AMT: a Property-based Monitoring Tool for Analog Systems*

Dejan Nickovic[1] and Oded Maler[1]

Verimag, 2 Av. de Vignate, 38610 Gières, France
[Dejan.Nickovic | Oded.Maler]@imag.fr

Abstract. In this paper we describe AMT, a tool for monitoring temporal properties of continuous signals. We first introduce STL/PSL, a specification formalism based on the industrial standard language PSL and the real-time temporal logic MITL, extended with constructs that allow describing behaviors of real-valued variables. The tool automatically builds property observers from an STL/PSL specification and checks, in an *offline* or *incremental* fashion, whether simulation traces satisfy the property. The AMT tool is validated through a Flash memory case-study.

1 Introduction

The algorithmic verification field has been centered around the decision procedures for model-checking temporal logic formulae. Temporal logic [MP95] is a rigorous specification formalism used to describe desired behaviors of the system. A number of efficient algorithms for translating temporal logic formulae into corresponding automata have been developed [VW86,SB00,GPVW95,GO01], resulting in the success of logics such as LTL and CTL and their common integration into main verification tools. The temporal logic-based formalisms were adopted by the hardware industry with the standard PSL [HFE04] specification language.

In order to reason about *timed* systems, a number of real-time formalisms have been proposed, either as extensions of temporal logics (MTL [Koy90],MITL [AFH96], TCTL [Y97]) or regular expressions (*timed* regular expressions [ACM02]). However, unlike in the untimed case, there is no simple correspondence between these logics and timed automata [AD94] used in the timed verification tools.

The verification in the *continuous* domain was made possible with the advent of *hybrid automata* [MMP92] as a model for describing systems that have continuous dynamics with switches, and the algorithms for exploring their state-space. Although a lot of progress has been done recently [ADF+06], the scalability still remains a major issue for the exhaustive verification of hybrid systems, due to the explosion of the state space. Moreover, property-based verification of hybrid systems is only at its beginning [FGP06].

Hence, the preferred validation method for continuous systems remains simulation/testing. However, it has been noted that the specification element of verification

* This work was partially supported by the European Community project IST-2003-507219 PROSYD (Property-based System Design).

can be exported to the simulation through property monitors. The essence of this approach is the automatic construction of an observer from the formula in the form of a program that can be interfaced with the simulator and alert the user if the property is violated by a simulation trace. This process is much more reliable and efficient than visual (graphical or textual) inspection of simulation traces, or manual construction of property monitors.

This procedure is called *lightweight verification*, where the property monitor checks whether a finite set of traces satisfy the property specification. In the framework of software *runtime verification*, temporal logic has been used as the specification language in a number of monitoring tools, including Temporal Rover (TR) [Dru00], FoCs [ABG+00], Java PathExplorer (JPaX) [HR01] and MaCS [KLS+02]. The extensions of temporal logics that deal with richer properties were also considered in monitoring tools such as LOLA [DSS+05].

In [MN04], we have introduced STL, a language for relating temporal behaviour of continuous signals via their *static abstractions* and a procedure for offline monitoring of specifications written in STL against continuous input traces. This paper extends the STL logic with an *analog layer* in which one can apply operations on continuous signals directly, as well as the *finitary interpretation* of the temporal operators in the spirit of PSL. The resulting logic is called STL/PSL. The original offline monitoring algorithm is extended to an *incremental* (semi-online) version. The main contribution of this paper is the implementation of a stand-alone Analog Monitoring Tool (AMT) which integrates results presented in [MN04] and this paper. Finally, a case-study on the behaviour of a FLASH memory cell is conducted in order to validate the performance of the tool.

The rest of the document is organized as follows: in Section 2, we introduce the STL/PSL logic along with its semantic domain. Section 3 discusses the offline property checking algorithm from [MN04] and presents its incremental extension. The AMT tool is presented in Section 4 and Section 5 describes the Flash memory case-study. Finally, in Section 6 we conclude with a discussion on the achievements and future work.

2 Signals and Their Temporal Logic

The specification of properties of continuous signals requires an adaptation of the semantic domain and the underlying logic.

2.1 Signals

Let the time domain $\mathbb{T}$ be the set $\mathbb{R}_{\geq 0}$ of non-negative real numbers. A finite length signal ξ over an abstract domain $\mathbb{D}$ is a partial function $\xi : \mathbb{T} \to \mathbb{D}$ whose domain of definition is $I = [0, r), r \in \mathbb{Q}_{>0}$. We say that the length of the signal ξ is r, and denote this fact by $|\xi| = r$. We use the notation $\xi[t] = \bot$ when $t \geq |\xi|$. In this paper, we restrict our attention to two particular types of signals, Boolean signals ξ_b with $\mathbb{D} = \mathbb{B}$, and continuous signals ξ_a with $\mathbb{D} = \mathbb{R}$.

