Towards Diagrammability and Efficiency in Event Sequence Languages

Kathi Fisler

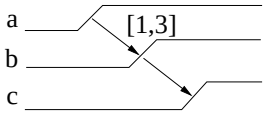
Department of Computer Science
WPI (Worcester, MA, USA)
kfisler@cs.wpi.edu

Abstract. Industrial verification teams are actively 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 account 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 [11], and Intel's FTL [3]. 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 [7] provide effectively linear symbolic verification algorithms [4]. That timing diagrams are not more widely used as event sequence languages suggests that they lack the expressiveness needed in industrial verification. 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?



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))))

Fig. 1. Expressing an event sequence in three languages.

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 an framework in which to assess both usability and efficiency of other event sequence languages; second, our characterization proves that timing diagrams can be extended with several new features—such as partial orders between events, multiple 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 3, 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 there exists a partition of the states Q into disjoint sets $Q_1, \ldots, Q_n$ such that (1) each Q_i is either contained in $\mathcal{F}$ or is disjoint from it, and (2) the Q_i's are partially ordered so that there is no transition from Q_i to Q_j unless $Q_i \leq Q_j$; in other words, an automaton is weak if each of its strongly connected components has either all states fair or all states non-fair [8]. Weak automata are attractive in the context of verification because symbolic cycle detection is effectively linear for weak automata, whereas existing algorithms for full LTL are quadratic [4].

Deterministic weak automata are also interesting in verification for their properties under complementation. Automata-based verification approaches must complement automata that capture properties as part of the verification process. In the general case, complementing a Büchi automaton can blowup its number of states exponentially. Complementing a deterministic weak automaton with a single fair set $\mathcal{F}$, however, requires only complementing $\mathcal{F}$; 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.

3 An Intermediate Language for Event Sequences

Event sequences define patterns of transitions and values on variables, as well as constraints on and exceptions to those patterns. This section presents a regular-expression-like syntax and semantics for event sequences. The semantics captures one pass over an event sequence, rather than the multiple passes needed to treat an event sequence as an invariant. The one-pass semantics offers two benefits: it provides a foundation for defining different multiple pass semantics [6], and it enables the mapping to weak automata. This limitation is not as serious as it might seem; in prior work [7], we showed that relabeling fair sets and adding a few select transitions constructs the automaton for a negated invariant event sequence (the machine most commonly needed for verification) from a one-pass (deterministic weak) automaton for that sequence.

3.1 Syntax

Like regular expressions, event chains capture sequences and repetitions of values on signals. Event chains, however, restrict disjunction to referring to ordering of

events, rather than their occurrence. Nesting event groups within event patterns restricts underspecified orders to occur within delimited intervals.[1]

Definition 1 Event chains 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; propositional variables annotated with arrows ($p\downarrow$, $p\uparrow$) denote falling and rising transitions, respectively.
- An *event group* is a nonempty set of events or a set of event groups.
- An *event pattern*, denoted P^M consists of an event group or a sequence of event patterns (P) and a positive number, $*$, or $+$ (M). Repetition markers $*$ and $+$ are called *unbounded*. We denote ordered sequences of event patterns using semicolons, as in $P_1^{M_1}; P_2^{M_2}; \ldots; P_k^{M_k}$. We omit M when $M = 1$.
- An *event chain* is a sequence of event patterns in which the last pattern does not have an unbounded repetition marker.

The set of all events, event groups, event patterns, and event pattern sequences in an event chain are called the *elements* of that chain.

An event sequence augments an event chain with three kinds of modifiers: *holding patterns* constrain variable values during event groups, *temporal constraints* restrict the amount of time (as a number of clock ticks) that can elapse between events, and *escape conditions* indicate special circumstances under which the chain should be considered rejected or satisfied. Temporal constraints may be relative to a designer-specified *event clock*, as captured by a Boolean expression (this is a common feature in many events sequence languages). The following definition formalizes each of these components.

Definition 2 An *event sequence* is a tuple $\langle E, H, T, S\rangle$ where E is an event chain, H (the *holding patterns*) is a partial function from event groups in E to propositional formulas, T is a set of temporal constraints and S is a set of escape conditions.

