FP6-IST-507219

PROSYD:

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Analog Case Study (Deliverable 3.4/2)

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	☒
PP	Restricted to other programme participants (including the Commission Services)	☐
RE	Restricted to a group specified by the consortium (including the Commission Services)	☐
CO	Confidential, only for members of the consortium (including the Commission Services)	☐

Table of Revisions

Version	Date	Description and reason	By	Affected sections
0.5	December 26, 2006	Complete Draft	Nickovic	All
0.8	December 28, 2006		Nickovic	Evaluation
1.0	December 31, 2006	Final Version	Nickovic	

Authors

Dejan Nickovic
Oded Maler
Andrea Fedeli
Pierluigi Daglio
Davide Lena

Executive Summary

This document describes the analog case study on a Flash memory simulation provided by ST Microelectronics Italy. The case study explores in practice the benefits of extending formal specification to analog designs, by expressing the desired continuous behavior of the Flesh memory cell with the STL/PSL specification language developed during the project and described in Deliverables 1.3/1 [M05] and 1.3/2 [NM$^+$06], and by checking its correctness with the STL/PSL Monitor tool presented in Deliverable 3.2/13 [NM06].

Purpose

The purpose of this document is to describe the efforts in the case study on property-based specification and checking of analog circuits using STL/PSL. It assesses the benefits of the analog extensions of PSL and of the lightweight verification tool developed during the project.

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

The background of the analog case study is based mainly on results achieved in the PROSYD project on the extension of property-based formal methods to analog systems. The case study is mainly an evaluation and a proof of concept of the STL/PSL language presented in Deliverable 1.3/2 [NM$^+$06] and of the STL/PSL Monitor tool (Deliverable 3.2/13 [NM06]). The other Deliverables related to this document are 1.3/1 [M05] and 3.2/6 [MNP06].

Contents

Table of Figures

List of Tables

Glossary

Assertion

A property that has to be fulfilled by the system under verification.

Exhaustive verification

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

Flash memory

Non-volatile computer memory that can be electrically erased and reprogrammed, primarily used in memory cards.

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**.

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.

1 Introduction

The property-based verification of digital designs is a mature domain that has been used in practice for many years. Specification formalisms based on temporal logics [MP95], such as LTL, CTL and PSL are commonly accepted and used in discrete verification tools. When considering *timed* models, the situation is less satisfactory, although a number of variants of real-time logics have been proposed, such as MTL [Koy90], MITL [AFH96], TCTL [Y97] or *timed regular expressions* [ACM02] (see [AH92, Hen98] for an extensive survey). The timed verification tool KRONOS [Y97] has integrated TCTL logic as a specification formalism.

The property-based verification of continuous systems is at its beginning and is restrained to small systems and very limited fragments of logics [FGP06]. Hence the preferred validation method for analog systems remains simulation and testing. The approach taken in the PROSYD project was to export the formal specification element to the analog simulation through property monitors. This procedure is called *lightweight verification*. In this framework, the property monitor is built automatically from the specification and it 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 logic STL/PSL that allows to reason about properties on continuous signals was developed and presented in Deliverable [NM$^+$06] (and based on results from [MN04, M05]). The STL/PSL Monitor tool [NM06] was implemented as a lightweight verification application that automatically builds observers of STL/PSL specifications and monitors their correctness with respect to continuous inputs.

The case study was done on a set of simulations of a Flash memory test cell provided by ST Microelectronics Italy. Its goal is to serve as a proof of concept of the property-based analog monitoring approach relying on the STL/PSL language. The main objectives of the case study is to evaluate the overall approach and tools and respond to the following questions:

- Which classes of properties can be written in STL/PSL?
- How difficult is to write correct STL/PSL specifications?
- How efficient is the automatic evaluation of STL/PSL specifications?

The rest of the document is organized as follows:

- Section 2 presents the Flash memory case study. It describes the properties specified in STL/PSL giving details about the results achieved.
- Section 3 evaluates the results achieved in the case study. It assesses the expressiveness of the STL/PSL language and common errors during the specification process. The evaluation also takes into account the computational efficiency of the STL/PSL Monitor tool, by analyzing the application's time and memory requirements.
- Finally, Section 4 gives the concluding remarks on the results achieved by the case study.

