

FP6-IST-507219

PROSYD:

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Manual for Property-based automatic generation of simulation monitors for digital, timed, and analog designs (Deliverable 3.2/13)

Due date of deliverable: January 1, 2007
Actual submission date: January 1, 2007

Start date of project: January 1, 2004

Duration: Three years

Organisation name of lead contractor for this deliverable: Verimag

Revision 1.0

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	<input checked="" type="checkbox"/>
PP	Restricted to other programme participants (including the Commission Services)	<input type="checkbox"/>
RE	Restricted to a group specified by the consortium (including the Commission Services)	<input type="checkbox"/>
CO	Confidential, only for members of the consortium (including the Commission Services)	<input type="checkbox"/>

Notices

For information, contact Oded Maler maler@imag.fr.

This document is intended to fulfil the contractual obligations of the PROSYD project concerning deliverable 3.2/13 described in contract number 507219.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

© Copyright PROSYD 2006. All rights reserved.

Table of Revisions

Version	Date	Description and reason	By	Affected sections
0.5	December 11, 2006	Complete Draft	Nickovic	All
0.6	December 28, 2006	Updates	Nickovic	All
1.0	December 31, 2006	Final Version	Nickovic	All

Authors

Dejan Nickovic
Oded Maler

Executive Summary

Common property verification tools are usually designed for checking the correctness of discrete, and to a lesser extent real-time systems. The STL/PSL Monitor tool extends the formal methods to the continuous and mixed-signal domain by providing lightweight verification of properties on continuous and Boolean dense signals expressed in the STL/PSL logic.

Purpose

The purpose of this document is to describe the work done on the lightweight verification of properties on continuous and dense Boolean signals, and to provide the user manual for the STL/PSL Monitor tool.

Intended Audience

This document is intended for formal methods researchers who are interested in the analysis of timed and analog systems and who want to apply property-based lightweight verification to the analog and mixed-signal designs.

Background

This document presents the implementation of the algorithms described in the Deliverable 3.2/6 [MNP06] and published in [MN04]. The specification language used by the tool is STL/PSL described in PROSYD Deliverables 1.3/1[M05] and 1.3/2[NM⁺06].

Contents

Table of Revisions	iii
Authors	iii
Executive Summary	iii
Purpose	iii
Intended Audience	iii
Background	iii
Contents	v
Table of Figures	vi
Glossary	vii
1 Introduction	1
2 STL/PSL Monitor Tool Manual	3
2.1 Signal Management	3
Offline Signal Input	4
Incremental Signal Input	4
2.2 Property Management	5
Property Format	6
Property Editing	6
Property List	7
Property Evaluation	8
2.3 Visualization of Results	10
3 Installation	13
3.1 System Requirements	13
3.2 Licence Issues	13
3.3 Installation	13
4 Implementation	15
5 Underlying Theory	17
5.1 Signal Representation	17
5.2 Checking STL/PSL Properties	19
STL/PSL Temporal Logic	20
Offline Marking	21
Combine Function	21
Incremental Marking	23
Optimization of Boolean operators	25
Freeze operator	25
Approximating events	27
6 Conclusion	29
7 References	31

Table of Figures

Figure 1 - STL/PSL Monitor Main Window	3
Figure 2 - Signal Load Frame	4
Figure 3 - Signal List Frame with Signal Names Changed	5
Figure 4 - Incremental Signal Packet Format	5
Figure 5 - Example STL/PSL Property	7
Figure 6 - Property Editor Frame	7
Figure 7 - Property List Frame	8
Figure 8 - Property Evaluation Frame	9
Figure 9 - Analog Plot Frame	11
Figure 10 - Boolean Plot Frame	12
Figure 11 - Distance Plot Frame	12
Figure 12 - STL/PSL Monitor Tool Architecture.....	16
Figure 13 - A signal ξ and its unitary decomposition (ξ^1, ξ^2, ξ^3)	18
Figure 14 - Marking for $p \cup_{[a,b]} q$ via marking for $\diamond_{[a,b]}(\pi \wedge p) \wedge p$: (a) with non-unitary signals we obtain wrong results; (b) with a unitary decomposition of p and q we obtain correct results. The computation with p_1 is omitted as it has an empty intersection with q	24
Figure 15 - A step in an incremental update: (a) A new segment α for φ is computed from Δ_{φ_1} and Δ_{φ_2} ; (b) α is appended to Δ_{φ} and the endpoints of χ_{φ_1} and χ_{φ_2} are shifted forward accordingly.	25
Figure 16 - Boolean Optimization: (a) Before the update (2) χ_{φ_2} is updated, and $\chi_{\varphi_1 \wedge \varphi_2}$ accordingly	26
Figure 17 - Freeze Operator	27
Figure 18 - Approximation of $raise(p)$	28

Glossary

Exhaustive verification

The process of proving the correctness of a system with respect to a formal specification by exhaustively exploring its mathematical model.

GUI

Graphical User Interface

Incremental marking

The lightweight verification by marking signals that are not available offline, but are rather fed to the verification tool incrementally. It is used mainly for online monitoring of ongoing simulations.

Lightweight verification

The procedure for proving the correctness of a single finite execution of a system with respect to a formal specification.

LTL

Linear-time Temporal Logic. A formal language commonly used to specify the properties that a system has to satisfy.

MITL

Metric Interval Temporal Logic. The dense-time extension of the LTL logic, allowing modalities ranging over a non-punctual interval.

Monitoring

See **lightweight verification**.

Offline marking

The lightweight verification by marking signals that are already available offline in one step. It is usually used for monitoring the results of simulations that are already available and stored on a device.

PSL

Property Specification Language. The formal language based on LTL and regular expressions upon which PROSYD is based.

STL/PSL

Signal Temporal Logic/Property Specification Language. The analog extension of PSL

TCP/IP

Transmission Control Protocol/Internet Protocol

Verification by Marking

Process of evaluating whether a set of signals satisfy a specified property. Unlike automata-based approaches, the lightweight verification by marking manipulates signals directly. See also **offline** and **incremental marking**.

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. Logics like LTL (linear-time temporal logic) or CTL (computation-tree logic) are commonly accepted and used in tools for the verification of discrete systems, and underly the industrial standard PSL [KCV04, HFE04] language. The success of temporal logics as property specification formalism used in verification is mainly due to the development of efficient algorithms for translating formulae into corresponding automata [VW86, SB00, GPVW95, GO01].

When considering *timed* models, a number of variants of real-time logics have been proposed, such as MTL [Koy90], MITL [AFH96], TCTL [Y97], as well as *timed regular expressions* [ACM02]. However, the correspondance between these simply defined logics and timed automata [AD94] (automata augmented with auxiliary variables called *clocks*) is not as simple as for the untimed case. As a consequence, until recently the only logic integrated into a timed verification tool was TCTL, used in KRONOS [Y97].

The verification in the *continuous* domain was made possible with the advent of *hybrid automata* as a model for describing systems that have continuous dynamics with switches, and the algorithms for exploring their state-space. However, the scalability is still 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 at its beginning, lacking proper specification formalisms dealing with continuous variables.

Hence, the preferred validation method for continuous systems remains simulation/testing. However, it has been noted that the specification element of verification can be exported to the simulation through property monitors. The essence of this approach is the automatic construction of a monitor 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 than manual (visual or textual) inspection of simulation traces, or manual construction of property monitors.

This procedure is usually called *lightweight verification* or *run-time verification* in the software context. In this framework, the property monitor checks whether a single trace (or a finite set of traces) satisfies the property specification. 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 STL/PSL Monitor tool that we present in this document is an effort to fill the existing gap in the property-based verification of continuous systems. It relies on the STL/PSL logic, an extension of PSL and the real-time temporal logic MITL, extended with constructs that allow the specify behaviours of continuous variables, developed during the PROSYD project and described in the Deliverable 1.3/2[NM⁺06]. The tool automatically constructs property monitors from an STL/PSL specification and checks whether the simulation traces satisfy the property.

	Present	Section
Mandatory Features		
Pointers to algorithms used	Yes	Section 5.2
List of target operating systems	Yes	Section 3.1
Explanations of coding standards	Yes	Section 4
Discussion of license issues	Yes	Section 3.2
User documentation and imported/exported file formats	Yes	Section 2.1, 2.2, 2.3
Test suite	Yes	D3.4/2 [NM06b]
Standard input language - PSL	Yes	Section 2.2
Support for analog extensions as described in D3.2/6	Yes	Section 2 and 5.2
Automatic generation of simulation monitors	Yes	Section 5.2
Nice to have features		
Support extra features like <code>rose()</code> and <code>fell()</code>	Partial	Section 5.2

The Section 2 is a user manual for the STL/PSL Monitor tool with its installation guide presented in Section 3. The implementation details are given in Section 4, while the underlying theory of the algorithms used by the tool is discussed in Section 5. Finally, we conclude the document in Section 6.