- A *temporal constraint* is a tuple $\langle e_1, e_2, l, u, c\rangle$ where e_1 and e_2 are (uniquely identified) events in E, l is a positive integer, u is either a positive integer at least as large as l or the symbol ∞, and c is a boolean expression (the clock for the constraint; `true` indicates the system clock). Events e_1 and e_2 may lie in different event groups, but if they do, then their nearest containing patterns may only have repetition markers of value 1.
- An *escape condition* has one of the following types, where X is a propositional expression over variable names and transitions on variables (variables need not be in the chain) and G is an event group:
  - "accept if don't complete G"
  - "reject if see X in G"
  - "accept if see X in G"

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

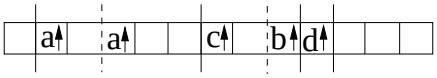


Fig. 2. A sample event sequence and an example of its semantics.

As an example, Figure 2 defines an event sequence comprised 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. In addition, the rising transition on d must occur between 2 and 5 ticks (inclusive) after the rising transition on c (the timing constraint), signal a must remain true until the rising transition on d occurs (the holding pattern constraint), and the rest of the sequence is only checked if a rising transition on a occurs (the escape condition).

3.2 Semantics

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

As an example of the semantics, consider the event sequence and the word shown in Figure 2. The figure shows how the word is divided into windows per event pattern (demarcated by solid lines), and subwindows as necessary for nested elements (demarcated by dashed lines). The definitions in this section formalize the mappings from event elements to windows.

Definition 3 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 $i_4 \leq i_2$.
- Given a window $w = [start, end]$, a sequence $[s_1, e_1], \ldots, [s_k, e_k]$ forms a *consecutive covering sequence of windows for* w if $s_1 = start$, $e_k = end$, and for all $1 \leq j < k$, $e_j = s_{j+1} - 1$.

Definition 4 Given an event sequence V and a word W, an *index assignment for* V *and* W is a function from the elements in V to non-empty sets of windows of W.

[1] While this property may not appear important in this paper, it helps maintain a mapping to reversal bounded counter machines when timing constraints are relaxed to also include variables, which impacts decidability of verification [6].

A window must meet certain requirements in order to capture the constraints of an event element. The following two definitions formalize those requirements for each type of element in an event chain.

Definition 5 Let $E = v_1 \wedge \ldots \wedge v_k$ be an expression where each v_i is a proposition, its negation, or a rising or falling transition on a propositional variable. Let W be a word and let i be 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 $v_i = p$, $W_i(p) = 0$ and $W_{i+1}(p) = 1$ if $v_i = p\uparrow$, and $W_i(p) = 1$ and $W_{i+1}(p) = 0$ if $v_i = p\downarrow$.

Definition 6 Let $V = \langle E, T, H, S\rangle$ be an event sequence, let W be a word, and let I be an index assignment for V and W. I is *valid* for V and W iff all of the following conditions hold:

1. I is minimal, in that (a) removing any window from $I(L)$ for any element L in E would render I invalid and (b) there exists no valid index assignment I' and element $L \in E$ such that for some window $[w_1, w_2] \in I(L)$, $I'(L) = [w_1, w_2']$ for some $w_2' < w_2$ and I' is equivalent to I on all windows that occur before w_1 (this requires I to assign the earliest possible matching windows to each element; it enables the recognizing automaton to be deterministic).[2]
2. I respects hierarchy, in that for all event elements L in E, if L is a subelement of (nested within) event element L', then for every window w in $I(L)$, there exists a window w' in $I(L')$ such that w' contains w.
3. I satisfies the structural requirements of V; namely:
   - (for event chains) $I(E) = \{[start, end]\}$ for some indices *start* and *end* into W such that no smaller choice of *start* or *end* yields a valid index assignment.
   - (for pattern sequences) For every event pattern sequence $P = P_1^{M_1}; \ldots; P_k^{M_k}$ and every w in $I(P)$ there exists sequence $w_1, \ldots, w_k$ of covering consecutive windows of w such that for all $1 \leq j \leq k, w_j \in I(P_j)$.
   - (for event patterns) For every event pattern G^M and wp in $I(G^M)$ there exists a sequence $wp_1, \ldots, wp_m$ of covering consecutive windows for wp such that each $wp_i \in I(G)$ and m is equal to M if M is a number or some natural (resp. positive) number if $M = *$ (resp $+$).
   - (for nested groups) For every event group G containing nested event groups $G_1, \ldots, G_k$ and every window $w \in I(G)$ there exists a sequence $w_1, \ldots, w_k$ of covering consecutive windows for w and a permutation $GP_1, \ldots, GP_k$ of $G_1, \ldots, G_k$ such that for all $i \leq j \leq k, w_j \in I(GP_j)$.
   - (for event groups) For every event group G consisting of a set $\{E_1, \ldots, E_k\}$ of events and every window $w = [start, end] \in I(G)$ there exists a permutation $EP_1, \ldots, EP_k$ of $E_1, \ldots, E_k$ such that $[start, start] \in I(EP_1)$ and for all $1 < j \leq k$, there exists index $i_j \in [start, end]$ such that $i_j \in I(EP_j)$, and $I(EP_j)$ is the smallest index in $[start, end]$ and larger than $I(EP_{j-1})$ that satisfies EP_j.

