Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Monitoring Properties of Analog and Mixed-Signal Circuits

Oded Maler[1], **Dejan Ničković**[2]

[1] CNRS-VERIMAG, University of Grenoble, France
[2] AIT Austrian Institute of Technology, Vienna, Austria

Received: date / Revised version: date

Abstract. In this paper, we present a comprehensive overview of the property-based monitoring framework for analog and mixed-signal systems. Our monitoring approach is centered around the Signal Temporal Logic (STL) specification language, and is implemented in a stand-alone monitoring tool (AMT). We apply this property-based methodology to two industrial case studies and briefly present some recent extensions of STL that were motivated by practical needs of analog designers.

1 Introduction

Verification of digital hardware has achieved in the past years a high level of automation, thanks to a mature tool and methodology support from the *electronic design automation* (EDA) industry. Formal verification techniques, such as *model* and *equivalence checking* or *theorem proving*, were integrated in numerous EDA toolkits. Despite this relative success of formal verification, lighter validation techniques based on simulation/testing remain popular amongst engineers, thanks to their simplicity. In this setting the system is seen as a "black-box" that generates a finite set of behaviors and a *monitor* checks each individual behavior against the specification for correctness. Although incomplete, monitoring is effectively used to catch faults in the system, without guaranteeing its full correctness.

The ongoing tendency of speeding-up the production of new devices of higher performance and reliability, while lowering their power consumption, results in the design process becoming more vulnerable to faults. Another effect of this trend is the integration of digital, analog and mixed-signal (AMS) components on the same chip which adds another level of complexity in the design process. Validation of AMS designs still relies mainly on simulation-based testing, combined with a number of analysis techniques, such as frequency-domain analysis, statistical measures, parameter extraction, eye diagrams etc. Unlike digital verification, the tool support for AMS validation is often specific to the class of properties considered and includes wave calculators, measuring commands as well as manually written scripts. These solutions are mostly ad-hoc and support minimal automation resulting in a time-consuming process that requires considerable (often non-reusable) user effort. The additional issue in AMS validation is the time required for the simulation of complex designs. A typical simulation of several nanoseconds of real-time transient behavior of a complex AMS circuit often takes hours or even days of simulation time.

In this paper, we provide an extensive overview of a framework for validation of AMS designs, that we developed with the aim to export some ingredients of the well-established verification methodology from digital to AMS systems. We adopt a property-based monitoring approach, in which the system behavior is simulated and checked with respect to a high-level specification expressed in a formal language. Our framework is centered around Signal Temporal Logic (STL) that was first introduced in [20]. STL extends the real-time temporal logic MTL [19] with numerical predicates, and thus allows expressing mixed-signal properties. In particular, STL naturally supports the class of mixed-signal specifications that involve describing temporal patterns between "events" that happen in analog and digital transient simulation traces.

Example 1. An example of a property that can be expressed in STL is a mixed signal stabilization property that has the following requirements:

- The absolute value of a continuous signal x is always less than 6
- When the (Boolean) trigger rises, within 600 time units $|x|$ has to drop below 1 and stay like that for at least 300 time units

Send offprint requests to:

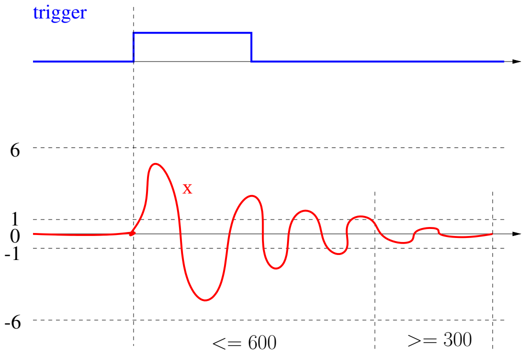


Fig. 1. Mixed signal stabilization property

This property is illustrated in Figure 1 and expressed in STL as:

$$\Box(|x| < 6 \wedge (\uparrow \mathit{trigger} \rightarrow \Diamond_{[0,600]} \Box_{[0,300]}(|x| < 1)))$$

Our property-based monitoring framework is supported by the tool AMT [24], that implements the algorithms presented in [20,22]. We have applied our validation approach to several case studies provided by industrial partners, in particular a FLASH memory design [24] and a DDR2 memory interface [16]. Apart from an extensive summary of our previous results on property-based monitoring of AMS systems, this paper also presents several features that were previously unpublished: (1) the monitoring algorithms for STL with future and past operators, strict temporal (until and since) operators and events; and (2) a completed version of the FLASH case study, that was only partially presented in [24].

Related work: Property-based verification of AMS systems was applied in [25], where Property Specification Language (PSL) was used as the specification language. However, that work is restricted to discrete-time analog systems, and does not treat continuous time. Simulation-based probabilistic model-checking of temporal and frequency domain properties of AMS systems is studied in [5]. In [13,12,14], the authors consider multi-valued semantics for MTL and provide robust interpretation of the logic over timed sequences of states. Real-time monitoring of the timed LTL (TLTL) logic is studied in [4]. TLTL specifications are interpreted over finite traces with the 3-valued semantics. The extensions of temporal logics that deal with richer properties were also considered in monitoring tools such as LOLA [6].

The question of extending the usual approximation and sampling theory of continuous signals and systems to those encompassing discontinuities was addressed in [17], where a topological framework derived from the family of Skorokhod distances was used to handle this type of systems in a uniform matter. Recently, a translation from hybrid data-flow models to hybrid automata was proposed in [26], where the special attention was given to approximations due to zero-crossings.

2 Signal Temporal Logic

Temporal logics MTL [19] and MITL [1] are popular real-time extensions of LTL. The principal modality of MTL and MITL is the *timed until* $\mathcal{U}_I$ where I is some interval (non-punctual in the case of MITL) with integer or rational endpoints. A formula $p\mathcal{U}_{[a,b]}q$ is satisfied by a signal at any time instant t that admits q at some $t' \in [t+a, t+b]$, and where p holds continuously from t to t'. The original version of MTL and MITL contained only future temporal operators, although an investigation of past and future versions of MITL was carried out in [2].

Signal temporal logic STL extends MTL and MITL with numerical predicates that allow to specify analog and mixed-signal properties. STL formulas are interpreted over Boolean and continuous signals.

2.1 Signals

A signal over a domain D is a function $w : \mathcal{T} \rightarrow D$ where $\mathcal{T}$ is the time domain, which is either the set $\mathcal{R}_{\geq 0}$ of non-negative real numbers in the case of infinite signals or an interval $[0, r)$ if the signal is of finite length. We focus on the finite length signals where D is the set $\mathbb{B}^n \times \mathbb{R}^m$ of vectors over n Boolean and m real valued variables. We abuse the notation, and denote by u the projection of w to its Boolean, and by ξ the projection of w to its real valued components.

Each finite Boolean signal u can be further decomposed into a *punctual signal*, defined only at 0 and denoted by $\dot{u}$, and an *open signal segment* defined over the interval $(0, r)$. We will denote such signal segments as $(u)_r$. The concatenation of a punctual signal and an open signal segment is a finite signal, and is simply their union. Concatenation of two finite Boolean signals u_1 and u_2 defined over $[0, r_1)$ and $[0, r_2)$, respectively, is the finite signal $u = u_1 \cdot u_2$, defined over $[0, r_1 + r_2)$ as

$$u[t] = \begin{cases} u_1[t] & \text{if } t < r_1 \\ u_2[t - r_1] & \text{otherwise} \end{cases}$$

A point-segment partition of $\mathcal{T}$ is an alternating sequence of adjacent points and open intervals of the form

$$J = \{t_0\}, (t_0, t_1), \{t_1\}, (t_1, t_2), \{t_2\}, \ldots$$

with $t_0 = 0$ and $t_i < t_{i+1}$. With respect to such a given time partition, a Boolean signal u can be written as an alternating concatenation of points and open segments:

$$u = \dot{u}_0 \cdot (u_0)_{r_0} \cdot \dot{u}_1 \cdot (u_1)_{r_1} \cdots$$

where $\dot{u}_i$ is the value of the signal at t_i and $(u_i)_{r_i}$ is the segment which corresponds to the restriction of u to the interval (t_i, t_{i+1}) whose duration is $r_i = t_{i+1} - t_i$. An interval splitting is the act of partitioning a segment (t_i, t_{i+1}) into $(t_i, t'), \{t'\}, (t', t_{i+1})$. We say that a time partition J' is a refinement of J, denoted by $J' \prec J$ if it can be obtained from J by one or more interval splittings. A time partition is *compatible* with a signal u if the value of u is uniform in each

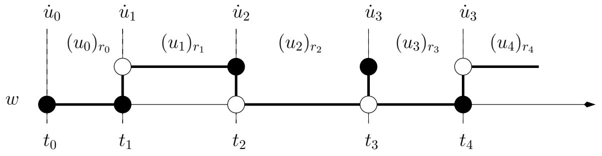


Fig. 2. The coarsest partition of a well-behaving signal w

open interval, that is, the segment $(u_i)_{r_i}$ is constant for every i.

The left and right limit of a signal u at point t are defined, as

$$u[\rightarrow t] = \lim_{r \rightarrow t^+} u[r] \quad \text{and} \quad u[t \leftarrow] = \lim_{r \rightarrow t^-} u[r],$$

respectively. We say that a time point t is *left-singular* with respect to u if $u[\rightarrow t] \neq u[t]$ and that it is *right-singular* if $u[t] \neq u[t \leftarrow]$. A point is singular if it is either left- or right-singular (or both). A point which is not singular is called *stationary*. Let us denote the sequence of singular points in u by $\mathcal{J}(u)$. A signal is *well-behaving* if the sequence $\mathcal{J}(u) = t_0, t_1, \ldots$ is finite or countable and diverging. We consider only finite, hence well-behaving signals.

Every well-behaving signal u with $\mathcal{J}(u) = t_0, t_1, \ldots$ induces a canonical time partition

$$J_u = \{t_0\}, (t_0, t_1), \{t_1\}, (t_1, t_2), \{t_2\}, \ldots,$$

which is the coarsest time partition compatible with u (see Figure 2 for an example).

Boolean signals can be combined and separated using the standard pairing and projection operators. Let $u_p : \mathcal{T} \rightarrow \mathbb{B}$, $u_q : \mathcal{T} \rightarrow \mathbb{B}$ and $u_{pq} : \mathcal{T} \rightarrow \mathbb{B}^2$ be signals. The pairing function is defined as

$$u_p \mid\mid u_q = u_{pq} \text{ if } \forall t \in \mathcal{T} \;\; u_{pq}[t] = (u_p[t], u_q[t])$$

and its inverse operation, projection as:

$$u_p = u_{pq}|_p \quad u_q = u_{pq}|_q$$

Note that the number of singular points in u_{pq} is at most the *sum* of the number of singular points in u_p and u_q, and that the number of singular points in $\textsc{op}(u_p, u_q)$, for a pointwise extension of a Boolean operator $\textsc{op}$ is at most that of u_{pq}. Hence well-behaving signals are closed under pairing, projection and Boolean operations. The Minkowski sum $A \oplus B$ of two sets is the set $\{a + b : a \in A, b \in B\}$.

We use the notation $[a, b] \ominus [c, d] = [a - c, b - d] \cap \mathcal{T}$ to denote the Minkowski difference with saturation at zero and $t \oplus [a, b]$ as a shorthand for $\{t\} \oplus [a, b]$.

When considering signals of finite length $|u| = r$, we use the notation $u[t] = \bot$ when $t \geq |r|$. The restriction of a signal of length d is defined as

$$u' = \langle u \rangle_d \text{ iff } u'[t] = \begin{cases} u[t] & \text{if } t < d \\ \bot & \text{otherwise} \end{cases}$$

When we apply operations on signals of different lengths, we use the convention

$$\textsc{op}(v, \bot) = \textsc{op}(\bot, v) = \bot$$

which guarantees that if $u = \textsc{op}(u_1, u_2)$ then we have $|u| = \min(|u_1|, |u_2|)$.