2 STL/PSL Monitor Tool Manual

This Section presents the user manual for the STL/PSL Monitor tool. The application takes an STL/PSL property and a set of analog/Boolean signals as inputs, and allows to monitor the correctness of the formula with respect to the input signals. Results are given both as a simple *correct/incorrect* answer, but also as plots showing representative signals of the property subformulae. The Figure 1 shows the main window of the STL/PSL Monitor tool.

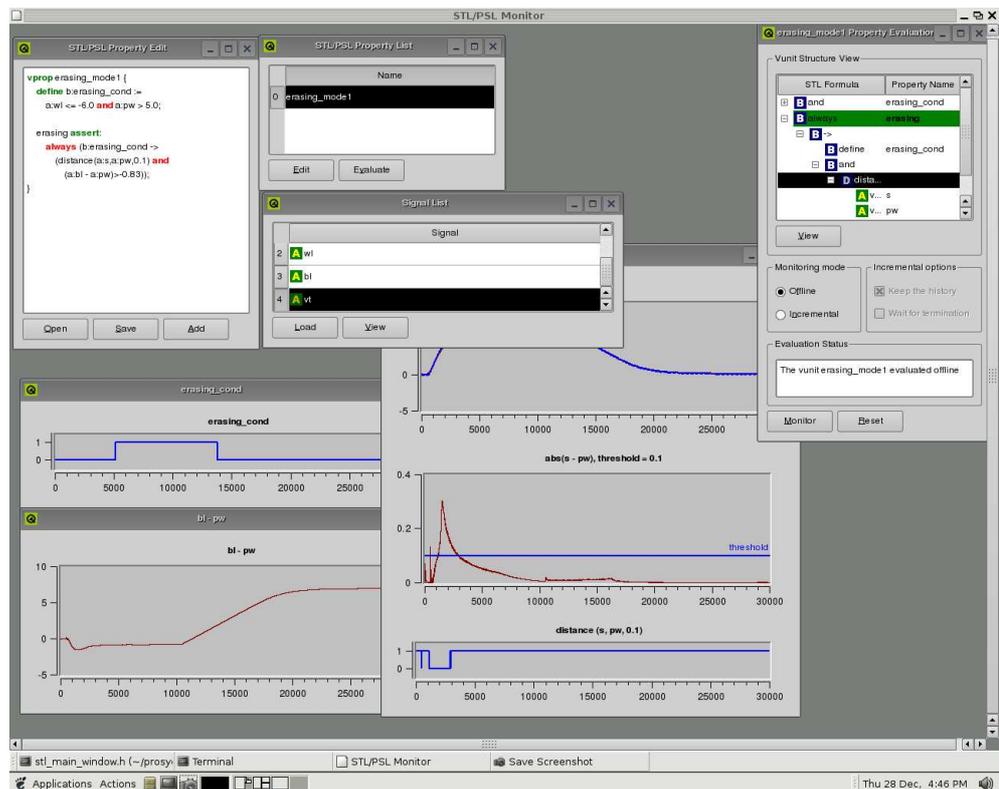


Figure 1: STL/PSL Monitor Main Window

2.1 Signal Management

The input signals in STL/PSL Monitor tool can be imported either statically from a file or dynamically, using a simple protocol built on top of the TCP/IP network communication protocol.

Offline Signal Input

Static import of signals is done via the `Signal List` frame shown in Figure 3. Two types of input files are supported:

out The output format of Nanosim simulations

vcd The analog subset of the Value Change Dump format commonly used by HDL and other simulators

The signals are imported in two steps. The first step consists in reading the header of the simulation file and showing the available signals to the user in a separate frame. The user makes a (possibly multiple) selection of signals that she needs for the monitoring purpose. Only the selected signals are imported into the application. The user can access the imported signal via the `Signal List` frame.

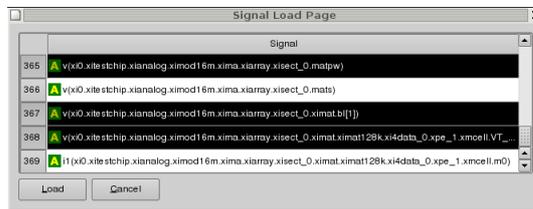


Figure 2: Signal Load Frame

The name of an imported signal can be changed. For this, the user needs to Double Click on the corresponding name in the `Signal View` frame, and to enter a new name. The new name is confirmed with the `Enter` key. If the new name already exists in the list of available signals, the renaming is cancelled and the old name is restored. Figure 3 shows a `Signal Frame` with some signals having their names shortened, in order to simplify the property specification.

To preview an imported signal from the `Signal View` frame, one needs to select the signal and press the `View` Button. The preview of signals is discussed in more details in Section 2.3.

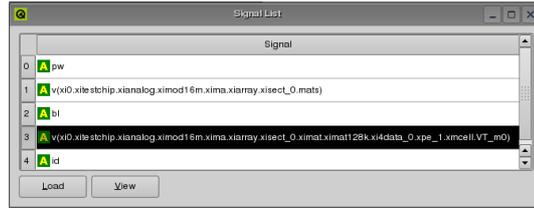


Figure 3: Signal List Frame with Signal Names Changed

Incremental Signal Input

The input signals can be imported into STL/PSL Monitor tool incrementally via a simple protocol built on top of the TCP/IP network. A simulator that provides the input signals needs to act as a client that connects to the STL/PSL Monitor tool when an STL/PSL property is set for incremental evaluation. For more details about the connection to the server, refer to the Section 2.2.

The client can send to the tool both Boolean and analog signals. They are encoded as a sequence of TCP/IP packets representing adjacent intervals in the format shown in Figure 4. All packets have a simple header containing their type which determines the size of the packet and the type of subsequent data fields:

Analog Packet: a sample of the analog signal, containing its name, time and the value

Boolean Packet: a Boolean interval, contains the name of the corresponding signal, the end points and the value of the interval

Termination Packet: Informs the tool that the simulation is over and no further packets are sent

	Packet Type	Variable Name	Interval Value		
Termination Packet	't'				
Analog Packet	'a'	Length of the Variable Name	time	value	
Boolean Packet	'b'	Variable Name	begin	end	value
	qchar	quint16	qreal	qreal	bool

Figure 4: Incremental Signal Packet Format

2.2 Property Management

The STL/PSL Monitor tool provides necessary components for managing the properties. The specification of STL/PSL properties is done in the STL/PSL Property Editor frame, discussed in Section 2.2. The properties that are ready to be evaluated are stored in the STL/PSL Property List frame (Section 2.2). Finally, the STL/PSL Property Evaluation frame is used for setting the monitoring options and for starting the actual evaluation of the selected property, as explained in Section 2.2.

Property Format

The syntax and semantic of the STL/PSL language supported by the tool is described in the Deliverable 1.3/2 [NM⁺06]. The tool extends the STL/PSL language by adding two constructs from the *verification* layer of PSL; the `assert` directive indicating that the property has to hold, and the `vprop` verification unit allowing to group together a set of STL/PSL assertions in order to monitor them in parallel. We also add the `define` directive which allows us to define an analog expression or an STL/PSL property and then refer to it as to a variable within the assertion. Note that Boolean variables have the prefix `b:` and analog variables the prefix `a:`.¹ The following production rules are used to specify a valid STL/PSL property:

```
Stl_Psl_Vprop ::=  
vprop IDENTIFIER { { Assert_Directive } }
```

```
Define ::=  
define Analog_Variable := Analog_Expression ;  
| define Boolean_Variable := Stl_Psl_Property ;
```

```
Analog_Variable ::= a:STRING  
Boolean_Variable ::= b:STRING
```

```
{ Define } Assert_Directive ::=  
IDENTIFIER assert : Stl_Psl_Property  
/* See Deliverable 1.3/2 [NM+06] for the production rules of Stl_Psl_Property and  
Analog_Expression */
```

An example of a valid STL/PSL property recognized by the tool is shown in Figure 5.

¹We need these prefixes in order to distinguish between Boolean and analog variables and hence avoid potential conflicts

```

vprop example {
  define b:asig_cond := a:asig1 <=3.5;

  first assert:
    always distance (a:asig1, a:asig2, 3.5);

  second assert:
    always (b:asig_cond ->
      eventually! [0:200] (a:asig2 > 5.5));
}

```

Figure 5: Example STL/PSL Property

Property Editing

STL/PSL Property Editor frame shown in Figure 6 is a simple editor of input properties. A property verification unit can be either written from scratch in the editor or loaded from a text file with the `stl` extension using the Open button. A `vprop` is saved to a file via the Save button. The Add button transforms the textual description of a property into its valid parse-tree that is added into the STL/PSL Property List frame.

