A Scenario-Matching Approach to the Description and Model Checking of Real-Time Properties

Victor Braberman, Nicolas Kicillof, and Alfredo Olivero

Abstract—A major obstacle in the technology-transfer agenda of behavioral analysis and design methods is the need for logics or automata to express properties for control-intensive systems. Interaction-modeling notations may offer a replacement or a complement, with a practitioner-appealing and lightweight flavor, due partly to the subspecification of intended behavior by means of scenarios. We propose a novel approach consisting of engineering a new formal notation of this sort based on a simple compact declarative semantics: VTS (Visual Timed event Scenarios). Scenarios represent event patterns, graphically depicting conditions over traces. They predicate general system events and provide features to describe complex properties not expressible with MSC-like notations. The underlying formalism supports partial orders and real-time constraints. The problem of checking whether a timed-automaton model has a matching trace is proven decidable. On top of this kernel, we introduce a notation to state properties over all system traces: conditional scenarios, allowing engineers to describe uniquely rich connections between antecedent and consequent portions of the scenario. An undecidability result is presented for the general case of the model-checking problem over dense-time domains, to later identify a decidable—yet practically relevant—subclass, where verification is solvable by generating antiscenarios expressed in the $VTS\text{-kernel}$ notation.

Index Terms—Requirements/specifications, model checking, formal methods, scenario-based verification.

1 Introduction

Stories describing global system behavior have proven an effective technique for expressing requirements and specifications. They motivated seminal interaction-based notations such as MSCs [1]. A fundamental cause of their success in the software industry is their unpretentious scope: Their canonic use consists of mere sketches or samples of relevant interaction, thus subspecifying behavior.

On the other hand, a frequent obstacle in the technology-transfer agenda of behavioral analysis and design methods is the need for logics or automata to express properties for control-intensive systems. The formal description of properties over models is sometimes a challenging task, even for trained people. Works that address this concern usually propose easy-to-apply patterns (e.g., [2]) or visual tools based on scenarios (e.g., [3], [4], [5]), to replace expressive logics (such as TCTL [6]) and virtual observers [7]. Interaction-modeling notations may offer a replacement or a complement. In fact, MSC-like interaction-based notations are being proposed for supporting many software development activities (see Section 6) and, as a consequence, semantic issues have arisen to which clear, unique answers have not always been given. Clearly, building an extension of a (de facto) standard which has not been originally created with a formal semantics in mind is a valuable and challenging research activity. However, in our experience, taking distance from standards and creating brand-new formal notational proposals often reveals new kinds of features, help in gaining theoretical insight on decidability limits and enabling new applications of a concept (in this case, scenarios). In particular, this sort of language may greatly benefit from being engineered with a formal but simple and practitioner-understandable semantics. Adhering to the view of scenarios as a subspecification of intended behavior, a declarative semantics seems a desirable characteristic, in contrast to the prescriptive view of state-based behavioral descriptions. Besides being backed up by our experience, this is actually a common practice in logical languages engineering, where semantics are usually declarative and can be stated in a few lines.

In this paper, we fully present VTS (Visual Timed event Scenarios): a visual language to define complex event-based requirements, devised as a contribution to the point of view stated in the previous paragraph and the technology-transference agenda of automatic behavioral analysis methods. It is used to describe *event patterns*, which can be regarded as simple, graphical depictions of predicates over traces (time-stamped executions), constraining expected behavior. A $VTS\text{-kernel}$ scenario language (based on [5]) basically features annotated partial order of relevant events, denoting a (possibly infinite) set of matching traces. Violation of verification goals for real-time systems, such as freshness, bounded response, or event correlation, can naturally be expressed using the notation. We consider

- *V. Braberman and N. Kicillof are with the Computer Science Department, School of Science, University of Buenos Aires, Argentina. E-mail: {vbraber, nicok}@dc.uba.ar.*
- *A. Olivero is with the Center for Advanced Studies, Universidad Argentina de la Empresa, Argentina. E-mail: aolivero@uade.edu.ar.*

Manuscript received 29 Apr. 2005; revised 8 Nov. 2005; accepted 15 Nov. 2005; published online 22 Dec. 2005.
Recommended for acceptance by S. Uchitel, M. Broy, I.H. Krüger, and J. Whittle.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0115-0404.

system events to be more general and abstract than interactions or message-passing collaborations as they do not require identifying the intervening entities (for example, to tell a story during the early stages of the software development process). We use the word *event* to refer to any observable change during a system execution that might be of relevance for the verification of its correct behavior. For example, an event may be a key press, a function call, or the firing of a sensor. An event can represent the sending or receiving of a message, but also a whole interaction or an internal state change. Scenarios written in VTS are generally interpreted according to a negative existential semantics: A scenario represents a set of subtraces that should never occur during the execution of a system.

The problem of checking whether a system under analysis (SUA) modeled as a timed-automaton (TA) [8]—arguably the best supported formal notation for state-based modeling of real-time systems (RTS)—satisfies a VTS-kernel scenario is shown to be decidable. Moreover, we provide a tool to visually edit a scenario and to translate it into a TA (observer) that recognizes matching runs. This automaton is composed with the SUA to check whether a violating execution is reachable in that behavioral model. This last step constitutes a common practice in the verification of concurrent systems: Safety and liveness requirements are usually expressed by verification engineers as virtual components (observers) and composed in parallel with the system to be verified [7]. In fact, it is believed that properties like "every p is followed by a q and, later, by an r and the difference between p and r is at most 10" cannot be expressed using logics with bounded temporal operators [9], [10] like TCTL (and logics supported by model checkers). Tool support for this process is provided by an application that was first introduced in [5] and further in [11], where we explained how it was bundled with other tools into a powerful suite for model-checking real-time systems. It consists of a Microsoft Visio front-end used as a syntax-driven graphical editor to draw scenarios that are exported to XML and a back-end that builds observer timed-automata from resulting XML documents in formats that are readable by existing real-time model checkers like Uppaal [12] and Kronos [13]. We are currently working on a version of the tool fully integrated in the Eclipse development environment.

In this paper, we introduce a novel notation, conditional scenarios, built on top of the VTS-kernel. It provides unprecedented and powerful ways to define triggered scenarios and to relate antecedent with consequent, while keeping the semantics neat. That is, they predicate system events and provide features to describe complex properties not expressible with MSC-like notations. An undecidability result is presented for the general case of the model-checking problem over dense-time domains, to later identify a decidable—yet practically relevant—subclass, where verification is solvable by generating antiscenarios expressed in the VTS-kernel notation.

The paper is structured as follows: We first recall basic (kernel) VTS scenarios by means of an example, together with their formal syntax and semantics. We then show the procedure to model check these scenarios. After that, we introduce conditional scenarios, an undecidability result for the general case, and an effective subclass to overcome this restriction. Then, the verification technique for this subclass by building kernel antiscenarios for a conditional scenario is explained. Finally, we relate this work to other similar efforts and draw conclusions.

2 VTS Kernel

2.1 Introduction

The basic elements of our graphical notation are points connected by lines and arrows. Points are labeled by (possibly empty) sets of events, meaning that the point stands for an occurrence of one of the events during execution. Our language has mechanisms to represent certain relations between these points, such as *precedence* and *temporal distance*, restrictions, such as *forbidden events*, and to identify special points as the *first* or *last* in a set. Relations and restrictions can also involve *begin* or *end* of system execution. Each of these features has a graphical representation introduced in the following example.

In a reengineering project for the electronic market at the Buenos Aires Stock Exchange, part of our team was involved in suggesting ways to rigorously specify real-time requirements for the problem of brokers placing bids and receiving updated data. The goal was to evaluate the success or failure of competing protocols at meeting those requirements. We have selected two protocols: a protocol that periodically broadcasts a token to collect the bids accumulated in brokers' machines (the broadcast acts as a logical tick) and a TRMP-based protocol ([14]). We are interested in four requirements:

Freshness of bids: When the data is processed, the age of the corresponding bid is bounded.

Bounded unfair time window: Two bids from different brokers collected in the same token broadcast round belong to the same logical time tick. Such a pair of bids might thus be matched to the offers in an incorrect order (with respect to the wall-clock time).

Bids are collected: When there is a submission by a broker, she will eventually receive the token.

Simultaneity: Remulticasts of messages for multiple receivers are almost simultaneous (the time elapsed between two different remulticasts is bounded).

In Fig. 1, we illustrate how these requirements are expressed in VTS as antiscenarios: Fig. 1a matches any violation of the bound on the age of the bid when the data is processed. Fig. 1b matches a violation of time limit for the unfair behavior (that is, the antiscenario detects two bids processed on the same round and submitted in two too-distant moments). Fig. 1c expresses the generic violation of the requirement that the token must eventually arrive; Fig. 1d shows a generic violation of the "simultaneity" requirement considering three receivers.

An arrow between two points indicates the *precedence* of the source with respect to the destination: For instance, in Fig. 1a, *Submission broker A* precedes *arrival token for broker A*. Events labeling the arrow are interpreted as *forbidden events* between both points: For instance, in the previous mentioned arrow, there is no *arrival token for broker A* event in

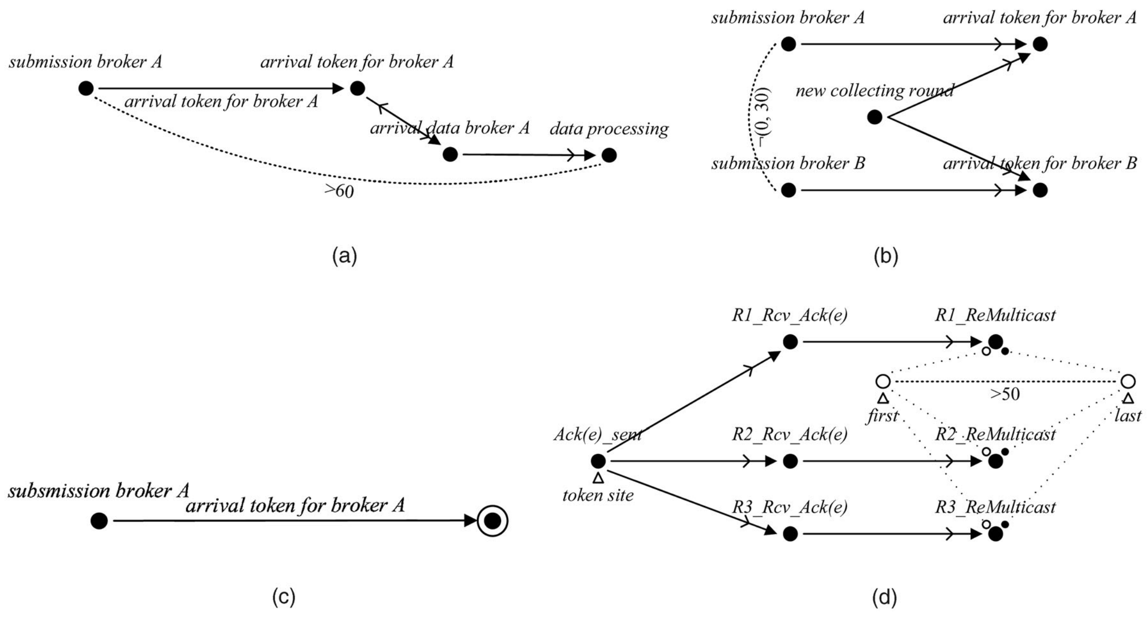

Fig. 1. Stock Exchange antiscenarios. (a) Bids are not fresh enough. (b) Unfair time-window is beyond threshold. (c) Bids are not collected. (d) Simultaneity violated.

between (i.e., the arrival associated to the point is the *next* event of this kind after the submission).