We first present some signal properties that are independent of the signal domain. The *restriction* of a signal ξ to length d is defined as

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

The *concatenation* $\xi = \xi_1 \cdot \xi_2$ of two signals ξ_1 and ξ_2 defined over the intervals $[0, r_1)$ and $[0, r_2)$ is a signal over $[0, r_1 + r_2)$ defined as

$$\xi[t] = \begin{cases} \xi_1[t] & \text{if } t < r_1 \\ \xi_2[t - r_1] & \text{otherwise} \end{cases}$$

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

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

The *Minkowski sum* and *difference* of two sets P_1 and P_2 are defined as

$$\begin{aligned} P_1 \oplus P_2 &= \{x_1 + x_2 : x_1 \in P_1, x_2 \in P_2\} \\ P_1 \ominus P_2 &= \{x_1 - x_2 : x_1 \in P_1, x_2 \in P_2\}. \end{aligned}$$

Signals can also be combined and separated using the standard operations of *pairing* and *projection* defined as

$$\begin{gathered} \xi_1 \,||\, \xi_2 = \xi_{12} \text{ if } \forall t\ \xi_{12}[t] = (\xi_1[t], \xi_2[t]) \\ \xi_1 = \pi_1(\xi_{12}) \ \ \xi_2 = \pi_2(\xi_{12}) \end{gathered}$$

In particular, $\pi_p(\xi)$ will denote the projection of the signal ξ on the dimension with domain $\mathbb{B}$ that corresponds to the proposition p (and likewise $\pi_s(\xi)$ denotes projection of the signal ξ on the dimension with domain $\mathbb{R}$ corresponding to the continuous variable s).

Non-Zeno Boolean signals of finite length admit a finite representation called *interval covering* defined as a sequence of intervals $I_0 \cdot I_1 \cdot \ldots \cdot I_k$ such that the value of ξ_b is constant in every interval, $\xi_b(I_i) = \neg\xi_b(I_{i+1})$ for all $i \in [0, k-1]$, $\bigcup_{i=0}^{k} I_i = I$ and $I_i \cap I_j = \emptyset$ for every $i \neq j$. An interval I is said to be *positive* if $\xi_b(I) = \mathrm{T}$ and *negative* otherwise. An interval covering $\mathcal{I}$ is said to be *consistent* with a signal ξ_b if $\xi_b[t] = \xi_b[t']$ for every t, t' belonging to the same interval I_i. We denote by $\mathcal{I}_{\xi_b}$ the *minimal* interval covering consistent with a finite variability signal ξ_b.

Unlike Boolean signals, continuous signals do not admit an *exact finite representation*. However, numerical simulators usually produce a *finite* collection of sampling pairs $(t, \xi_a[t])$ with t ranging over some interval $[0, r) \subseteq \mathbb{T}$. This finite representation is in contrast to continuous signals defined as *ideal mathematical objects* consisting of an uncountable number of pairs $(t, \xi_a[t])$ for all $t \in [0, r)$. We adopt the approach of representing continuous signals of finite length by using a finite set of sampling points. The signal value at the missing time instants $t \in (t_i, t_{i+1})$ corresponds to the interpolation between sample points $(t_i, \xi_a[t_i])$ and $(t_{i+1}, \xi_a[t_{i+1}])$.

2.2 STL/PSL Specification Language

In this section we describe the STL/PSL logic, as an extension of MITL [AFH96] and STL [MN04] logics. We use a layered approach in the fashion of PSL [HFE04], with the *analog layer* allowing to reason about continuous signals and the *temporal layer* relating the temporal behavior of different input traces. The “communication” between

the two layers is done via *static abstractions* that partition the continuous state space according to the satisfaction of some inequality constraints on the continuous variables.

Since STL/PSL is targeted for specifying properties to be used for *lightweight verification* over *finite* traces, we adopt the finitary interpretation used in PSL, by defining *strong* and *weak* forms of the temporal operators. The strong form of an operator requires the terminating condition to occur before the end of the signal, while the weak form makes no such requirements. In PSL for example, `until!` and `until` represent the strong and the weak forms of the until operator, respectively.

The *analog layer* of STL/PSL is defined by the following grammar:

$$\phi :== s \mid \texttt{shift}(\phi, \texttt{k}) \mid \phi_1 \star \phi_2 \mid \phi \star \texttt{c} \mid \texttt{abs}(\phi)$$

where s belongs to a set $S = \{s_1, s_2, \ldots, s_n\}$ of continuous variables, $\star \in \{\texttt{+}, \texttt{-}, \texttt{*}\}$, $\texttt{c} \in \mathbb{Q}$ and $\texttt{k} \in \mathbb{Q}^+$. Note that the analog operators defined above are the ones currently supported by the AMT tool, but can be easily extended to new ones.

The semantics of the analog layer of STL/PSL is defined as an application of the analog operators to the input signal ξ:

$$\begin{array}{ll} s[t] & = \pi_s(\xi)[t] \\ \texttt{shift}(\phi, \texttt{k})[t] & = \phi[t + \texttt{k}] \\ (\phi_1 \star \phi_2)[t] & = \phi_1[t] \star \phi_2[t] \\ (\phi \star \texttt{c})[t] & = \phi[t] \star \texttt{c} \\ \texttt{abs}(\varphi)[t] & = \begin{cases} \phi[t] & \text{if } \phi[t] \geq 0 \\ -\phi[t] & \text{otherwise} \end{cases} \end{array}$$

The *temporal layer* of STL/PSL is defined as follows:

$$\begin{array}{c} \varphi :== p \mid \phi \circ \ \texttt{c} \mid \texttt{not}\ \varphi \mid \varphi_1\ \texttt{or}\ \varphi_2 \mid \texttt{eventually!}\ \varphi \mid \\ \texttt{eventually![a:b]}\ \varphi \mid \texttt{eventually[a:b]}\ \varphi \mid \\ \varphi_1\ \texttt{until!}\ \varphi_2 \mid \varphi_1\ \texttt{until![a:b]}\ \varphi_2 \end{array}$$

where p belongs to a set $P = \{p_1, p_2, \ldots, p_n\}$ of propositional variables, $\texttt{a}, \texttt{b}, \texttt{c} \in \mathcal{Q}$ and $\circ \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}$. Note that we include explicitly in the syntax *weak* and *strong* versions of *eventually* operators[1].

