constraint according to the algorithm in Figure 8. Let W be a word such that every index that satisfies e_1 has the proposition see-e_1 and every index that satisfies e_2 has the proposition see-e_2. W satisfies the timing constraint iff M reaches its done state when run on W.*

Proof: Let i_1 be the first index into W that satisfies e_1 (labeled see-e_1) and let e_2 be the first index larger than e_1 that satisfies e_2 (labeled see-e_2). The definition of constraint validity requires that the number of indices between e_1 and e_2 (inclusive) that satisfy *clk* lie between l and u (inclusive). The transitions in M guarantee that if M is in state s_i (as named in the algorithm), then *clk* has occurred i times since see-e_1 occurred (and see-e_2 has not yet occurred); the examples in Figure 10 illustrate this claim.

Recall that the algorithm defined n to be either the lower bound l if the upper bound is ∞ or the upper bound u if u is a number; the number of appearances of *clk* between see-e_1 and see-e_2 (inclusive) must be at least n if $n = l$ and at most n if $n = u$. If $n = u$, then any occurrence of *clk* in state s_n sends M to the error state (step 7); this is consistent with violating constraint validity. The only transitions into the done state come from states s_i where i is at least $l-1$ (step 8). The guards on each of these transitions guarantee that the *clk* count remains between l and u, inclusively. If $n \neq u$, then $n = l$ and $u = \infty$. In this case, M contains only two transitions to the done state (steps 7 and 8): one from s_{n-1} which requires see-e_2 and *clk*

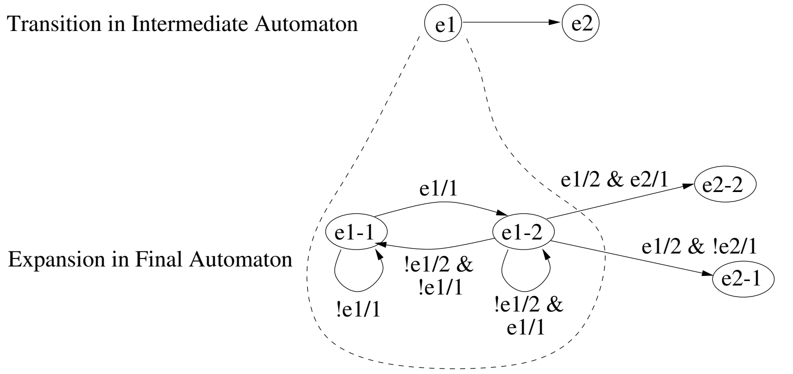


Figure 9: Example of expanding a transition between intermediate states representing events. This example shows only how to handle the events themselves; it does not illustrate handling holding patterns or escape conditions. Separate automata account for timing constraints.

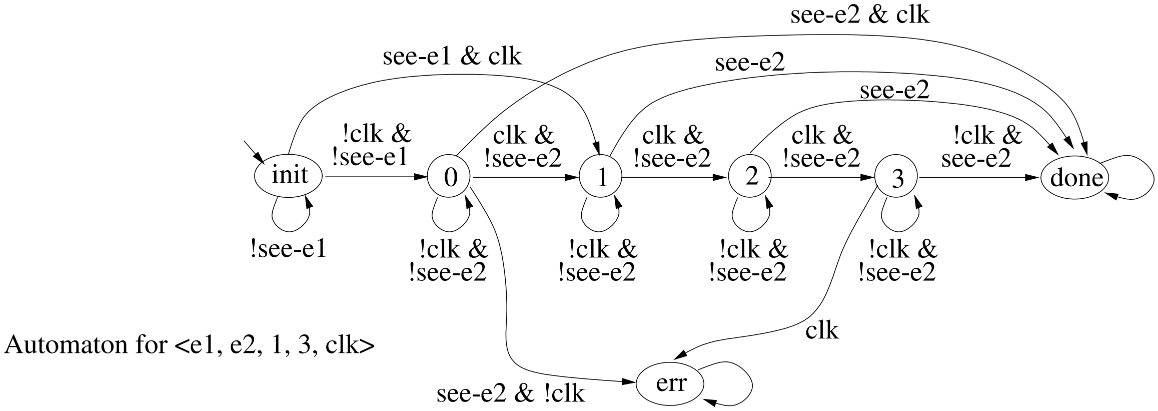


Figure 10: Realizing a timing constraint as an automaton.

to occur simultaneously, and one from s_n which requires only see-e_2. In either case, this restricts the number of clk occurences to lie between l and u inclusively. M therefore reaches the done state iff W satisfies the definition of constraint validity for this timing constraint.