[2] Window minimality resembles Sugar's notion of expressions "holding tightly".

- (for events) For every event Ev, and every index i such that $[i, i] \in I(Ev)$, i satisfies Ev (Defn 5).

4. I satisfies the holding pattern constraints, in that for every holding pattern h corresponding to an event group G and every window $[w_1, w_2] \in I(G)$, every index $w_1 \leq i \leq w_2$ satisfies h.
5. I satisfies the timing constraints, in that for every time constraint $\langle e_1, e_2, l, u, c \rangle$ 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 element containing both e_1 and e_2, the number of indices satisfying c between t_1 and t_2 (inclusive) is within the range $[l, u]$.

The previous definition formalizes how an index assignment captures an event sequence without considering escape conditions. The next two definitions handle escape conditions. The minimality restrictions support a semantics of event sequence concatenation, which we omit here for sake of space. Definition 9 relates words and event sequences based on the existence of index assignments that may or may not invoke escape conditions.

Definition 7 Let V be an event sequence, W be a word, and I be an index assignment for V and W. Let C be an escape condition in V of the form "accept/reject if see X in G". Index i into W *invokes* C *under* I if $i \in I(G)$, i satisfies X, I is valid for all windows that occur before that for $I(G)$ that contains i, and no index into W smaller than i invokes an escape condition under I. If there exists an index i in I that invokes some escape condition C, we say that I *invokes an escape condition of* V.

Definition 8 Let V be an event sequence, W be a word, and I be an index assignment for V and W. I *loops under escape condition* $C \in V$ if C is of the form "accept if don't complete G", I is valid for all windows that occur before that for $I(G)$ that contains i, and I is not valid for the prefix of V up to and including G.

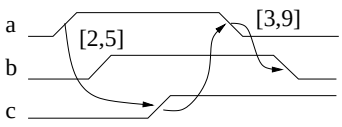
Definition 9 Let V be an event sequence and let W be a word. $W \models V$ if there exists an index assignment I for V and W such that I is valid for V, I loops under some escape condition in V, or I invokes some escape condition in V.

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

Timing diagrams map naturally into the intermediate language. Figure 3 shows a timing diagram and its representation as an event sequence. Mapping the depicted transitions into event groups is the only subtle task. Visually, the event



$$E = \{a\uparrow, b\uparrow, c\uparrow, a\downarrow\}; \{b\downarrow\}$$
$$T = \{\langle a\uparrow, c\uparrow, 2, 5, \mathsf{true}\rangle,$$
$$\langle c\uparrow, a\downarrow, 1, \infty, \mathsf{true}\rangle,$$
$$\langle a\downarrow, b\downarrow, 3, 9, \mathsf{true}\rangle\}$$

Fig. 3. A timing diagram with partial orders and its mapping into an event sequence.

chain $\{a\uparrow\}; \{b\uparrow\}; \{c\uparrow\}; \{a\downarrow\}; \{b\downarrow\}$ appears appropriate, but is incorrect because the rising transitions on a and b can occur in any order since no constraint orders them. The correct event chain respects the partial order in the diagram.

The language presented here extends our previous results on the relationship between timing diagrams and weak automata [7] 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"). The results in Section 5 show that timing diagrams with partial event orders and multiple input assumptions on the environment also map to deterministic weak automata. We view environment assumptions as events that are only constrained if they occur [5]; unlike other events, their failure to occur does not violate the diagram's requirements. For the diagram in Figure 3, we could treat the two transitions on a as environment assumptions by rewriting the event chain using nested event groups (as $\{\{a\uparrow\}, \{b\uparrow\}, \{c\uparrow\}, \{a\downarrow\}\}; \{b\downarrow\}$) and adding "accept-if-don't-complete" escape conditions on the two groups corresponding to transitions on a.