The satisfaction relation $(\xi, t) \models \varphi$, indicating that signal ξ satisfies φ at time t is defined inductively as follows:

[1] Untimed eventually exists only in its strong form. Weak eventually is trivially satisfied by any finite trace ξ

$(\xi, t) \models p$ iff $\pi_p(\xi)[t] = \mathrm{T}$
$(\xi, t) \models \phi \circ \mathtt{c}$ iff $\phi[t] \circ \mathtt{c}$
$(\xi, t) \models \mathtt{not}\ \varphi$ iff $(\xi, t) \not\models \varphi$
$(\xi, t) \models \varphi_1\ \mathtt{or}\ \varphi_2$ iff $(\xi, t) \models \varphi_1$ or $(\xi, t) \models \varphi_2$
$(\xi, t) \models \mathtt{eventually!}\ \varphi$ iff $\exists t' \geq t\ \mathtt{st}\ t' < |\xi|\ \mathtt{and}\ (\xi, t') \models \varphi$
$(\xi, t) \models \mathtt{eventually![a:b]}\ \varphi$ iff $\exists t' \in t \oplus [a, b]\ \mathtt{st}\ t' < |\xi|\ \mathtt{and}\ (\xi, t') \models \varphi$
$(\xi, t) \models \mathtt{eventually[a:b]}\ \varphi$ iff $\exists t' \in t \oplus [a, b]\ \mathtt{st}\ t' \geq |\xi|\ \mathtt{or}\ (\xi, t') \models \varphi$
$(\xi, t) \models \varphi_1\ \mathtt{until!}\ \varphi_2$ iff $\exists t' \geq t$ st $t' < |\xi|$ and $(\xi, t') \models \varphi_2$ and $\forall t'' \in [t, t']\ (\xi, t'') \models \varphi_1$
$(\xi, t) \models \varphi_1\ \mathtt{until![a:b]}\ \varphi_2$ iff $\exists t' \in t \oplus [a, b]$ st $t' < |\xi|$ and $(\xi, t') \models \varphi_2$ and $\forall t'' \in [t, t']\ (\xi, t'') \models \varphi_1$

An STL/PSL specification φ_{prop} is an STL/PSL temporal formula. The signal ξ satisfies the specification φ_{prop}, denoted by $\xi \models \varphi_{\text{prop}}$, iff $(\xi, 0) \models \varphi_{\text{prop}}$. Note that our definition of the semantics of the *until* and timed *until* operators differs slightly from their conventional definition since it requires a time instant t where both $(\xi, t) \models \varphi_2$ and $(\xi, t) \models \varphi_1$. From the basic STL/PSL operators, one can define standard Boolean and temporal operators, namely *always* and *weak until*, as well as *weak* and *strong* forms of timed *always* operators.

A large part of analog design is based on comparing waveforms (signals) with some reference signal that specify a desired behavior. These notions are formalized using a distance function (metric) which quantifies numerically the resemblance of two signals. Mathematically speaking, a metric space is a pair (X, d) such that X is the domain and $d : X \times X \to \mathbb{R}_+$ is a function satisfying: $d(x, x) = 0$; $d(x, y) = d(y, x)$ and $d(x, y) + d(y, z) \geq d(x, z)$. There are many ways to define distance functions on waveforms, by taking the maximum of the pointwise distance at every time t, summing/integrating over the pointwise distance, etc. Once such a distance d is defined, it can be used to define distance-based logical operators of the form $d(\xi, \xi') < c$ for some positive constant c. Below we define three such operators, the first is based on the maximal pointwise distance while the two others are based on the metric defined in [KC06a] which "tolerates" large pointwise deviations between the two signals if they last for a time shorter than `t` and occur at most once every `T-t` units. As one can see these operators constitute a syntactic sugar as they can be expressed in STL/PSL.

$\mathtt{distance}(\phi_1, \phi_2, \mathtt{c}) = \mathtt{abs}(\phi_1\text{-}\phi_2)\ \mathtt{<=\ c}$
$\mathtt{distance}(\phi_1, \phi_2, \mathtt{c}, \mathtt{t}, \mathtt{T}) = \mathtt{abs}(\phi_1\text{-}\phi_2)\ \mathtt{>\ c\ ->\ eventually![<=t]}\ \mathtt{always[<=T\text{-}t]}(\mathtt{abs}\ (\phi_1\text{-}\phi_2)\ \mathtt{<=\ c})$
$\mathtt{distance}(\varphi_1, \varphi_2, \mathtt{t}, \mathtt{T}) = (\varphi_1\ \mathtt{xor}\ \varphi_2)\mathtt{->\ eventually![<=t]}\ \mathtt{always[<=T\text{-}t]}\ (\varphi_1\ \mathtt{iff}\ \varphi_2)$

3 Checking STL/PSL Properties

In this section we describe two algorithms for checking STL/PSL properties. Both algorithms are based on a process that we call *marking*, namely determining truth value

of each subformula at every time instant t. The marking is a doubly-recursive process going from the atomic propositions upward to the top formula, and, due to the nature of future temporal logic, from truth values at time t to truth values at time $t' \leq t$. The marking process terminates when the value of the top formula at time 0 is determined.