Theorem 2 *Let V be an event sequence and let M be the automaton obtained for V from Algorithm 1. Let W be an infinite word. M accepts W iff $W \models V$.*

Proof: Assume that M accepts W. Then there exists a fair path P through M that corresponds to the run of M on W. We will construct an index assignment I from P and argue that I is locally minimal, valid and either total or uses (invokes or loops under) an escape condition. For each state s in P at which M recognized event e, add window $[i, j]$ to $I(e)$, where i is the (0-based) index of s in P and j is one index before that containing a next event of e. For each maximal subpath $s_1, \ldots, s_k$ of P whose states all correspond (were expanded from) a cluster C, assign window $[i, j]$ to $I(C)$, where i is the position of s_1 and in P and j is one index prior to that assigned to the next event after state s_k.

I is structurally valid by construction: every path through M corresponds to an instance of V by construction, so P must consist of subpaths (which yield sequences of windows) that capture repeated and unordered clusters; furthermore, constructing I around maximal subpaths guarantees that all subclusters of each cluster have been assigned windows. I is constraint valid because P is fair: if a timing constraint were violated, the automaton for that constraint would have reached its error state (by Theorem 1) thus rendering

P unfair (by the definition of the fair states in Algorithm 1). I is locally minimal because by construction M advances through states at the first occurrences of events (non-determinism may govern which schedule of an unordered cluster is followed, but does not delay matching events). Since M accepts W, M must visit some fair state infinitely often. If that fair state uses the state *Final*, then I must be total because *Final* is not reachable until all clusters in V are recognized. If that fair state arises from an "accept if see" escape condition, then some index i into P must have invoked that escape condition and all prior indicies into P satisfied V; in this case, I would invoke an escape condition at i. If the fair state arises from a looping escape condition, then by construction P does not reach the state for some event, but satisfied all requirements for prior events; I therefore loops on an escape condition. I is therefore sufficient to establish that $W \models V$.

For the other direction, assume that $W \models V$. To prove that M accepts W, we will argue that the index assignment I that witnesses $W \models V$ generates a fair path through M. Let V' be the instance of V that corresponds to I. By Lemma 2, M contains a path P that also corresponds to I. To complete the proof, we must argue that P is fair. We first argue that P satisfies the fairness requirements on the timing automata. By definition, I satisfies all timing constraints of V; By Theorem 1, all timing constraints on pairs of events within I are in their done states. Timing constraints on pairs of events not captured within I are still in their initial states. If I invokes or loops under an escape condition after seeing the first event in a timing constraint but not the second, the corresponding automata will be in an intermediate (non error) state. In each of these cases, the timing constraint automata are (and remain) in states that satisfy the definitions of fair states from Algorithm 1.

Since $W \models V$, I is either total or uses an escape condition. If I is total, then the path satifies all events in V; by construction, M transitions to the fair state *Final* after recognizing all events in V, so the theorem holds in this case. If I invoked an escape condition at index i while processing cluster C, the theorem also holds because by construction, M must have a transition from each state it could be in while processing C to a fair state for the escape condition. If I loops under an escape condition, then that cycle is also marked fair, which is sufficient to render the entire run fair. The theorem therefore holds in this direction.

5.2 Characterization of Determinism

Theorem 3 *Given an event chain, Algorithm 1 produces a deterministic automaton if all of the following conditions are satisfied:*

- *For every unordered cluster $\{C_1, \ldots, C_k\}$, the first events of each C_i are pairwise logically inconsistent with those of each $C_j \neq C_i$ unless a timing constraint orders C_i and C_j.*
- *For each repeated cluster C^*, the first events of C are pairwise logically inconsistent with the first events of each next cluster of C^* (other than C).*
- *For each "accept/reject when see X in C" escape condition, X is logically inconsistent with all holding patterns for C.*

Proof: The machine is deterministic if the choice among multiple next states is deterministic. The construction yields multiple next states in four cases: possible transitions to the *Final* state, when choosing between schedules for an unordered cluster, possible skips of repeated clusters, and when invoking escape conditions. The restriction to event chains guarantees that states with transitions to *Final* have no other outgoing transitions. By construction, transitions into the states that expand clusters occur when a first event is recognized for that cluster. If these events are logically inconsistent, then the corresponding transitions must be deterministic. This covers the remaining cases.

To approach this result from another perspective, we also prove that these conditions are sufficient to yield a unique minimal index assignment.