The *d-suffix* of a signal u is the signal $u' = d \backslash u$ obtained from u by removing the prefix $\langle u \rangle_d$ from u, that is,

$$u'[t] = u[t + d] \text{ for every } t \in [0, |u| - d).$$

2.2 Syntax, Semantics and Rewriting Rules

We consider the STL logic with both *future* and *past* operators. Let $X = \{x_1, \ldots, x_m\}$ be the set of real valued variables and $P = \{p_1, \ldots, p_n\}$ the set of STLpropositions. A *Boolean constraint* over X is a predicate of the form $x \circ c$, where $x \in X$, $\circ \in \{<, \leq, =, \geq, >\}$ and $c \in \mathbb{Q}$. The syntax of an STL formula φ over X and P is defined by the grammar

$$\begin{array}{l} \alpha := p \mid x \circ c \\ \varphi := \alpha \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathcal{U}_I \varphi_2 \mid \varphi \mathcal{S}_I \varphi_2 \end{array}$$

where $p \in P$, $x \in X$, $c \in \mathbb{Q}$ is a constant and I is an interval of the form $[a, b]$, $[a, b)$, $(a, b]$, (a, b), $[a, \infty)$ or (a, ∞) where $0 \leq a \leq b$ are rational numbers. As in LTL, basic STL operators can be used to derive other standard Boolean and temporal operators, in particular the time-constrained *eventually* ($\diamondsuit$), *once* ($\diamondminus$), *always* ($\Box$), and *historically* ($\boxminus$) operators:

$$\begin{array}{ll} \diamondsuit_I \, \varphi = \mathit{true} \; \mathcal{U}_I \; \varphi & \diamondminus_I \, \varphi = \mathit{true} \; \mathcal{S}_I \; \varphi \\ \Box_I \, \varphi = \neg \diamondsuit_I \neg\varphi & \boxminus_I \, \varphi = \neg \diamondminus_I \neg\varphi \end{array}$$

The semantics of an STL formula φ with respect to an n-dimensional signal w is described via the satisfiability relation $(w, t) \models \varphi$, indicating that the signal w satisfies φ at time t, according to the following recursive definition, where $\mathcal{T}$ is the time domain.

$$\begin{array}{lll} (w,t) \models x \circ c & \leftrightarrow & w|_x[t] \circ c \\ (w,t) \models p & \leftrightarrow & p[t] = 1 \\ (w,t) \models \neg\varphi & \leftrightarrow & (w,t) \not\models \varphi \\ (w,t) \models \varphi_1 \vee \varphi_2 & \leftrightarrow & (w,t) \models \varphi_1 \text{ or } (w,t) \models \varphi_2 \\ (w,t) \models \varphi_1 \; \mathcal{U}_I \; \varphi_2 & \leftrightarrow & \exists\, t' \in (t \oplus I) \cap \mathcal{T} \; (w,t') \models \varphi_2 \text{ and} \\ & & \forall\, t'' \in (t, t') \; (w, t'') \models \varphi_1 \\ (w,t) \models \varphi_1 \; \mathcal{S}_I \; \varphi_2 & \leftrightarrow & \exists\, t' \in (t \ominus I) \cap \mathcal{T} \; (w,t') \models \varphi_2 \text{ and} \\ & & \forall\, t'' \in (t', t) \; (w, t'') \models \varphi_1 \end{array} \tag{1}$$

A formula φ is satisfied by w if $(w, 0) \models \varphi$. The satisfaction relation can be viewed as a characteristic function χ^φ mapping signals over $\mathbb{B}^n \times \mathbb{R}^m$ into Boolean signals, such that $u = \chi^\varphi(w)$ meaning that for every $t \geq 0$, $u[t] = 1$ if and only if $(w, t) \models \varphi$. The definitions of $\mathcal{U}_I$ and $\mathcal{S}_I$ are *strict* as originally proposed in [1], meaning that φ_1 need not hold at t and neither at the moment t' when φ_2 becomes true. Note also that when I is left-open with a bound a, the truth of φ_2 at $t + a$ does not count for satisfaction. Note that we define *strong* finitary semantics for $\varphi_1 \mathcal{U} \varphi_2$, whose satisfaction requires φ_2 to be satisfied before the end of the finite length signal. A more detailed discussion about interpretation of temporal logic over finite traces can be found in [22].

Untimed *strict* temporal operators $\mathcal{U}$ and $\mathcal{S}$ can be expressed using the timed operators where the interval is $(0, \infty)$ Similarly, we can define *non-strict* untimed temporal operators $\underline{\mathcal{U}}$ and $\underline{\mathcal{S}}$ (which are the commonly-used interpretations of $\mathcal{U}$ and $\mathcal{S}$ in LTL) in terms of the strict ones.

$$\begin{array}{l}\varphi_1 \; \mathcal{U} \; \varphi_2 = \varphi_1 \; \mathcal{U}_{(0,\infty)} \; \varphi_2 \\ \varphi_1 \; \mathcal{S} \; \varphi_2 = \varphi_1 \; \mathcal{S}_{(0,\infty)} \; \varphi_2 \\ \varphi_1 \; \underline{\mathcal{U}} \; \varphi_2 = \varphi_2 \vee (\varphi_1 \wedge (\varphi_1 \; \mathcal{U} \; \varphi_2)) \\ \varphi_1 \; \underline{\mathcal{S}} \; \varphi_2 = \varphi_2 \vee (\varphi_1 \wedge (\varphi_1 \; \mathcal{S} \; \varphi_2))\end{array}$$

Note that $\underline{\mathcal{U}}$ differs from $\mathcal{U}_{[0,\infty)}$.

In what follows we show that some of the timed operators ($\mathcal{U}_I$ and $\mathcal{S}_I$, each with all types of intervals) can be written in terms of simpler ones, which will allow us to simplify our monitoring procedures for STL. We start with the following lemma, proved also in [11,21], which shows that the timed *until* can be expressed by a combination of untimed *until* and timed *eventually*.

Lemma 1 ($\mathcal{U}_I$ can be expressed by $\mathcal{U}$ and $\Diamond_I$). *For every signal w, the identities in Figure 3 hold.*

Proof. We prove the first of these identities, the others are similar. One direction of the equivalence follows directly from the semantics of timed *until*, so we consider only the other direction which is proved via the following observations:

1. If $w \models \Box_{(0,a]} \varphi_1$, then φ_1 holds continuously throughout $(0, a]$
2. If $w \models \Box_{(0,a]}(\varphi_1 \; \mathcal{U} \; \varphi_2)$, then $\varphi_1 \; \mathcal{U} \; \varphi_2$ has to hold at a and there exists $t' > a$ such that φ_2 is *true* and φ_1 holds during (a, t')
3. Formula $\Diamond_{(a,b)} \varphi_2$ requires the existence of $t' \in (a, b)$ such that φ_2 holds at t'

Combining these observations we see that $w \models \Box_{(0,a]} \varphi_1 \wedge \Box_{(0,a]}(\varphi_1 \; \mathcal{U} \; \varphi_2) \wedge \Diamond_{(a,b)} \varphi_2$ implies that there exists $t' \in (a, b)$ such that φ_2 is *true* at t' and φ_1 holds continuously during $(0, t')$, which is the semantic definition of $\varphi_1 \; \mathcal{U}_{(a,b)} \; \varphi_2$. ▟

Consequently, the operators $\mathcal{U}$, $\mathcal{S}$, $\Diamond_I$ and $\diamondminus_I$, where I ranges over the interval types $[a, b]$, $[a, b)$, $(a, b]$ and (a, b), are sufficient to express any STL property.

2.2.1 Expressing Events

STL does not provide constructs that allow to reason explicitly about *instantaneous events* which can be viewed as taking place in singular intervals of zero duration. A natural way to introduce them is to consider the instants when a signal changes its value. To this end we propose two unary operators, *rise* ↑ and *fall* ↓, which hold at the rising and falling edges of a Boolean signal, respectively. However, since we allow singular points to be equal to their left neighborhood, $\uparrow p$ may hold at t even if $w|_p[t] = 0$ as illustrated in Figure 4. Intuitively, $\uparrow \varphi$ holds at t if φ is false at t and true in a right neighborhood of t, or if φ is true at t and false in a left neighborhood of t. These operators can be expressed in STL if we allow *both* future and past operators, as follows:

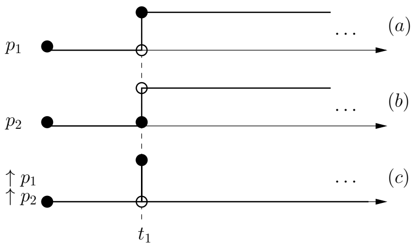


Fig. 4. Two signals p_1 and p_2 that differ at time t where both $\uparrow p_1$ and $\uparrow p_2$ hold.

$$\begin{array}{l}\uparrow \varphi = (\varphi \wedge (\neg\varphi \, \mathcal{S} \; \mathrm{T})) \;\vee\; (\neg\varphi \wedge (\varphi \, \mathcal{U} \; \mathrm{T})) \\ \downarrow \varphi = (\neg\varphi \wedge (\varphi \, \mathcal{S} \; \mathrm{T})) \;\wedge\; (\varphi \wedge (\neg\varphi \, \mathcal{U} \; \mathrm{T}))\end{array}$$

2.3 *Some Properties of $p\mathcal{S}q$ and $p\mathcal{U}q$*

In this section we prove some semantic properties of $p\mathcal{S}q$ and $p\mathcal{U}q$. In particular, we show that their satisfiability is uniform in all open time segments where their input does not change.

Lemma 2 (Since is Left-continuous). *Let $u = \dot{u}_0 \cdot (u_0)_{r_0} \cdot \dot{u}_1 \cdot (u_1)_{r_1} \cdots = \chi^{p\mathcal{S}q}(w)$. Then, $\dot{u}_0 = 0$ and for any $i \geq 1$, $\dot{u}_i = u_{i-1}$.*

Proof. The proof for $\dot{u}_0 = 0$ is trivial and follows directly the semantics of $p\mathcal{S}q$ evaluated at time 0, whose satisfaction requires the existence of $t' < 0$ which is not the case. For $i \geq 1$, assume first that $\dot{u}_i = 1$. Then there exist $t' < t_i$ such that q is satisfied at t' and that p holds continuously throughout the interval (t', t_i). Then, it follows that $(w, t) \models p\mathcal{S}q$ everywhere in (t', t_i) and, consequently $u_{i-1} = 1 = \dot{u}_i$. If $\dot{u}_i = 0$, there are two possibilities, either q was never true at any $t' \in [0, t_i)$, and hence u was false in the whole interval $(0, t_i)$, or that for any $t'' \in [0, t_i)$ where q was true, there is $t' \in (t'', t_i)$ where p was false, implying that $p\mathcal{S}q$ was not satisfied at (t', t_i) and $u_{i-1} = 0 = \dot{u}_i$. ▟

Lemma 3 (Until is Right-continuous). *Let $u = \dot{u}_0 \cdot (u_0)_{r_0} \cdot \dot{u}_1 \cdot (u_1)_{r_1} \cdot = \chi^{p\mathcal{U}q}(w)$. Then, for any $i \geq 0$, $\dot{u}_i = u_i$.*

Proof. Assume first that $\dot{u}_i = 1$. Then there exists $t' > t_i$ such that q is satisfied at t' and that p holds continuously throughout the interval (t_i, t'). Then, it follows that $(w, t) \models p\mathcal{U}q$ everywhere in (t_i, t') and, consequently $u_i = 1 = \dot{u}_i$. If $\dot{u}_i = 0$, there are two possibilities, either q never becomes true at any $t' > t_i$ and hence u is false in the whole open interval (t, ∞), or for any $t'' > t_i$ where q is true there is $t' \in (t_i, t'')$ where p does not hold which implies that $p\mathcal{U}q$ is not satisfied at (t_i, t') and $u_i = 0 = \dot{u}_i$. ▟

Lemma 4 (Semantic Rules for Since). *Let $w|_{pq} = \dot{w}_0 \cdot (w_0)_{r_0} \cdot \dot{w}_1 \cdot (w_1)_{r_1} \cdots$ be a Boolean signal and let $u = \dot{u}_0 \cdot (u_0)_{r_0} \cdot \dot{u}_1 \cdot (u_1)_{r_1} \cdots = \chi^{p\mathcal{S}q}(w)$. Then, for every $i \geq 0$,*

$$\begin{array}{lcl}
w \models \varphi_1 \ \mathcal{U}_{(a,b)}\ \varphi_2 & \leftrightarrow & w \models \Box_{(0,a]}\,\varphi_1 \wedge \Box_{(0,a]}(\varphi_1\ \mathcal{U}\ \varphi_2) \wedge \Diamond_{(a,b)}\,\varphi_2 \\
w \models \varphi_1 \ \mathcal{U}_{(a,b]}\ \varphi_2 & \leftrightarrow & w \models \Box_{(0,a]}\,\varphi_1 \wedge \Box_{(0,a]}(\varphi_1\ \mathcal{U}\ \varphi_2) \wedge \Diamond_{(a,b]}\,\varphi_2 \\
w \models \varphi_1 \ \mathcal{U}_{[a,b)}\ \varphi_2 & \leftrightarrow & w \models \Box_{(0,a)}\,\varphi_1 \wedge \Box_{(0,a]}(\varphi_1\ \underline{\mathcal{U}}\ \varphi_2) \wedge \Diamond_{[a,b)}\,\varphi_2 \\
w \models \varphi_1 \ \mathcal{U}_{[a,b]}\ \varphi_2 & \leftrightarrow & w \models \Box_{(0,a)}\,\varphi_1 \wedge \Box_{(0,a]}(\varphi_1\ \underline{\mathcal{U}}\ \varphi_2) \wedge \Diamond_{[a,b]}\,\varphi_2 \\
w \models \varphi_1 \ \mathcal{U}_{(a,\infty)}\ \varphi_2 & \leftrightarrow & w \models \Box_{(0,a]}\,\varphi_1 \wedge \Box_{(0,a]}(\varphi_1\ \mathcal{U}\ \varphi_2) \\
w \models \varphi_1 \ \mathcal{U}_{[a,\infty)}\ \varphi_2 & \leftrightarrow & w \models \Box_{(0,a)}\,\varphi_1 \wedge \Box_{(0,a]}(\varphi_1\ \underline{\mathcal{U}}\ \varphi_2)
\end{array}$$

$$\begin{array}{lcl}
w \models \varphi_1 \ \mathcal{S}_{(a,b)}\ \varphi_2 & \leftrightarrow & w \models \boxminus_{(0,a]}\,\varphi_1 \wedge \boxminus_{(0,a]}(\varphi_1\ \mathcal{S}\ \varphi_2) \wedge \diamondminus_{(a,b)}\,\varphi_2 \\
w \models \varphi_1 \ \mathcal{S}_{(a,b]}\ \varphi_2 & \leftrightarrow & w \models \boxminus_{(0,a]}\,\varphi_1 \wedge \boxminus_{(0,a]}(\varphi_1\ \mathcal{S}\ \varphi_2) \wedge \diamondminus_{(a,b]}\,\varphi_2 \\
w \models \varphi_1 \ \mathcal{S}_{[a,b)}\ \varphi_2 & \leftrightarrow & w \models \boxminus_{(0,a)}\,\varphi_1 \wedge \boxminus_{(0,a]}(\varphi_1\ \underline{\mathcal{S}}\ \varphi_2) \wedge \diamondminus_{[a,b)}\,\varphi_2 \\
w \models \varphi_1 \ \mathcal{S}_{[a,b]}\ \varphi_2 & \leftrightarrow & w \models \boxminus_{(0,a)}\,\varphi_1 \wedge \boxminus_{(0,a]}(\varphi_1\ \underline{\mathcal{S}}\ \varphi_2) \wedge \diamondminus_{[a,b]}\,\varphi_2 \\
w \models \varphi_1 \ \mathcal{S}_{(a,\infty)}\ \varphi_2 & \leftrightarrow & w \models \boxminus_{(0,a]}\,\varphi_1 \wedge \boxminus_{(0,a]}(\varphi_1\ \mathcal{S}\ \varphi_2) \\
w \models \varphi_1 \ \mathcal{S}_{[a,\infty)}\ \varphi_2 & \leftrightarrow & w \models \boxminus_{(0,a)}\,\varphi_1 \wedge \boxminus_{(0,a]}(\varphi_1\ \underline{\mathcal{S}}\ \varphi_2)
\end{array}$$

Fig. 3. Identities for bounded until and since

Case	$\dot{w}_i$	w_i	u_i
1	$*$	$\overline{p}$	0
2	$*$	pq	1
3a	$\overline{pq}$	$p\overline{q}$	0
3b	q		1
3c	$p\overline{q}$		$\dot{u}_i$

Fig. 5. $p\,\mathcal{S}\,q$ rules for determining u_i

Case	u_i	w_i	$\dot{w}_{i+1}$
1	0	$\overline{p}$	$*$
2	1	pq	$*$
3a	0	$p\overline{q}$	$\overline{pq}$
3b	1		q
3c	$\dot{u}_{i+1}$		$p\overline{q}$

Fig. 6. $p\,\mathcal{U}\,q$ rules for determining u_i

1. if $w_i = \overline{p}$, then $u_i = 0$,
2. if $w_i = pq$, then $u_i = 1$
3. if $w_i = p\overline{q}$, there are three possibilities:
(a) if $\dot{w}_i = \overline{pq}$, then $u_i = 0$
(b) if $\dot{w}_i = q$, then $u_i = 1$
(c) if $\dot{w}_i = p\overline{q}$, then $u_i = \dot{u}_i$.