Offline marking: This procedure assumes that the multi-dimensional input signal ξ is already available, and the marking procedure is applied to the entire signal, propagating backward at once the values of subformulae, up to obtaining the truth value of the main formula.

Incremental marking: The incremental procedure updates the marking each time a new segment of the input signal is observed. It is useful in detecting early violation of an STL/PSL property and can be applied in parallel with the simulation process. It can also be used for monitoring real, rather than simulated systems.

The offline marking procedure takes as arguments a temporal STL/PSL specification φ_{prop} and the input signal ξ that we treat as a global data structure and do not pass it explicitly as an argument to the procedure. The algorithm computes, from the bottom-up, a signal $\chi_\psi(\xi)$ for each subformula ψ of φ_{prop}.[2] If ψ is a temporal STL/PSL formula φ, $\chi_\varphi(\xi)$ is called the *satisfaction signal*. This signal satisfies $\chi_\varphi(\xi)[t] = 1$ iff $(\xi, t) \models \varphi$. If ψ is a formula ϕ from the analog layer of STL/PSL, $\chi_\phi(\xi)$ is the result of applying the operator ϕ to the (continuous) signal ξ. Whenever the identity of ξ is clear from the context, we will use the shorthand notation χ_ψ.

The algorithm is decomposed into two methods OFFLINE-T and OFFLINE-A as shown in Algorithm 1, computing the χ_ψ corresponding to the formula ψ from the temporal and the analog layer of STL/PSL, respectively. The top level formula φ_{prop} is monitored by invoking OFFLINE-T(φ_{prop}).

Algorithm 1: OFFLINE-T and OFFLINE-A

```
input : STL/PSL Temporal Formula φ and signal ξ
switch φ do
    case p
        χ_φ := π_p(ξ);
    end
    case φ ∘ c
        OFFLINE-A (φ);
        χ_φ := COMBINE(∘c, χ_φ);
    end
    case OP_1(φ_1)
        OFFLINE-T (φ_1);
        χ_φ := COMBINE(OP_1, χ_φ1));
    end
    case OP_2(φ_1, φ_2)
        OFFLINE-T (φ_1, φ_2);
        χ_φ := COMBINE(OP_2, χ_φ1, χ_φ2));
    end
end

input : STL/PSL Analog Formula ϕ and signal ξ
switch ϕ do
    case s
        χ_ϕ := π_s(ξ);
    end
    case OP_1(ϕ_1)
        OFFLINE-A(ϕ_1);
        χ_ϕ := COMBINE(OP_1, χ_ϕ1);
    end
    case OP_2(ϕ_1, ϕ_2)
        OFFLINE-A(ϕ_1, ϕ_2);
        χ_ϕ := COMBINE(OP_2, χ_ϕ1, χ_ϕ2);
    end
end
```

[2] The notation ψ is used whenever it is not important whether ψ is a temporal or an analog layer formula

Most of the work is done in the COMBINE procedure which takes one or two signals (possibly of different length) and computes from them a new signal based on the specific operation. The approach is based on [MN04] with some extensions to deal with both strong and weak operators. We illustrate the procedure on few representative operations:

$\chi_\varphi :=$ **COMBINE**(**or**$, \chi_{\varphi_1}, \chi_{\varphi_2})$ For the disjunction we first construct a refined interval covering $\mathcal{I} = \{I_1, \ldots, I_k\}$ for $\chi_{\varphi_1} || \chi_{\varphi_2}$ so that the mutual values of both signals become uniform in every interval. Then we compute the disjunction interval-wise, that is, $\varphi(I_i) = \varphi_1(I_i) \vee \varphi_2(I_i)$. Finally we merge adjacent intervals having the same Boolean value to obtain the minimal interval covering $\mathcal{I}_{\chi_\varphi}$.

$\chi_\varphi :=$ **COMBINE**(**eventually![a,b]**$, \chi_{\varphi_1})$ For every positive interval $I \in \chi_{\varphi_1}$ we compute its *back shifting* $I \ominus [a, b] \cap \mathbb{T}$ and insert it to χ_φ. Overlapping positive intervals in χ_φ are merged to obtain a minimal consistent interval covering. In the process, all the negative intervals shorter than $b - a$ disappear.[3]

$\chi_\phi :=$ **COMBINE**$(\star, \chi_{\phi_1}, \chi_{\phi_2})$ For the pointwise arithmetic operations on continuous signals χ_{ϕ_1} and χ_{ϕ_2}, we first take the union of their sampling points and extend each signal to the new points by interpolation. The signal $\chi_{\phi_1 \star \phi_2}$ is computed by applying the pointwise arithmetic operation to each pair of corresponding sampling points. An example of the arithmetic operation is shown in Figure 1.