We use an abbreviation for this frequent subpattern: A certain point represents the *next* occurrence of an event after another. The abbreviation is a second (open) arrow near the destination point (in the text $\rightarrow\!\!\!\rightarrow$), and is equivalent to adding its set of labels as forbidden events on the arrow (the abbreviated form can be seen in Fig. 1a between *arrival data broker A* and *data processing*). Conversely, to represent the *previous* occurrence of a (source) event, there is a symmetrical notation: an open arrow near the source extreme (in the text $\leftarrow\!\!\!\rightarrow$). Fig. 1a includes this kind of abbreviation when relating *Arrival token for broker A* and *Arrival data broker A*.

A dashed line linking two points allows us to express a *temporal distance* between them. In Fig. 1b, the $\neg(0, 30)$ restriction means that the temporal distance between the responses is $\leq$ 0 t.u. or $\geq$ 30 t.u.. We use a line instead of an arrow because these must match two different points in the execution, without a known precedence (their relative order is unimportant). We can also include forbidden events on a dashed line.

Another useful idiom asserts that something happens (or not) since the beginning or until the end of an execution. VTS has two special symbols: a big full circle for *begin* and two concentric circles for *end*. For example, the scenario in Fig. 1c states that there is no token arrival for a *submission broker A*.

VTS can also identify the *first* or the *last* event in a set. The graphical representation for the first event in a set is a point linked to each point in the set by dotted lines ending in small empty circles. The notation for the last event uses full circles. The scenario of Fig. 1d expresses a temporal distance between the first and last *ReMulticast* events to occur in a set of three.

Triangles below points are used to assign them an optional name.

Fig. 2 shows the complete graphical notation of VTS.

2.2 VTS Formal Syntax

Definition 2.1 (Scenario). *A* scenario *is a tuple* $\langle \Sigma, P, \ell, \not\equiv, <, <_F, <_L, \gamma, \delta \rangle$, *where* Σ *is a finite set of events,* P *is a finite set of points;* $\ell : P \rightarrow 2^{\Sigma}$ *is a function that labels each point with a set of events,* $\not\equiv \subseteq P \times P$ *is an asymmetric relation among points (inequality);* $< \subseteq (P \uplus \{\mathbf{0}\} \times P \uplus \{\infty\}) \setminus \{\langle \mathbf{0}, \infty \rangle\}$ *is a precedence relation between points (*$\mathbf{0}$ *and* ∞ *represent the beginning and the end of execution, respectively);* $<_F \subseteq P \times P$ *links a point representing the first in a set with each of its members;* $<_L \subseteq P \times P$ *links each point in a set to another one representing the last to occur;* $\gamma : (\not\equiv \cup <) \rightarrow 2^{\Sigma}$ *assigns to each pair of points, related by inequality or precedence, the set of events forbidden between them; and* $\delta : (\not\equiv \cup <) \rightarrow \Phi$ *(see below) assigns to each inequality or precedence a restriction for the time elapsed between the two points.*

We call $\mathcal{I}_{\mathrm{N}}$ the set of integer-bounded intervals of positive real numbers. A *time restriction* is a formula of the form θ or $\neg\theta$, where θ is an interval in $\mathcal{I}_{\mathrm{N}}$. Φ is the set of all time restrictions. Given a nonnegative real number t and an interval θ, we say $t \models \theta$ iff $t \in \theta$ and $t \models \neg\theta$ iff $t \notin \theta$.

2.3 VTS Semantics

Definition 2.2 (Sequence). *Given a set* C, *a* sequence over C *is a (possibly infinite) sequence of elements from* C. *Given a sequence* s, $|s|$ *is its length (we say that* $|s| \overset{def}{=} \infty$ *when* s *is infinite) and* $\Pi(s) \overset{def}{=} \{i \in \mathrm{N} \ / \ 0 \leq i < |s|\}$ *is the set of positions of* s. *Given* $i, j \in \Pi(s)$, s_i *is the* ith *element of* s, $s_{i]}$ *is the prefix ending at position* i, $s_{[i}$ *is the suffix starting at position* i, *and* $s_{[i,j]}$ *is the subsequence from position* i *to position* j *(if* $i > j$, $s_{[i,j]} \overset{def}{=} s_{[j,i]}$*). Using "(" or ")" instead of "[" or "]" means the corresponding subsequence does not*

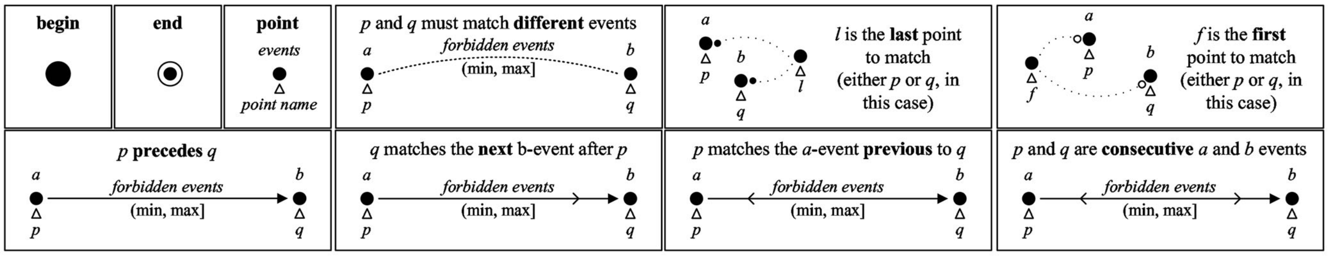
begin
end
point
events
point name
p and q must match different events
a
forbidden events
b
(min, max]
p
q
l is the last point to match (either p or q, in this case)
f is the first point to match (either p or q, in this case)
p precedes q
q matches the next b-event after p
p matches the a-event previous to q
p and q are consecutive a and b events

Fig. 2. VTS graphical notation.

include its border(s). We call $first(s)$ the first element of s. If s is finite, $last(s)$ is its last element.

Definition 2.3 (Temporal Sequence). *A* temporal sequence *is a weakly increasing sequence of timestamps (i.e., nonnegative real numbers). Given a finite temporal sequence τ, we define $\Delta(\tau)$ as the time elapsed during τ: $\Delta(\tau) = last(\tau) - first(\tau)$ or 0 if $|\tau| = 0$. A temporal sequence τ can be* shifted *by a real number ϵ, producing a temporal sequence called $\tau + \epsilon$ such that $\forall i \in \Pi(\tau); (\tau + \epsilon)_i = \tau_i + \epsilon$. We define the* temporal concatenation *of two temporal sequences τ and τ' as $\tau \rhd \tau' = \tau(\tau' + last(\tau))$. For example, $(0\ 2\ 3\ 5.5) \rhd (1\ 5.3) = (0\ 2\ 3\ 5.5\ 6.5\ 10.8)$.*

Definition 2.4 (Trace). *A* trace *over a set C is a pair $\langle s, \tau \rangle$, where s is a sequence over $C \cup \{\lambda\}$ and τ is a temporal sequence of the same length (λ stands for an instant in a trace with no associated event-label).*

The semantics of VTS assigns to each scenario a set of traces satisfying it. Labeled points represent events in the traces; they can match a particular position in a trace if the event in that position is among the allowed events associated to the point by the labeling function ℓ. Unlabeled points are called *instants*. They represent moments in the execution when no event occurs (λ).

We define the set of *first-representatives* (respectively, *last*) as $FirstReps = Dom(<_F)$ (respectively, $LastReps = Ran(<_L)$). A point $\mathtt{p}$ in these sets is called a *representative* as it represents the first (respectively, last) point in the set $R = \{\mathtt{p}'/\mathtt{p} <_F \mathtt{p}'\}$ (respectively, $\{\mathtt{p}'/\mathtt{p}' <_L \mathtt{p}\}$) to be matched in a trace. The sets R for a (first or last) representative $\mathtt{p}$ are denoted $FirstOf(\mathtt{p})$, $LastOf(\mathtt{p})$, respectively. The remaining points are called *concrete*.

Intuitively, a *matching* is a mapping between points in a scenario and positions in a trace, exhibiting how the trace satisfies the scenario.

Definition 2.5 (Matching). *Given a scenario $\mathcal{S} = \langle \Sigma, P, \ell, \not\equiv, <, <_F, <_L, \gamma, \delta \rangle$, a trace $\sigma = \langle s, \tau \rangle$ over Σ and a mapping $\hat{ } : P \rightarrow \Pi(\sigma)$, we say that $\hat{ }$ is a* matching *between $\mathcal{S}$ and σ iff, for all points $\mathtt{p}, \mathtt{q} \in P$:* **M1** *if $\ell(\mathtt{p}) = \emptyset$, then $s_{\hat{\mathtt{p}}} = \lambda$, otherwise $s_{\hat{\mathtt{p}}} \in \ell(\mathtt{p})$;* **M2** *if $\mathtt{p} \not\equiv \mathtt{q}$, then $\hat{\mathtt{p}} \neq \hat{\mathtt{q}}$;* **M3** *if $\mathtt{p} < \mathtt{q}$, then $\hat{\mathtt{p}} < \hat{\mathtt{q}}$;* **M4** *$s_{(\hat{\mathtt{p}}, \hat{\mathtt{q}})} \cap \gamma(\mathtt{p}, \mathtt{q}) = \emptyset$;* **M5** *$s_{\hat{\mathtt{p}})} \cap \gamma(0, \mathtt{p}) = s_{(\hat{\mathtt{p}}} \cap \gamma(\mathtt{p}, \infty) = \emptyset$;* **M6** *$\Delta(\tau_{[\hat{\mathtt{p}}, \hat{\mathtt{q}}]}) \models \delta(\mathtt{p}, \mathtt{q})$;* **M7** *$\Delta(\tau_{\hat{\mathtt{p}}]}) \models \delta(\mathbf{0}, \mathtt{p})$;* **M8** *if $\mathtt{p} \in FirstRep$ (respectively, LastRep), then $\hat{\mathtt{p}} = \min\{\hat{\mathtt{r}}/\mathtt{r} \in FirstOf(\mathtt{p})\}$ (respectively,* max *and LastOf).*

Definition 2.6 (Existential VTS Semantics). *We say that a trace σ* satisfies *a scenario $\mathcal{S}$ (noted $\sigma \models \mathcal{S}$) iff there exists at least one matching between them. We say that a timed automaton satisfies a scenario ($\mathcal{A} \models \mathcal{S}$) iff it exhibits at least a time-divergent[1] trace satisfying the scenario.*

Note that a representative should be typically labeled with all labels associated to the points it represents since it is used to represent an event in those sets. In our graphical notation, representatives receive no label in order to keep diagrams clean.

3 Model Checking for Existential VTS Semantics

In this section, we provide a mapping from VTS scenarios to timed automata. Given a scenario, we describe a tableau that recognizes all traces matching the scenario.

3.1 Preliminaries: Timed Automata

Timed automata are a widely used formalism to model and analyze timed systems. They are supported by several tools (e.g., [12], [13]). Their semantics is based on labeled state-transition systems and time-divergent runs over them. A complete formal presentation can be found in [8], [13].