Theorem 4 *Let V be an event sequence, W be a word, and Σ be the set of index assignments for V and W that are valid and either total or use an escape condition. Σ has a unique minimal index assignment I under $\prec$.*

Proof: We prove that for every non-equivalent I_1 and I_2 in Σ that are unordered under $\prec$, there exists another I in Σ such that $I \prec I_1$ and $I \prec I_2$. This is sufficient to guarantee the existence of a unique minimal element of Σ.

The proof is by construction: we define $\overline{I}$ from $\overline{I_1}$ and $\overline{I_2}$, then argue that I must lie in Σ. The proof of Lemma 1 shows that I_1 and I_2 are potentially unordered only if $RgW(I_1) = RgW(I_2)$ (this condition captures cases in which I_1 and I_2 assign different events to the same set of windows). Let w be the earliest window such that $\overline{I_1}(w) \neq \overline{I_2}(w)$. Define $\overline{I}(w)$ to be $\overline{I_1}(w) \cup \overline{I_2}(w)$; for all other windows w' in $RgW(I_1)$, define $\overline{I}(w')$ to be $\overline{I_1}(w')$.

$I \prec I_1$ because $RgW(I) = RgW(I_1)$ and $\overline{I_1}(w) \subset \overline{I}(w)$ by construction (since $\overline{I_2}(w) \neq \overline{I_1}(w)$, the $\subset$ relation must hold). $I \prec I_2$ for similar reasons (since w is also the smallest window on which $\overline{I}$ and $\overline{I_2}$ disagree).

It remains only to show that I must lie in Σ. Since I_1 and I_2 are valid, I is also valid because every window it assigns to an event is assigned by one of I_1 or I_2, while respecting all timing constraints. I therefore lies in Σ by definition.

5.3 Characterization of Weakness

We call a cluster C *fair* if there exists an escape condition of the form "accept if don't complete C". A cluster is *all-fair* if it is either fair or all of its sub-clusters are all-fair. A cluster is *non-fair* if neither it nor any of its sub-clusters is fair.

Lemma 3 *If an event sequence contains no all-fair repeated clusters, then the automaton from Algorithm 1 requires only one fair set.*

Proof: If no cycle contains states from more than one fair set, then a single fair set suffices. Cycles can contain states from multiple fair sets under two conditions. First, two "accept don't complete" conditions could exist for clusters C_1 and C_2 where C_1 contains C_2. In this case, a cycle that satisfies C_2 satisfies C_1, so only one fairness constraint is required. Second, a repeated cluster could have all sub-clusters fair, thus creating a cycle that visits each sub-cluster then self-loops for the repeated cluster. The theorem statement rules out this case.

Theorem 5 *Given an event chain, Algorithm 1 produces a weak automaton iff every repeated cluster in the chain is non-fair.*

Proof: Non-trivial strongly-connected components (SCCs) arise from abstract states with self-loops, which in turn arise from expanding states for repeated clusters. With the exception of the *Final* state and the states for "accept/reject if see" escape conditions (which form their own SCCs), states are marked fair only if they correspond to or expand from clusters that have "accept if don't complete" conditions. If a repeated cluster is non-fair, then it has no fair SCCs embedded within self-loops (other, larger SCCs). If a repeated cluster is all-fair, it requires multiple fair sets and is not weak by definition. All other repeated clusters contain cycles with both fair and non-fair states.

Our mapping to deterministic weak automata is not complete; in other words, our language does not logically characterize deterministic weak automata. Consider the regular expression $ab^* + bc^*$: a deterministic weak automaton accepts it, but it is not expressible in our language due to the use of disjunction.

6 Related Work

We are unaware of logical characterizations of weak automata, much less ones that account for diagrammability or other forms of usability. The original work on the efficiency of verifying weak automata is due to Bloem, Ravi and Somenzi [5]. Other timing diagram formalizations have supported some of the language extensions discussed here [2, 6, 12], but none related the diagrammatic features of these languages to efficiency in verification.