Proof. The value of u in the i^{th} segment is determined with respect to the values of inputs p and q in the same segment w_i and at the preceding singular point $\dot{w}_i$. It is not hard to see that the 5 cases for values of $\dot{w}_i$ and w_i shown in Figure 5 cover all 16 possible combinations of values for p and q at the i^{th} singular point and the adjacent open segment. For any $t \in (t_i, t_{i+1})$ in the i^{th} segment, we have

Case 1: For any $t' < t$ which is in (t_i, t_{i+1}), by definition p does not hold throughout (t', t), hence $(w, t) \not\models p\,\mathcal{S}\,q$, that is $u_i = 0$.

Case 2: There exists $t' < t$ which is also in (t_i, t_{i+1}), where by definition q holds at t' and p holds continuously throughout (t', t). Hence $(w, t) \models p\,\mathcal{S}\,q$ for all such t and $u_i = 1$.

Case 3-(a): p was false at t_i and q does not hold anywhere in the interval (t_i, t), which implies that $p\,\mathcal{S}\,q$ is not satisfied throughout (t_i, t_{i+1}) and $u_i = 0$.

Case 3-(b): q was true at t_i and p was continuously true during (t_i, t), implying that $p\,\mathcal{S}\,q$ is satisfied at (t_i, t_{i+1}) and $u_i = 1$.

Case 3-(c): p holds and q remains false throughout $[t_i, t)$. Hence, $p\,\mathcal{S}\,q$ holds at t iff there is $t' \in [0, t_i)$ where q holds, and p remains true during (t', t_i), that is iff $p\,\mathcal{S}\,q$ holds at t_i. This implies that $p\,\mathcal{S}\,q$ is satisfied at (t_i, t_{i+1}) iff it is satisfied at t_i and $u_i = \dot{u}_i$. ▟

Lemma 5 (Semantic Rules for Until). *Let $w|_{pq} = \dot{w}_0 \cdot (w_0)_{r_0} \cdot \dot{w}_1 \cdot (w_1)_{r_1} \cdots$ be a Boolean signal and let $u = \dot{u}_0 \cdot (u_0)_{r_0} \cdot \dot{u}_1 \cdot (u_1)_{r_1} \cdots = \chi^{p\mathcal{U}q}(w)$. Then, for every $i \geq 0$,*

1. if $w_i = \overline{p}$, then $u_i = 0$,
2. if $w_i = pq$, then $u_i = 1$
3. if $w_i = p\overline{q}$, then either w_i is the last segment in w and $u_i = 0$, or:
(a) if $\dot{w}_{i+1} = \overline{pq}$, then $u_i = 0$
(b) if $\dot{w}_{i+1} = q$, then $u_i = 1$
(c) if $\dot{w}_{i+1} = p\overline{q}$, then $u_i = \dot{u}_i$.

Proof. The value of u in the i^{th} segment is determined with respect to the values of inputs p and q in that same segment w_i and the next singular point $\dot{w}_{i+1}$. It is not hard to see that the 5 cases for values of w_i and $\dot{w}_{i+1}$ cover all 16 possible combinations of values for p and q at w_i and $\dot{w}_{i+1}$. For any $t \in (t_i, t_{i+1})$ in the i^{th} segment, we have

Case 1: For any $t' > t$ in (t_i, t_{i+1}), and by definition p does not hold throughout (t, t'), hence $(w, t) \not\models p\,\mathcal{U}\,q$ and $u_i = 0$.

Case 2: There exists $t' > t$ in (t_i, t_{i+1}) such that by definition q holds at t' and p holds continuously throughout (t, t'). Hence $(w, t) \models p\,\mathcal{U}\,q$ for all such t and $u_i = 1$.

Case 3-(a): By definition p is false at t_{i+1} and q does not hold anywhere in the interval (t, t_{i+1}), which implies that $p\mathcal{U}q$ is not satisfied throughout (t_i, t_{i+1}) and $u_i = 0$.

Case 3-(b): q is true at t_{i+1} and p continuously holds during (t, t_{i+1}), implying that $p\mathcal{U}q$ is satisfied at (t_i, t_{i+1}) and $u_i = 1$.

Case 3-(c): p holds and q remains false throughout $(t, t_{i+1}]$. Hence, $p\mathcal{U}q$ holds at t iff there is $t' > t_{i+1}$ where q holds, and p remains true during (t_{i+1}, t'), that is iff $p\mathcal{U}q$ holds at t_{i+1}. This implies that $p\mathcal{U}q$ is satisfied at (t_i, t_{i+1}) iff it is satisfied at t_{i+1} and $u_i = \dot{u}_{i+1}$.

The only remaining case is when $w_i = p\overline{q}$ and it is the last segment in the signal w (end of the signal since w is of finite length). Since there is no $t' > t_i$ where q is true, $u_i = 0$. ∎

3 Monitoring STL Specifications

In this section, we describe two procedures for monitoring STL properties. These procedures are:

1. An *offline* procedure that propagates truth values upwards from propositions via super-formulae up to the main formula. The offline monitoring method is presented in section 3.1
2. An *incremental* marking procedure that updates the marking each time a new segment of the input signal is observed. Section 3.2 describes the incremental monitoring algorithm.

Unlike automata-based monitoring algorithms, the procedures that we propose are directly applied to signals. A central notion in these algorithms is that of the *satisfaction signal* $u_\varphi = \chi^\varphi(w)$ associated with a formula φ and a signal w. We remind the reader that this signal satisfies $u_\varphi[t] = 1$ iff $(w, t) \models \varphi$. Due to the non-causality of future temporal operators of STL, the value of $u_\varphi[t]$ is not necessarily known at time t, that is, after observing $w[t]$, and may depend on future values of w.

We consider signals of the form $w : \mathcal{T} \to \mathbb{B}^n \times \mathbb{R}^m$ and STL formulas that express mixed signal properties of w via predicates over its real valued components. A predicate over a set X of real valued variables is a function from X to $\mathbb{B}$. We consider a finite set of such predicates such that by applying them pointwise we obtain Boolean signals describing the evolution over time of the truth values of these predicates with respect to w. Hence, a numerical predicate in STL has a similar role as an atomic proposition.

Events such as *rising* and *falling* in the Boolean signal correspond to some *qualitative* changes in the real-valued signal, for example threshold crossing of some continuous variable.

The monitoring of STL can be reduced to Booleanization and monitoring against the MTL-skeleton of the formula, hence in the remainder of the section, we describe the monitoring algorithms for MTL formulas.

3.1 *Offline Marking*

The *offline marking* algorithm works as follows. It has as input an MTL formula and an n-dimensional signal w of length r. For every sub-formula ψ of φ it computes its satisfiability signal $u_\psi = \chi^\psi(w)$ (we will use u when ψ is clear from the context). The procedure is recursive on the structure (parse tree) of the formula (see Algorithm 1). It goes down until the propositional variables whose values are determined directly by w, and then propagates values as it comes up from the recursion. We use OP$_1$ and OP$_2$ for arbitrary unary and binary logical or temporal operators. As a preparation for the incremental version, we do not pass w and u_φ as input or output parameters but rather store them in global data structures.

Algorithm 1: OFFLINEMTL

```
input   : an MTL Formula $\varphi$ and signal $w$
switch $\varphi$ do
    case $p$
        $u_\varphi := w|_p$;
    case OP$_1(\varphi_1)$
        OFFLINEMTL ($\varphi_1$);
        $u_\varphi :=$ COMBINE(OP$_1, u_{\varphi_1}$);
    case OP$_2(\varphi_1, \varphi_2)$
        OFFLINEMTL ($\varphi_1$);
        OFFLINEMTL ($\varphi_2$);
        $u_\varphi :=$ COMBINE (OP$_2, u_{\varphi_1}, u_{\varphi_2}$);
end
```

Most of the work in this algorithm is done by the COMBINE function which for $\varphi = \text{OP}_2(\varphi_1, \varphi_2)$ computes u_φ from the signals u_{φ_1} and u_{φ_2}. We describe how this function works for each of the operators, and for the sake of readability we omit the description of various optimizations. We have shown in Lemma 1 that timed until and since operators are redundant and consequently, in the remainder of the section it is sufficient to describe the COMBINE function for the following operators:

- Negation $\neg\varphi$ and disjunction $\varphi_1 \vee \varphi_2$
- Untimed *since* $\varphi_1 \mathcal{S} \varphi_2$ and *until* $\varphi_1 \mathcal{U} \varphi_2$
- Timed *once* $\diamondminus_I \varphi$ and *eventually* $\Diamond_I \varphi$

3.1.1 Combine function for $\neg\varphi$ and $\varphi_1 \vee \varphi_2$

The negation $\varphi = \neg\varphi_1$ is simply computed with $u_\varphi :=$ COMBINE$(\neg, u_{\varphi_1})$, by changing the Boolean value of each singular point and open segment in the representation of u_{φ_1}.

For the disjunction $\varphi = \varphi_1 \vee \varphi_2$, the function $u_\varphi :=$ COMBINE$(\vee, u_{\varphi_1}, u_{\varphi_2})$ first refines the point-segment representation of the signals for the pairing $u' = u_{\varphi_1} || u_{\varphi_2}$, by computing a finer point-segment representation of u' such that the value of both signals u_{φ_1} and u_{φ_2} becomes uniform (does not vary) within every open segment. Then, we compute the disjunction at every point/segment, concatenating

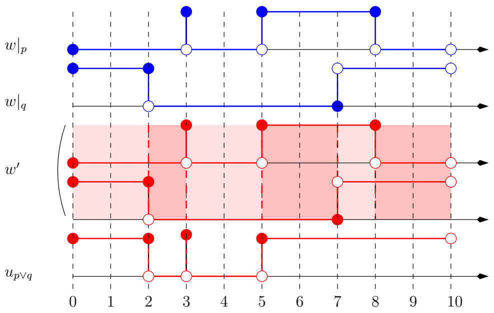


Fig. 7. Computing $u = \chi^{p\vee q}(w)$

them in order to obtain u_φ. This procedure is illustrated in Figure 7.

3.1.2 Combine function for $\varphi_1 \mathcal{S} \varphi_2$ and $\varphi_1 \mathcal{U} \varphi_2$

We assume a finite signal $w = \dot{w}_0 \cdot w_0^{r_0} \cdots \dot{w}_k \cdot w_k^{r_k}$ of length $|w| = r_0 + \cdots + r_k = r$. We have shown in Lemma 2 that $p \mathcal{S} q$ operator is left continuous, meaning that the satisfaction of the operator at any singular point cannot differ from its satisfaction during the previous open segment. We have also shown in Lemma 4 that there is a finite number of rules that determine the value of u in open segments depending on the past observations of p and q. The combination of these two results gives us a straightforward recipe for computing $u = \chi^{p \mathcal{S} q}(w)$.

Now we can describe how the function that computes the value of $u = \chi^{p \mathcal{S} q}(w)$ works. We start reading the signal w from the beginning towards the end. Following Lemma 2, $\dot{u}_0 = 0$, regardless of w. For every subsequent singular point, the value $\dot{u}_i$ is equal to u_{i-1}, the value of u during the previous open segment. When a new open segment of w_i is read, the procedure applies the rules of Lemma 4 to compute u_i, the value of u in the same segment. If p is false in w_i, then u_i is also false. Similarly, if both p and q hold in w_i, then the segment u_i is set to be true. Finally, if p holds during w_i and q is false throughout the same segment, there are three possibilities: 1) either both p and q were false at the previous singular point $\dot{w}_i$ and then u_i is set to be false; 2) q was true at $\dot{w}_i$ so u_i is set to true or 3) p was true and q false at $\dot{w}_i$ and u_i has the same value as in the previous singular point $\dot{u}_i$.

Computing the COMBINE function for $p \mathcal{U} q$ operator is symmetric to the $p \mathcal{S} q$ case. We have shown in Lemma 3 that *until* is right continuous, meaning that the satisfaction of the operator at any singular point is identical to its satisfaction during the next open segment. In Lemma 5 we provided a finite number of rules to determine the value of u in the open segments depending on the future observations of p and q. The combination of these two results provide rules for computing $u = \chi^{p \mathcal{U} q}(w)$.