Definition 3.1 (Timed Automation). *A* timed automaton *is a tuple $\mathcal{A} = \langle L, X, \Sigma, E, I, \mathrm{l}_0 \rangle$, where L (denoted $locs(\mathcal{A})$) is a finite set of locations, X is a finite set of clocks (nonnegative real variables), Σ (denoted $label(\mathcal{A})$) is a set of labels, E is a finite set of edges, $I : L \overset{tot}{\rightarrow} \Psi_X$ is a total function associating to each location a clock constraint (see below) called the location's invariant, and $\mathrm{l}_0 \in L$ is the initial location (denoted $init(\mathcal{A})$). Each edge in E is a tuple $\langle \mathrm{l}, a, \psi, \alpha, \mathrm{l}' \rangle$, where $\mathrm{l} \in L$ is the source location, $\mathrm{l}' \in L$ is the target location, $a \in \Sigma$ is the label, $\psi \in \Psi_X$ is the guard, $\alpha \subseteq X$ is the set of clocks reset at the edge. The set of clock constraints Ψ_X for a set of clocks X is defined according to the following grammar: $\Psi_X \ni \psi ::= x \prec c | \psi \wedge \psi | \neg \psi$, where $x \in X, \prec \in \{<, \leq\}$ and $c \in \mathrm{N}$.*

Usually, a TA $\mathcal{A}$ has an associated mapping $Pr : locs(\mathcal{A}) \mapsto 2^{Props}$, which assigns to each location a subset of a set of propositional variables ($Props$). The parallel composition $\mathcal{A}_1 \parallel \mathcal{A}_2$ of TAs $\mathcal{A}_1$ and $\mathcal{A}_2$ is defined using the synchronous product of automata [13].

3.2 Tableau Construction

For the rest of the section, when we refer to a scenario $\mathcal{S}$, we are referring to the tuple $\langle \Sigma, P, \ell, \not\equiv, <, <_F, <_L, \gamma, \delta \rangle$.

1. τ is a time-divergent trace iff, for any real number T, there exists a position k such that $\Delta(\tau_{k]}) > T$.

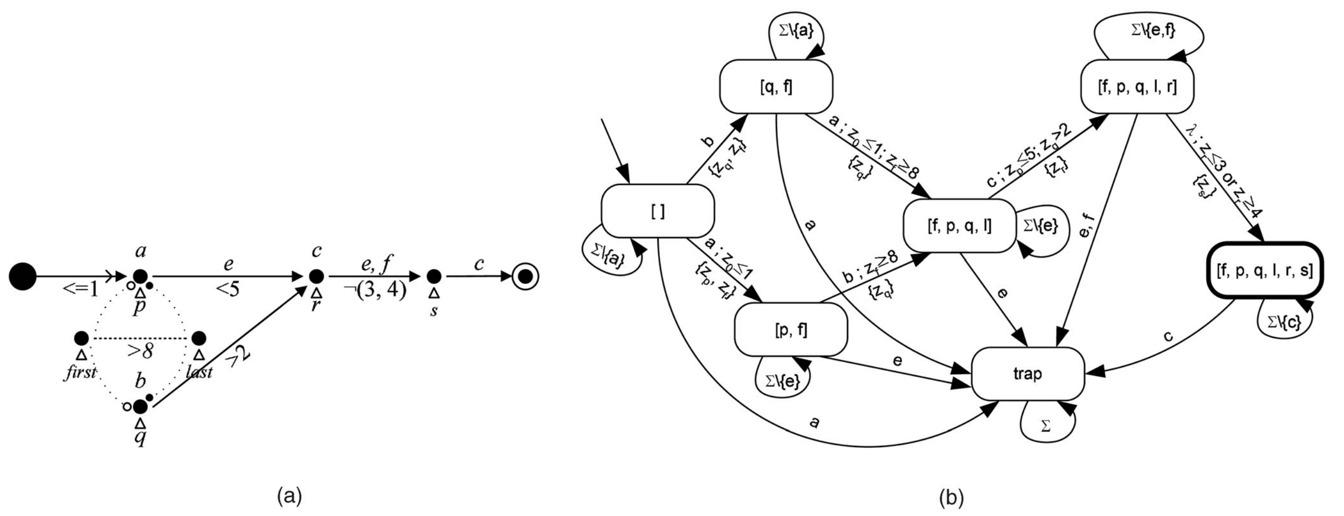

Fig. 3. VTS scenario and corresponding tableau. (a) Scenario. (b) Tableau.

A *configuration* Θ of a scenario $\mathcal{S}$ is a subset of P such that: **C1** Θ is left-closed under the relation $(< \cup <_F \cup <_L)$; **C2** if $\mathrm{p} \in FirstRep$ and $\mathrm{p} \in \Theta$, then $FirstOf(\mathrm{p}) \cap \Theta \neq \emptyset$; and **C3** if $\mathrm{q} \in LastRep$ and $LastOf(\mathrm{q}) \subseteq \Theta$, then $\mathrm{q} \in \Theta$. We call $Conf \subseteq 2^P$ the set of all configurations.

A pair $\langle a, F\rangle \in \Sigma \times 2^P$ is a *single step extension* of $\Theta \in Conf$ (denoted $\Theta \xrightarrow{a} \Theta \cup F$) iff **E1** $\Theta \cup F \in Conf$; **E2** $(\nexists \mathrm{p}, \mathrm{q} \in F)\langle \mathrm{p}, \mathrm{q}\rangle \in (\not\equiv \cup <)$; and **E3** $(\forall \mathrm{p} \in F)(a \in \ell(\mathrm{p}))$. We say that $\Theta \xrightarrow{\lambda} \Theta \cup F$ iff conditions **E1** and **E2** hold, while **E′3** $(\forall \mathrm{p} \in F)(\ell(\mathrm{p}) = \emptyset)$.

We define the set of *active event restrictions* for a configuration Θ, $AR(\Theta) \subseteq \not\equiv \cup <$ as follows: $\langle \mathrm{p}, \mathrm{q}\rangle \in AR(\Theta)$ iff $\mathrm{p} \in \Theta, \mathrm{q} \notin \Theta$; $\langle \mathbf{0}, \mathrm{p}\rangle \in AR(\Theta)$ iff $\mathrm{p} \notin \Theta$; and $\langle \mathrm{p}, \infty\rangle \in AR(\Theta)$ iff $\mathrm{p} \in \Theta$. We call $\Gamma(\Theta) \overset{def}{=} \bigcup \gamma(\mathrm{p}, \mathrm{q})$ for all $\langle \mathrm{p}, \mathrm{q}\rangle \in AR(\Theta)$. Furthermore, the set of *strictly active event restrictions* for a configuration Θ with respect to a set of points F is defined as $SAR(\Theta, F) \overset{def}{=} \{\langle \mathrm{p}, \mathrm{q}\rangle \in AR(\Theta) / \ q \notin F\}$. We call $\Gamma_F(\Theta) \overset{def}{=} \bigcup \gamma(\mathrm{p}, \mathrm{q})$ for all $\langle \mathrm{p}, \mathrm{q}\rangle \in SAR(\Theta, F)$.

Given a time restriction φ and a clock x, we define $\psi_x(\varphi)$ as the clock constraint over x such that, for every nonnegative real number t, $\psi_x(\varphi)[x \backslash t]$ is true iff $t \models \varphi$. For example, $\psi_x((a, b])$ is the constraint $a < x \wedge x \leq b$.

Given a scenario $\mathcal{S}$ and $F \subseteq P$, we define the set $R_F \overset{def}{=} \{x_{\mathrm{p}} / \mathrm{p} \in F \wedge \exists \mathrm{q}.\delta(\mathrm{p}, \mathrm{q}) \neq [\mathbf{0}, \infty]\}$. Given a configuration Θ and an extension F, we define

$$\psi_\Theta^F \overset{def}{=} \bigwedge_{\mathrm{p} \in \Theta \uplus \{\mathbf{0}\}, \mathrm{q} \in F} \psi_{x_\mathrm{p}}(\delta(\mathrm{p}, \mathrm{q})).$$

Definition 3.2 (Tableau construction). *The* tableau $\mathcal{T}_\mathcal{S}$ *for VTS scenario $\mathcal{S}$ is a timed automaton $\langle L, X, \Sigma, E, I, \mathrm{l}_0\rangle$ such that:* **T1** $L = Conf \uplus \{\mathrm{s}_{\mathrm{trap}}\}$, *we call* $\mathrm{s}_{\mathrm{accept}}$ *the configuration P (i.e., the configuration with all the points in the scenario);* **T2** $X = \{x_\mathrm{p} / \mathrm{p} \in P \uplus \{\mathbf{0}\}\}$; **T3**

$$\begin{aligned} E = & \{\langle \Theta, a, \psi_\Theta^F, R_F, \Theta'\rangle / \Theta \xrightarrow{a} \Theta' \wedge a \notin \Gamma_F(\Theta)\} \\ & \cup \{\langle \Theta, \lambda, \psi_\Theta^F, R_F, \Theta'\rangle / \Theta \xrightarrow{\lambda} \Theta'\} \\ & \cup \{\langle \Theta, a, true, \emptyset, \Theta\rangle / a \in \Sigma \wedge a \notin \Gamma(\Theta)\} \\ & \cup \{\langle \Theta, a, true, \emptyset, \mathrm{s}_{\mathrm{trap}}\rangle / a \in \Gamma(\Theta)\} \\ & \cup \{\langle \mathrm{s}_{\mathrm{trap}}, a, true, \emptyset, \mathrm{s}_{\mathrm{trap}}\rangle / a \in \Sigma\} \text{ where } \Theta' = \Theta \cup F; \end{aligned}$$

T4 $(\forall \mathrm{l} \in L)(I(\mathrm{l}) \equiv true)$; **T5** l_0 *is the empty configuration.*

Fig. 3b shows the resulting tableau for the VTS scenario in Fig. 3a.

Theorem 3.3 (Model checking VTS**).** *Given a Timed Automaton $\mathcal{A}$ and a scenario $\mathcal{S}$, with $\Sigma \subseteq label(\mathcal{A})$, $Pr(accept) = locs(\mathcal{A}) \times \{\mathrm{s}_{\mathrm{accept}}\}$ and $Pr(init) = \{(init(\mathcal{A}), init(\mathcal{T}_\mathcal{S}))\}$:*

$$\mathcal{A} \models \mathcal{S} \qquad \text{iff} \qquad \mathcal{A} || \mathcal{T}_\mathcal{S} \models init \Rightarrow \exists\Diamond(\exists\Box\, accept).$$

Since the $accept$ proposition is associated to P (the whole set of points), the satisfaction of the TCTL [6] property $init \Rightarrow \exists\Diamond\, accept$ ("$accept$ is reachable from $init$") would mean that we can match the scenario. However, we have to check whether the execution can stay in that acceptance node without receiving any forbidden events (i.e., events that should not occur until ∞), that is, $init \Rightarrow \exists\Diamond(\exists\Box\, accept)$ ("It is possible to evolve from $init$ to $accept$ and remain there forever"). The complete decidability proof and details related to the tableau construction can be found in [15].