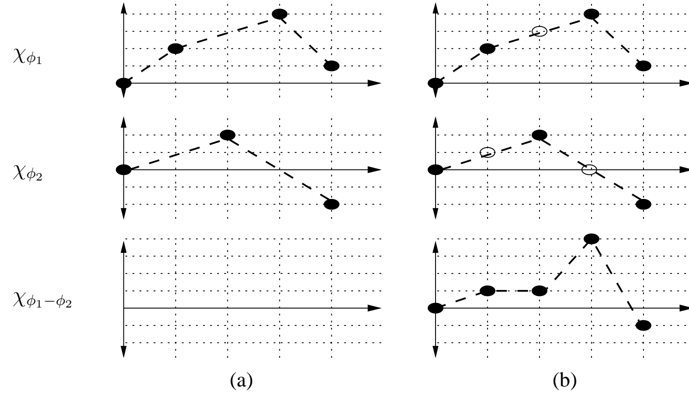

Fig. 1. Combining $\phi = \phi_1 - \phi_2$: (a) Input signals χ_{ϕ_1} and χ_{ϕ_2} sampled at different rates; (b) Refinement of χ_{ϕ_1} and χ_{ϕ_2} and computation of $\chi_{\phi_1 - \phi 2}$

Incremental marking is performed using a kind of piecewise-online procedure invoked each time a new segment of ξ, denoted by Δ_ξ, is observed. For each subformula ψ the algorithm stores its already-computed associated signal partitioned into a concatenation of two signals $\chi_\psi \cdot \Delta_\psi$ with χ_ψ consisting of values already propagated to

[3] Another way to see it is as shifting the *negative* intervals by $[b, a]$.

the super-formula of ψ, and Δ_ψ, consisting of values that have already been computed but which have not yet propagated to the super-formula and can still influence it.

Initially all signals are empty. Each time a new segment Δ_ξ is read, a recursive procedure similar to the offline one is invoked, which updates every χ_ψ and Δ_ψ from the bottom up. The difference with respect to the offline algorithm is that only segments of the signal that has not been propagated upwards participate in the update of their super-formulae. This may result in a considerable saving when the signal is very long.

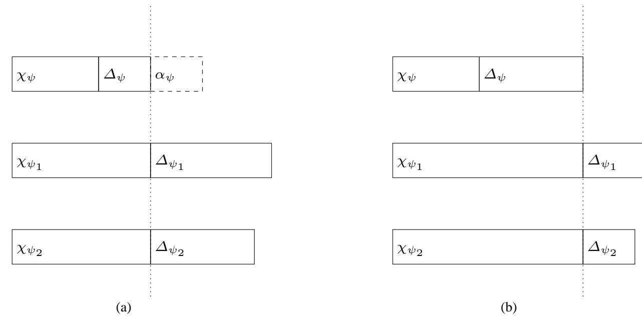

Fig. 2. A step in an incremental update: (a) A new segment α_ψ for ψ is computed from Δ_{ψ_1} and Δ_{ψ_2}; (b) α_ψ is appended to Δ_ψ and the endpoints of χ_{ψ_1} and χ_{ψ_1} are shifted forward accordingly.

As an illustration consider $\psi = \text{OP}(\psi_1, \psi_2)$ and the corresponding truth signals of Figure 2-(a). Before the update we always have $|\chi_\psi \cdot \Delta_\psi| = |\chi_{\psi_1}| = |\chi_{\psi_2}|$: the parts Δ_{ψ_1} and Δ_{ψ_2} that may still affect ψ are those that start at the point from which the value of χ_ψ is still unknown. We apply COMBINE procedure on Δ_{ψ_1} and Δ_{ψ_2} to obtain a new (possibly empty) segment α_ψ of Δ_ψ. This segment is appended to Δ_ψ in order to be propagated upwards, but before that we need to shift the borderline between χ_{ψ_1} and Δ_{ψ_1} (as well as between χ_{ψ_2} and Δ_{ψ_2}) in order to reflect the update of Δ_ψ. The procedure is detailed in Algorithm 2.

Note that if $\chi_{\varphi_{\text{prop}}}$ becomes determined for time 0, the incremental procedure can be stopped. The finitary interpretation of temporal operators is used only if $\chi_{\varphi_{\text{prop}}}$ has not been determined after the end of simulation.

4 Overview of the AMT tool

AMT is a stand-alone tool with a graphical user interface which implements the above algorithms with respect to sampled continuous signal inputs. AMT was written in C++ for GNU/Debian Linux x86 machines. The user interface is based on the library QT[4], while QWT[5] was used for visualizing plots.