These examples help illustrate the influence of timing diagrams in the design of the proposed event sequence language. The language as proposed is, however, more expressive than our current timing diagram formalization. Consider the event chain $a\uparrow^*; b\uparrow$. The current timing diagram semantics requires all depicted transitions to occur unless an escape condition matches, so this chain (without escape conditions) is not expressible as a timing diagram. Similar examples involving repetitions also exist. Enriching the timing diagram notation could resolve some of these issues; this remains an issue for future work.

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 combination so 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 a yields a weak automaton and corresponds to event chain $(\{p\}^*; \{q\})^*; \{r\}$. The second cannot be captured with a weak automaton and is not expressible in our language.

Fig. 4. Automata for two LTL formulas.

One key difference between these two formulas is that the second contains a repetition in its last pattern, 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 explains the restriction in the definition of event chains that the final pattern not have 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. Augmenting the chain $(p \mathbin{\mathsf{U}} q) \mathbin{\mathsf{U}} r$ with an 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 described in the introduction to 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 patterns. 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. The construction proceeds in phases, building a series of abstract machines, each of which refines the one from the previous phase. Figure 5 illustrates the intuition behind the phases: phase 1 captures event patterns, phase 2 handles nested event groups, phase 3 expands

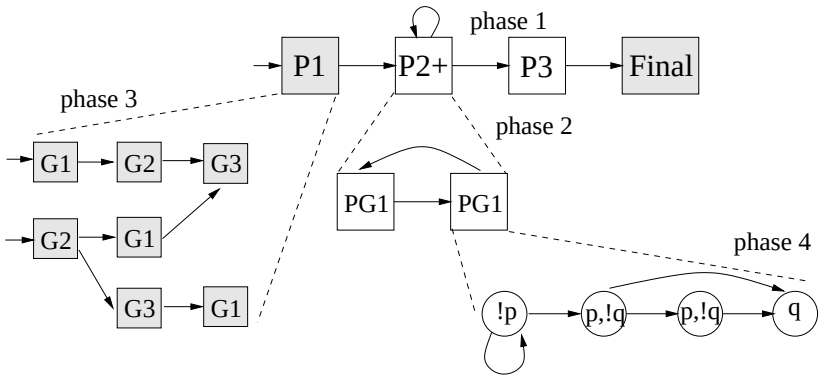


Fig. 5. Phases in the state machine construction algorithm.

non-nested event groups into their permutations and phase 4 expands sets of events into states that reflect timing constraints. Edges in the abstract machines reflect edges in the actual machine: if there is no edge from one abstract state to another, then there will be no edge from any state in the expansion of the first to the expansion of the second. For sake of space, we present the detailed algorithm only up through phase 3; this is sufficient for the results on weakness.

Phases may add labels and annotations (to denote initial or fair) to states. Expanding an abstract state copies all such markings to each expanded state. The construction uses the following definition regarding adjacency of patterns.

Definition 10 Let P_i be an event pattern in an event pattern sequence $P = P_1^{M_1}; \ldots; P_k^{M_k}$. The *next patterns* of P_i are defined as follows. If $i < k$, then the next patterns of P_i is the set of all P_j such that $i \leq j \leq k$ and for all h such that $i < h < j$, P_h has the repetition marker $*$. If $i = k$ or if P_k has repetition marker $*$, the next patterns of P_i includes the next patterns for the pattern sequence P' that directly contains P.[3] If P_i has an unbounded repetition marker, then P_i is a next pattern of itself. The *previous patterns* are defined similarly.

Example: In $P_1; P_2; P_3^*; P_4^+$, the next patterns of P_2 are P_3 and P_4; the next patterns of P_3 are P_3 are P_4. Given sequence $P_1; \langle P_{21}; P_{22}^* \rangle; \langle P_{31}; P_{32} \rangle^*; P4$, the next patterns of P_{21} are P_{22}, P_{31} and P_4.

Preprocessing phase: repetition expansion. Expand all event patterns that repeat a concrete number of times $n > 1$ with n copies of the pattern. In the resulting event sequence, all repetition markers are either the number 1 or unbounded. The remaining phases assume that all event patterns have this format.

[3] Such a containing pattern much exist for all sequences other than the overall event chain, which by definition cannot end with an unbounded pattern.