3.3 Example

The Remote Sensing case study is an abstraction of an industrial distributed real-time control system adhering to a fixed-priority scheduling architectural style. This system basically consists of a central component and two remote sensors ($Sensorx$ with x = 1, 2). Sensors periodically sample a set of environmental variables ($SampleVx$ event) and store the values in shared memory ($StoreVx$ event). When the central component needs them, it broadcasts a signal to the sensors with a minimal interarrival time ($RqstSent$ event). When the signal arrives ($RqstArrivesAtSx$ event) and there is budget to run a sporadic thread devoted to handling this message, that task is awakened ($UpRx$ event),

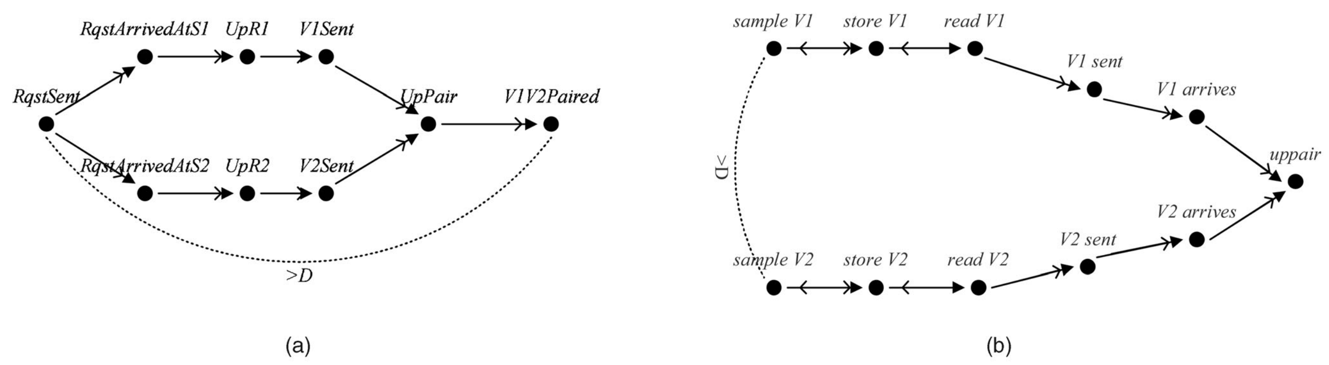

Fig. 4. Remote sensing properties. (a) Bounded response. (b) Correlation.

reads the last stored value from shared memory ($ReadVx$ event), and sends it back to the central component ($VxSent$ event). Upon arrival of both pieces of data ($V1Arrives$ and $V2Arrives$ events) and when the budget is refreshed, a sporadic task is awakened ($UpPair$ event). That task pairs the readings ($V1V2Paired$ event) so that another process can use that piece of information to perform certain actions on some actuators.

The formal model for this example was generated using the technique introduced in [16] to translate real-time system designs based on fixed-priority scheduling into timed automata. The idea underlying the translation is the use of fixed-priority scheduling theory to build a conservative (and practically analyzable) model of behavior which can be checked by timed model checkers such as Kronos and Uppaal.

In this case, the model is comprised of 12 timed automata (one for each design component) interacting asynchronously.

The antiscenario of Fig. 4a shows a pattern that, if matched, would indicate that a request to collect a pair of data items from the central components takes more than D time units to wake up the receiving task in the central processor.

A correlation requirement is informally stated as "paired values should have been sampled in not-too-distant time instants." Fig. 4b illustrates how to compactly formalize the antiscenario for correlation in VTS.

Table 1 summarizes the translation and verification results for some of the scenarios using a Windows-based version of the Uppaal tool on a Pentium IV 1.6 Mhz PC with 512 MB. It includes the size of the observers, as well as the verification options and optimization techniques used.[2]

TABLE 1
Verification Results

Bounded Response - Tableau size: # locs = 29 and # trans = 520				
Value for D	Transl. time	Verification		
		Match?	time	options
380	0.8 sec.	Yes	5.6 secs.	ObsSlice + -Was
390	0.8 sec.	No	5.8 secs.	ObsSlice + -Was
Correlation - Tableau size: # locs = 57 and # trans = 959				
Value for D	Transl. time	Verification		
		Match?	time	options
185	1.1 sec.	Yes	147 secs.	ObsSlice + -WasdC
195	1.1 sec.	No	6 secs.	-Was -A

4 VTS Conditional Scenarios

4.1 Introduction

In this section, we present an extension to VTS introducing a new kind of scenarios interpreted as conditional predicates on scenario matchings, universally quantified over all system traces.

In a few words, conditional scenarios are useful to state that, whenever a matching is encountered in a given trace for a subscenario (the antecedent), the same matching must be extensible to cover one in the set of consequent scenarios. This notion is more flexible than other known trigger-based approaches (e.g., [3], [18], [19], [20]) since extensions are not necessarily subsequent (future) events: They can predicate about previous occurrences of events (e.g., "if ack is found, then $request$ must have been issued at least 10 t.u. before") and they can even add constraints to the antecedent event-pattern (e.g., "if two events a and b are encountered in a trace, then either no c event can be found in between or the distance is greater than 5 t.u.").

Definition 4.1 (Scenario Specialization). *Given two scenarios, $\mathcal{S}_1, \mathcal{S}_2$, we say that $\mathcal{S}_2$ specializes $\mathcal{S}_1$ (denoted $\mathcal{S}_2 <: \mathcal{S}_1$) iff* **Sp1** $P_1 \subseteq P_2$;[3] **Sp2** $\Sigma_1 \subseteq \Sigma_2$; **Sp3** $\ell_2(\mathrm{p}) \subseteq \ell_1(\mathrm{p})$ *for all* $\mathrm{p} \in P_1$; **Sp4** $<_1 \subseteq <_2$; **Sp5** $<_{F1} \subseteq <_{F2}$; **Sp6** $<_{L1} \subseteq <_{L2}$; **Sp7** $\gamma_1(\mathrm{p},\mathrm{q}) \subseteq \gamma_2(\mathrm{p},\mathrm{q})$ *for all* $\mathrm{p},\mathrm{q} \in P_1$; **Sp8** $\delta_2(\mathrm{p},\mathrm{q}) \subseteq \delta_1(\mathrm{p},\mathrm{q})$ *for all* $\mathrm{p},\mathrm{q} \in P_1$; **Sp9** $\not\equiv_1 \subseteq \not\equiv_2$.

It is easy to see that the following property holds:

Property 4.2. *Given $\mathcal{S}_1$ and $\mathcal{S}_2$, two scenarios such that $\mathcal{S}_2 <: \mathcal{S}_1$, σ a trace over Σ_2, and $\hat{\ }$ a matching between $\mathcal{S}_2$ and σ, then $\hat{\ }|_{P_1}$ ($\hat{\ }$ restricted to P_1) is a matching between $\mathcal{S}_1$ and σ.*

Definition 4.3 (Conditional Scenario). *Given a scenario $\mathcal{S}_0$ (antecedent) and an indexed set of scenarios $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k$ (consequents) such that $\mathcal{S}_i <: \mathcal{S}_0$ for $i \in 1 \ldots k$ and $P_i \cap P_j = P_0$ for $i \neq j \in 1 \ldots k$, we call $\mathcal{C} = \langle \mathcal{S}_0, \{\mathcal{S}_i\}_{i=1\ldots k} \rangle$ a* Conditional Scenario (CS).

2. We use *ObsSlice*, a slicer for timed models based on the topology of the observer (see [17]) that reduces the amount of work to be done by the model checker (often rendering the analysis feasible).

3. Note that, to avoid unnecessary formal noise, we define in this paper these notions based on set inclusion; a more general treatment requires an embedding function between points of the scenarios.

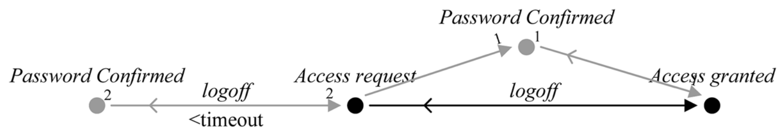

Fig. 5. Authorization.

Definition 4.4 (Semantics of CSs). *A trace $\sigma = \langle s, \tau \rangle$ satisfies a CS $\mathcal{C} = \langle \mathcal{S}_0, \{\mathcal{S}_i\}_{i=1\dots k} \rangle$ ($\sigma \models \mathcal{C}$) iff, for every matching $\hat{\cdot}$ between $\mathcal{S}_0$ and σ there exists $\hat{\hat{\cdot}}$ a matching between $\mathcal{S}_i$ and σ,* **for some** *$i \in \{1..k\}$, such that $\forall \mathrm{p} \in P_0 . \hat{\mathrm{p}} = \hat{\hat{\mathrm{p}}}$ (i.e., $\hat{\hat{\cdot}}$* extends *$\hat{\cdot}$).*

Fig. 5 shows a graphical example of a conditional scenario. In our graphical notation, all consequents are depicted in the same picture sharing a common antecedent. We use black for all elements in the antecedent scenario (which is also included in all consequents). Elements in consequent (alternative) scenarios are drawn in gray, with a superscript number identifying the scenario they belong to. The interpretation of the scenario in Fig. 5 is that, whenever an *Access request* event is followed by an *Access granted* event (without a *logoff* in between), one of two other event sequences must be observed, too. One of them (consequent 1) requires that, after the access has been requested, a valid password is entered. The other one (consequent 2) allows for the case where a valid password had been entered before access to the resource was requested. Observe the power of our trigger notation, where the antecedent need not precede the consequents in time.

Another sample scenario for a robot case study (Fig. 6) states that either an error message is prompted (scenario 2) or both motors must start in any order (consequent 1), with the time between the first to start and the last to stop being shorter than 100 t.u. and no error prompted.

Languages defined by CSs are not necessarily timed regular. For instance, the language containing all timed words where each a is followed in one t.u. by a b is not expressible as a timed automata (see [21]), while it is indeed expressible as the language accepted by the simple conditional scenario shown in Fig. 7a or the complement language for the scenario in Fig. 7b. This implies that a model-checking strategy for CSs based on a tableau construction for their language or its complement is unfeasible.

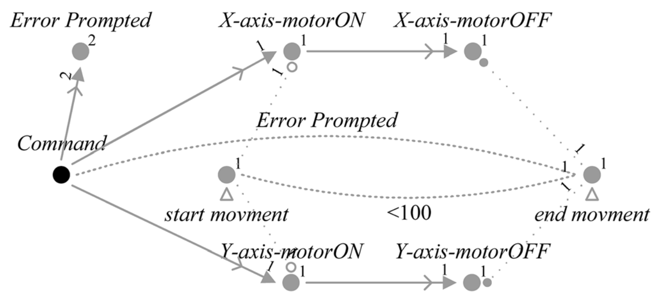

Fig. 6. Moving robot.

4.2 Undecidability Result: Model-Checking Conditional Scenarios over Dense-Time Domains

The problem of checking timed-automata models for adherence to CSs is highly undecidable; more precisely, it is a $\prod_i^i$-hard problem (the same can be said about satisfiability over dense-time domains). The proof is inspired in the proof for the universality problem of Büchi Timed Automata exhibited in [8]. We use the very same encoding of nondeterministic 2-counter machine computations using timed words. That is, a configuration $\langle i, c, d \rangle$ (location and counters) is represented with the sequence $b_i(a_1)^c(a_2)^d$. We require that the subsequence of σ corresponding to the time interval $[j, j+1)$ encode the jth configuration of the computation.

The denseness of the underlying time domain allows the counter values to get arbitrarily large. The ability to feature punctual intervals as constraints on time distances leads to undecidability of the satisfaction and other related problems for many real-time logics (see [22]) and the same can be said of CSs.