[4] http://www.trolltech.com
[5] http://qwt.sourceforge.net

Algorithm 2: INCREMENTAL-T and INCREMENTAL-A

```
input : STL/PSL Temporal Formula φ and increment Δ_ξ
switch φ do
    case p
        Δ_φ := Δ_φ · π_p(Δ_ξ);
    end
    case φ ∘ c
        INCREMENTAL-A (φ);
        α_φ := COMBINE(∘c, χ_φ);
        d := |α_φ| ;
        Δ_φ := Δ_φ · α_φ ;
        χ_φ := χ_φ · ⟨Δ_φ⟩_d ;
        Δ_φ := d\Δ_φ ;
    end
    case OP1(φ1)
        INCREMENTAL-T (φ1);
        α_φ := COMBINE(OP1, χ_φ1));
        d := |α_φ| ;
        Δ_φ := Δ_φ · α_φ ;
        χ_φ1 := χ_φ1 · ⟨Δ_φ1⟩_d ;
        Δ_φ1 := d\Δ_φ1 ;
    end
    case OP2(φ1, φ2)
        INCREMENTAL-T (φ1, φ2);
        α_φ := COMBINE(OP2, χ_φ1, χ_φ2));
        d := |α_φ| ;
        Δ_φ := Δ_φ · α_φ ;
        χ_φ1 := χ_φ1 · ⟨Δ_φ1⟩_d ;
        Δ_φ1 := d\Δ_φ1 ;
        χ_φ2 := χ_φ2 · ⟨Δ_φ2⟩_d ;
        Δ_φ2 := d\Δ_φ2
    end
end

input : STL/PSL Analog Formula ϕ and increment Δ_ξ
switch ϕ do
    case s
        Δ_ϕ := Δ_ϕ · π_s(Δ_ξ);
    end
    case OP1(ϕ1)
        INCREMENTAL-A(ϕ1);
        α_ϕ := COMBINE(OP1, χ_ϕ1);
        d := |α_ϕ| ;
        Δ_ϕ := Δ_ϕ · α_ϕ ;
        χ_ϕ1 := χ_ϕ1 · ⟨Δ_ϕ1⟩_d ;
        Δ_ϕ1 := d\Δ_ϕ1 ;
    end
    case OP2(ϕ1, ϕ2)
        INCREMENTAL-A(ϕ1);
        INCREMENTAL-A(ϕ2);
        α_ϕ := COMBINE(OP2, χ_ϕ1, χ_ϕ2);
        d := |α_ϕ| ;
        Δ_ϕ := Δ_ϕ · α_ϕ ;
        χ_ϕ1 := χ_ϕ1 · ⟨Δ_ϕ1⟩_d ;
        Δ_ϕ1 := d\Δ_ϕ1 ;
        χ_ϕ2 := χ_ϕ2 · ⟨Δ_ϕ2⟩_d ;
        Δ_ϕ2 := d\Δ_ϕ2 ;
    end
end
```

The main window of the application is partitioned into five frames that allow the user to manage STL/PSL properties and input signals, evaluate the correctness of the simulation traces with respect to a specification and finally visualize the results. The **property edit** frame contains a text editor for writing, importing and exporting STL/PSL 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 STL/PSL specification is imported into the **property evaluation** frame for its monitoring with respect to a set of input simulation traces, in either *offline* or *incremental* modes. The static import of the input traces is done via 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 3.

4.1 Property Management

The specifications in AMT are written in a simple editor with syntax highlighting for the extended STL/PSL language described below. An STL/PSL specification is then transformed into a structure adapted for the monitoring purpose, 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.

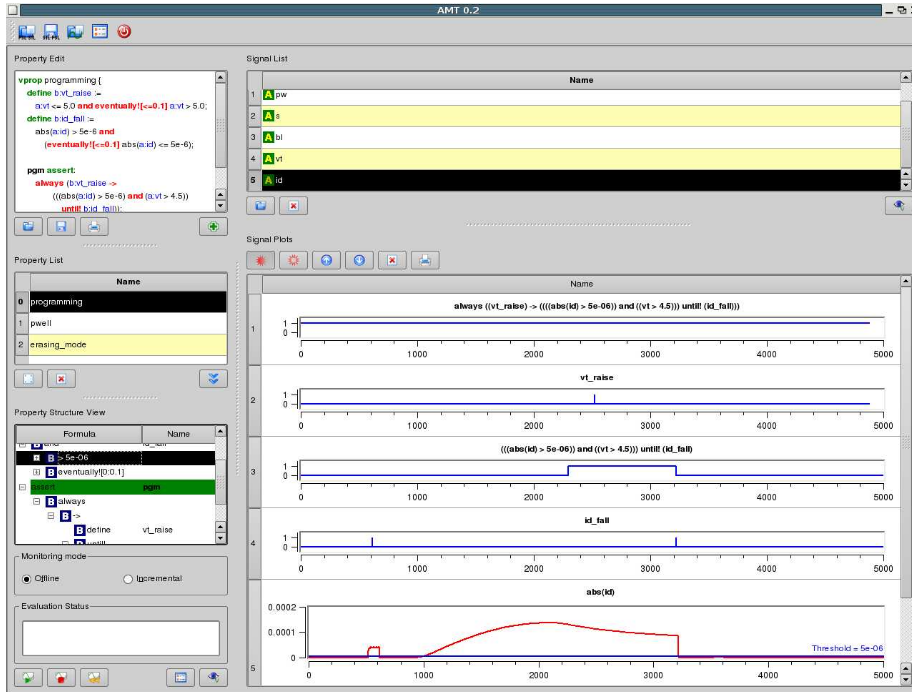

Fig. 3. AMT Main Window

Property Format AMT tool extends the STL/PSL language described in Section 2.2 with additional constructs that simplify the process of property specification. Each top-level STL/PSL property is declared as an *assertion*, and a number of assertions can be grouped into a single logical unit in order to monitor them together at once. We also add a definition directive which allows the user to declare a formula and give it a name, and then refer to it as a variable within the assertions. The extended STL/PSL is defined with the following production rules