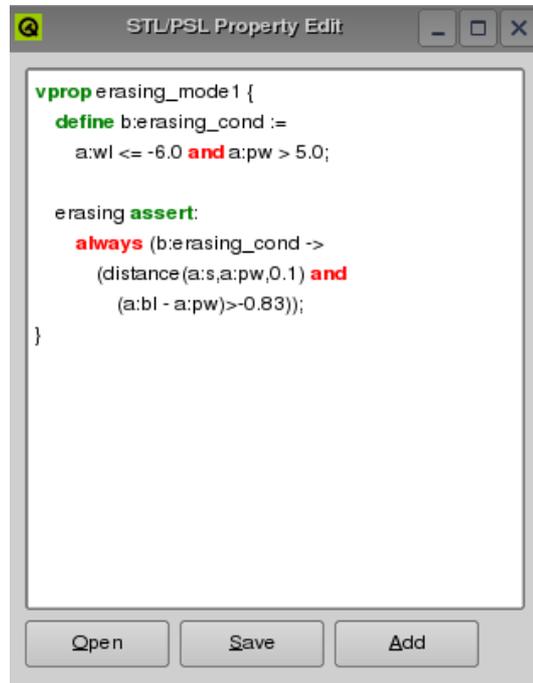


Figure 6: Property Editor Frame

Property List

The STL/PSL Property List frame (Figure 7) stores the list of valid STL/PSL property parse-trees. The textual description of the property can be loaded back into the editor with the View button. The evaluation frame of the selected property is invoked by the Evaluate button.

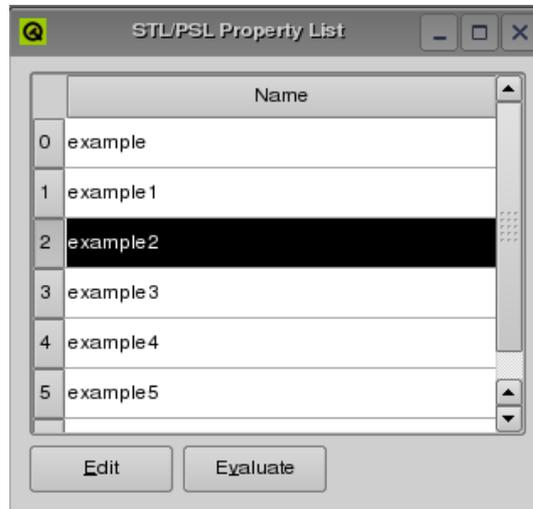


Figure 7: Property List Frame

Property Evaluation

The Property Evaluation frame allows the user to monitor the correctness of a set of assertions defined in an STL/PSL `vprop` verification unit with respect to the input traces. The frame contains the following parts:

Vrop Structure View: Shows the `vprop` set of assertions in a tree view, following the parse structure of the formula. It also allows to view the signal plot of each evaluated sub-formula.

Monitoring Options: Allows the user to choose between *offline* and *incremental* evaluation. For the *incremental* evaluation, some additional options are available.

Evaluation Status: Contains some useful messages about the current evaluation status of the monitoring process.

Monitoring Buttons: The Monitor button starts the evaluation process of the STL/PSL formula. The Reset button clears the current results, allowing to reevaluate the same formula (for example with different input signals).

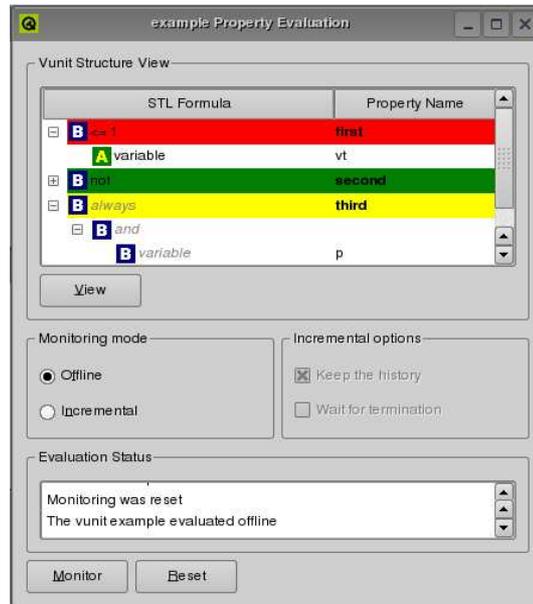


Figure 8: Property Evaluation Frame

An example Property Evaluation is shown in Figure 8.

The vprop unit is shown in the evaluation frame in a form of a tree, where nodes correspond to different sub-formulae. Each node contains three distinctive information:

- The icon showing the type of the subformula:
 - **A** Analog subformula
 - **B** Temporal subformula
 - **D** Distance-based subformula
- The name of the operator with its parameters, if any.
- The name of the subformula, if any (only *assertions* and *variables* have a specific name).

The *assertions* are considered as particular nodes in the vprop unit parse tree, as they represent the top-level formulae. The user is interested in their correctness. We show visually whether an assertion within a vprop is true by highlighting it using the following color scheme:

Yellow: Marks an *undetermined* assertion. An assertion is undetermined if it has not been evaluated yet, either because the monitoring procedure has not yet been started, some input variables defined in the assertion are missing or the vprop unit is in the process of evaluation in the incremental mode, and there has not been yet enough input to determine the correctness of the assertion.

Green: The assertion is *correct*, ie. its evaluation signal is *true* at time 0.

Red: The assertion is *incorrect*, ie. its evaluation signal is *false* at time 0.

The user has the option to conduct the evaluation of an STL/PSL property either *offline* or *incrementally*. In the offline monitoring case, the input signals are taken from the available signals in the Signal View Frame and the property is evaluated with respect to them. If one or more input signals are missing, the monitoring procedure tries to evaluate the vprop unit anyways, but without the guarantee to bring any conclusive result. However, in some cases a definite response can be given even if some inputs are missing. Consider as an example the assertion *always (p and q)*. If the signal *p* is false at some time, we don't need the input signal *q* to conclude that the assertion is *false*.

The incremental procedure requires an online feed of input signals, through TCP/IP protocol. There are two additional options for the incremental evaluation of a property:

Keep the history: The incremental marking procedure separates each evaluation signal into two parts, the first part that has already been used by and is not relevant anymore to the parent formula, and the second part that is still needed by the parent formula in order to make its own updates. By checking this option, the user enforces the tool to keep in memory portions of signals that are not needed anymore by the incremental marking procedure. This option is useful for understanding the reasons of an evaluation that fails by providing the complete history of the monitoring execution.

Wait for termination: The incremental procedure stops by default the evaluation as soon as all the top formulae (assertions) are determined. This is especially helpful when dealing with monitoring assertions that are falsified before the end of a (potentially long) simulation. However, in some cases, the user would like to persue with the monitoring until the end of the simulation. This options allows to force waiting for the end of the simulation (termination packet).

The Monitor button is used to start the evaluation procedure. In the *offline* case, it applies the marking algorithm on the parse tree of the formula, using the available signals as inputs. In the case of the *incremental* marking, a server thread is launched, listening for a new client (a simulator for example) to connect. The default TCP/IP port on which the tool listens for the client is 8022. The client has to provide the inputs incrementally, by sending the TCP/IP packets encoded in the format described in Section 2.1. Each packet that is received is processed, and the incremental procedure tries to update different subformula based on new information.

The results of an evaluation can be cleared with the Reset button. The visualize the results, the user selects the relevant subformula in the parse tree view, and clicks on the View button.

2.3 Visualization of Results

Each operator in STL/PSL (Boolean, temporal or analog) contains an associated *representative* signal. The signals corresponding to different operators are visualized as plots. The STL/PSL Monitor tool provides three types of plots:

Analog Plot: Shows the values of the variables and functions from the *analog layer* of STL/PSL (Figure 9).

Boolean Plot: Shows the *satisfaction signal* (truth values) of Boolean variables and expressions, as well as of temporal operators (Figure 10).

Distance Plot: Provides the visualisation of *threshold* and *threshold delay* distance operators. Consists of three distinct plots, as shown in Figure 11. The first one displays the reference signal with an “envelop” of *threshold* size around it, as well as the input signal. The second plot shows the absolute difference between the input and the reference signal and the threshold value. In this plot we can see how the distance between the reference signal and the input evolves over time with respect to the threshold. Finally, the third plot contains the satisfaction signal of the distance operator.