The computation of $u = \chi^{p \mathcal{U} q}(w)$ works as follows. The signal w is read from the end towards the beginning. We determine the value of every open segment u_i according to the rules of Lemma 5. If p is false in w_i, then u_i is also false. Similarly, if both p and q hold in w_i, then the value of the segment u_i is set to be true. For segments w_i where p is true and q is false, there are four possibilities: 1) w_i is the last open segment in w and u_i is false; 2) both p and q are false in $\dot{w}_{i+1}$ and u_i is set to false; 3) q is true at $\dot{w}_{i+1}$ so u_i is also set to true or 4) p is true and q false at $\dot{w}_{i+1}$ and u_i has the same value as in the next singular point $\dot{u}_{i+1}$. Every singular point $\dot{u}_i$ is set to the value of the succeeding open segment u_i, as shown by Lemma 3.

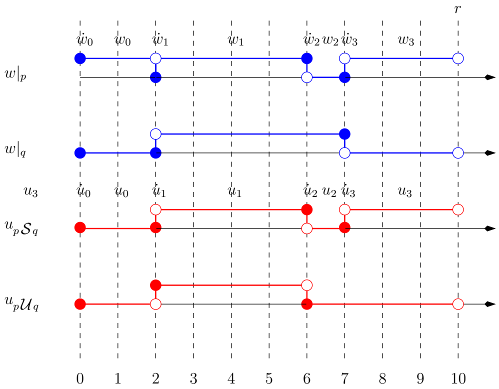


Fig. 8. Computing $u = \chi^{p\,\mathcal{S}\,q}(w)$ and $u = \chi^{p\,\mathcal{U}\,q}(w)$

Examples of computing $u = \chi^{p \mathcal{S} q}(w)$ and $u = \chi^{p \mathcal{U} q}(w)$ are shown in Figure 8.

3.1.3 Combine function for $\Diamond_I \varphi$ and $\diamondminus_I \varphi$

To compute $u = \chi^{\Diamond_I \varphi}(u_\varphi)$ and $u = \chi^{\diamondminus_I \varphi}(u_\varphi)$ we first observe that whenever φ holds in an interval J, u holds in the interval $J \ominus I \cap \mathcal{T}$ (respectively $J \oplus I \cap \mathcal{T}$). Hence, the essence of the procedure is to "propagate" the intervals in u_φ where φ holds either forward or backward. We employ the auxiliary concept of *interval covering* of a signal.

Definition 1 (Interval covering). For a Boolean signal u of finite length defined over $\mathcal{T} = [0, r)$, its interval covering is a sequence $\mathcal{I}_u = I_0, \ldots, I_k$ such that $\bigcup I_i = \mathcal{T}$ and $I_i \cap I_j = \emptyset$ for any $i \neq j$. An interval covering is said to be consistent with a finite length signal u if $u[t] = u[t']$ for every t, t' that belong to the same interval $I_i \in \mathcal{I}_u$. We denote by $\mathcal{I}_u$ the minimal interval covering consistent with the signal u. The set of positive intervals in $\mathcal{I}_u$ is $\mathcal{I}_u^+ = \{I \in \mathcal{I}_u | \forall t \in I, u[t] = 1\}$ and the set of negative intervals is $\mathcal{I}_u^- = \mathcal{I}_u - \mathcal{I}_u^+$.

Let us assume that $\mathcal{I}_{u_\varphi}$ is the minimal interval covering consistent with u_φ. Then $u = \chi^{\Diamond_I \varphi}(u_\varphi)$ is computed using the following procedure. For every positive interval $I^+ \in \mathcal{I}_\varphi^+$, we compute its back shifting (Minkowski difference saturated by $\mathcal{T}$) $I^+ \ominus I \cap \mathcal{T}$ and insert it to $\mathcal{I}_u^+$. This set represents the intervals where $\Diamond_I \varphi$ is satisfied, and the property is violated outside these intervals. Overlapping positive intervals in $\mathcal{I}_u^+$ are merged to obtain the minimal interval covering[1] of u.

[1] Note that the similar operation can be applied to negative intervals in $\mathcal{I}_\varphi^-$, in order to directly compute intervals where $\Diamond_I \varphi$ is violated. This

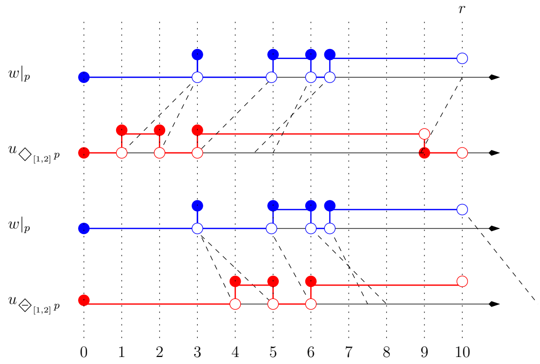


Fig. 9. Computing $u = \chi^{\Diamond_{[1,2]}p}(w)$ and $u = \chi^{\diamondminus_{[1,2]}p}(w)$

The combine function for $u = \chi^{\diamondminus_I \varphi}(u_\varphi)$ is computed in a similar way. For every positive interval $I^+ \in \mathcal{I}_\varphi^+$, we compute its forward shifting (Minkowski sum saturated by $\mathcal{T}$) $I^+ \oplus I \cap \mathcal{T}$ and insert it to $\mathcal{I}_u^+$, and merge the overlapping positive intervals in $\mathcal{I}_u^+$ to obtain the minimal interval covering of u.

An example of computing $u = \chi^{\Diamond_I \varphi}(u_\varphi)$ and $u = \chi^{\diamondminus_I \varphi}(u_\varphi)$ is shown in Figure 9.

3.2 *Incremental Marking*

This approach combines the simplicity of the offline procedure with the advantages of online monitoring in terms of early detection of violation/satisfaction and typically smaller memory requirements. After observing a prefix $w[0, t_1]$ of the signal we apply the offline procedure (without applying the finitary interpretation rules for future temporal operators, these are applied only at the end of the input trace). If, as a result, $u_\varphi = \chi^\varphi(w)$ is determined at time 0 we are done. Otherwise, we observe a new segment $w[t_1, t_2]$ and then apply the same procedure based on $w[0, t_2]$.

A more efficient implementation of this procedure need not start the computation from scratch each time a new segment is observed. It will be often the case that $u_\psi = \chi^\psi(w)$ for some sub-formulae ψ is already determined for some subset of $[0, t_1]$, based on $w[0, t_1]$. In this case we only need to propagate upwards new information obtained from $w[t_1, t_2]$, combined possibly, with some residual information from the previous segments that was not sufficient for determination of the satisfaction of the super formula. The choice of granularity (lengths of segments) in which this procedure is invoked depends on trade-offs between the computational cost and the importance of early detection.

The essence of the incremental marking procedure lies in the observation that the evaluation of a Boolean or future temporal formula φ at time t, depends on the values of their subformulae at $t' \geq t$. This implies that if u_φ is already determined at some interval $[0, t_1]$, we only need to keep the values of the satisfaction signal of its subformulae after t_1. Similarly, a past temporal operator ψ depends on the satisfaction of its subformulae at $t' \leq t$. The algorithm needs minor (and symmetric) adaptations between incremental marking for future and past temporal operators, and in the remaining of the section we focus on the procedure dealing with future temporal formulae.

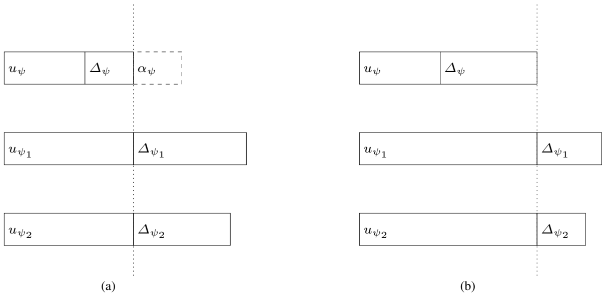


Fig. 10. A step in an incremental update: (a) A new segment α for ψ is computed from Δ_{ψ_1} and Δ_{ψ_2}; (b) α is appended to Δ_ψ and the endpoints of u_{ψ_1} and u_{ψ_1} are shifted forward accordingly.

Incremental marking is done using a piecewise-online procedure invoked each time a new segment of w, denoted by Δ_w, is observed. For each sub-formula ψ the algorithm stores its already-computed satisfaction signal partitioned into a concatenation of two signals $u_\psi \cdot \Delta_\psi$ with u_ψ consisting of values that have been already propagated to the super-formula of ψ, and Δ_ψ consists of values that have already been computed but which have not yet been propagated to the super-formula and can still influence its satisfaction.

Initially all signals are empty. Each time a new segment Δ_w is read, a recursive procedure similar to the offline procedure is invoked, which updates every u_ψ and Δ_ψ from the bottom up. The difference with respect to the offline algorithm is that only the segments of the signal that have not been propagated upwards participate in the update of their super-formulae. This may result in a lot of saving when the signal is very long (the empirical demonstration of this claim is given in section 6.1.4).

As an illustration consider $\varphi = \text{OP}(\varphi_1, \varphi_2)$ and the corresponding truth signals of Figure 10-(a). Before the update we always have $|u_\varphi \cdot \Delta_\varphi| = |u_{\varphi_1}| = |u_{\varphi_2}|$: the parts Δ_{φ_1} and Δ_{φ_2} that may still affect φ are those that start at the point from which the satisfaction of φ is still unknown. We apply the COMBINE procedure on Δ_{φ_1} and Δ_{φ_2} to obtain a new (possibly empty) segment α of Δ_φ. This segment is appended to Δ_φ in order to be propagated upwards, but before that we need to shift the borderline between u_{φ_1} and Δ_{φ_1} (as well as between u_{φ_2} and Δ_{φ_2}) in order to reflect the update of Δ_φ. The procedure is described in Algorithm 2.

Example 2. We illustrate the incremental monitoring procedure on the MTL formula $\varphi = \Box(p \rightarrow \Diamond_{[1,2]} q)$. The input signal w is split into three segments Δ_w^1, Δ_w^2 and Δ_w^3 and the incremental marking procedure is applied upon the arrival of each such segment:

procedure is not necessary for offline monitoring, but is useful for the incremental version of the algorithm

Algorithm 2: INC-OFFLINE-MTL

```
input : an MTL Formula $\varphi$ and an increment $\Delta_w$ of a
        signal
switch $\varphi$ do
    case $p$
        $\Delta_\varphi := \Delta_\varphi \cdot w_p(\Delta_w)$;
    case $\text{OP}_1(\varphi_1)$
        INC-OFFLINE-MTL $(\varphi_1)$;
        $\alpha :=$ COMBINE$(\text{OP}_1, \Delta_{\varphi_1})$;
        $d := |\alpha|$ ;
        $\Delta_\varphi := \Delta_\varphi \cdot \alpha$ ;
        $u_{\varphi_1} := u_{\varphi_1} \cdot \langle \Delta_{\varphi_1} \rangle_d$ ;
        $\Delta_{\varphi_1} := d \backslash \Delta_{\varphi_1}$
    case $\text{OP}_2(\varphi_1, \varphi_2)$
        INC-OFFLINE-MTL $(\varphi_1)$;
        INC-OFFLINE-MTL $(\varphi_2)$;
        $\alpha :=$ COMBINE$(\text{OP}_2, \Delta_{\varphi_1}, \Delta_{\varphi_2})$;
        $d := |\alpha|$ ;
        $\Delta_\varphi := \Delta_\varphi \cdot \alpha$ ;
        $u_{\varphi_1} := u_{\varphi_1} \cdot \langle \Delta_{\varphi_1} \rangle_d$ ;
        $\Delta_{\varphi_1} := d \backslash \Delta_{\varphi_1}$ ;
        $u_{\varphi_2} := u_{\varphi_2} \cdot \langle \Delta_{\varphi_2} \rangle_d$ ;
        $\Delta_{\varphi_2} := d \backslash \Delta_{\varphi_2}$
end
```

1. The first step of the monitoring procedure is computed when the first segment $\Delta_w^1 = \binom{\dot{\bar{p}}}{q} \cdot \binom{\bar{p}}{q}^2 \cdot \binom{\dot{p}}{q} \cdot \binom{p}{q}^{0.5} \cdot \binom{\dot{p}}{\bar{q}} \cdot \binom{p}{\bar{q}}^{1.5}$ is appended to w. After applying recursively the marking procedure and computing u_ψ for the subformulae ψ of φ. Figure 11-(a) shows the computed signals and as we can see, u_φ for the top level formula φ remains empty. Note that the segment of w defined over $[0, 2)$ as well as the entire computed segment of $u_{\Diamond_{[1,2]}} q$ can be discarded, since they do not affect any more the satisfaction of their corresponding super-formulae.
2. The segment $\Delta_w^2 = \binom{\dot{\bar{p}}}{q} \cdot \binom{\bar{p}}{q}^3 \cdot \binom{\dot{p}}{q} \cdot \binom{p}{q}^{0.5} \cdot \binom{\dot{p}}{\bar{q}} \cdot \binom{p}{\bar{q}}^{0.5}$ is appended to the previous segment of w, and the incremental marking procedure is applied again, computing new segments of satisfaction signals for sub-formulae of φ. The satisfaction of the top formula remains undetermined. The satisfaction signals for subformulae of φ after applying the marking procedure are shown in Figure 11-(b).
3. Finally, the third segment $\Delta_w^2 = \binom{\dot{\bar{p}}}{\bar{q}} \cdot \binom{\bar{p}}{\bar{q}}^2$ is appended to w and the incremental marking procedure is applied again. Now, all the subformulae of φ, including the top level formula itself can be updated, and since u_φ is false at $t = 0$ (see Figure 11-(c)), we can conclude that the formulae is violated by w and stop the procedure.