Fig. 8a and Fig. 8b illustrate how to encode, as a CS, the rules that recognize any timed word that either is not a correct encoding of a computation or is not recurring (i.e., the first location is not traversed infinitely often). Note that, if the model checking problem were decidable, checking the universal automaton against those CSs would answer whether or not there is a recurring computation since the universal language necessarily includes all valid 2-counter machine computations (similarly for the *or* satisfiability problem).

Let us see some details of the rules encoded in Fig. 8a. Consequent 1 detects cases where b_1 is not the first instruction, while consequent 2 matches traces where the first b_1 is not observed in time $t = 1$ (we identify two instants, surrounding $t = 1$, and forbid b_1 between them). Consequent 3 is matched whenever the first interval is ill-formed due to an incorrectly placed symbol. Consequent 4 is matched whenever there exists a pair of consecutive configurations not separated by 1 t.u., while consequent 5 checks the violation of the $(a_1)^*(a_2)^*$ pattern. Consequent 6 accepts finite computations (a case of ill-formedness). Finally, consequent 7 matches a nonrecurring situation.

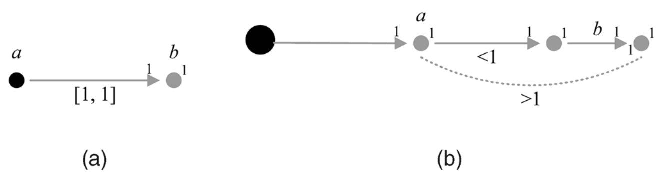

Fig. 7. Scenarios accepting a non-timed-regular (NTR) language and its complement. (a) NTR. (b) NTR complement.

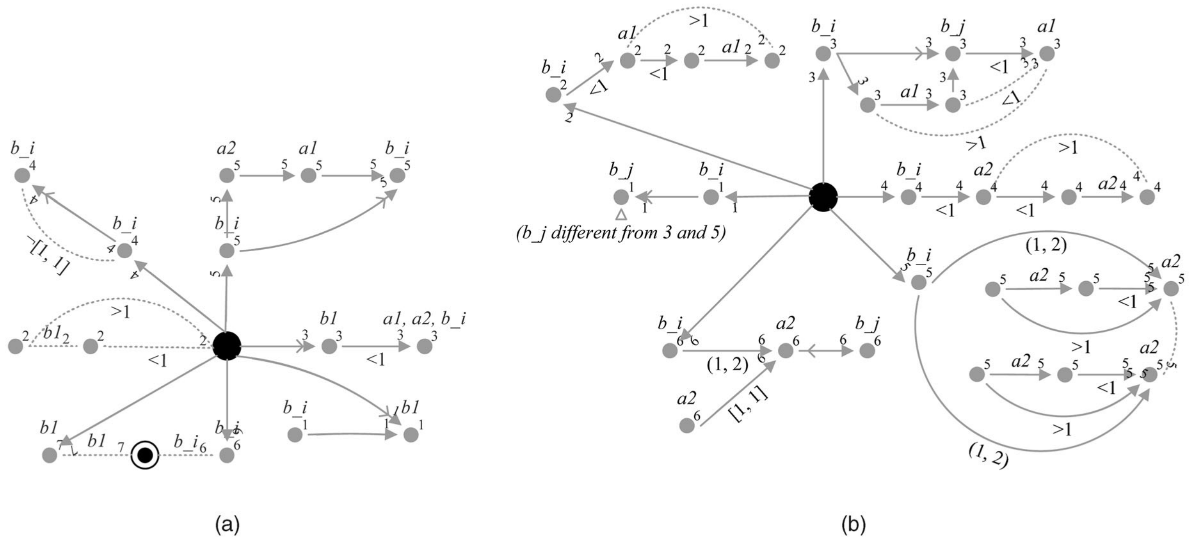

Fig. 8. Coding two counter nondeterministic machine. (a) Common rules. (b) Instruction rates.

For each instruction $1 \leq i \leq n$, a set of consequents is added to detect cases where b_i occurs at time t and the configuration in $[t+1, t+2)$ does not follow from the configuration corresponding to the subsequence $[t, t+1)$ by executing instruction i.

As in [8], we show, in Fig. 8b, a construction for a sample instruction, say, "Increment the counter D and jump nondeterministically to instruction 3 or 5," In this figure, each consequent corresponds to one of the timed automata built in the proof of [8]. Consequent 1 is matched when there exists a b_i instance at time t and there is neither a b_3 nor a b_5 at time $t+1$. Consequent 2 accepts a run iff there is some b_i at time t followed by an a_1 at time $t < t+1$ such that there is no event matching a_1 at time $t+1$. Similarly, consequent 3 accepts a run if there is some b_i at time t and, for some $t < t+1$, there is no a_1 at time t, but there is an a_1 at time $t+1$. The complements of consequents 2 and 3 together ensure proper matching of a_1s. On the other hand, consequent 4 detects that, for some b_i at time t, there is an a_2 at some $t < t+1$ with no match at $t+1$. Consequent 5 is matched when, for some b_i at time t, there are two a_2s in $(t+1, t+2)$ without matches in $(t, t+1)$. Consequent 6 says that, for some b_i at time t, the last a_2 in the interval $(t+1, t+2)$ has a matching a_2 in $(t, t+1)$. It is not difficult to see that consequents 4, 5, and 6 capture all situations where the encoding of a_2s in the intervals $(t, t+1)$ and $(t+1, t+2)$ does not match according to the desired scheme (see [8]).

5 Deterministic Conditional Scenarios

5.1 Introduction

The undecidability result motivates the identification of a decidable, in practice relevant, fragment of the VTS CS language. We are particularly interested in a case we call deterministic CSs (DCSs), where points in the consequent have at most a single way to be matched, provided that the antecedent has been matched. What makes them interesting is that they are not only useful in practice, but also amenable to model checking by building a set of basic scenarios that stand for the possible counterexamples of the DCS's satisfaction.[4] From a syntactical perspective, points in the consequents of a DCS are reachable by following "next" and "previous" precedences from some point in the antecedent. There is no restriction on the structure of the antecedent. The authorization and the robot examples of Fig. 5 and Fig. 6 are actually DCSs since points in their consequents are reachable from their antecedent patterns following paths for which edges are either previous or next precedences.

To formalize this notion, we need to introduce some definitions.

Definition 5.1 (Pointwise determination). *Given two points* p_1 *and* p_2 *(where* p_2 *is a concrete and noninstant point) in a scenario, we say that* p_1 determines p_2 *(denoted* $\mathrm{p}_1 \hookrightarrow \mathrm{p}_2$*) iff either* $\ell(\mathrm{p}_2) \subseteq \gamma(\mathrm{p}_1, \mathrm{p}_2)$ *and* $\mathrm{p}_1 < \mathrm{p}_2$*, or* $\ell(\mathrm{p}_2) \subseteq \gamma(\mathrm{p}_2, \mathrm{p}_1)$ *and* $\mathrm{p}_2 < \mathrm{p}_1$*. In the graphical notation, this can be expressed as* $\mathrm{p}_1 \twoheadrightarrow \mathrm{p}_2$ *or* $\mathrm{p}_2 \longleftrightarrow \mathrm{p}_1$*, respectively.*[5]

Extending this notion to a set of points yields:

Definition 5.2 (Determination). *Given a point* p *and a set of points* P*, we say that* P determines p *(denoted as* $P \hookrightarrow \mathrm{p}$*) iff*

1. $\mathrm{p} \in P$ *or*
2. *there exists a point* $\mathrm{p}' \in P$ *such that* $\mathrm{p}' \hookrightarrow \mathrm{p}$ *or*
3. $FirstOf(\mathrm{p}) \subseteq P$ *(when* p *is a first-rep.) or*
4. $LastOf(\mathrm{p}) \subseteq P$ *(when* p *is a last-rep.).*

Definition 5.3 (Point Ranking). *Given a set of points* P*, a ranking* $\lessdot$ *on* P *is a total order over* P*.*

Definition 5.4 (Deterministic Scenario Specialization). *Given* $\mathcal{S}_1$ *and* $\mathcal{S}_2$*, two scenarios such that* $\mathcal{S}_2 <: \mathcal{S}_1$*, we say*

4. Note that the classic approach to checking triggered scenarios consists in building an automaton to recognize the complement of the conditional scenario language. The technique is not trivially imported to our setting since timed automata are not, in general, determinizable (see [8]).

5. An operator for expressing the earliest event of given type, similar to the "next" arrow, appears in the state-clock logics of [23].

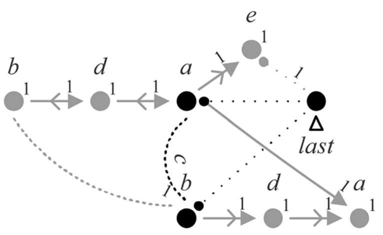

Fig. 9. DCS to illustrate some antiscenario construction cases.

that $\mathcal{S}_2$ *deterministically specializes* $\mathcal{S}_1$ *(denoted* $\mathcal{S}_2 <:: \mathcal{S}_1$*), iff there exists a ranking* $\lessdot$ *over* P_2 *such that:* $\forall \mathrm{p}_1 \in P_1, \mathrm{p}_2 \in P_2 \setminus P_1 \,.\, \mathrm{p}_1 \lessdot \mathrm{p}_2$ *and* $\lessdot\mathrm{p}_2 \hookrightarrow \mathrm{p}_2$ *(where* $\lessdot\mathrm{p}_2 = \{\mathrm{p} \in P_2 | \, \mathrm{p} \lessdot \mathrm{p}_2\}$*).*

In other words, concrete points in $\mathcal{S}_2$ *are reachable from points in* $\mathcal{S}_1$ *through* $\twoheadrightarrow$ *or* $\longleftrightarrow$ *arrows.*

Definition 5.5 (Deterministic CS (DCS)). *A conditional scenario* $\mathcal{C} = \langle \mathcal{S}_0, \{\mathcal{S}_i\}_{i=1\ldots k} \rangle$ *is a* Deterministic Conditional Scenario *iff for all* $i \in \{1..k\}$, $\mathcal{S}_i <:: \mathcal{S}_0$.

5.2 Verifying Deterministic Conditional Scenarios by Negative Scenario Generation

In this section, we show how to build a set of negative scenarios, a.k.a. antiscenarios, (VTS-kernel scenarios) that stand for all the ways in which a Deterministic Conditional Scenario may be violated. Antiscenarios are not a mere technical and hidden intermediate artifact, they are also valuable constructions for engineers. For instance, the generated antiscenarios to be checked could be identified as satisfiable in a simple inspection performed by the designer, thus avoiding the model-checking step altogether. In a few words, antiscenarios generated from conditional scenarios are a human-readable description of how things could go wrong (for instance, antiscenarios previously shown were generated from DCSs [15]).

The algorithm is explained by means of the authorization (Fig. 5) and the robot examples (Fig. 6), together with an artificial example (Fig. 9) added in order to cover most cases treated by the technique (see [15] for a general formalization).

For each consequent, the algorithm proceeds by building a set of negative scenarios that, if satisfied, would indicate there is a way to match the antecedent in a trace, and some of the points of a consequent are not found in the relative positions where they should be (unmatched points, e.g., see Fig. 10a, Fig. 11a, and Fig. 11c). To build these antiscenarios the algorithm uses the ranking order in order to enumerate the precedence path leading to the missing point.