To save the visualization plots as Portable Network Graphics png files, the user has to Right Click on the plot and choose the Save option.

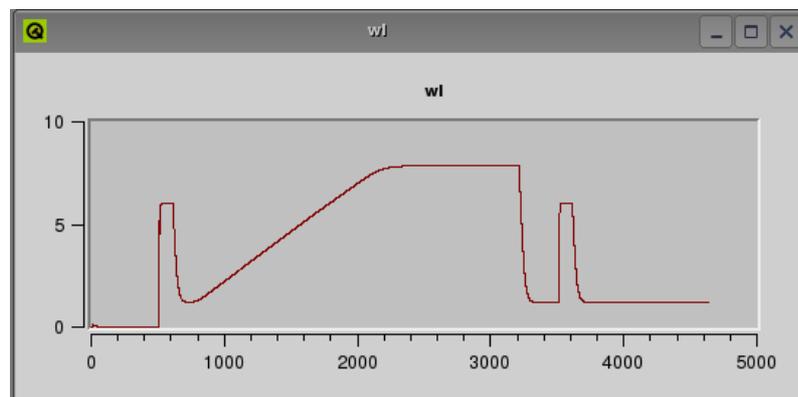


Figure 9: Analog Plot Frame

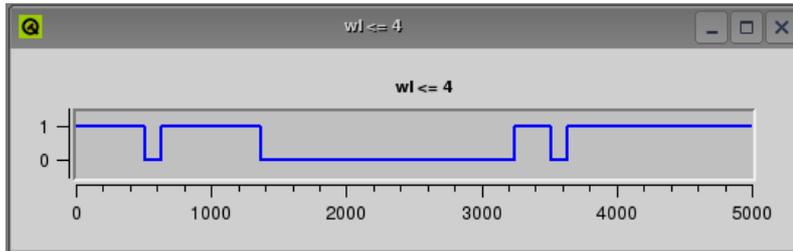


Figure 10: Boolean Plot Frame

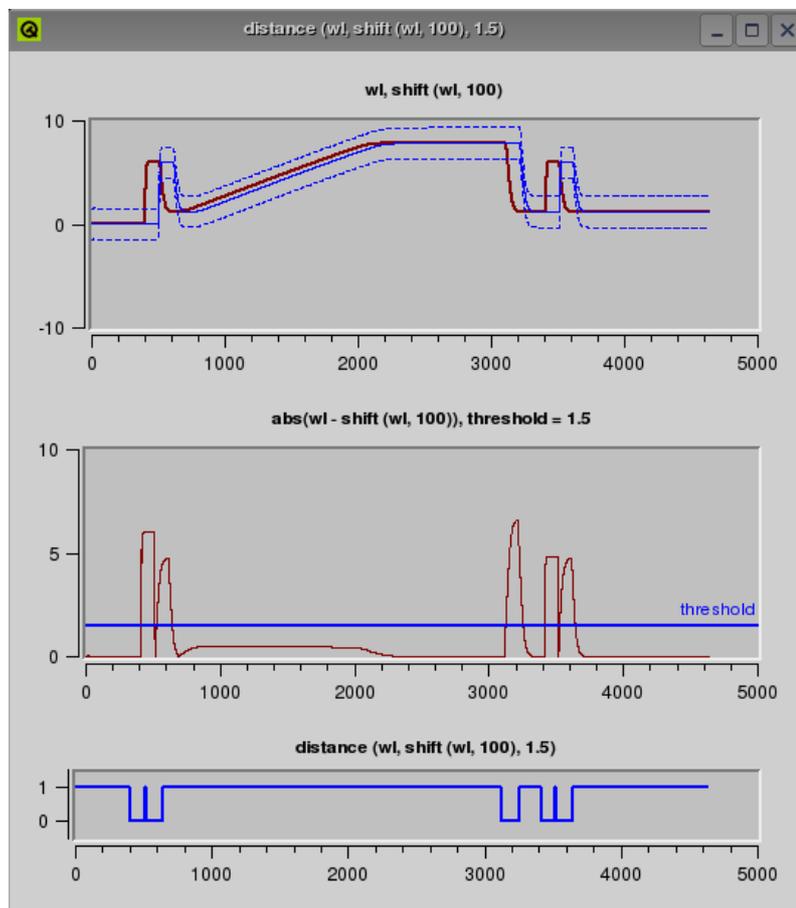


Figure 11: Distance Plot Frame

3 Installation

3.1 System Requirements

The STL/PSL Monitor tool was developed on a Debian GNUL/Linux x86 machine. It requires the following libraries for its installation:

- Qt [QT] (v. 4.1.0) library used for the Graphical User Interface
- Qwt [QWT] (v. 5.0.0rc) library providing the plotting widgets for Qt
- Flex [FLX] lexical scanner (v. 2.5.31) and Bison [BIS] parser (v. 1.875d) for reading STL/PSL properties and signal files

The tool was tested with the versions of external libraries that appear between the brackets.

3.2 Licence Issues

The STL/PSL Monitor tool is distributed under the following licence:

Copyright (C) Verimag/IMAG. All rights reserved.

This software may be freely used, copied and redistributed without fee for non-commercial purposes provided that this copyright notice is preserved intact on all copies and modified copies.

There is no warranty or other guarantee of fitness of this software. It is provided solely "as is". The author disclaims all responsibility and liability with respect to this software's usage or its effect upon hardware or computer systems.

3.3 Installation

In order to install STL/PSL Monitor tool:

1. Unzip `stl-monitor.tar.gz`
`tar xvfz stl-monitor.tar.gz`
2. Set the environmental variable `QWT` to point to the directory where the Qwt library is installed
3. Goto the `stl-monitor` directory
`cd stl-monitor`
4. Create a Makefile from the Qt project file
`qmake stl-monitor.pro`
5. Make the binaries
`make`
6. Launch the application
`./stl-monitor`

4 Implementation

STL/PSL Monitor tool was implemented on a GNU/Linux x86 machine in C++ with the Qt library standing as the basis for its Graphical User Interface.

C++ was chosen mainly for its object-oriented programming model. It allowed us to treat in a uniform manner different types of STL/PSL operators and signals and to simplify the recursive evaluation algorithm.

The graphical components of the tool are based on the Qt library, as one of the de-facto standards of GUI application development for C/C++ under Linux. We use the most important features of Qt, such as the graphical component communication via Signals and Slots, and to some extent the Model/View architecture for the separation of data from the way it is presented to the user.

Figure 12 presents a high-level view of the tool's architecture. It shows main components of the application and their inter-communication, hiding the implementation details. Boxes represent modules that manipulate data-structures shown as wavy polygones. Separate threads in the application are shown as dotted boxes.

The application takes two types of inputs, the STL/PSL Property and the input signals. Properties are edited in the STL/PSL Property Edit module, and can be either written back to a file, or transformed in the STL/PSL Property Parse-Tree data structure, stored internally by the tool.

Input signals are loaded by the Signal Input module in an offline fasion, either from a NanoSim file or a Value Change Dump file and are stored internally as Signal data structure. Signals can also be imported incrementally, via the Server module, which establishes a TCP/IP connection with a simulator or any other client which can provide inputs according to the predefined protocol. The Server module is a separate thread, created by the main application. This way, receiving and processing input packets can be done concurrently.

The STL/PSL Property Evaluator monitors an input property stored in STL/PSL Property Parse-Tree with respect to the input signals. It has two modes, Offline and Incremental. Depending on the mode, the input signal is either loaded from the Signal structure, or is incrementally updated from the Server. Finally, Signal Plot data structure is a visual representation of the Signal structure.