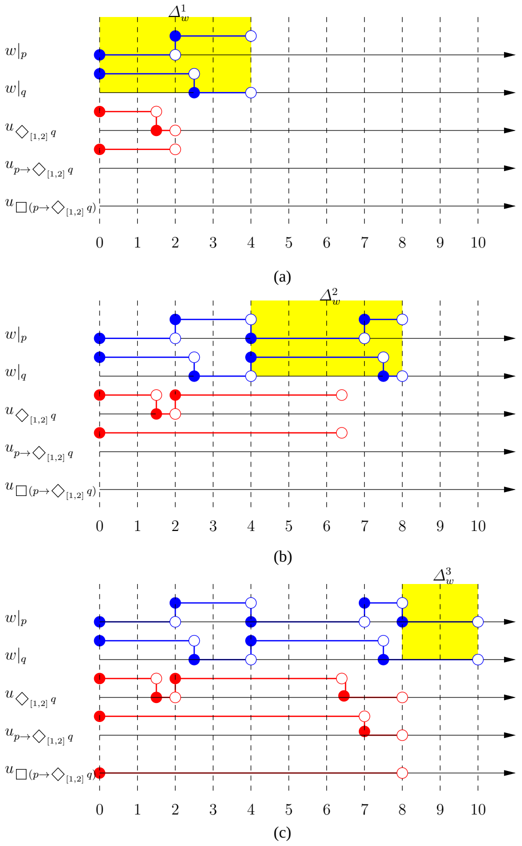


Fig. 11. Satisfaction signals u_ψ for sub-formulae ψ of $\varphi = \Box(p \rightarrow \Diamond_{[1,2]} q)$ computed incrementally upon receiving (a) Δ_w^1 (b) Δ_w^2 and (3) Δ_w^3

4 Continuous Signals and their Representation

Section 2 defined the satisfaction relation of STL and we have seen in Section 3, that monitoring of STL can be simply reduced to monitoring of MTL, by booleanizing real valued variables via numerical predicates. However, to really implement a monitoring procedure, we have to cope with some technical problems related to the computer representation of continuous signals.

As we have seen in Section 2.1, finite non-Zeno Boolean signals, albeit the fact that they are defined over dense time domain, admit an *exact finite representation* via the switching (singular) points and the open segments that define their *true* and *false* intervals. This is no longer the case for continuous signals where we have a contrast between the *ideal mathematical object*, consisting of an uncountable number of pairs $(t, \xi[t])$ with t ranging over some interval $[0, r) \subseteq \mathcal{T}$, and any *finite* representation which consist of a collection of such pairs, with t restricted to range over a finite set of *sampling points*.

The values of ξ at sampling points t_1 and t_2 do not determine the values of ξ inside the interval (t_1, t_2). They may, at most, impose some constraints on these values. Such constraints can be based on the dynamics of the generating system and on the manner in which the numerical simulator

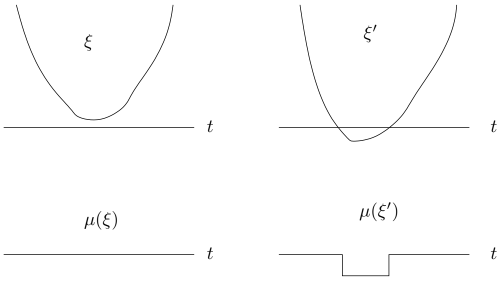


Fig. 12. Two signals which are close from a continuous point of view, one satisfying the property $\Box(x > 0)$ and one violating it.

produces the signal values at the sampling points. Numerical analysis is a very mature domain studying algorithms for numerical approximations with a lot of accumulated experience concerning tradeoffs between accuracy and computation time.

In order to speak quantitatively about the approximation of a signal by another we need the concept of a *distance/metric* imposed on the space of continuous signals. A metric is a function that assigns to two signals ξ_1 and ξ_2 a non-negative value $\rho(\xi_1, \xi_2)$ which indicates how they resemble each other. Using metrics one can express the "convergence" of a numerical integration scheme as the condition that $\lim_{d \to 0} \rho(\xi, \xi_d) = 0$ where ξ is the ideal mathematical signal and ξ_d is its numerical approximation using an integration step d.

Metrics and norms for continuous signals are used extensively in circuit design, control and signal processing. There are, however, major problems concerned with their application to property monitoring due to the incompatibility between the *continuous* nature of the signals and the discrete nature of their Booleanization, a phenomenon which is best illustrated using the following simple example. Consider the property $\Box(x > 0)$ and an ideal mathematical signal ξ that satisfies the property but which passes very close to zero at some points. We can easily deform ξ into a signal ξ' which is *very close* to ξ under any reasonable continuous metric, but according to the metric induced by the property, these signals are as distant as can be: one of them satisfies the property and the other violates it (see Figure 12).

Moreover, if the sojourn time of a signal below zero is short, an arbitrary shift in the sampling can make the monitor miss the zero-crossing event and declare the signal as satisfying (see Figure 13). In this sense properties are not *robust* as small variations in the signal may lead to large variations in its property satisfaction. Let us mention some interesting ideas [18] concerning new metrics for bridging the gap between the continuous and the discrete points of view, by extending the usual approximation and sampling theory of continuous signals and systems to those encompassing discontinuities.

We handle the above-mentioned issues pragmatically. The following assumptions facilitate the monitoring of sampled continuous signals against STL properties, passing through Booleanization:

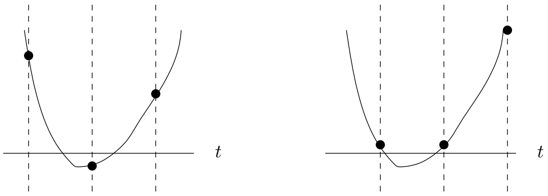


Fig. 13. Shifting the sampling points, zero crossing can be missed.

1. *Sufficiently-dense sampling*: the simulator detects every change in the truth value of any of the predicates appearing in the formula at a sufficient accuracy. This way the positive intervals of all the Boolean signals that correspond to these predicates are determined. This requirement imposes some level of sophistication on the simulator that has to perform several back-and-forth iterations to locate the time instances where a threshold crossing occurs. Many simulation tools used in industry have such event-detection features. For instance, VERILOG-AMS [27] provides event-detection feature using constructs such as `@cross`, `@last_crossing` or `@above` which allow to detect the crossings of thresholds with arbitrary precision, by forcing the simulator to make smaller time steps around the defined threshold. A survey of the treatment of discontinuous phenomena by numerical simulators can be found in [23].
2. *Bounded variability*: some restrictive assumptions can be made about the values of the signal between two sampling points t_1 and t_2. For example one may assume that ξ is monotone so that if $\xi[t_1] \leq \xi[t_2]$ then $\xi[t'_1] \leq \xi[t'_2]$ for every t'_1 and t'_2 such that $t_1 < t'_1 < t'_2 < t_2$. An alternative condition could be a condition a-la Lipschitz: $|\xi[t_2] - \xi[t_1]| \leq K|t_2 - t_1|$. Such conditions guarantee that the signal does not get wild between the sampling points, otherwise property checking based on these values may become useless.

Under such assumptions every continuous signal given by a discrete-time representation, based on sufficiently-dense sampling, induces a well-defined Boolean signal ready for MTL monitoring. When there is no direct connection with the simulator available, we replace the hypothesis of sufficiently-dense sampling by *interpolation*. That is, when we have two consecutive sampling points $t_1 < t_2$ such that one satisfies a predicate and the other does not, we use linear interpolation to "compute" the value of the signal throughout the interval (t_1, t_2) and detect the singular point t' where the value of the predicate changes. The procedure is illustrated in Figure 14.

Let us add at this point a general remark that the standards of exactness and exhaustiveness as maintained in discrete verification cannot and should not be exported to the continuous domain, and even if we are not guaranteed that all events are detected, we can compensate for that by using safety margins in the predicates and properties.

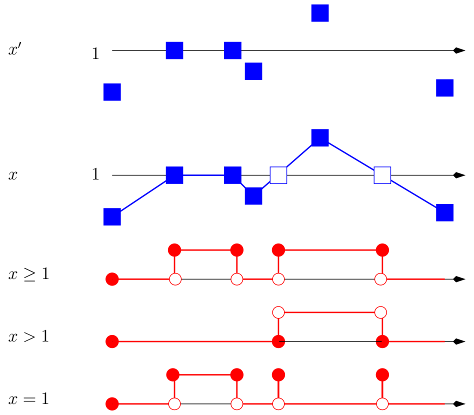


Fig. 14. Transformation of a continuous signal to its Boolean abstraction via interpolation and numerical predicates. The signal indicated by x' was not sufficiently dense with respect to the predicates $x \circ 1$ and hence two additional sampling points were added.

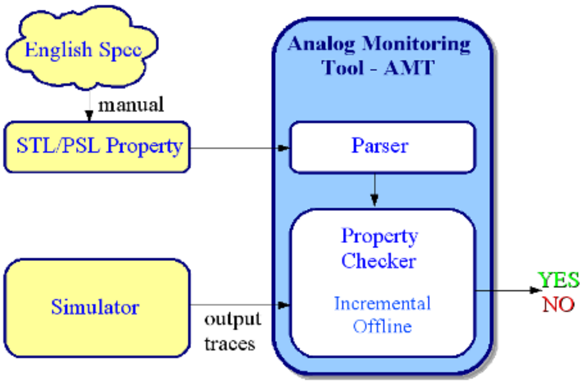


Fig. 15. Architecture of the AMT tool

5 AMT - Analog Monitoring Tool

In this section, we describe the *analog monitoring tool* AMT that implements the monitoring algorithms presented in Section 3. AMT is a stand-alone tool with a graphical user interface (GUI) written in C++ for the GNU/Debian Linux x86 platform. We used the QT[2] library to develop the tool GUI, and QWT[3] library to implement visualization of Boolean and continuous signals.

Figure 15 illustrates the general architecture of AMT. The tool takes as inputs: (1) a formal specification; and (2) a set of simulation traces. The formal specification is expressed in STLPSL, a textual language that extends STL with syntactic sugar. The STLPSL property usually results from a manual formalization of an informal specification. The STLPSL specification is automatically translated into a property checker that monitors the input simulation traces and checks whether they satisfy the required properties.

The main window of the application is partitioned into five frames that allow the user to manage STLPSL properties and input signals, evaluate the correctness of the simulation traces with respect to the input specification and, finally, visualize the results. The **property edit** frame contains a text editor for writing, importing and exporting STLPSL specifications, which are then translated into an internal data structure based on the parse-tree of the formula stored in the **property list** frame. An STLPSL specification is imported into the **property evaluation** frame for monitoring with respect to a set of input simulation traces, in either *offline* or *incremental* mode. The static import of the input traces is done through the **signal list** frame. The imported input signals, as well as signals associated to the subformulae of a specification can be visualized by the user from the **signal plots** frame. A screenshot of the main window is shown in Figure 16.

5.1 *Property Management*

The specifications in AMT are written in a simple editor with syntax highlighting for the STLPSL language. An STLPSL specification is transformed into an internal data structure that is adapted for monitoring, following the parse-tree of the formula. The user can hold more than one specification that is ready for evaluation in the property list frame.

5.2 *Property Format*

The tool supports as input specification the STLPSL language, that extends STL with syntactic sugar inspired by the Property Specification Language (PSL). We group single properties (declared as *assertions*) into a single logical unit in which they can be monitored simultaneously. We also add a definition directive that allows the user to declare a formula and give it a name, and then refer to it as a variable within other assertions. The syntax of STLPSL is defined with the following production rules

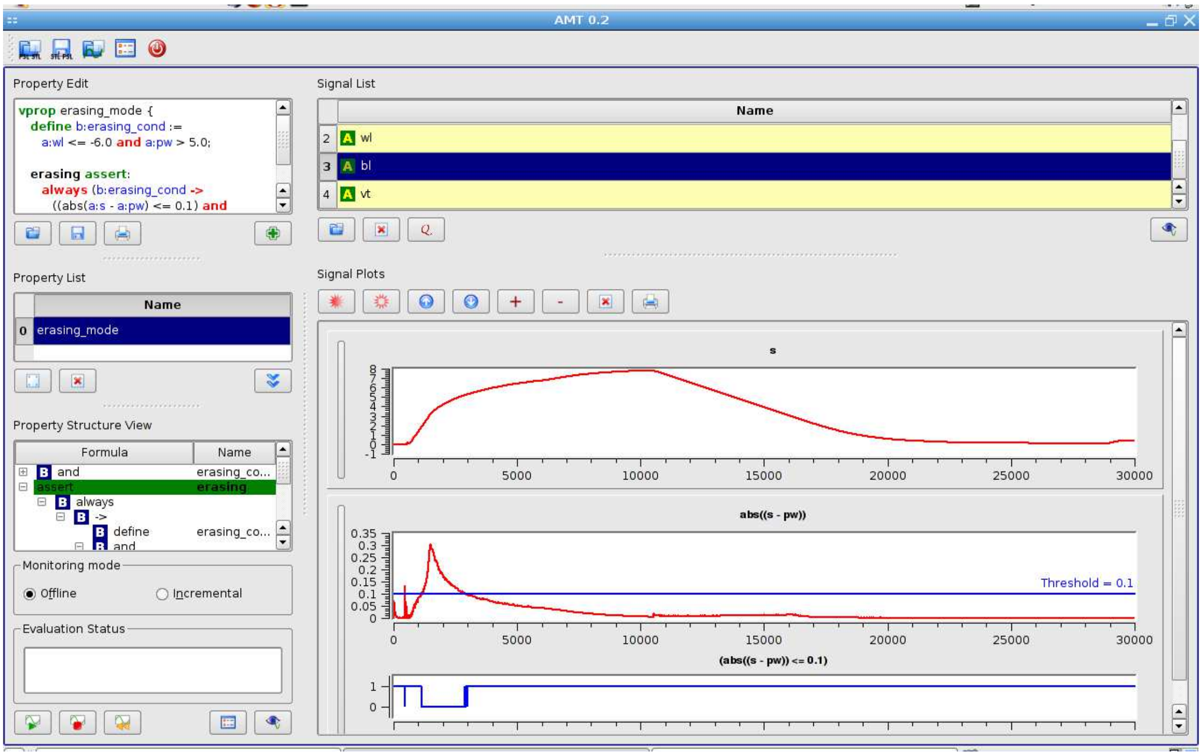
```
varphi :==
  vprop NAME {
    { define_directive }
    { assert_directive }
}

define_directive :==
  define b:NAME := varphi
  | define a:NAME := phi

assert_directive :==
  NAME assert : varphi
```

where `varphi` corresponds to a temporal property and `phi` to an analog operation. We omit the full list of operators supported in STLPSL and refer the reader to the tool documentation.