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

define_directive :==
  define b:NAME := stl_psl_property
  | define a:NAME := analog_expression

assert_directive :==
  NAME assert : stl_psl_property
```

where `stl_psl_property` and `analog_expression` correspond to φ and ϕ from Section 2.2, respectively.

Property Evaluation The correctness of an STL/PSL specification with respect to input traces is monitored through the property evaluation frame. The frame shows the set of assertions in a tree view, following the parse structure of the formula. The user can choose between *offline* and *incremental* evaluation of the specification.

In the offline case, the input signals are fetched from the signal list frame and the assertions are checked with respect to them. 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 *timeout* value that defines the period between two consecutive evaluations. In between two such periods, the monitor accumulates input received from the simulator. There are three manners to end the incremental monitoring procedure: 1) All assertions become determined and AMT stops the evaluation and closes the connection with the simulator; 2) The special termination packet is received from the simulator and 3) The user explicitly stops the procedure via the GUI.

AMT shows visually the evaluation result of an assertion, choosing a different color scheme for *undetermined*, *correct* and *incorrect* assertions. Each subformula of the specification has an associated signal with it, which can be visualized within the signal plots frame. The visualization of the associated signals can be used for understanding why an assertion holds/fails. During the incremental evaluation, all the signals within the signal plots frame are updated in real-time as new results are computed. The user can switch off the accumulation of intermediate results for better memory performance, thus discarding signals as soon as they are not needed anymore for the evaluation of super-formulae. In that case, the only output of the tool is the final answer.

4.2 Signal Management

The signals in AMT can be either continuous or Boolean. Signals are input traces that can be imported into the tool in an offline or incremental fashion. But signals are also associated to each subformula of an STL/PSL specification. The user can visualize them from the signal plots frame.

Offline Signal Input Signals can be statically loaded from the signal list frame. Two file formats are currently supported by AMT:

- **out** The output format of the Nanosim simulations. 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.

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.

5 A FLASH Memory Case Study

The subject of the case study is the “Tricky” technology FLASH memory test chip in 0.13*us* process developed in ST Microelectronics Italy. The FLASH memory presents an advantage for the analog case study, in that it is a digital system whose logical behavior is implemented at the analog level. Hence, it is a good link between the analog and the digital world.

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

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

The memory cell was simulated in the *programming* and the *erasing* modes for the case study, with the simulation time being 5000 *us* and 30000 *us* respectively. Four STL/PSL properties were written 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.

A detailed description of the properties and the monitoring results can be found in [NMF+06]. As an example, we consider the *erasing property*. The informal description of the property first defines the erasing condition, which is characterized by the wordline signal **wl** being lower than -6 and p-well **pw** above 5. Whenever the 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 corresponding STL/PSL specification is:

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

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

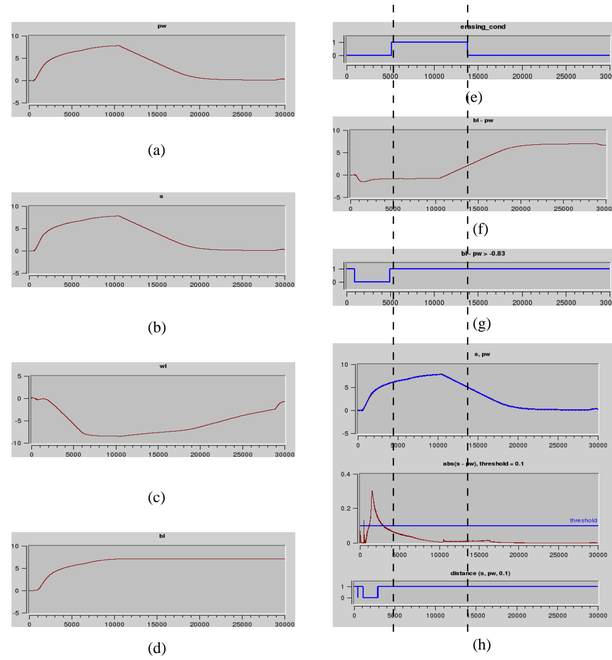

Fig. 4. Erasing Property: (a) pw; (b) s; (c) wl; (d) bl; (e) erasing_cond; (f) bl-pw; (g) bl-pw>=-0.83; (h) distance(s,pw,0.1)

Figure 4 shows some of the representative signals of the erasing property. We can mainly see that, whenever the *erasing condition* in Figure 4 (e) holds (denoted between two dashed lines), the pointwise distance between **s** and **pw** remains smaller than 0.1 (Figure 4 (h)) and the difference between **bl** and **pw** stays above the -0.83 threshold.

5.1 Tool Evaluation

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 [MN04] where k is the number of sub-formulae and m is the number of intervals.

Table 1 shows the size of the input signals (number of intervals). 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 number of intervals 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 simulator which, in most cases, is much more com-

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

Table 1. Input Size

property	time (s)	# intervals
programming1	0.14	99715
programming2	0.42	405907
p-well	0.12	89071
decay	0.50	594709
erasing	2.35	2968578

Table 2. Offline Algorithm Evaluation

Property	Offline t = total # intervals	Incremental m = max # active intervals	m/t * 100
programming1	99715	65700	65.9
programming2	594709	242528	40.8
p-well	89071	8	0.01
decay	594709	279782	47.1

Table 3. Offline/Incremental Space Requirement Comparison