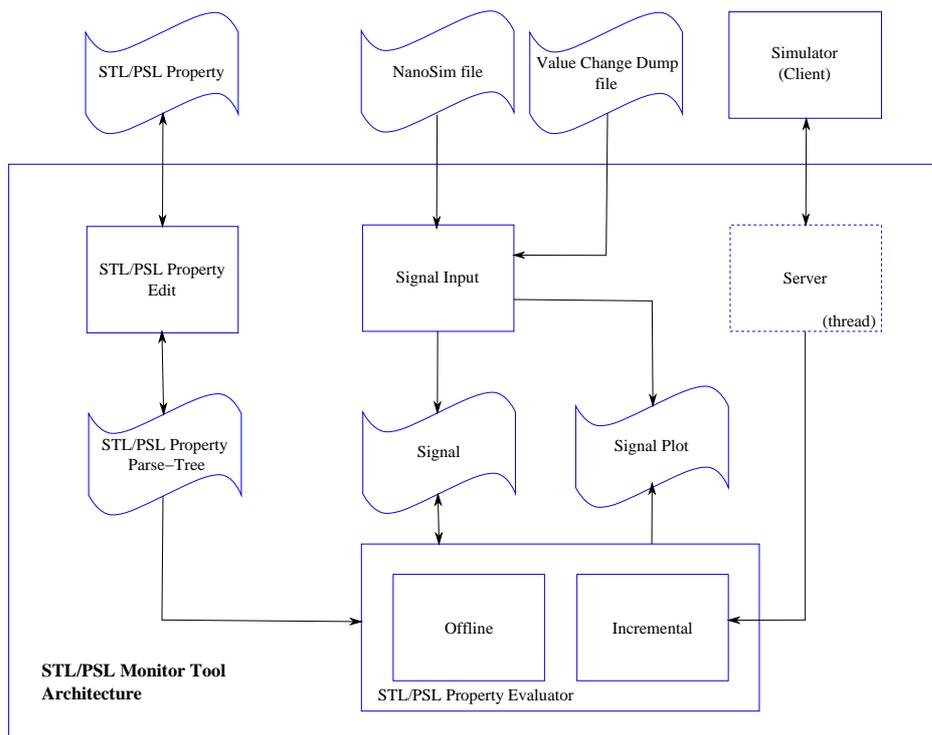


Figure 12: STL/PSL Monitor Tool Architecture

5 Underlying Theory

This Section describes various algorithms used in the STL/PSL tool. It represents mainly the adaptation and the extension of the algorithms described in Sections 6 and 9 of the PROSYD Deliverable 3.2/6 [MNP06].

5.1 Signal Representation

A signal ξ of finite length over an arbitrary domain \mathbb{D} is defined as a partial function $\xi: \mathcal{X}_{\geq 0} \rightarrow \mathbb{D}$ whose domain of definition is the interval $I = [0, r)$. We are interested in two particular types of signals and their effective representation, the Boolean signals ξ_b ($\mathbb{D} = \mathbb{B}$) and the continuous signals ξ_a ($\mathbb{D} = \mathbb{R}$).

Some properties that are used in the algorithms presented are valid for arbitrary signals. A signal whose value is defined on an interval $[0, r)$ is called finite, its length is denoted by $|\xi| = r$ and we use the notation $\xi[t] = \perp$ when $t \geq |\xi|$. The restriction of a signal to length d is defined as

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

The *concatenation* $\xi = \xi_1 \cdot \xi_2$ of two signals ξ_1 and ξ_2 defined over the intervals $[0, r_1)$ and $[0, r_2)$ respectively 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 $\xi' = d \setminus \xi$ obtained from ξ by removing the prefix $\langle \xi \rangle_d$ from ξ , that is,

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

Different signals (Boolean or continuous) can be combined and separated using the standard operations of *pairing* and *projection* defined as

$$\begin{aligned} \xi_1 \parallel \xi_2 = \xi_{12} & \text{ if } \forall t \xi_{12}[t] = (\xi_1[t], \xi_2[t]) \\ \xi_1 = \pi_1(\xi_{12}) \quad \xi_2 = \pi_2(\xi_{12}) & \end{aligned}$$

In particular, $\pi_p(\xi_b)$ will denote the projection of the Boolean signal ξ_b on the dimension that corresponds to proposition p (and conversely $\pi_a(\xi_a)$ denotes projection of the continuous signal ξ_a on the dimension corresponding to the analog variable a). For a union of Boolean and analog signals $\xi = \xi_a \cup \xi_b$, we assume that $\pi_a(\xi)$ is equivalent to $\pi_a(\xi_a)$ and $\pi_p(\xi)$ is equivalent to $\pi_p(\xi_b)$.

The dense time domain presents a number of potential problems for the representation of Boolean signals. However, even in the dense time domain, non-Zeno Boolean signals² of finite length admit a finite representation that we call *interval covering* as a sequence of intervals $\xi_b = I_0 \cdot I_1 \cdot \dots \cdot I_k$ such that $I_0 = [0, t_1)$, $I_i = [t_i, t_{i+1})$, $I_k = [t_{k-1}, r)$, the value of ξ_b is constant in every interval, $\xi_b(I_{i+1}) = \neg \xi_b(I_i)$, $\bigcup I_i = I$ and $I_i \cap I_j = \emptyset$ for every $i \neq j$. An interval covering I' is said to *refine* I , denoted by $I' \prec I$ if $\forall I' \in I' \exists I \in I$ such that $I' \subseteq I$.

An interval covering 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 . In that case we can use the notation $\xi_b(I_i)$. Clearly, if I is consistent with ξ_b , so is any I' satisfying $I' \prec I$. We restrict ourselves to signals of *finite variability*, that is, signals admitting a finite consistent interval covering. We denote by I_{ξ_b} the *minimal* interval covering consistent with a finite variability signal ξ_b . The set of positive intervals of ξ_b is $I_{\xi_b}^+ = \{I \in I_{\xi_b} : \xi_b(I) = 1\}$ and the set of negative intervals is $I_{\xi_b}^- = I_{\xi_b} - I_{\xi_b}^+$.

A signal ξ_b is said to be *unitary* if $I_{\xi_b}^+$ is a singleton. Any finite-variability signal ξ_b can be decomposed into a union of k unitary signals such that $\xi_b = \xi_b^1 \vee \dots \vee \xi_b^k$, see Figure 13.

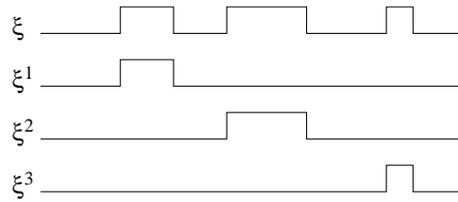


Figure 13: A signal ξ and its unitary decomposition (ξ^1, ξ^2, ξ^3) .

Unlike Boolean signals, continuous signals do not admit an *exact finite representation* and numerical simulators usually produce a *finite* collection of sampling pairs $(t, \xi_a[t])$ with t ranging over some interval $[0, r) \subseteq \mathcal{R}_{\geq 0}$. This finite representation is in contrast to continuous signals defined in the STL/PSL language (see Deliverable 1.3/2 [NM⁺06] as *ideal mathematical objects* consisting of an uncountable number of pairs $(t, \xi_a[t])$ for all $t \in [0, r)$.

Hence, STL/PSL Monitor tool represents continuous signals internally as a finite sequence of sampling points $\xi_a = (t_0, \xi_a[t_0]) \cdot (t_1, \xi_a[t_1]) \cdot (t_2, \xi_a[t_2]) \dots$ with $t_0 < t_1 < t_2 \dots$. The values of $\xi_a[t]$ in between two sample points t_i and t_{i+1} with $t_i < t < t_{i+1}$, are approximated by the tool using the *linear interpolation*. The

²Boolean signals that cannot switch their value infinitely many times in a bounded interval of time

analog domain does not maintain the same standards of exactness and exhaustiveness as in the digital verification domain, and the linear interpolation was chosen as a good tradeoff between approximation quality and computational efficiency. The linear interpolation is implicit in the representation of the continuous signals, and is effectively taken into account when manipulating such signals via the threshold-based Boolean abstraction and the built-in functions of the *analog layer* of STL/PSL. For a detailed discussion on the problems induced by the finite representation of continuous signals, refer to the Section 8 of the PROSYD Deliverable 3.2/6 [MNP06].

In the rest of this document, the notation ξ is used for a multidimensional signal containing signals with the arbitrary domain, ξ_b for Boolean and ξ_a for continuous signals. The abstraction of the domain is useful for the implementation of the algorithm as it allows to treat uniformly operations on Boolean and continuous signals.

5.2 Checking STL/PSL Properties

The STL/PSL Monitor tool implements and extends the marking procedures for checking timed properties described in the Section 6 of the PROSYD Deliverable 3.2/6 [MNP06]. They are based on the bottom-up propagation of values of subformulae in the parse-tree of an STL/PSL property. Two algorithms have been implemented:

Offline marking: This procedure assumes that the input multi-dimensional signal $\xi = \xi_a \cup \xi_b$ is already available, and the marking procedure is applied on the entire signal, propagating 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.

Each subformula of an STL/PSL property has an associated *representative signal* ξ' . A Boolean representative signal is called the *satisfaction signal* $\xi'_b = \chi_\varphi(\xi_b)$. The signal $\xi'_b = \chi_f(\xi_b)$ satisfies $\xi'_b[t] = 1$ iff $(\xi_b, t) \models \varphi$. The continuous representative signal $\xi'_a = \chi_\varphi(\xi_a)$ corresponds to the result of applying the operation φ to the input signal ξ_a . Due to the non-causality of STL/PSL, the value of $\xi'[t]$ is not necessarily known at time t , that is, after observing $\xi[t]$, and may depend on future values of ξ .