5.3 *Property Evaluation*

The property evaluation frame provides the monitoring features for checking correctness of an STLPSL specification

[2] http://www.trolltech.com
[3] http://qwt.sourceforge.net



```
vprop erasing_mode {
  define b:erasing_cond :=
    a:wl <= -6.0 and a:pw > 5.0;

  erasing assert:
    always (b:erasing_cond ->
      ((abs(a:s - a:pw) <= 0.1) and
```

	Name
0	erasing_mode

	Name
2	A wl
3	A bl
4	A vt

Formula	Name
⊞ B and	erasing_co...
⊟ assert	erasing
⊟ B always	
⊟ B ->	
B define	erasing_co...
⊟ B and	

Monitoring mode: ● Offline ○ Incremental

Evaluation Status

Fig. 16. AMT main window

with respect to input simulation traces. The frame shows the set of assertions in a tree view, following the parse structure of the formulas. The user can choose in particular between *offline* and *incremental* evaluation of the specification.

In the offline case, the assertions are monitored with respect to input signals that are fetched from the signal list frame. If one or more signals are missing, the monitoring procedure still tries to evaluate the property, but without guaranteeing a conclusive result.

For the incremental procedure, AMT acts as a server that waits for a connection from a simulator. Once the connection is established, the simulator sends input segments incrementally. The monitor alternates between reception of new input segments and incremental evaluation of the assertions. The user can configure the following parameters for the incremental evaluation:

- The user can set the TCP/IP port that is used for communication between the tool and the simulator
- *timeout* value defines the time period between two consecutive evaluations. Simulations of analog circuit often have tens or even hundreds of thousands of samples per signal. Hence, it is usually not wise to re-evaluate the property upon receiving every single new individual sample. This option enables to accumulate inputs received from the simulator between any two consecutive periods defined by the timeout value and apply the incremental checking procedure only at instants when the timer expires. There is no pre-defined optimal value for the timeout, and it represents a trade-off between the frequency of computation and the potential possibility of earlier detection of violation/satisfaction of a property
- The incremental procedure often provides better memory management than its offline counterpart. This happens because parts of the signals that have been fully determined and are not needed by their super-formulae can be discarded. However, in some situations, one would prefer to keep the entire signal for visualization and debugging purposes. The tool allows the user to choose through the "keep history" option whether the entire signal is kept, or only its segments that are needed for subsequent evaluations

There are three ways to terminate the incremental monitoring procedure:

1. All assertions become determined and AMT stops the evaluation, closing the connection with the simulator
2. The special termination packet is received from the simulator, indicating the end of the input traces. In that case the tool completes the evaluation of assertions with respect to the finitary semantics of the specification language operators
3. The user explicitly stops the procedure before the end of simulation, by resetting the monitoring. In that case the connection with the simulator is closed and the evaluation remains undetermined;

AMT favors visual evaluation of monitoring results, by choosing a different color scheme for *undetermined*, *satisfied* and *violated* assertions. Each subformula of the specification has an associated signal, which can be visualized within the signal plots frame. The visualization of the associated signals can be used for debugging and understanding the reasons behind satisfaction or violations of assertions. During the incremental evaluation, if the "keep history" option is enabled, all the signals within the signal plots frame are updated in real-time as new results are computed.

5.4 Signal Management

The signals in AMT can be either real-valued or Boolean. Signals are input traces that can be imported into the tool in an offline or incremental fashion. Signals are also associated to subformulas of STLPSL specifications. The user can visualize signals from the signal plots frame.

5.4.1 Offline Signal Input

Signals can be statically loaded from the signal list frame. AMT supports the following input formats:

out The output format of the NANOSIM simulator. The *current* and *voltage* signals are loaded, while *logical* signals are ignored.
vcd The subset of Value Change Dump file format including real and 2-valued Boolean signals, commonly used for dumping simulations.
txt A simple Ascii format that is dumped by the COSMOSSCOPE wave calculator tool.

The analog simulation traces are usually very large. A typical file generated by the simulation of a complex AMS circuit contains hundreds of signals, and often exceeds hundreds of megabytes of data. AMT has been designed to be able to deal with very large files and has been tested with simulation dumps exceeding $2GB$ of memory. While a standard simulation file contains hundreds of signals, an STLPSL specification usually refers only to a small subset. Hence, there is a need to efficiently navigate through the list of available signals. For this purpose, AMT provides an option for multiple selection of signals, as well as the selection of signals by a filter. For instance, the filter `*data*1*` selects all signals that have the pattern `data` withing their names followed (not necesserily immediately) by `1`. Moreover, an additional window shows the list of currently selected signals.

5.4.2 Incremental Signal Input

Signals can be imported incrementally to AMT, via a simple TCP/IP protocol. A simulator that produces input signals needs to connect to AMT during the incremental evaluation and send packets containing signal updates to the tool. The packets can be either Boolean or continuous signal updates, or a special termination packet, informing the tool that the simulation is over.

6 Case Studies

In this section, we present two case studies intended to evaluate the usefulness of our property-based approach for checking the correctness of AMS simulation traces. The first case study is described in Section 6.1 and involves checking properties of a FLASH memory design with the simulation traces provided by ST Microelectronics. The second case study is presented in Section 6.2 and involves monitoring specifications of a DDR2 memory interface component from Rambus.

The properties used in the FLASH memory case study were provided by ST Microelectonics analog designers in form of informal specifications written in English language. These properties were translated to STLPSL, matching the original requirements expressed by the designers. This process took several iterations involving clarifications with designers on the exact meaning of the specifications. The main objective of this case study is the evaluation of the AMT tool.

The DDR2 memory interface case study rather concentrates on the specification of complex timing properties from the official specification document [15] in STLPSL. The objective is to evaluate the expressiveness of STLPSL with respect to a realistic example used in the analog industry and identify potential weaknesses of the approach, providing useful feedback about missing features that should be taken into consideration in future research.

6.1 FLASH *Memory Case Study*

The subject of the case study is the "Tricky" technology FLASH memory test chip in $0.13\mu s$ process developed in ST Microelectronics. The FLASH memory is a digital system whose logical behavior is implemented at the analog level. Hence, it presents a direct link between the analog and the digital worlds.

For monitoring, the system under test is seen as a black box, and we do not need to know details chip architecture, apart from its monitoring interface. The memory cell can be in one of the *programming*, *reading* or *erasing* modes. The correct functioning of the design at its analog level of abstraction in a given mode is determined by the behavior of its interface signals extracted during the simulation:

bl:	matrix bit line terminal
pw:	matrix p-well terminal
wl:	matrix word line
s:	matrix source terminal
vt:	threshold voltage of cell
id:	drain current of cell

The memory cell was simulated in the *programming* and *erasing* modes for the case study, with the simulation time being $5000\mu s$ and $30000\mu s$ respectively. Four STLPSL properties were specified to describe the correct behavior of the cell in the *programming* mode and one property in the *erasing mode*. The AMT monitoring was done on a Pentium 4 HT 2.4GHz machine with 2Gb of memory. All the properties were found to be *correct* with respect to the input traces.

6.1.1 Programming Mode

The first property requires that whenever the **vt** signal crosses the threshold of 5, both **vt** and **id** have to remain continuously above 4.5 and $5 \cdot 10^{-6}$ respectively, until **id** falls below $5e-6$ (see Figure 17 for the resulting signals after the evaluation).

The STLPSL specification for this property[4] is:

```
vprop programming1 {

  pgm1 assert:
    always (rise(a:vt>5) ->
      ((abs (a:id) > 5e-6) and (a:vt>4.5))
      until (fall(a:id>5e-6)));
}
```

The second property is split into two assertions. The first assertion **pgm1** requires that whenever the wordline **wl** is below 0.1 but will jump to above 3.8 within $15\mu s$ and the cell is not in the programming mode (translated by the absolute value of the source current **id** being below $30 \cdot 10^{-6}$), the bitline signal **bl** should cross 3.8 before the end of the simulation, and remain above that threshold continuously until the word line **wl** goes above 6, which should happen within 300 and $1500\mu s$ from the **bl** crossing. The results of the evaluation are shown in Figure 18.

The second assertion **pgm2** specifies that whenever the programming procedure starts (translated by the crossing of 3.8 threshold by the bitline signal **bf**), bitline should not fall below that threshold until the signal **vt** becomes higher than 5 and the absolute value of the source current **id** goes below $5 \cdot 10^{-6}$. Figure 19 shows the results of the **pgm2** assertion of the property.

We use the following STLPSL specification to express the second property:

```
vprop programming2 {

  define b:not_pgm :=
    rise((a:wl <= 0.1) and
      eventually[0:15]
      (a:wl >= 3.8 and a:id >= 30e-6));

  pgm1 assert:
    always (b:not_pgm ->
      eventually (rise(a:bl>=3.8) and
        ((a:bl>=3.8) until[300:1500]
          (a:wl >= 6))));

  pgm2 assert:
    always (rise(a:bl >= 3.8) ->
      (not (a:bl <= 0.1) until (a:vt >=5
        and abs(a:id) <= 5e-6)));
}
```

6.1.2 Erasing Mode

We first define the erasing condition that holds whenever the wordline signal **wl** is lower than -6 and p-well **pw** is above 5. The main property states that whenever an erasing condition holds, the pointwise distance between the source **s** and p-well **pw** voltages has to be smaller than 0.1 and the value of **pw** should not be greater than 0.83 from the value of bitline **bl**.

The STLPSL specification of the property is as follows:

```
vprop erasing {
  define b:erasing_cond :=
    a:wl <= -6 and a:pw > 5;

  erasing assert:
    always (b:erasing_cond ->
      (abs (a:s-a:pw) <= 0.1)
      and (a:pw-a:bl)<0.83));
}
```

Figure 20 shows some of the representative signals of the erasing property.

6.1.3 P-Well Driving During Programming

This property requires that whenever the bitline **bl** and wordline **wl** signals are above 2.5 threshold, the p-well signal **pw** has to be below -0.5. The evaluation results for the p-well property are shown in Figure 21.

The p-well property is expressed in STLPSL as follows:

```
vprop pwell {
  p_well assert:
    always ((a:bl>2.5 and a:wl>2.5) ->
      a:pw<=-0.5);
}
```

6.1.4 Tool Performance

The time and space requirements of AMT were studied with both *offline* and *incremental* algorithms. The complexity of the algorithm used in AMT is shown to be $O(k \cdot m)$ in [20] where k is the number of sub-formulae and m is the size of the input signal (number of singular points and open segments).

Table 1 shows the size of the input signals (number of singular points and segments). We can see that the *erasing* mode simulation generated 10 times larger inputs from the *programming* mode simulation. Table 2 shows the evaluation results for the *offline* procedure of the tool. Monitoring the properties for the programming mode required less than half a second. Only the `erasing` property took more than 2 seconds, as it was tested against a larger simulation trace. We can also see that the evaluation time is linear in the size of signals generated by the procedure and can deduce that the procedure evaluates about 1,000,000 intervals per second.