Phase 1: Capture Event Patterns. Let $P_1^{M_1}; \ldots; P_k^{M_k}$ be the event patterns that form the event chain in V. Create an abstract state for each P_i and an edge from each P_i to P_{i+1} (for $1 \leq i < k$). Create a final state and mark it as a fair state; include an edge from P_k to the final state and an edge from the final state to itself. Mark the state for P_1 as an initial state. For each pattern $P_i = G_i^{M_i}$

- If M_i is unbounded, add an edge from P_i to itself. If $M_i = *$, add an edge from each previous pattern of P_i to each next pattern of P_i.
- If G_i is a sequence of event patterns $PG_1; \ldots; PG_n$, repeat this phase up to this point to create a chain of states for $PG_1, \ldots, PG_n$; replace the state for P_i with this chain. Any incoming edges to the state for P_i other than self-loops should point to the state for PG_1. The outgoing edge from P_i to P_{i+1} becomes an edge from PG_n to P_{i+1}. If P_i had a self-loop, insert an edge from PG_n to PG_1.

Phase 2: Expand Nested Event Group States. This phase expands states for nested event groups into states for simple event groups. The construction relies on the following definition:

Definition 11 A permutation $G_1 \ldots G_k$ of event groups *violates a timing constraint* $\langle e_1, e_2, l, u, c \rangle$ if there exists G_i and G_j such that $1 \leq i < j \leq k$, $e_2 \in G_i$ and $e_1 \in G_j$.

- For every abstract state S_G corresponding to an event group G that contains a set of event groups $G_1, \ldots, G_k$, and every permutation $GP_1, \ldots, GP_k$ of $G_1, \ldots, G_k$ that does not violate a timing constraint, create a chain of abstract states $GPS_1, \ldots, GPS_k$. For every non-self-loop edge coming into S_G, add an edge from the same source to GP_1. For every non-self-loop edge leaving S_G, add an edge from GPS_k to the target of the original edge.[4]
- 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 S_G had an edge to itself, add an edge from each final state in the subgraph that expands S_G to each initial state in the subgraph that expands S_G.
- Remove S_G from the abstract state graph.
- For each holding pattern h for event group G and each abstract state S_G corresponding to or expanded from G, add h as a propositional label to S_G.

Phase 3: Handle Escape Conditions.

- For each escape condition C of the form "reject if see X in G", create a new abstract state S_C for C, label S_C with X, add an edge from each abstract state corresponding to G to S_C and add a self-loop at S_C.

[4] To reduce the machine size, we could perform a bisimilarity minimization on the subgraph of all states that expanded S_G.

– For each escape condition C of the form "accept if see X in G", create a new abstract state S_C for C, label S_C with X, add an edge from each abstract state corresponding to G to S_C, add a self-loop at S_C, and mark S_C as fair.
– For each escape condition C of the form "accept if don't complete G", mark every abstract state corresponding to G as fair.

This machine is not necessarily weak, due to potential escape conditions on nested repeated patterns. It is, however, the SCC quotient graph for the final machine, so the characterization theorem for when the algorithm yields a weak automaton is defined on the abstract machine.

Phase 4: Expand Event Sets. All abstract states now correspond to sets of events. This phase instantiates abstract states with their actual events and handles timing constraints. Since the events in an event set may occur in any order, we first generate all partial orders that are logically consistent within each unordered set over the events in the group; a partial order is logically consistent if no two unordered events are logically inconsistent.[5] Eliminate all such orders that violate a timing constraint. In a previous paper, we presented a technique for generating Büchi automata for timing diagrams with timing constraints and a total ordering on events [7]. The current problem reduces to that one by conjoining events that are in the same cell of a partial order into a single event. For sake of space, and since the expansion into events does not affect weakness by construction, we do not reproduce the details here. Intuitively, the algorithm uses the timing constraints to generate combinations of time spent in each set of events, then creates the number of states needed to count out these durations of time.

To handle the event clock c in a timing constraint over events e_1 and e_2, this phase adds a unique label for c to each state between e_1 and e_2, and creates an automaton that outputs this label whenever c is true. The final step cross-products the core machine with the clock machines; this does not affect weakness.

5.1 Soundness

Lemma 1. *One fair set is sufficient for event sequence automata.*

Proof Sketch: The automaton only needs to satisfy any one fairness condition to accept a word, so the argument depends on whether different fair sets could interact in the same cycle. By construction, the only possible overlap between fair sets is if two "accept don't complete" conditions exist for groups G_1 and G_2 where G_1 contains G_2. In this case, a cycle that satisfies G_2 satisfies G_1, so only one fairness constraint is required.

Theorem 1. *Let V be an event sequence and let M be the automaton constructed for V. Let w be an infinite word. M accepts w iff $w \models V$.*

[5] Unlike permutations, partial orders allow events to occur simultaneously.