Some extensions have been done to the algorithms presented in the Deliverable 3.2/6 [MNP06]. The marking procedure implemented in the tool supports the strong and weak interpretation of temporal STL/PSL operators and updates accordingly the truth values of the corresponding satisfaction signal according. Furthermore, certain Boolean operators in the temporal layer of STL/PSL are optimized in order to update the truth values of their satisfaction signal even when the input signals are not complete.

STL/PSL Temporal Logic

In this Section, we describe the syntax and semantics of the basic subset of STL/PSL. We restrict the discussion to this subset, as the other operators are simply derivable from the basic operators. Refer to the Deliverable 1.3/2 [NM⁺06] for the definition of full STL/PSL. The syntax of STL/PSL is defined by the grammar

$$\begin{aligned}\phi &:= a \mid \phi_1 \star \phi_2 \mid ddt(\phi_1) \mid \phi_1[c] \mid abs(\phi_1) \\ \varphi &:= p \mid \phi \circ c \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 u_{[a,b]}^s \varphi_2 \mid \varphi_1 u^s \varphi_2\end{aligned}$$

where p belongs to a set $P = \{p_1, \dots, p_n\}$ of propositional variables, a to a set $A = \{a_1, \dots, a_m\}$ of analog variables, $\star \in \{-, +, *\}$, $\circ \in \{\leq, >\}$, $c \in \mathcal{Q}$ and $b > a \geq 0$ are rational numbers (in fact, it is sufficient to consider integer constants). A valid STL/PSL property needs to have φ as the top-level operator. We will use the notation $\psi = \phi \cup \varphi$ when we don't need to distinguish between ϕ and φ types of subformulae.

From basic STL/PSL operators one can derive other standard Boolean and temporal operators, in particular the *eventually* and *always* operators:

$$\begin{aligned}\diamond_{[a,b]}^s \varphi &= \mathbb{T} u_{[a,b]}^s \varphi & \text{and} & & \square_{[a,b]}^w \varphi &= \neg \diamond_{[a,b]}^s \neg \varphi \\ \diamond^s \varphi &= \mathbb{T} u^s \varphi & \text{and} & & \square^w \varphi &= \neg \diamond^s \neg \varphi\end{aligned}$$

We interpret STL/PSL over n -dimensional union of analog and Boolean signals. The semantics of formulae in ϕ is defined as follows:

$$\begin{aligned}a(t) &= \pi_a(\xi(t)) \\ (\phi_1 \star \phi_2)(t) &= \phi_1(t) \star \phi_2(t) \\ ddt(\phi_1)(t) &= \frac{d \phi_1 t}{dt} \\ \phi_1[c](t) &= \phi_1(t+c) \\ abs(\phi_1)(t) &= \begin{cases} \phi_1(t) & \text{if } \phi_1(t) \geq 0 \\ -\phi_1(t) & \text{otherwise} \end{cases}\end{aligned}$$

Let us denote $\mathbb{T}^f = [0, |\xi|)$. The satisfiability relation of φ is defined similarly to PSL.

$$\begin{aligned}
(\xi, t) \models p &\leftrightarrow p[t] = \\
(\xi, t) \models \phi \circ c &\leftrightarrow \phi[t] \circ c \\
(\xi, t) \models \neg \varphi &\leftrightarrow (\xi, t) \not\models \varphi \\
(\xi, t) \models \varphi_1 \vee \varphi_2 &\leftrightarrow (\xi, t) \models \varphi_1 \text{ or } (\xi, t) \models \varphi_2 \\
(\xi, t) \models \varphi_1 \text{ u }^s \varphi_2 &\leftrightarrow \exists t' \in [t, \infty) \cap \mathbb{T}^f (\xi, t') \models \varphi_2 \text{ and} \\
&\quad \forall t'' \in [t, t'], (\xi, t'') \models \varphi_1 \\
(\xi, t) \models \varphi_1 \text{ u }_{[a,b]}^s \varphi_2 &\leftrightarrow \exists t' \in t \oplus [a, b] \cap \mathbb{T}^f (\xi, t') \models \varphi_2 \text{ and} \\
&\quad \forall t'' \in [t, t'], (\xi, t'') \models \varphi_1 \\
\\
(\xi, t) \models \diamond^s \varphi &\leftrightarrow \exists t' \in [t, \infty) \cap \mathbb{T}^f (\xi, t') \models \varphi \\
(\xi, t) \models \diamond_{[a,b]}^s \varphi &\leftrightarrow \exists t' \in t \oplus [a, b] \cap \mathbb{T}^f (\xi, t') \models \varphi
\end{aligned}$$

Offline Marking

This algorithm presents an extension of the one presented in [MN04] and the Deliverable 3.2/6 [MNP06]. Its input is an STL/PSL formula ψ and an n -dimensional signal $\xi = \xi_a \cup \xi_b$ of length r containing both Boolean and analog inputs. They are distinguished by projecting the right dimension of ξ to the proper domain \mathbb{D} with respect to the subformula type ($\psi = \phi$ or $\psi = \varphi$). For every sub-formula ψ_1 of ψ it computes its representative signal $\chi_{\psi_1}(\xi)$. In the Deliverable 3.2/6 [MNP06] we assumed that the temporal operators of STL/PSL were bounded, with the guarantee that the properties were fully determined for signals that are long enough. We include in the tool the unbounded *until* as well, and maintain the guarantee of determined properties by using *weak* and *strong* interpretation of the temporal operators in the style of PSL.

The algorithm for marking the property is recursive on the structure (parse-tree) of the formula. It goes down until the propositional and analog variables, whose values are determined directly by ξ , and then propagates values as it comes up from the recursion. We will use OP for arbitrary logical, temporal or analog operators. We also use a list $List(T)$ to group together objects of the same type T . A useful example is $List(\psi_c)$, a list of children subformulae ψ_c ³. As a preparation for the incremental version, we do not pass ξ and χ_φ as input or output parameters but rather store them in global data structures.

The main calculations in this algorithm are done by the COMBINE function which for an operator OP and its subformulae $List(\psi_c)$ computes χ_ψ from the signals χ_{ψ_i} where $\psi_i \in List(\psi_c)$. Note that different χ_{ψ_i} may differ in length. Detailed description on the implementation of the COMBINE function is given in the next Section.

³This way we represent uniformly operators of any cardinality

Algorithm 1: OFFLINESTLPSL

```
input : an STL/PSL Formula  $\psi$ 
switch  $\psi$  do
  case  $p$  propositional variable
  |  $\chi_\psi := \pi_p(\xi)$ ;
  end
  case  $a$  analog variable
  |  $\chi_\psi := \pi_a(\xi)$ ;
  end
  case  $OP(List(\psi_c))$  logical, temporal or analog operator
  | foreach  $\psi_i \in List(\psi_c)$  do
  | | OFFLINESTLPSL ( $\psi_i$ );
  | end
  |  $\chi_\psi := COMBINE(OP, List(\chi_{\psi_c}))$ ;
end
end
```

Combine Function

In this Section, we describe how the COMBINE function works for different operators of STL/PSL. In the general algorithm, we used abstract notations such as ψ and lists of arguments, allowing us to unify the COMBINE procedure for all the analog, Boolean and logical operators. When describing COMBINE to each particular operator, we instantiate the types and replace the lists by explicit enumeration of the arguments according to the cardinality of the operator.

$\chi_\phi := COMBINE(\star, \chi_{\phi_1}, \chi_{\phi_2})$ The pointwise arithmetic operations on continuous signals χ_{ϕ_1} and χ_{ϕ_2} , is done by applying the operation to their sampling points with matching time values. Since the two signals may not be sampled at the same rate, we first refine them. The refinement is done by taking each signal, and adding to it the samples at times where the other signal is originally sampled. The values of new samples are approximated using the linear interpolation. The resulting signal is calculated by taking pointwise arithmetic operation of each corresponding pair of samples.

$\chi_\phi := COMBINE(abs, \chi_{\phi_1})$ The absolute value χ_ϕ of χ_{ϕ_1} is calculated by taking the absolute value of each sample in χ_{ϕ_1} and appending it to ξ_ϕ .

$\chi_\phi := COMBINE(ddt, \chi_{\phi_1})$ The derivative of χ_{ϕ_1} is approximated by taking consecutive samples $\chi_{\phi_1}[t_i]$ and $\chi_{\phi_1}[t_{i+1}]$ and calculating the value $v = \frac{\chi_{\phi_1}[t_{i+1}] - \chi_{\phi_1}[t_i]}{t_{i+1} - t_i}$.