The execution times of the incremental algorithm are less meaningful because the procedure works in parallel with the

[4] This property is expressed in STL as follows: $\Box(\uparrow (vt > 5) \rightarrow ((|id| > 5 \cdot 10^{-6}) \wedge (vt > 4.5)\, \mathcal{U} \downarrow (id > 5 \cdot 10^{-6}))$

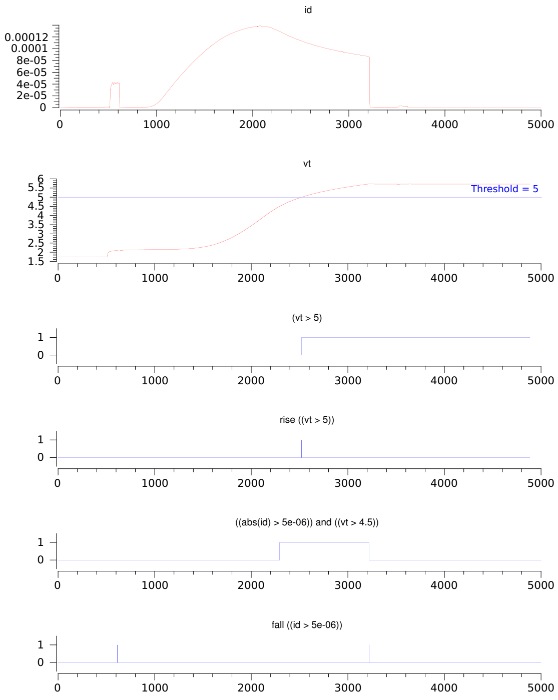


Fig. 17. Evaluation results for the **programming1** property

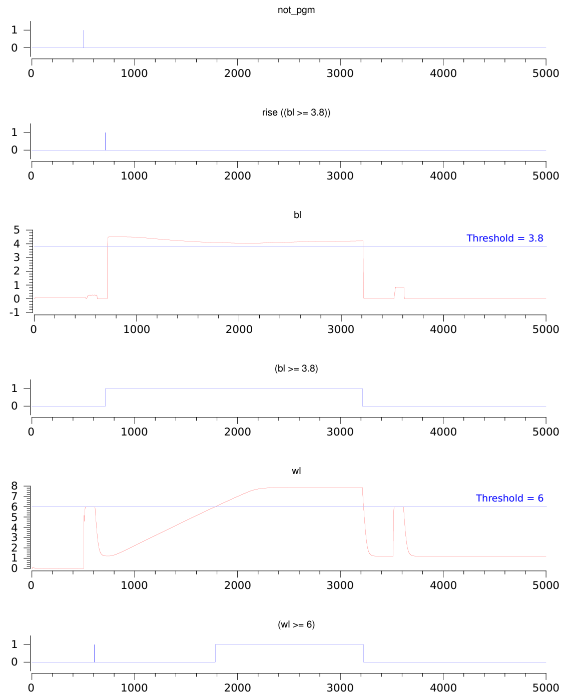


Fig. 18. Evaluation results for the **programming2** property (assertion **pgm1**)

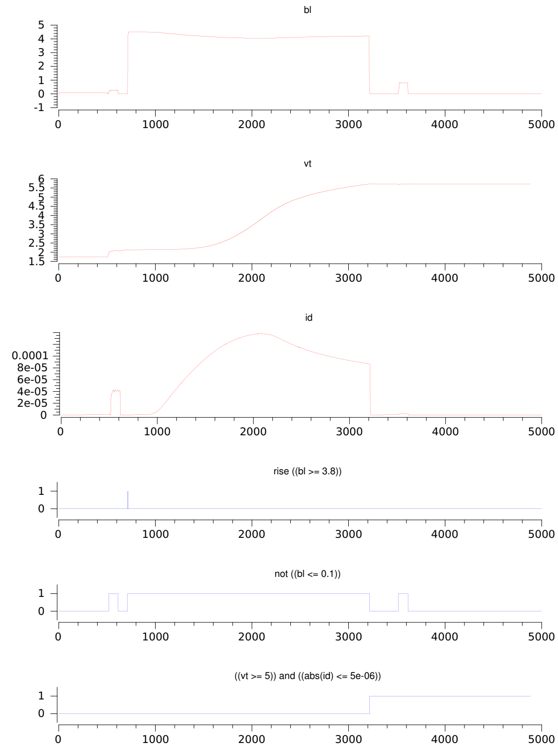


Fig. 19. Evaluation results for the **programming2** property (assertion **pgm2**)

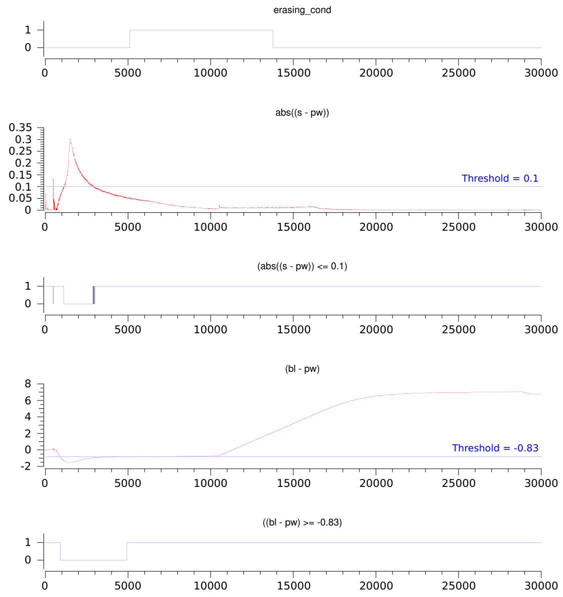


Fig. 20. Evaluation results for the **erasing** property

simulator and the evaluation time depends on the frequency of the incoming input. In fact, a major advantage of the incremental procedure is its ability to detect property violation in the middle of the simulation and save simulation time. Another advantage of the incremental algorithm is its reduced space requirement as we can discard parts of the simulation after they have been fully used. Table 3 compares the memory consumptions of the offline and incremental procedures. For the former we take the total number of signal segments generated by the tool, while for the latter we take the maximal number of signal segments kept simultaneously in memory. We can see that this ratio varies a lot from one property to another, going from 0.01% up to almost 70%. The general observation is that pointwise operators require considerably less memory in the incremental mode, while properties in-

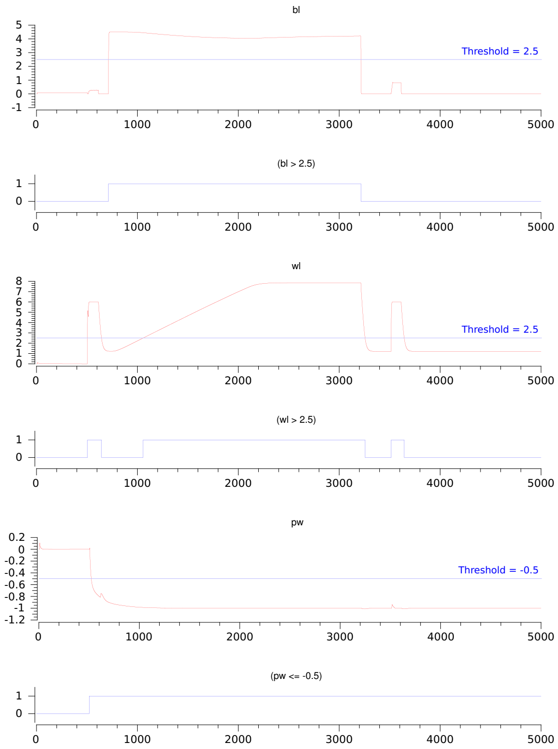


Fig. 21. Evaluation results for the **pwell** property

	pgm sim	erase sim
name	input size	input size
wl	34829	283624
pw	25478	283037
s	33433	282507
bl	32471	139511
id	375	n/a

Table 1. Input Size

property	time (s)	size
pgm1	0.14	99715
pgm2	0.42	405907
p-well	0.12	89071
erasing	2.35	2968578

Table 2. Offline algorithm evaluation

	Offline	Incremental	
Property	t	m	m/t * 100
pgm1	99715	65700	65.9
pgm2	594709	242528	40.8
p-well	89071	8	0.01

Table 3. Offline/incremental space requirement comparison, where t = total size and m = maximal active size

Threshold	Value (V)
V_{DDQ}	1.8
$V_{IH(AC)_{min}}$	1.25
$V_{IH(DC)_{min}}$	1.025
$V_{REF(DC)}$	0.9
$V_{IL(DC)_{max}}$	0.775
$V_{IL(AC)_{max}}$	0.65

Table 4. Threshold values for DQ and DQS

volving the nesting of untimed temporal properties often fail to discard their inputs until the end of the simulation.

6.2 DDR2 *Case Study*

This case study focuses on a DDR2 memory interface developed at Rambus. The memory interface acts as a bus between the memory and other components in the circuit and exhibits communication of digital data implemented at the analog level. Hence, correct functioning of a DDR2 memory interface largely depends on the appropriate timing of different signals within the circuit. In Section 6.2.1, we describe an alignment specification that expresses a typical DDR2 property and different steps needed for translating it in a formal STLPSL specification. The experimental results are presented in 6.2.2.

6.2.1 Alignment Between Data and Data Strobe Signals

In DDR2, data access is controlled by a single-ended or differential data strobe signal, which acts as an asynchronous clock. The official JEDEC DDR2 specification [15] describes, amongst others, a number of properties that involve timing relationship between events that happen in data and data strobe signals. In this case study, we are in particular interested in a property that defines the correct alignment between these two signals. The case study considers the specification parameters for the single-ended data strobe $DDR2-400$ memory interface, which is part of the JEDEC standard.

The DDR2 specification contains a number of relevant thresholds, shown in Table 4. The temporal relationship between data signal DQ and data strobe signal DQS is defined with respect to the crossings of these thresholds.

The general definition of the alignment of data DQ and data strobe DQS signals is shown in Figure 22. The proper alignment between the two signals is determined by two values, the *setup* time tDS and *hold* time tDH. The setup and hold times of DQ and DQS are checked both on their *falling* and *rising* edges, but we only consider, for the sake of simplicity, the specification of the setup time at the falling edge property. The other cases are similar and symmetric.

Informally, the setup property at the falling edge requires that whenever DQS crosses the $V_{IH(DC)_{min}}$ threshold from above, the previous crossing of $V_{IL(AC)_{max}}$ by the signal DQ from above should precede it by at least a period of time of tDS. This property is formalized in STLPSL as follows:

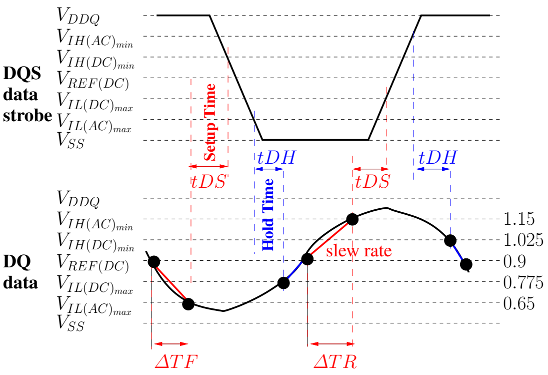


Fig. 22. Data DQ and data strobe DQS alignment

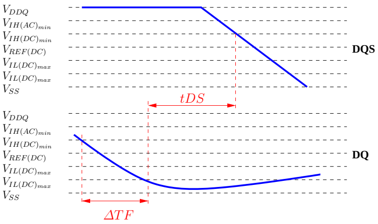


Fig. 23. DQ/DQS falling setup time tDS and the relation between slew rate and ΔTF

```
define b:dqs_above_vihdcmin :=
  (a:DQS >= 1.025);
define b:dqs_above_vilacmax :=
  (a:DQ >= 0.65);

always (fall(b:dqs_above_vihdcmin)
  -> historically[0:tDS] not
    fall(b:dq_above_vilacmax));
```

Unfortunately, the above property, naturally expressed in STLPSL, does not express the full reality. In practice, setup time tDS is not a constant value, but rather varies according to the slew rates (slopes) of DQ and DQS signals. For example, when DQ and DQS fall more sharply, the required tDS increases. Setup time tDS is equal to the sum of a (constant) *base term* $tDS(base)$ and a (variable) *correction term* ΔtDS

$$tDS = tDS(base) + \Delta tDS$$

The setup base term $tDS(base)$ is equal to $150ps$ for the single-ended DDR2-400. The correction term ΔtDS is a value that depends directly on slew rates of DQ and DQS, with the setup slew rate of a falling signal being defined as

$$sr = \frac{V_{REF(DC)} - V_{IL(AC)_{max}}}{\Delta TF} \quad (2)$$

where ΔTF is the time that the signal spends between $V_{REF(DC)}$ and $V_{IL(AC)_{max}}$. As we can see, the falling setup slew rate sr of a signal can be deduced from ΔTF.

In order to extract the setup correction term ΔtDS from the actual slew rates of DQ and DQS (sr_{DQ} and sr_{DQS}), we can use a specification table from [15], partially reproduced in Table 5. According to the JEDEC specification, ΔtDS corresponding to the slew rates not listed in Table 5 should be linearly interpolated. Consequently, we can apply the following sequence of computations in order to determine the correct value of tDS at any time

$$\Delta TF \rightarrow \text{setup falling slew rate} \rightarrow \text{correction term} \rightarrow tDS$$

To summarize, tDS is a value that varies during the simulation as a function of slew rates of DQ and DQS ($tDS = f(sr_{DQ}, sr_{DQS})$). The problem is that STLPSL cannot capture parameterized time bounds and therefore we have to use approximations in order to express a similar alignment property that still preserves some guarantees. We can subdivide the domain of slew rates (say $R = [sr_{min}, sr_{max}]$) into n regions $R_1, \ldots, R_n$. For each pair (R_i, R_j) of DQ/DQS slew rate regions, we assign a separate constant setup time tDS_{ij}. Instead of one property, we obtain $n \times n$ properties of the form:

"whenever DQS crosses the $V_{IH(DC)_{min}}$ threshold from above, DQ slew rate sr_{DQ} is in R_i and DQS slew rate sr_{DQS} is in R_j, the previous crossing of $V_{IL(AC)_{max}}$ by the signal DQ from above should precede it by at least a period of time of tDS_{ij}."

The proper constant value for tDS_{ij} for a pair of slew rate regions (R_i, R_j) can be chosen in two manners. The first solution consists in computing tDS_{ij} from the maximum correction term for the DQ and DQS slew rates that are in the R_i and R_j regions, respectively. This corresponds to an over-approximation of the original specification, and if this property is violated, we don't know if it is a real failure or a false alarm. On the other hand, the satisfaction of the over-approximated property implies that the original one holds too. Conversely, the computation of tDS_{ij} from the minimum correction term defined for the slew rates in the pair of regions (R_i, R_j) yields to an under-approximation of the original property. If the new property is falsified, we know that it corresponds to a real violation, while if it passes, we cannot say whether we are indeed safe.

As an example, consider the highlighted range of Table 5, which we call the "top-left" range, where the setup falling slew rates of DQ and DQS are between 1 and 2 V/ns. For the conservative approximation of tDS, with slew rates falling in that range, we choose the worst-case ΔtDS as the correction term, that is $188ps$. Hence, the approximated falling setup time tDS_{TL} for all DQ and DQS with falling slew rates between 1 and $2V/ns$ would be equal to $tDS_{TL} = 150 + 188 = 338ps$.