Then, for each pair of points in the consequent, antiscenarios are built expressing the cases where although these points in the consequent are found following the precedence-path from the antecedent, a pairwise constraint that should hold according to the conditional scenario is actually violated. Pairwise constraints include precedence (e.g., Fig. 11b, Fig. 13b), forbidden events (e.g., Fig. 10c, Fig. 11e), temporal constraints (e.g., Fig. 10b, Fig. 11d), and inequalities (Fig. 13a). Indeed, consequent patterns are made up of a conjunction of these atomic pairwise constraints together with representative constraints (which may actually involve many points). Regarding the last case, *first* and *last* constructs allow the designer to express situations like a representative point belonging to the antecedent that should also be a representative of points contained in the consequent, points that are both last and first-representatives in the consequent, etc. That is, *first* and *last* constructs may establish aliasing constraints ("the first in this set is also the last in this other set"). In those cases, it is necessary to build scenarios violating the underlying aliasing constraint (e.g., Fig. 13c).

It can be seen that, if there were just one consequent scenario, these generic counterexamples would summarize all the ways the DCS could be violated. If there are n alternative consequents, counterexamples must be ob-

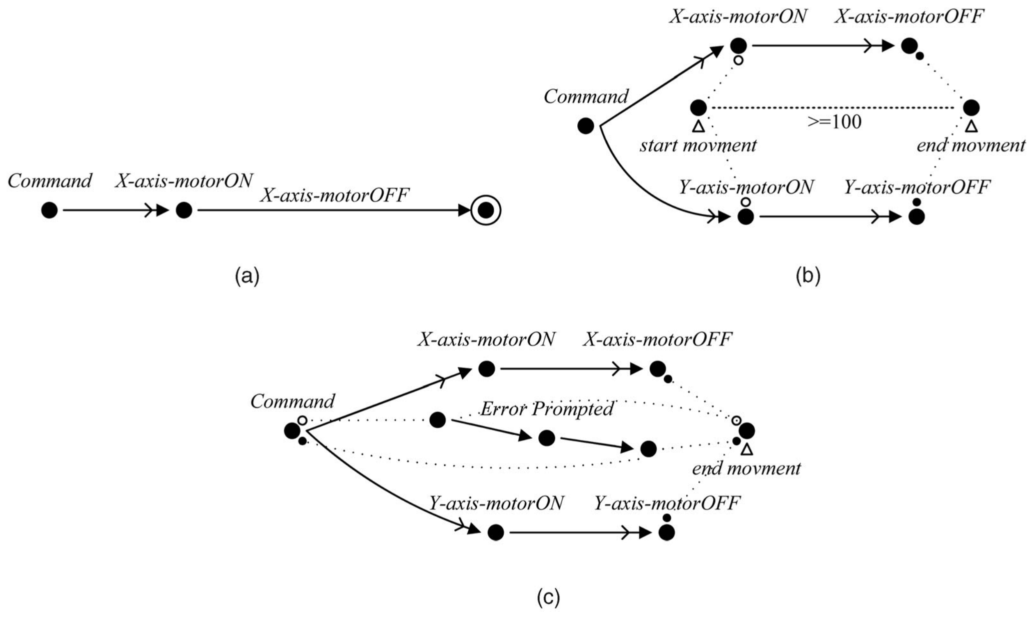

Fig. 10. Some intermediate antiscenarios for the robot example. (a) X-motor never turns off. (b) To slow. (c) Error prompted while moving.

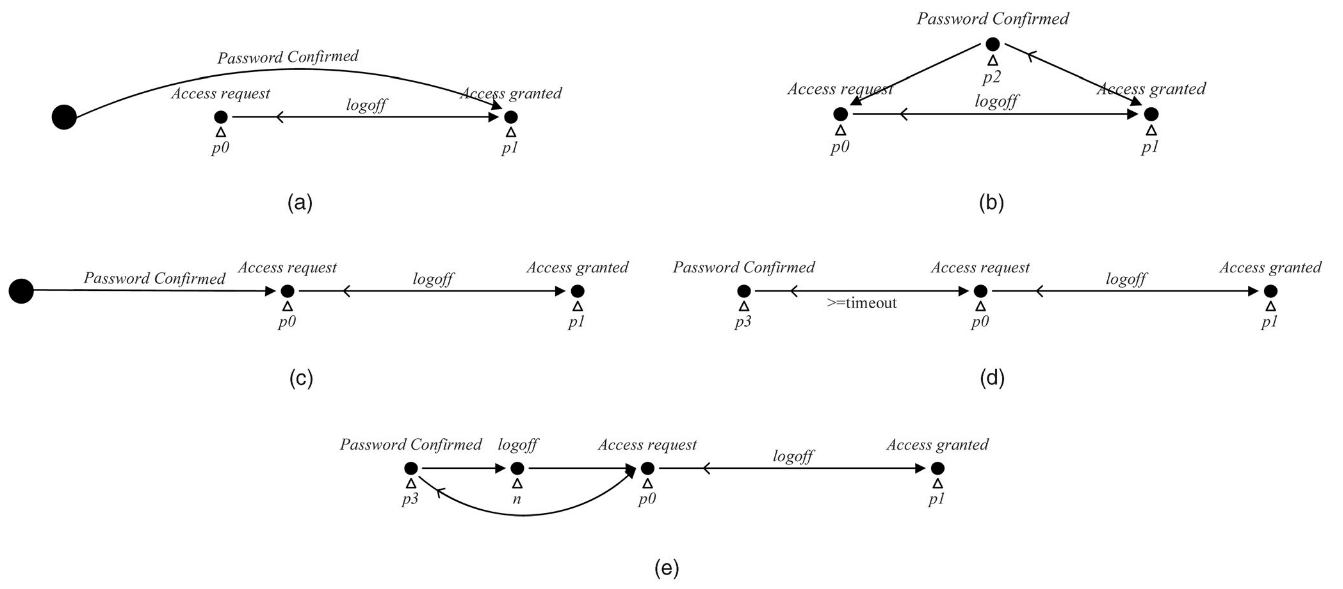

Fig. 11. Intermediate antiscenarios for the authorization example. (a) Passwd not confirmed (cons. 1). (b) Passwd confirmed before (cons. 1). (c) Passwd not confirmed before access reqst. (cons. 2). (d) Deadline violated since passwd. Confirmed (cons. 2). (e) Log off after last confirmation (cons. 2).

tained by producing all n-fusions (see next) that result from choosing an antiscenario for each of the n consequents. That is, consequent counterexamples are merged since we are negating a disjunction of conjunctions (see Fig. 12). As already mentioned, many of these scenario combinations are, in practice, inconsistent by construction and, thus, are not required to be submitted to the model checking phase (Fig. 12d is one of the three unfeasible final antiscenarios). The scenarios that need to be model checked are thus only the first three in Fig. 12.

Now, we introduce the operation of *fusion* to obtain a scenario by combining two or more scenarios. This operation is based on set union;[6] so, if two combined scenarios share points and arrows, these will appear once in the final construction. Thus, fusion is stronger than language intersection (the latter, achieved by disjoint union of patterns); it requires not only the existence of matchings for both patterns, but it states that they should coincide in the common subpattern (in our case, the antecedent).

Definition 5.6 (Fusion). *Given $\mathcal{S}_1$ and $\mathcal{S}_2$ two scenarios,[7] we define the fusion between $\mathcal{S}_1$ and $\mathcal{S}_2$ as the scenario*

$$\mathcal{S}_1 \oplus \mathcal{S}_2 = \langle \Sigma_3, P_3, \ell_3, \not\equiv_3, <_3, <_{F3}, <_{L3}, \gamma_3, \delta_3 \rangle,$$

where $\Sigma_3 = \Sigma_1 \cup \Sigma_2$; $P_3 = P_1 \cup P_2$; $\ell_3(\mathrm{p}) = \ell_1(\mathrm{p})$ *if* $\mathrm{p} \in P_1 \setminus P_2$, $\ell_3(\mathrm{p}) = \ell_2(\mathrm{p})$ *if* $\mathrm{p} \in P_2 \setminus P_1$, *or* $\ell_3(\mathrm{p}) = \ell_1(\mathrm{p}) \cap \ell_2(\mathrm{p})$ *if* $\mathrm{p} \in P_1 \cap P_2$; $\not\equiv_3 = (\not\equiv_1 \cup \not\equiv_2)$; $<_3 = (<_1 \cup <_2)$; $<_{F3} = (<_{F1} \cup <_{F2})$; $<_{L3} = (<_{L1} \cup <_{L2})$; $\gamma_3 = \gamma_1 \cup \gamma_2$; $\delta_3(\mathrm{p}) = \delta_1(\mathrm{p})$ *if* $\mathrm{p} \in P_1 \setminus P_2$, $\delta_3(\mathrm{p}) = \delta_2(\mathrm{p})$ *if* $\mathrm{p} \in P_2 \setminus P_1$, *or* $\delta_3(\mathrm{p}) = \delta_1(\mathrm{p}) \cap \delta_2(\mathrm{p})$ *if* $\mathrm{p} \in P_1 \cap P_2$.

It is easy to see that $\mathcal{S}_1 \oplus \mathcal{S}_2 <: \mathcal{S}_1$ and $\mathcal{S}_1 \oplus \mathcal{S}_2 <: \mathcal{S}_2$. The fusion $\mathcal{S}_1 \oplus \mathcal{S}_2$ is a *minimal specialization* of both $\mathcal{S}_1$ and $\mathcal{S}_2$, that is, for any other scenario $\mathcal{S}$ such that $\mathcal{S} <: \mathcal{S}_1$, and $\mathcal{S} <: \mathcal{S}_2$; then $\mathcal{S} <: \mathcal{S}_1 \oplus \mathcal{S}_2$.

As can be easily observed, the size of the tableau for a single antiscenario depends on the number of configurations, which ranges from linear in the number of points to exponential, depending on the degree of concurrency. Worst-case exponential complexity is not rare when dealing with linear-time property specification formalisms. Fortunately, patterns generated by humans tend to be of small size. The number of antiscenarios constructed for each consequent is, roughly speaking, quadratic on the number of events strictly belonging to the consequent. Note that this is a better situation than the traditional approach of complementing the tableau which recognizes the language of satisfying traces (exponential in the size of the tableau). Finally, the number of fusions is combinatorial in the quantity of antiscenarios for each consequent. The reason for this can be found in the analogy with translating CNF into DNF. Again, the number of consequents is usually small in practice (often amounting to one for regular operation and one for error or exceptional circumstances). Fortunately, it is typically the case that fusion of antiscenarios yields unfeasible scenarios, which might be discarded before reaching the model-checking step by some sort of safe and efficient static analysis.[8] Notice that antiscenario construction procedures involve several steps that can be parallelized and/or performed on-the-fly during model checking (even using SAT-solving methods). On the other hand, the true concurrent nature of patterns makes them particularly suitable for applying partial order techniques. The fact that antiscenarios share common subpatterns

6. A pushout construction is also suitable to define this operation.

7. In case $\not\equiv_1$ and $\not\equiv_2$ contain a given pair and its converse, the scenarios must be made compatible before performing the fusion by choosing a given direction and consequently inverting the pairs in the corresponding definition of γ and δ in order to maintain asymmetry. Also, if the scenarios share a point at which the labeling functions do not yield any label in common, their fusion yields an inconsistent scenario.

8. Note that, when tableau procedure is feed with a scenario featuring either cyclic precedence relation or inconsistencies between precedence and forbidden constraints, it yields observers where the acceptance configuration is unreachable.