$\chi_\phi := COMBINE([c], \chi_{\phi_1})$ The shifting of continuous signal χ_{ϕ_1} is computed by assigning $\chi_\phi[t - c] = \chi_{\phi_1}[t]$ for all samples in χ_{ϕ_1} , except the ones where $t < c$, in which case the sample is discarded. The only subtle point is when we have two samples $\chi_{\phi_1}[t_i]$ and $\chi_{\phi_1}[t_{i+1}]$ such that $t_i < 0$ and $t_{i+1} > c$. In that case, we need to approximate the value of χ_ϕ at time 0 by linearly interpolating the value of χ_{ϕ_1} at time c .

$\chi_\varphi := \text{COMBINE}(\leq c, \chi_{\varphi_1})$ The Boolean abstraction of a continuous signal χ_{φ_1} with respect to a constant c is done by comparing consecutive samples $\chi_{\varphi_1}[t_i]$ and $\chi_{\varphi_1}[t_{i+1}]$ to c . When both values are smaller or equal than c , the corresponding Boolean interval $I = [t_i, t_{i+1})$ in χ_φ is assigned to *true*, and vice versa. If the value of c is crossed by χ_{φ_1} in between t_i and t_{i+1} , the linear interpolation is used to find the time of crossing t . In that case, two intervals $I_1 = [t_i, t), I_2 = [t, t_{i+1})$ of opposite values (depending whether $\chi_{\varphi_1}[t_i] \leq c$ or $\chi_{\varphi_1}[t_{i+1}] \leq c$) are appended to χ_φ .

$\chi_\varphi := \text{COMBINE}(\neg, \chi_{\varphi_1})$ The negation is computed by simply changing the Boolean value of each minimal interval in the representation of χ_{φ_1} .

$\chi_\varphi := \text{COMBINE}(\vee, \chi_{\varphi_1}, \chi_{\varphi_2})$ For the disjunction we first construct a refined interval covering $I = \{I_1, \dots, I_k\}$ 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 $I\chi_\varphi$.

$\chi_\varphi := \text{COMBINE}(\diamond_{[a,b]}^s, \chi_{\varphi_1})$ This is the most important part of our procedure which computes $\chi_\varphi := \text{SHIFT}_{[a,b]}(\xi_{\varphi_1})$. For every positive interval $I \in I^+_{\varphi_1}$ we compute its back shifting $I \ominus [a, b] \cap \mathcal{T}$ and insert it to I^+_φ . Overlapping positive intervals in I^+_φ are merged to obtain a minimal consistent interval covering. In the process, all the negative intervals shorter than $b - a$ disappear.⁴

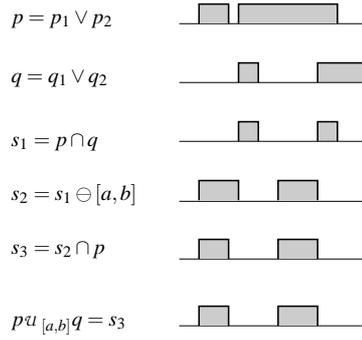
$\chi_\varphi := \text{COMBINE}(u_{[a,b]}^s, \chi_{\varphi_1}, \chi_{\varphi_2})$ The implementation of the timed *until* operator is based on the equivalence $\varphi_1 u_{[a,b]}^s \varphi_2 \leftrightarrow \diamond_{[a,b]}^s(\varphi_1 \wedge \varphi_2) \wedge \varphi_1$ when χ_{φ_1} is a unitary signal. This is because for a unitary signal, if φ_1 holds at t_1 and at t_2 it must hold during all the interval. This does not hold for arbitrary signals, see Figure 14. In order to treat the general case where χ_{φ_1} is an arbitrary signal we first need to decompose it into the unitary signals $\chi_{\varphi_1}^1, \dots, \chi_{\varphi_1}^k$ and then compute

$$\chi_\varphi^i = \text{SHIFT}_{[a,b]}(\chi_{\varphi_1}^i \wedge \chi_{\varphi_2}) \wedge \chi_{\varphi_1}^i$$

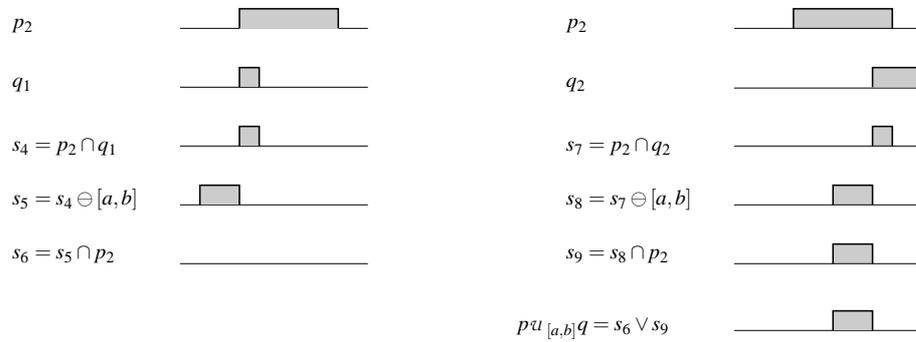
for each $i \in [1, k]$. Finally we recombine the resulting signals by as

$$\chi_\varphi = \bigvee_{i=1}^k \chi_\varphi^i.$$

⁴Another way to see it is as shifting the *negative* intervals by $[b, a]$.



(a)



(b)

Figure 14: Marking for $p u_{[a,b]} q$ via marking for $\diamond_{[a,b]}(\pi \wedge p) \wedge p$: (a) with non-unitary signals we obtain wrong results; (b) with a unitary decomposition of p and q we obtain correct results. The computation with p_1 is omitted as it has an empty intersection with q .

Incremental Marking

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_1 the algorithm stores its already-computed satisfying signal partitioned into a concatenation of two signals $\chi_{\psi_1} \cdot \Delta_{\psi_1}$ with χ_{ψ_1} consisting of values already propagated to the super-formula of ψ_1 , and Δ_{ψ_1} , consists 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 procedure is invoked, which updates every χ_{ψ_1} and Δ_{ψ_1} from the bottom up. The difference with respect to the offline algorithm is that only the segments of the signal that has 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.

As an illustration consider a temporal binary formula $\varphi = \text{OP}(\varphi_1, \varphi_2)$ and the corresponding truth signals of Figure 15-(a). Before the update we always have $|\chi_\varphi \cdot \Delta_\varphi| = |\chi_{\varphi_1}| = |\chi_{\varphi_2}|$: 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 χ_{φ_1} and Δ_{φ_1} (as well as between χ_{φ_2} and Δ_{φ_2}) in order to reflect the update of Δ_φ . The procedure is described in Algorithm 2.

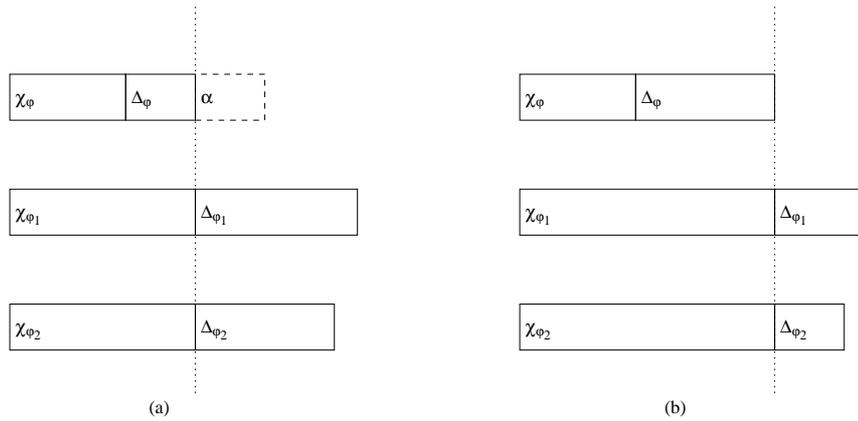


Figure 15: A step in an incremental update: (a) A new segment α for φ is computed from Δ_{φ_1} and Δ_{φ_2} ; (b) α is appended to Δ_φ and the endpoints of χ_{φ_1} and χ_{φ_2} are shifted forward accordingly.

Optimization of Boolean operators

In the case of certain binary Boolean operators, the resulting truth value at a time t can be deduced even if one of the input values is not known at time t . We use the following equivalences in order to be able to make updates when one of the inputs is unknown:

$$\begin{aligned} F \wedge \perp &= F \\ T \vee \perp &= T \\ F \rightarrow \perp &= T \end{aligned}$$

An example of how the optimization works is shown in Figure 16 for the conjunction of two formulae. The consequence of the optimization is that certain properties can be determined even if all the inputs are not present. Note that, some other operators, such as timed until, use indirectly this optimization, because of the equivalence $\varphi_1 \text{ u }_{[a,b]}^s \varphi_2 = \varphi_1 \wedge \diamond_{[a,b]}^s (\varphi_1 \wedge \varphi_2)$, when χ_{φ_1} is a unitary signal.