In order to determine the falling slew rates of DQ and DQS, we need to detect how much time these signals remain in their falling slew region (between $V_{REF(DC)}$ and $V_{IL(AC)_{max}}$ crossing $V_{REF(DC)}$ from above). This can be done with the following formula:

```
define b:dq_in_fsr :=
  ((a:DQ <= 0.9) and (a:DQ >= 0.65))
    since (a:DQ >= 0.9);
```

		DQS Single-Ended Slew Rate tDS			
		2V/ns	1.5V/ns	1V/ns	0.9V/ns
DQ	2V/ns	**188**	167	125	
Single-Ended	1.5V/ns	146	125	83	81
Slew Rate	1V/ns	63	42	0	-2
tDS	0.9V/ns		31	-11	13

Table 5. Correction terms for setup time

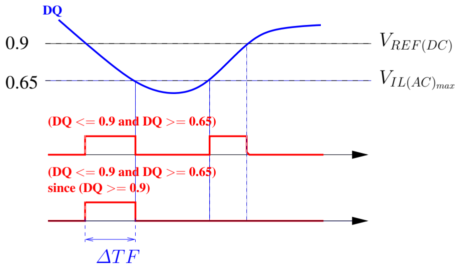


Fig. 24. Falling slew region and ΔTF

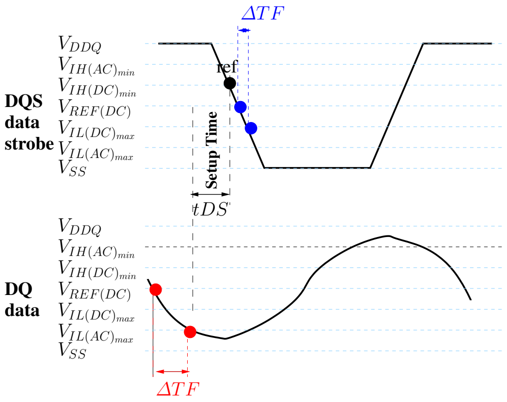


Fig. 25. Relation between the reference point and the corresponding ΔTF of DQ and DQS

```
define b:dqs_in_fsr :=
  ((a:DQS <= 0.9) and (a:DQS >= 0.65))
    since (a:DQS >= 0.9)
```

which holds if the signal is in the falling slew region, as shown in Figure 24.

Note that according to equation (2), DQ and DQS have their slew rates in the range between 1 and $2V/ns$ if their respective ΔTF is between 125 and $250ps$. Moreover, the value of tDS is determined at the crossing of $V_{REF(DC)}$ by DQS from above (point **ref** in Figure 25) with respect to the previous falling setup slew rate of DQ and the next falling setup slew rate of DQS, as shown in Figure 25. Hence, the falling slew rates of DQ and DQS are in the range between 1 and $2V/ns$ if the following formulas hold:

```
define b:dq_slew_rate_in_1_2 :=
  not b:dq_in_fsr since
    (b:dq_in_fsr since[125:250)
      (rise(b:dq_in_fsr)));

define b:dqs_slew_rate_in_1_2 :=
  not b:dqs_in_fsr until
    (b:dqs_in_fsr until[125:250)
      (fall(b:dqs_in_fsr)));

define b:top_left_region :=
    b:dq_slew_rate_in_1_2 and
      b:dqs_slew_rate_in_1_2;
```

Finally, the main property for the falling setup time, provided that DQ and DQS falling slew rates are in the range between 1 and $2V/ns$, is expressed as:

```
define b:dqs_above_vihdcmin :=
  (a:DQS >= 1.025);
define b:dqs_above_vilacmax :=
  (a:DQ >= 0.65);

always ((fall(b:dqs_above_vihdcmin)
  and b:top_left_region)
    -> historically[0:338] not
      fall(b:dq_above_vilacmax));
```

with similar properties that have to be written for each range of DQ and DQS slew rates.

6.2.2 Experimental Evaluation

Property-based monitoring of AMS behaviors is a novel approach and it is worth discussing some methodological aspects related to this case study. The process started by investigating the validation methods that are currently used by analog designers and understanding what are the actual difficulties that they encounter in checking the correctness of their designs. The next step required to identify the type of application whose validation is not fully covered by existing tools and that could benefit from assertion-based monitoring techniques, which led us to consider the DDR2 memory interface. With the help of analog designers we were able to study in detail different properties that are defined in the official DDR2 specification, and consequently understand how to translate them into STLPSL assertions. This preparation process of the case study is difficult to quantify, although it clearly took orders of magnitude more time than the actual writing and evaluation of the assertions that describe DDR2 properties. Despite the length of this pre-processing, it was a

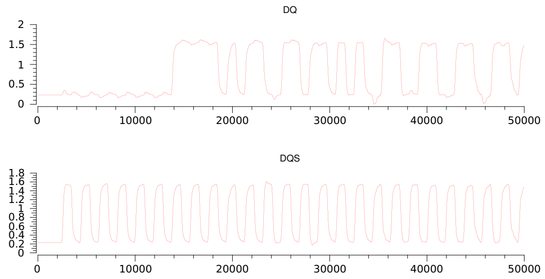


Fig. 26. Segment of DQ and DQS simulation traces

crucial step in understanding relevance, strengths and weaknesses of the property-based analog monitoring framework.

In this case study, we considered a single-ended DDR2-1066 memory interface, which is not yet a JEDEC standard. Hence the exact specification parameters could not be obtained for that particular version of DDR2, and we used instead the official specification parameters for the single-ended DDR2-400 presented in Section 6.2.1, assuming that these parameters would be conservative enough. The simulation traces contained about 180,000 samples per signal. We used the offline monitoring for this case study because the DDR2 simulation traces were already available.

The translation of the alignment property into a set of STLPSL assertions started by splitting the main property into four different ranges, taking an over-approximated tDS value for each slew rate range. The evaluation of each property took about 7 seconds. Since some of the over-approximating properties were shown to be false, we decomposed them further in three iterations into a total of seven properties before being able to show that the simulation traces satisfy the specification. The properties were refined manually and this proved to be a tedious task.

7 Framework Extensions

In this section, we give a brief overview of several recent extensions of the framework presented in this paper. that were recently added to our framework . These extensions, all centered around the STL specification language, are partly motivated by practical needs that were identified, together with engineers and designers, while evaluating our property-based monitoring approach.

7.1 Quantitative Semantics for STL

The evaluation of a signal against an STL property results in a Boolean yes/no answer, and this contrasts with quantitative nature of real-valued variables in continuous systems. AMS designs are subject to noise and numerical errors, which makes the STL monitoring procedure potentially sensitive to small variations in input signals, as discussed in Section 4. In [10], partly inspired by [13], an alternative *quantitative* semantics for STL has been proposed which returns a positive or negative real number whose sign indicates satisfaction/violation and its magnitude stands for the *robustness* of satisfaction: how far a satisfying signal is, in time and space, from violating the property. The quantitative semantics for STL allows for example to capture the variance in time of the expected occurence of an event, where a simple yes or no answer does not give enough information. It turns out that these quantitative measures propagate from sub-formulae to formulae in a similar manner as satisfaction with $\min$ and $\max$ replacing disjunction and conjunction, respectively. An efficient monitoring procedure which extends Algorithm 1 to the quantitative semantics has been implemented in the tool Breach [7] and has been applied to a biological model [9].

7.2 Parameter Identification for STL

In digital systems engineering, the role of specification is quite clear: they express what sequences of input and output events are allowed in the interaction of numerous *well-defined* discrete elements. In the analog setting (and in fact in any engineering disciplines whose components are not abstracted into logical functions) the behaviors of the components themselves is not precisely known as they depend on technology and process variations, are sensitive to particular sizing of transistors and interconnections between elements used within the component. In such a context, a temporal formula can serve as a high-level abstract (and non-deterministic) description of the component, which can be sufficient for proving the correct interaction of the component with the rest of the system. The inverse problem of inferring a formula compatible with a set of simulation traces is referred to as *identification* or *learning*. To support this process we defined in [3], a *parametric* variant of STL, where both *amplitude* (voltage, current) thresholds and *time bounds* can remain unspecified. We developed techniques for automatic extraction of tightest parameter values for which a parametric STL specification is satisfied by a set of simulation traces.

7.3 Time-Frequency Logic

The main motivation for STL is the transfer of semi-formal validation techniques from digital to AMS domain, aiming to formalize, improve and automate current validation practices for AMS designs. Like any other temporal logic, STL is tied to *time-domain* properties of continuous signals and is mainly adapted to express complex temporal patterns between "events" that happen in signals. In the world of signal processing (and the AMS circuits that realize them) many interesting properties are expressed much more naturally in the language of the *frequency-domain*. For example, the filtering of noise from a signal is done by applying the Fourier transform to the signal, removing high-frequency elements from the spectrum and translating back to the time-domain. Moreover, in many applications one needs to combine time and frequency, for example to characterize musical melodies and this can be facilitated by bounded-window transforms such as the Short-Time Fourier Transform or *Wavelets*. A first step toward this fusion of time- and frequency-domain analysis has been recently made in [8] by defining *Time Frequency Logic* (TFL) and implementing its monitoring procedure.

8 Conclusion

In this paper, we overviewed of the property-based monitoring framework for AMS systems centered around the STL specification language. Apart from theoretical foundations for formal specifications of analog and mixed signal behaviors, we presented the AMT tool which implements the monitoring algorithms and two industrial case studies that show the benefits and pitfalls of our approach. This framework has been developed over years and benefitted from interactions with analog designers that helped us understand the real needs in the domain of AMS validation. Many of the our designs choices, as well as the extensions of the framework presented in this paper, result from such interactions.

Although our framework has reached a certain level of maturity, there are many research directions to explore:

- Tighter integration of the monitoring procedure with AMS simulators, by making the simulators aware of the temporal properties and thus steering their simulation steps in order to find counter-examples more efficiently;
- Better explanation of specification violations by causality analysis of simulation traces;
- Further exploration of time-frequency analysis by considering multi-dimensional temporal (and spatial) logic formalisms.

References

1. Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The Benefits of Relaxing Punctuality. *J. ACM*, 43(1):116–146, 1996.
2. Rajeev Alur and Thomas A. Henzinger. Back to the Future: Towards a Theory of Timed Regular Languages. In *FOCS*, pages 177–186, 1992.
3. Eugène Asarin, Alexandre Donzé, Oded Maler, and Dejan Nickovic. Parametric Identification of Temporal Properties. In *RV*, 2011.
4. Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of Real-Time Properties. In *FSTTCS*, pages 260–272, 2006.
5. Edmund M. Clarke, Alexandre Donzé, and Axel Legay. On Simulation-Based Probabilistic Model Checking of Mixed-Analog Circuits. *Formal Methods in System Design*, 36(2):97–113, 2010.
6. Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: Runtime Monitoring of Synchronous Systems. In *TIME*, pages 166–174, 2005.
7. Alexandre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *CAV*, pages 167–170, 2010.
8. Alexandre Donzé, Oded Maler Ezio Bartocci, Dejan Nickovic, Radu Grosu, and Scott Smolka. On Temporal Logics and Signal Processing. In *(submitted)*, 2012.
9. Alexandre Donzé, Eric Fanchon, Lucie Martine Gattepaille, Oded Maler, and Philippe Tracqui. Robustness analysis and behavior discrimination in enzymatic reaction networks. *PLoS ONE*, 6(9):e24246, 09 2011.
10. Alexandre Donzé and Oded Maler. Robust Satisfaction of Temporal Logic over Real-Valued Signals. In *FORMATS*, pages 92–106, 2010.
11. Deepak D'Souza and Nicolas Tabareau. On Timed Automata with Input-Determined Guards. In *FORMATS/FTRTFT*, pages 68–83, 2004.
12. Georgios E. Fainekos, Antoine Girard, and George J. Pappas. Temporal Logic Verification Using Simulation. In *FORMATS*, pages 171–186, 2006.
13. Georgios E. Fainekos and George J. Pappas. Robustness of Temporal Logic Specifications. In *FATES/RV*, pages 178–192, 2006.
14. Georgios E. Fainekos and George J. Pappas. Robustness of Temporal Logic Specifications for Continuous-Time Signals. *Theor. Comput. Sci.*, 410(42):4262–4291, 2009.
15. JESD79-2C. *DDR2 SRAM Specification*. Jedec Standard, 2008.
16. Kevin D. Jones, Victor Konrad, and Dejan Nickovic. Analog Property Checkers: a DDR2 Case Study. *Formal Methods in System Design*, 36(2):114–130, 2010.
17. Chiheb Kossentini and Paul Caspi. Mixed delay and threshold voters in critical real-time systems. In *FORMATS/FTRTFT*, pages 21–35, 2004.
18. Chiheb Kossentini and Paul Caspi. Approximation, Sampling and Voting in Hybrid Computing Systems. In *HSCC*, pages 363–376, 2006.
19. Ron Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*, 2(4):255–299, 1990.
20. Oded Maler and Dejan Nickovic. Monitoring Temporal Properties of Continuous Signals. In *FORMATS/FTRTFT*, pages 152–166, 2004.
21. Oded Maler, Dejan Nickovic, and Amir Pnueli. From MITL to Timed Automata. In *FORMATS*, pages 274–289, 2006.
22. Oded Maler, Dejan Nickovic, and Amir Pnueli. Checking Temporal Properties of Discrete, Timed and Continuous Behaviors. In *Pillars of Computer Science*, pages 475–505, 2008.
23. Pieter J. Mosterman. An Overview of Hybrid Simulation Phenomena and Their Support by Simulation Packages. In *HSCC*, pages 165–177, 1999.
24. Dejan Nickovic and Oded Maler. AMT: a Property-Based Monitoring Tool for Analog Systems. In *FORMATS*, pages 304–319, 2007.
25. Ghiath Al Sammane, Mohamed H. Zaki, Zhi Jie Dong, and Sofiène Tahar. Towards Assertion Based Verification of Analog and Mixed Signal Designs Using PSL. In *FDL*, pages 293–298, 2007.
26. Peter Schrammel and Bertrand Jeannet. From hybrid data-flow languages to hybrid automata: a complete translation. In *HSCC*, pages 167–176, 2012.
27. Verilog AMS 2.3. *Language Reference Manual*. Accellera Systems Initiative, 2008.