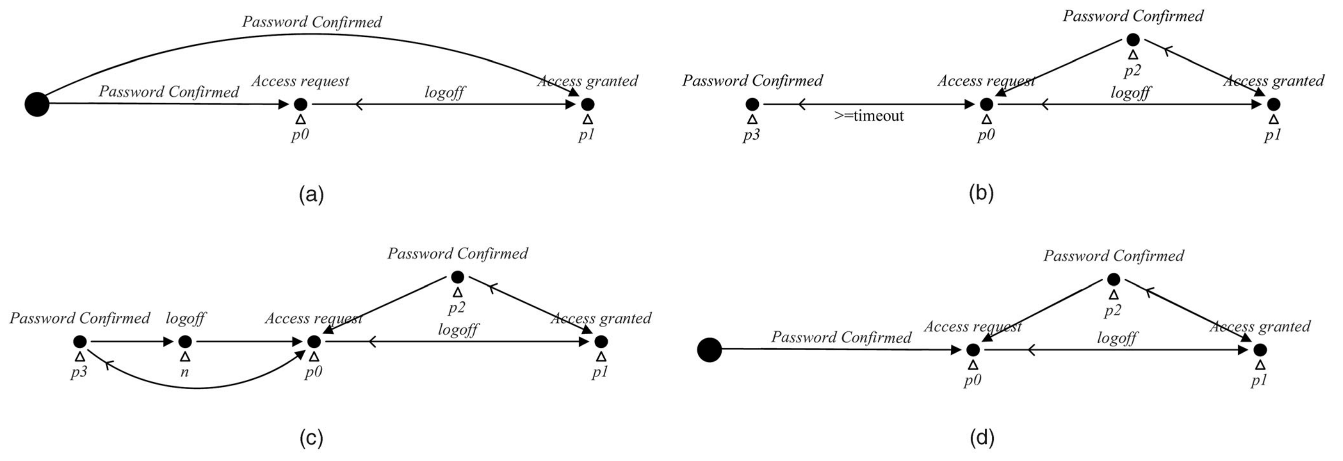

Fig. 12. Some final antiscenarios for the authorization example. (a) Final antiscenario ($\mathrm{a} \oplus \mathrm{c}$). (b) Final antiscenario ($\mathrm{b} \oplus \mathrm{d}$). (c) Final antiscenario ($\mathrm{b} \oplus \mathrm{e}$). (d) Final -unfeasible- antiscenario ($\mathrm{b} \oplus \mathrm{c}$).

makes them suitable for some data-reuse strategy during model checking.

6 Related Work

TimeEdit (see [4]) is a tool also meant to simplify the task of expressing never-claims (for the Spin or FeaVer model-checking tools). Something similar could be said about an event-based, real-time variant of GIL (Graphical Interval Logic) [24], [25]. Both TimeEdit and RT-GIL are based on timeline diagrams and do not feature partial ordering of events. TimeEdit does not support timing constraints. RT-GIL is a graphical interval logic that shares some common features with VTS. It provides search operators to locate end points of intervals (similar to *next* and *previous* in VTS). However, its *previous* operator cannot be applied freely as in VTS: Interval recognition always starts forward from a generic (or the first) point in the enclosing interval. Thus, for instance, it is not possible to express freshness, correlation constraints or to assert the existence of a past event in general, all of which are easily expressible in VTS. In order to mimic triggered scenarios in RT-GIL, logical implication could be used by stacking formulas. Nevertheless, the antecedent would need to be repeated in each consequent and it must be deterministic to ensure that the right interval is matched. GIL can constrain duration of intervals which are bounded by contiguous events. More precisely, in [25], it is remarked that RT-GIL users can construct and determinize Hybrid Automata oracles by hand when all intervals to be timed are specified using a bidirectional search operator. Unlike VTS, RT-GIL features the eventually and the henceforth modalities.

Timing Diagrams are a known notation in the context of hardware design. Regular Timing Diagrams (see [19]) (and extensions) are a good example of this class of visual notation suited for asynchronous systems. They are basically a set of independent waveforms on signals where events (change of values) can be causally constrained—and time-constrained by a number of ticks of a given clock—or defined as concurrent. In this setting, RTDs induce deterministic matchings and this implies sacrificing expressive power in favor of an efficient model checking.

Perhaps the most notable and widespread example of a visual formalism for scenario-based specifications are Message Sequence Charts and UML versions [1]. Unlike our approach, MSCs are not used to describe negative scenarios or conditional scenarios, but to describe some or all possible behaviors of a protocol in a systemic view.

TMCS, LCS, and Negative Scenarios (see [3], [18], [20], respectively) are three representative extensions of MSCs that feature some sort of conditional scenario mechanism.

TMCS extends MSCs with partial and conditional scenarios, together with process-algebra-like operators to compose these building blocks. It does not feature real-time constraints. Formal semantics is based on acceptance trees and a "must" preorder (for refinement) (see [20]). Conditional scenarios in this setting are primarily meant to refine nondeterministic base specifications to constrain and retain desirable sequences (i.e., unlike ours, their aim is not to describe verification goals). In conditional scenarios, each instance's action sequence is partitioned into a trigger prefix part and an action part. Events in the action part must happen if the trigger part is performed by the instance.

Uchitel et al. introduced, in [18], a language to describe nontimed negative scenarios based on MSCs to elicit requirements. They use a trigger-like precondition (after/until).

LCS (see [3]) includes trigger-like notations as charts and precharts. They seek to distinguish between mandatory (universal) and provisional (existential) behavior. Activation charts are used as a means to express when the body of the chart will occur during a system run. Semantics was formally given originally by deriving a Timed Büchi Automata (see [26]) and, recently, a claimed efficient embedding into CTL^* was given in [27].[9]

Although we share with all these approaches the idea of using partial orders to describe scenarios, our work differs in several aspects. On the one hand, our language is meant to express properties to be checked against a model or an implementation under analysis; we neither focus on creating an executable modeling language for different

9. Trace semantics in [27] is actually given resorting to a notion similar to the configurations we use for tableaux construction.

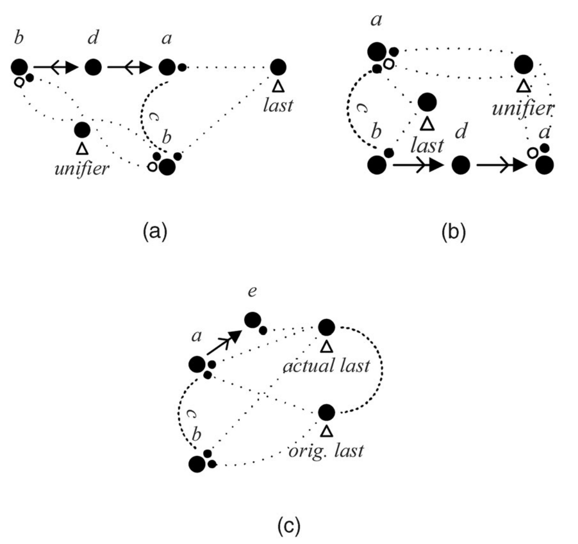

Fig. 13. Some intermediate antiscenarios for the artificial example. (a) Inequality violated. (b) Precedence violated (matching some event). (c) Alias violated.

phases of the development process nor on eliciting requirements. On the other hand, we are neither constrained to describe message interchange nor to define the instances that perform events. Moreover, event visibility is treated quite differently in our case: Two contiguous events in a scenario do not need to match contiguous events of the same kind in the run except when explicitly noted. The first/last feature of VTS is not present in the cited approaches. One notable difference is that, in VTS conditional scenarios, the antecedent pattern is not required to predicate on a prefix of the behavior. Our consequents can refer to events occurring before the trigger or interleaved with its events. Additionally, our notation and tools declaratively deal with real-time constraints in a dense-time domain without using timers, as is the case in [3]. Finally, our notation can be used to express some classes of liveness constraints, such as a pattern stating that a given stimulus is never answered. Our semantic definitions are given in a declarative way based on pattern matching; in our experience, this approach is more likely to be understood by a practitioner than encodings in temporal logics, automata, or process algebraic approaches. In fact, our tableau procedure is presented, also declaratively, just in order to show the algorithmic solution to the checking problem.

Firley et al. mix UML sequence diagrams and model checking in [28]. Diagrams are translated into observers in the Uppaal formalism and Uppaal models are instrumented to be composed with the observer. Again, no semantics is given for the checking problem beyond the algorithmic translation itself and a set of proof obligations in branching logics. They follow the "contiguous semantics": the sequence specifies exactly the message exchange that may occur—thus featuring a sort of loop construct—except for the starting event in the optional interpretation. The presented construction only supports totally ordered sets of events. UML sequence diagrams do not support the concept of instants, conditional scenarios, or liveness constraints.

Similarly to our approach, [29] introduces a notation for template charts, i.e., charts with only partial information about participating events and their interrelated order. However, their goal is to search portions of MSCs that match those templates. Besides, the template syntax is that of MSCs, so the explained differences with VTS apply.

A syntactical notion of refinement for Timed MSCs is presented in [30]. Since the authors pursue a development process based on refining charts, these notions distinguish whether the timing constraint is an assumption about the environment or a bound on process performance. Refinement is not the goal of our specialization concept.

In [5], VTS is also compared with different works on monitoring and debugging of distributed systems, event correlation detection, and runtime verification.

7 Conclusions

Though other notations were proposed for visually describing negative scenarios (e.g., [3], [4], [18], [28]), the VTS-kernel event-pattern notation we show here combines existing and novel features in a way that makes it a powerful tool for the verification of real-time applications: a simple formal syntax based on partial orders, an underlying declarative and compact semantics based on the intuitive idea of matching over traces, and an operational semantics based on Timed Automata. Besides, events are neither restricted to be communication-events nor to be consecutive[10] and the negation of event occurrence in intervals permits identifying next or previous events of a given sort. The concept of representative events and instants are novel features, too, as are the begin and end points (especially suited for negating progress). Moreover, VTS deals with real-time constraints in a dense-time domain without using timers.

Notice that VTS-kernel can also be used to express positive scenarios that represent admissible system execution as MSCs are usually employed to convey use cases during a requirement engineering phase. This function is less likely to serve as input for automatic software quality assurance tools because the mere statement that a certain interaction is allowed to happen does not provide enough information to validate an application or a model. However, this task can greatly benefit from a notation that possesses a clear formal semantics and is able to succinctly express typical real-time requirements.

Additionally, we introduce conditional scenarios (CSs), a unique notation that enables expressing alternatives that may be nontrivially linked to the trigger part of the scenario. That expressive power is a consequence of viewing semantics of scenarios from a novel pattern-matching perspective. Beyond description of verification goals, CSs offer an interesting and formally supported approach for the partial specification of system behavior, partiality being a key feature for industrial adoption.

10. This feature explains why we do not need to provide an iteration operator, which is often used in other approaches to express that a visible event occurs an unbounded number of times in an interval.

From a theoretical perspective, CSs constitute a scenario notation based on simple features which may help to understand the limits of decidability for these sort of formalisms. Indeed, we prove that the satisfaction and the model-checking problems are highly undecidable over dense-time domains. Moreover, we identify a large and practical fragment (the DCSs) which may be model checked by verifying a set of negative VTS-kernel scenarios. This fragment is such that consequents have at most one possible matching extension given a matching on the antecedent. To our knowledge, the translation into antiscenarios is a novel and useful approach to verification since, as a byproduct, generic and human-readable potential counterexample-patterns are generated. Besides, that translation shows how some mentioned features, such as first, last, begin, and end points, play key roles in building counterexamples of conditional scenarios, thus further justifying language design decisions in VTS-kernel.