Amla *et al.'s* work on modular timing diagrams has much in common with this work [3]. Their work makes timing diagrams more expressive by combining them through non-diagrammatic operators for conjunction, iteration, and deterministic choice. Expressions in their language encompass several timing diagrams, while our work pushes the limits of a single timing diagram. At a technical level, our work uses standard Büchi automata as a semantic model, while their work uses ∀-automata [11]. The ∀-automata provide a more natural and computationally efficient formalism for non-deterministic modular timing diagrams than Büchi automata. Our work targets the class of event sequences recognizable by deterministic weak automata because we are interested in understanding when a particular algorithmic optimization applies to verification of event sequences; ∀-automata don't offer benefits in our context due to our restriction to deterministic systems. Overall, the core differences between our works appear to be philosophical; ours focuses on understanding the interplay between diagrammability and efficiency, while theirs focuses on building a practical verification framework around timing diagrams. Our two works are complementary, and both provide useful insights for developing verification frameworks around timing diagrams. Understanding the interplay between these two sets of results is an interesting open question for future work.

7 Conclusions and Future Work

The relationships between timing diagrams and deterministic weak automata suggest that there exist formal models of event sequences that simultaneously address both usability and efficiency. A traditional theoretical approach to designing languages towards efficiency would be to find a syntactic (logical) characterization of weak automata. This approach, however, fails to account for the usability of that logical characterization. This is perhaps justifiable, as "usability" is an inherently informal notion. If we refine our notion of usability to mean diagrammability, however, formal models become possible. Formal characterizations of diagrammability usually rely on topological or spatial arguments [10]; appropriate characterizations for discrete linear events remain an open problem.

The event sequence language proposed in this paper targets diagrammability by allowing only a restricted form of disjunction; in particular, disjunction governs the *ordering* of events, but not their *occurrence*. This is consistent with diagrams' tendency to imply that all depicted items actually exist (maps, for example, indicate that all depicted features are actually there). Such nuances in the different uses of logical operations appear fundamental to formal models of diagrammability. This limited nature of disjunction also targets efficiency by supporting our criteria for deterministic automata. Restricted forms of iteration enable the mapping to weak automata. Single timing diagrams support limited forms of iteration, and hence satisfy the criteria for weakness. Overall, the generality of our language substantially enriches the set of features our timing diagrams can support while retaining efficiency for verification.

Several avenues remain open for future work. Given that the proposed language is more expressive than our current timing diagrams, characterizing diagrammability is an important next problem in this project. We expect restrictions on cluster nesting similar to those in timing diagrams to be key to such a characterization. We also plan to explore formal relationships between other event sequence languages and ours; this would help identify subsets of other languages that could be visualized and verified efficiently through a mapping to weak automata. Finally, many general questions remain regarding the nature of diagrammatic representations and their relationship to computational concerns such as efficiency and decidability that are so important in verification. We hope that our work will contribute to better understanding of these issues.

References

[1] Accellera Working Group. Property specification language reference manual (version 1.0). Available at `http://www.eda.org/vfv/docs/psl_lrm-1.0.pdf`, 2003.

[2] N. Amla, E. A. Emerson, and K. S. Namjoshi. Efficient decompositional model checking for regular timing diagrams. In *IFIP Conference on Correct Hardware Design and Verification Methods*, 1999.

[3] N. Amla, E. A. Emerson, K. S. Namjoshi, and R. J. Trefler. Visual specifications for modular reasoning about asynchronous systems. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 226–242, 2002.

[4] R. Armoni et al. The ForSpec temporal logic: A new temporal property-specification language. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 296–211, 2002.

[5] R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In *International Conference on Computer-Aided Verification*, number 1633 in Lecture Notes in Computer Science, pages 222–235. Springer-Verlag, 1999.

[6] K. Feyerabend and B. Josko. A visual formalism for real-time requirement specifications. In M. Bertran and T. Rus, editors, *Transformation-Based Reactive Systems Development, Proc. 4th International AMAST Workshop on Real-Time Systems and Concurrentand Distributed Software, ARTS'97*, volume 1231, pages 156–168. Springer-Verlag, 1997.

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

[8] K. Fisler. On tableau constructions for timing diagrams. In *NASA Langley Formal Methods Workshop*, 2000.

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

[10] O. Lemon. Comparing the efficacy of visual languages. In D. Barker-Plummer, D. I. Beaver, J. van Benthem, and P. S. di Luzio, editors, *Words, Proofs, and Diagrams*, pages 47–70. CSLI Publications, 2002.

[11] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by ∀-automata. In *ACM Symposium on Principles of Programming Languages*, pages 1–12, 1987.

[12] Y. Ramakrishna, L. Dillon, L. Moser, P. Melliar-Smith, and G. Kutty. A real-time interval logic and its decision procedure. In *Proc. Thirteenth Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 173–192. Springer-Verlag, December 1993.

[13] Synopsys, Inc. Openvera assertions. Available online for download at `http://www.open-vera.com/technical/technical.html`, 2002.