Proof Sketch: Intuitively, the proof develops a correspondence between states in the abstract machines and the windows in the co-domain of an index assignment for W and V. The theorem follows from an argument that the windows occurring in accepted (resp. rejected) words correspond to accepting (resp. rejecting) paths through the automaton.

Although the mapping is sound, it 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.

5.2 Proof of Determinism

Definition 12 Let P^M be an event pattern. The *first events* of P^M is the set of events in P if P is an event group. If P is an event pattern sequence $P_1^{M_1}; \ldots; P_k^{M_k}$, then let P_h be the first pattern in the sequence with a bounded repetition marker (or P_k if no such pattern exists). The first events of P^M is the union of the first events for all $P_i^{M_i}$ where $1 \leq i \leq h$.

Example: Given pattern sequence $P_1; P_2; P_3$ where $P1 = a \uparrow$ and $P_2 = P_{21}^* P_{22}$, the first events of P_1 is the set $\{a \uparrow\}$ and the first events of P_2 is the union of the first events of P_{21} and P_{22}.

Theorem 2. *The construction algorithm produces a deterministic automaton if all of the following conditions are satisfied:*

- *Each pair of events contained within any one event group (including nested event groups) is pairwise logically inconsistent unless a timing constraint orders the two events.*
- *For each event pattern $P_i^{M_i}$ with an unbounded repetition marker in an event pattern sequence $P_1^{M_1}; \ldots; P_k^{M_k}$, the first events of P_i are pairwise logically inconsistent with the first events of each next pattern of P_i.*
- *For each "accept/reject when see X in G" escape condition, X is logically inconsistent with all holding patterns for G.*

Proof Sketch: The machine is deterministic if the choice among multiple next states is deterministic. In the event group graph, multiple next states arise in three cases: from event groups, unbounded repetition markers and escape conditions. By construction, transitions into the states that expand event groups or patterns occur when a first event is recognized for that pattern or group. If these events are logically inconsistent, then the corresponding transitions must be deterministic. Similar arguments cover the transitions to different events within an event set and to escape condition states.

5.3 Proof of Weakness

Theorem 3. *The construction algorithm produces a weak automaton if the sequence has no escape condition of the form "accept if don't complete G" where G is strictly contained in a pattern with an unbounded repetition marker.*

Proof Sketch: The proof follows from the relationship between the SCC quotient graph of the final automaton and the abstract machine after phase 3. The final automaton refines the structure of the event-group level automaton by construction; this means that all SCCs either lie within abstract states of the event-group machine, or fall within SCCs in the event-group machine. By construction, fair markers in the abstract machine propagate to the states that expand on the abstract states. An SCC in the final machine is therefore fair if and only if its corresponding SCC in the abstract machine is fair. The result therefore holds if we can show that the restriction in the theorem statement are sufficient to guarantee that the event-group machine is a weak automaton.

Three kinds of abstract states are marked fair: the final state (which by construction has no edges to other states and hence forms its own SCC), the "accept/reject if see" escape condition states (which also have no edges to other states), and the "accept if don't complete" escape conditions, which could lie in cycles with other states. The restriction in the theorem rules out the case which would lead to a cycle of both fair and non-fair states; the "strictly contained" requirement forces a non-fair state into the same cycle.

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 [4]. Other timing diagram formalizations have supported some of the language extensions discussed here [2, 5, 10], but none related the diagrammatic features of these languages to efficiency in verification.

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 [9]; 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 our reading of diagrams to imply that all depicted items actually exist (maps, for example, suggest that all depicted features are actually there). Such nuances in the different ways that we use logical operations are fundamental to

formal models of diagrammability. The language targets efficiency by restricting repetitions within sequences to forms that yield weak automata. Although the resulting language does not fully characterize weak automata, it does allow us to substantially enrich the set of timing diagrams that can be verified efficiently.

Several avenues remain open for future work. Establishing formal relationships between other event sequence languages and the proposed one would identify subsets of those other languages that could be verified efficiently through a mapping to weak automata. We would like to prove that the current language is maximal, in the sense that any further extension would violate either diagrammability or weakness. Finally, more 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.

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. 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.
4. 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.
5. 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.
6. K. Fisler. Timing diagrams: Formalization and algorithmic verification. *Journal of Logic, Language, and Information*, 8:323–361, 1999.
7. K. Fisler. On tableau constructions for timing diagrams. In *NASA Langley Formal Methods Workshop*, 2000.
8. 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.
9. 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.
10. 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.
11. Synopsys, Inc. Openvera assertions. Available online for download at `http://www.open-vera.com/technical/technical.html`, 2002.