8 Future Work

Our research group agenda includes the addition of syntactic sugar to the language, such as providing modularity, parametrization, and extended event-name matching capabilities, as well as enhancements of expressive power without sacrificing decidability, feasibility, and ease of use. As future work, we also include topics such as: static analysis of pattern feasibility, tableau determinization for certain classes of patterns (e.g., those with a bounded number of overlapping matchings), construction of tableaux for the general case of conditional scenarios when there are no timing constraints involved, avoiding the conversion from CNF to DNF, and leveraging the partial-order nature of scenarios during model checking.

We are also exploring the definition of composed events (standing for a whole scenario). Allowing recursion in this definitions will enhance the expressive power of the notation since composed events will permit referring to the characteristics of an unbounded history of events (e.g., "If an odd number of requests is received, then the response should be issued within 10 t.u."). In relation to these extensions, we plan to add the definition of state (à la "fluents," see [31]) and its use to express properties on intervals.

In order to provide more evidence supporting the practical relevance of the approach, we plan to study how some patterns (e.g., [2]) translate into scenarios to help the construction of requirements (some examples can be found in [32]). We also intend to compare the expressive power of VTS against finite-trace linear-time logics (e.g., [33]), identifying equally expressive fragments.

On the other hand, we are experimenting with the use of scenario notations to finetune model checking by combining them with model-slicing technology like the *ObsSlice* tool [17], [34]. This novel application takes full advantage of the richness in the way antecedents and consequents can be combined in our notation.

We also conjecture that VTS scenarios are amenable to serving as verification objectives in a runtime verification tool or as a sort of test purposes to guide the extraction of test cases from formal models.

Some theoretical questions will be addressed in the near future: the connection between DCSs and determinizable classes of TA (e.g., [35]), whether relaxing punctuality would also turn CSs satisfaction decidable, as in the case of MITL (see [22]), on which fragment of timed automata the model-checking problem is actually decidable, and whether CS fragments could be embedded into some logic that does not use bounded temporal operators like TPTL (which is undecidable for dense-time models, see [9]) or logics that incorporate automata in their descriptions (e.g., [36], [10]).

Acknowledgments

The authors would like to acknowledge Daniel Monteverde and the anonymous reviewers for their insightful comments. This research was partially supported by ANPCyT project BID-PICT 11738, UBACyT X020, a Microsoft Research Embedded Systems Innovation Excellence Award, and the IBM Eclipse Innovation Program.

References

[1] ITU-T, "Recommendation Z. 120. Message Sequence Charts," Technical Report Z-120, Int'l Telecomm. Union—Standardization Sector, Genève, 2000.

[2] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett, "Patterns in Property Specifications for Finite-State Verification," *Proc. 21st ACM/IEEE Int'l Conf. Software Eng. (ICSE '99),* pp. 411-420, 1999.

[3] D. Harel and R. Marelly, "Playing with Time: On the Specification and Execution of Time-Enriched lscs," *Proc. 10th IEEE/ACM Int'l Symp. MASCOTS '02,* pp. 193-202, 2002.

[4] M.H. Smith, G.J. Holzmann, and K. Etessami, "Events and Constraints: A Graphical Editor for Capturing Logic Requirements of Programs," *Proc. Fifth IEEE Int'l Symp. Requirements Eng. (RE '01),* pp. 14-22, 2001.

[5] A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero, "Visual Timed Event Scenarios," *Proc. 26th ACM/IEEE Int'l Conf. Software Eng. (ICSE '04),* 2004.

[6] R. Alur, C. Courcoubetis, and D.L. Dill, "Model-Checking in Dense Real-Time," *Information and Computation,* vol. 104, no. 1, pp. 2-34, 1993.

[7] B. Alpern and F. Schneider, "Verifying Temporal Properties without Temporal Logic," *ACM Trans. Programming Languages and Systems,* vol. 11, no. 1, pp. 147-167, 1989.

[8] R. Alur and D.L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science,* vol. 126, no. 2, pp. 183-235, 1994.

[9] R. Alur and T.A. Henzinger, "A Really Temporal Logic," *Proc. IEEE Symp. Foundations of Computer Science,* pp. 164-169, 1989.

[10] T. Wilke, "Specifying Timed State Sequences in Powerful Decidable Logics and Timed Automata," *Formal Techniques in Real-Time and Fault-Tolerant Systems,* H. Langmaack, W.-P. de Roever, and J. Vytopil, eds., pp. 694-715, Lbeck: Springer, 1994.

[11] A. Alfonso, V. Braberman, D. Garbervetsky, N. Kicillof, A. Olivero, and F. Schapachnik, "Vintime: Combining High-Level Finesse with Low-Level Muscle to Verify Real-Time Systems," *Proc. PriSE '04, First Conf. PRInciples of Software Eng.,* 2004.

[12] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL—A Tool Suite for Automatic Verification of Real-Time Systems," *Proc. Int'l Conf. Hybrid Systems,* pp. 232-243, 1995.

[13] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A Model-Checking Tool for Real-Time Systems," *Proc. 10th Int'l Conf. CAV '98,* pp. 546-550, 1998.

[14] N.F. Maxemchuk and D.H. Shur, "An Internet Multicast System for the Stock Market," *ACM Trans. Computer Systems,* vol. 19, no. 3, pp. 384-412, 2001.

[15] V. Braberman, N. Kicillof, and A. Olivero, "Visual Timed Event Scenarios," Technical Report 05-011, Computer Science Dept. School of Science, Univ. of Buenos Aires, 2005.

[16] V. Braberman and M. Felder, "Verification of Real-Time Designs: Combining Scheduling Theory with Automatic Formal Verification," *Proc. Seventh ACM/SIGSOFT Int'l Conf. ESEC/FSE '99,* pp. 494-510, 1999.
[17] V. Braberman, D. Garbervetsky, and A. Olivero, "Improving the Verification of Timed Systems Using Influence Information," *Proc. Eighth Int'l Conf. TACAS '02,* pp. 21-36, 2002.
[18] S. Uchitel, J. Kramer, and J. Magee, "Negative Scenarios for Implied Scenario Elicitation," *Proc. 10th ACM/SIGSOFT Int'l Conf. FSE '02,* pp. 109-118, 2002.
[19] N. Amla, E.A. Emerson, and K.S. Namjoshi, "Efficient Decompositional Model Checking for Regular Timing Diagrams," *Proc. Int'l Conf. Correct Hardware Design and Verification Methods,* pp. 67-81, 1999.
[20] B. Sengupta and R. Cleaveland, "Triggered Message Sequence Charts," *Proc. SIGSOFT FSE,* pp. 167-176, 2002.
[21] R. Alur and P. Madhusudan, "Decision Problems for Timed Automata: A Survey," *Proc. SFM,* M. Bernardo and F. Corradini, eds., pp. 1-24, 2004.
[22] R. Alur, T. Feder, and T.A. Henzinger, "The Benefits of Relaxing Punctuality," *Proc. Symp. Principles of Distributed Computing,* pp. 139-152, 1991.
[23] J.-F. Raskin and P.-Y. Schobbens, "State Clock Logic: A Decidable Real-Time Logic," *Proc. HART,* O. Maler, ed., pp. 33-47, 1997.
[24] L.E. Moser, Y.S. Ramakrishna, G. Kutty, P.M. Melliar-Smith, and L.K. Dillon, "A Graphical Environment for the Design of Concurrent Real-Time Systems," *ACM Trans. Software Eng. and Methodology,* vol. 6, no. 1, 1997.
[25] G.S. Avrunin, J.C. Corbett, and L.K. Dillon, "Analyzing Partially-Implemented Real-Time Systems," *Proc. 18th ACM/IEEE Int'l Conf. Software Eng. (ICSE '97),* pp. 228-238, 1997.
[26] J. Klose and H. Wittke, "An Automata Based Interpretation of Live Sequence Charts," *Proc. Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS),* T. Margaria and W. Yi, eds., pp. 512-527, 2001.
[27] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps, "Temporal Logic for Scenario-Based Specifications," *Proc. 11th Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05),* 2005.
[28] T. Firley, M. Huhn, K. Diethers, T. Gehrke, and U. Goltz, "Timed Sequence Diagrams and Tool-Based Analysis—A Case Study," *Proc. Second Int'l Conf. UML '99,* pp. 645-660, Oct. 1999.
[29] V. Levin and D. Peled, "Verification of Message Sequence Charts via Template Matching," *Proc. TAPSOFT,* M. Bidoit and M. Dauchet, eds., pp. 652-666, 1997.
[30] T. Zheng, F. Khendek, and B. Parreaux, "Refining Timed mscs," *Proc. SDL Forum,* R. Reed and J. Reed, eds., pp. 234-250, 2003.
[31] D. Giannakopoulou and J. Magee, "Fluent Model Checking for Event-Based Systems," *Proc. ACM/SIGSOFT Int'l Conf. ESEC/FSE 2003,* pp. 257-266, Sept. 2003.
[32] A. Alfonso, "Un Lenguaje Visual para la Especificación y Verificación Automática de Requerimientos de Tiempo Real Complejos," master's thesis, FCEyN. Univ. de Buenos Aires, 2003.
[33] K. Havelund and G. Rosu, "Synthesizing Monitors for Safety Properties," *Proc. Eighth Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02),* pp. 342-356, 2002.
[34] V. Braberman, D. Garbervetsky, and A. Olivero, "Obsslice: A Timed Automata Slicer Based on Observers," *Proc. 16th Int'l Conf. Computer-Aided Verification (CAV '04),* 2004.
[35] R. Alur, L. Fix, and T.A. Henzinger, "Event-Clock Automata: A Determinizable Class of Timed Automata," *Theoretical Computer Science,* vol. 211, nos. 1-2, pp. 253-273, 1999.
[36] A. Bouajjani, Y. Lakhnech, and S. Yovine, "Model-Checking for Extended Timed Temporal Logics," *Proc. Int'l Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT),* B. Jonsson and J. Parrow, eds., pp. 306-326, 1996.

Victor Braberman received the PhD degree in computer science from the School of Sciences, University of Buenos Aires, Argentina, in 2000, where he currently holds a professor position. He also holds a researcher position at CONICET, Argentina. His research is primarily concerned with providing theory, techniques, and algorithms for the automatic analysis and synthesis of software-intensive systems. He has headed the development of novel proof-of-concept tools based on model checking, static, and dynamic analysis for real-time, distributed, and embedded software. He also has varied industrial experience as an independent consultant on software engineering topics.

Nicolas Kicillof received the MS degree in computer science from the School of Sciences, University of Buenos Aires, Argentina, where he is currently a professor and PhD candidate. His research topics are formal notations for state and interaction-based requirement description, tools for automatic conformance checking and model-based testing, and aspect-oriented development. He has recently completed a research internship with the Foundations of Software Engineering Group, Microsoft Research Redmond Lab. He has varied industrial experience as an independent consultant on software engineering topics.

Alfredo Olivero received the PhD degree from the Institut National Polytechnique de Grenoble, France. He is a full-time research associate at the Universidad Argentina de la Empresa. His research interests include requirements engineering, analysis techniques, formal methods, and verification tools. He is one of the authors of Kronos, a model checker for verifying complex real-time systems. He has headed the development of other tools for the analysis of real-time systems. He has varied industrial experience as an independent consultant on software engineering topics. He was the general chair of JAIIO 2003 (Argentine Conference on Computer Science and Operational Research).