Algorithm 2: INC-OFFLINE-STLPSL

input : an STL/PSL Formula ψ and an increment Δ_ξ of a signal

```
switch  $\psi$  do
  case  $p$ 
    |  $\Delta_\psi := \Delta_\psi \cdot \pi_p(\Delta_\xi)$ ;
  end
  case  $a$ 
    |  $\Delta_\psi := \Delta_\psi \cdot \pi_a(\Delta_\xi)$ ;
  end
  case  $OP(List(\psi_c))$ 
    | foreach  $\psi_i \in List(\psi_c)$  do
      | INC-OFFLINE-STLPSL ( $\psi_i$ );
    end
    |  $\alpha := COMBINE(OP, List(\Delta_{\psi_c}))$ ;
    |  $d := |\alpha|$ ;
    |  $\Delta_\varphi := \Delta_\varphi \cdot \alpha$ ;
    | foreach  $\psi_i \in List(\psi_c)$  do
      |  $\chi_{\psi_i} := \chi_{\psi_i} \cdot \langle \Delta_{\psi_i} \rangle d$ ;
      |  $\Delta_{\psi_i} := d \setminus \Delta_{\psi_i}$ ;
    end
  end
end
end
```

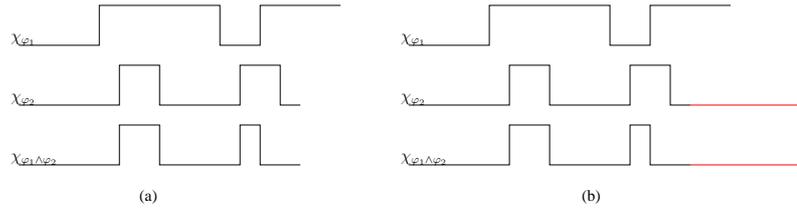


Figure 16: Boolean Optimization: (a) Before the update (2) χ_{φ_2} is updated, and $\chi_{\varphi_1 \wedge \varphi_2}$ accordingly

Freeze operator

The STL/PSL Monitor tool implements the *freeze operator* which is not part of the STL/PSL language. The syntax of the operator is `freeze(x, cond, c)` where x is an analog expression, $cond$ is a Boolean expression and c is a constant (time delay).

`freeze(x, cond, c)` returns the value of x during the interval $[0, t + c)$ where t is the time point where the freeze condition $cond$ becomes *true*. At $t + c$ the current value of x is frozen, and that value is returned until the end of the trace. We can see an example of the freeze operator in Figure 17.

The freeze operator was defined and implemented into the tool, as a request of analog designers of ST Italy as part of the effort on the analog case study presented in Deliverable 3.4/2 [NM06b]. However, its semantics are difficult to define formally in the spirit of temporal logics, and hence the freeze operator has not been

officially included in the STL/PSL language described in Deliverable 1.3/2 ??.

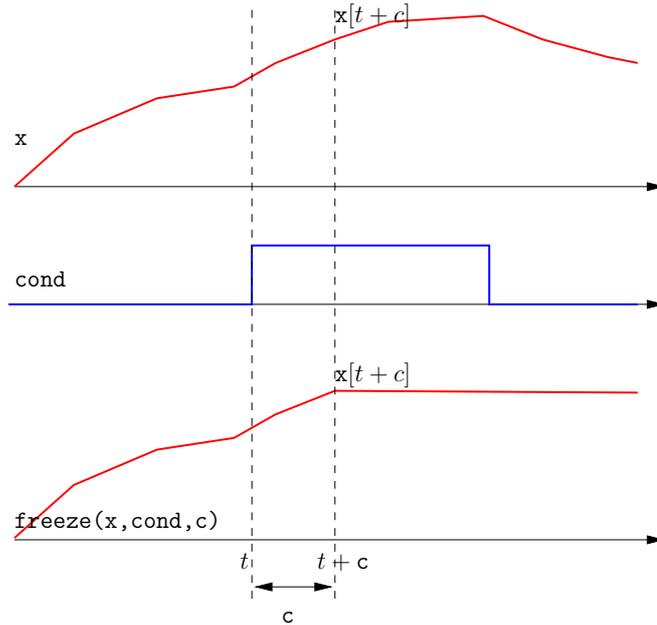


Figure 17: Freeze Operator

Approximating events

The Boolean signals in STL/PSL language are defined as a concatenation of left-closed right-open intervals $I = [t_1, t_2)$ such that $t_2 > t_1$. This means that point-wise intervals (or events) cannot be represented as an interval $I = [t_1, t_1)$ is empty. Hence we cannot express in STL/PSL the raise and fall of a Boolean signal. However, raise and fall operations can be approximated to arbitrary precision using the following formulae:

$$\begin{aligned} \text{raise}(p) &\approx \neg p \wedge \diamond_{\leq \varepsilon}^s p \\ \text{fall}(p) &\approx p \wedge \diamond_{\leq \varepsilon}^s \neg p \end{aligned}$$

The property $\neg p \wedge \diamond_{\leq \varepsilon}^s p$ is evaluated to *true* in the interval $[t - \varepsilon, t)$ where t is the exact point where p becomes *true*. We can see that by choosing a sufficiently small ε , the *raise*(p) property is well approximated. As an example, consider the signal p in Figure 18(a). The perfect raise of p is captured in Figure 18(b). Figures 18(c) and (d) show two approximations of *raise*(p) with approximation error ε_1 and ε_2 respectively, such that $\varepsilon_2 \ll \varepsilon_1$. We can see that by decreasing the size of the ε constant, the approximation gets better.

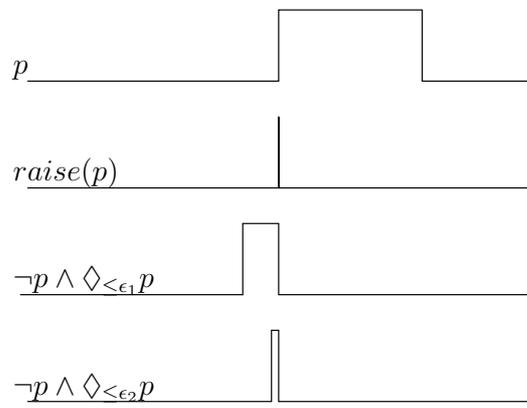


Figure 18: Approximation of $raise(p)$

6 Conclusion

The STL/PSL Monitor tool is, to the best of our knowledge, the only property-based lightweight verification tool for the continuous systems. As a pioneering work, we hope that it will help promote formal methods beyond their traditional domains. The simple and elegant marking procedure is interesting in itself as it provides solid bases for also checking purely timed MITL properties.

7 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.
- [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*).
- [BIS] <http://www.gnu.org/software/bison/>
- [Dru00] D. Drusinsky. The Temporal Rover and the ATG Rover. In *Proc. SPIN'00*, pages 323–330. LNCS 1885, Springer, 2000.
- [FLX] <http://www.gnu.org/software/flex/>
- [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.
- [KCV04] A. Kumari B. Cohen and S. Venkataramanan, *Using PSL/Sugar for Formal and Dynamic Verification*, VhdlCohen Publishing, 2004.
- [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.
- [M05] O. Maler, *Extending PSL for Analog Circuits*, PROSYD Deliverable D1.3/1, 2005.
- [MN04] O. Maler and D. Nickovic, Monitoring Temporal Properties of Continuous Signals, *FORMATS/FTRTFT'04*, 152-166, LNCS 3253, 2004.
- [MNP06] O. Maler, D. Nickovic and A. Pnueli, *Checking Digital, Timed and Analog PSL Properties*, PROSYD Deliverable D3.2/6, 2006.

- [MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.
- [NM06a] D. Nickovic and O. Maler, *Final Proposal for PSL Analog extensions*, PROSYD Deliverable D1.3/2, 2006.
- [NM06b] D. Nickovic and O. Maler, *Analog Case Study*, PROSYD Deliverable D3.4/2, 2006.
- [NM⁺06] D. Nickovic, O. Maler, A. Pnueli, P. Caspi and A. Girard, *Final Proposal for PSL Extensions*, PROSYD Deliverable D1.3/2, 2006.
- [QT] <http://www.trolltech.com/products/qt>
- [QWT] <http://qwt.sourceforge.net/>
- [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.