putationally demanding. In fact, one major attraction of the incremental procedure is the 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 intervals generated by the tool while for the latter we take the maximal number of intervals kept simultaneously in memory. We can see that this ratio varies a lot from one property to another, going from 0.01% up to 70%. The general observation is that pointwise operators require less memory in the incremental mode, while properties involving the nesting of untimed temporal properties often fail to discard their inputs until the end of the simulation.

6 Conclusions

The main contribution of this paper is the implementation of the AMT tool that monitors temporal properties of continuous and mixed signals. The specification language for describing desired behaviors of continuous signals supported by the tool is STL/PSL, a subset of PSL, properly extended to express sequential properties of such signals. The monitoring algorithms used by AMT are the offline marking procedure from [MN04] and its incremental extension described in this paper. The tool is integrated with numerical simulators by supporting some standard input formats for continuous simulations and by direct communication between the two using a simple protocol built on top of TCP/IP.

AMT was validated through a FLASH memory case-study. The results show that the tool can be effectively used in both its offline and incremental modes. A number of in-

teresting properties concerning transient behavior of continuous signals were described in STL/PSL. Combinations of operators from the analog and temporal layers allow expressing properties such as ramp detection in an input trace, conditional distance-based comparisons between a reference and an input signal, or a stabilization of an input signal around an arbitrary value. The main class of properties that cannot be expressed in STL/PSL are those dealing with the frequency spectrum of signals. A typical English specification of such a property would be ”At least 60% of the energy power spectrum of a signal is within its frequency band between 300 and 1500Hz”. We hope to introduce such properties into future versions of the tool.

Acknowledgments We would like to thank Andrea Fedeli, Pierluigi Daglio and Davide Lena from ST Microelectronics Italy, for providing us with the simulations of the FLASH memory cell and their precious help in formulating the properties. We would also like to thank Qiang Lu from Synopsis for providing us with a description of the Nanosim output file format.

References

[ABG+00] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal. FoCs: Automatic Generation of Simulation Checkers from Formal Specifications. In *Proc. CAV'00*, pages 538–542. LNCS 1855, Springer, 2000.

[ACM02] E. Asarin, P. Caspi and O. Maler, Timed Regular Expressions, *The Journal of the ACM* **49**, 172–206, 2002.

[AD94] R. Alur and D.L. Dill, A Theory of Timed Automata, *Theoretical Computer Science* **126**, 183–235, 1994.

[ADF+06] E. Asarin, T. Dang, G. Frehse, A. Girard, C. Le Guernic and O. Maler, Recent Progress in Continuous and Hybrid Reachability Analysis, *CACSD*, 2006.

[AFH96] R. Alur, T. Feder, and T.A. Henzinger, The Benefits of Relaxing Punctuality, *Journal of the ACM* **43**, 116–146, 1996 (first published in *PODC'91*).

[Dru00] D. Drusinsky. The Temporal Rover and the ATG Rover. In *Proc. SPIN'00*, pages 323–330. LNCS 1885, Springer, 2000.

[DSS+05] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra and Z. Manna, LOLA: Runtime Monitoring of Synchronous Systems, In *TIME'05*, 166–174, 2005

[FGP06] G. Fainekos, A. Girard and G. Pappas Temporal Logic Verification Using Simulation In *Proc. FORMATS'06*, pages 171–186. LNCS 4202, Springer, 2006.

[GO01] P. Gastin and D. Oddoux, Fast LTL to Büchi Automata Translation, *CAV'01*, 53–65, LNCS 2102, 2001.

[GPVW95] R. Gerth, D.A. Peled, M.Y. Vardi and P. Wolper, Simple On-the-fly Automatic Verification of Linear Temporal Logic, *PSTV*, 3–18, 1995.

[HFE04] J. Havlicek, D. Fisman and C. Eisner, Basic results on the semantics of Accellera PSL 1.1 foundation language, *Technical Report 2004.02*, Accelera, 2004.

[HR01] K. Havelund and G. Rosu. Java PathExplorer - a Runtime Verification Tool. In *Proc. ISAIRAS'01*, 2001.

[KC06a] C. Konsentini and P. Caspi, Sampling and Voting in Hybrid Computing Systems In *Proc. HSCC'06*.

[KLS+02] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, Checking, and Steering of Real-time Systems. In *Proc. RV'02*. ENTCS 70(4), 2002.

[Koy90] R. Koymans, Specifying Real-time Properties with Metric Temporal Logic, *Real-time Systems* **2**, 255–299, 1990.

[MMP92] O. Maler, Z. Manna and A. Pnueli, From Timed to Hybrid Systems *Real-Time: Theory in Practice*, 447-484, LNCS 600, 1992.

[MN04] O. Maler and D. Nickovic, Monitoring Temporal Properties of Continuous Signals, *FORMATS/FTRTFT'04*, 152-166, LNCS 3253, 2004.

[NMF+06] D. Nickovic, O. Maler, A. Fedeli, P. Daglio and D. Lena, *Analog Case Study*, PROSYD Deliverable D3.4/2, 2006. http://www.prosyd.org/twiki/pub/Public/DeliverablePageWP3/prosyd3.4_2.pdf

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[SB00] F. Somenzi and R. Bloem, Efficient Büchi automata from LTL formulae, *CAV'00*, 248–263, LNCS 1855, 2000. 1855.

[VW86] M.Y. Vardi and P. Wolper, An Automata-theoretic Approach to Automatic Program Verification, *LICS'86*, 322–331, 1986.

[Y97] S. Yovine, Kronos: A Verification Tool for Real-time Systems, *International Journal of Software Tools for Technology Transfer* **1**, 123–133, 1997.