2 Flash Memory Case Study

The subject of the analog case study is the “Tricky” technology flash memory test chip in 0.13um process developed in ST Microelectronics Italy and shown in Figure 1. The flash memory presents an advantage for the analog case study, in that it is a digital system whose discrete behavior is implemented at the analog level. Hence, it is a good link between the analog and the digital world. The “Tricky” main blocks are:

- 16Mbit memory flash array
- Partial flash array used in simulation
- Full row and column decoders
- LV sense amplifiers (1.2 volt)
- Simple synthesized control logic for testing purposes
- Analog/digital transistor level pads
- No charge pumps and oscillators

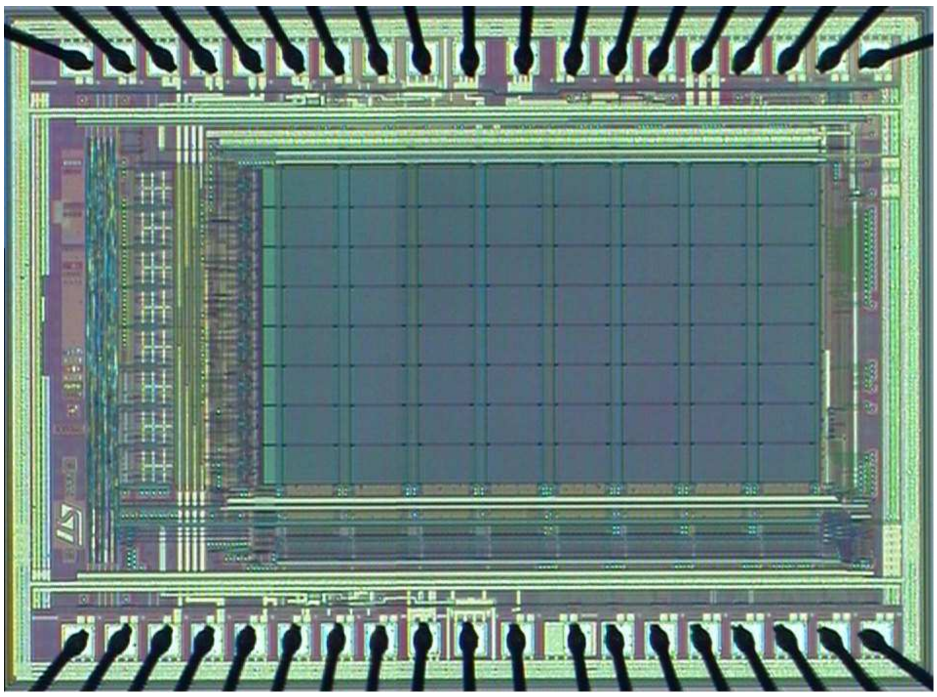


Figure 1: Tricky Flash Memory Cell

The analog case study is based on lightweight verification of the test chip. In the lightweight verification, the system under test is viewed as a black box, and we don't need to know further details about the underneath chip architecture. The correct functioning of the chip at the analog level is determined by the behavior of a number of signals, which are extracted during the simulation:

bl: matrix bit line terminal (flash cell drain)

pw: flash matrix p-well terminal (flash cell bulk)

wl: matrix word line (flash cell gate)

s: flash matrix source terminal (flash cell source)

vt: threshold voltage of flash cell

id: drain current of flash cell

The flash memory cell can be in one of the *programming*, *reading* or *erasing* modes. In each mode, the above mentioned signals should have a particular behavior. The flash cell was simulated in the *programming* and the *erasing* mode for the case study. The length of the *programming* mode simulation was 5000us, and 30000us for the *erasing* mode.

There are four STL/PSL properties written to describe the correct behavior of the *programming* mode and one property for the *erasing* mode. The specification of the properties is a joint effort of analog designers of ST Microelectronics Italy and Verimag. The ST Microelectonics provided the specification in plain English and pseudo-STL/PSL code. The specifications were adapted at Verimag to the actual syntax of STL/PSL. Some properties required several feedback-loops in order to get corrected. For each property, we note the changes, if any, that were brought with respect to the initial specification.

2.1 Programming Properties

STL/PSL Specification

```
vprop programming1 {
  define b:vt_raise :=
    a:vt <= 5.0 and eventually![<=0.1] a:vt > 5.0;

    define b:id_fall :=
        abs(a:id) > 5e-6 and
          (eventually![<=0.1] abs(a:id) <= 5e-6);

      pgm assert:
        always (b:vt_raise ->
          (((abs(a:id) > 5e-6) and (a:vt > 4.5))
            until! b:id_fall));
  }


  vprop programming2
  {
    define b:raise_wl_and_not_pgm :=
      a:wl<=0.1 and
        eventually![<=15.0]
          (a:wl>3.8 and a:id > 30e-6);

    define b:start_prog :=
      abs(a:bl) <= 0.1 and
        (eventually![<=10.0] a:bl > 3.8);

    define b:fall_bl :=
      a:bl>3.8 and eventually![<=10.0] abs(a:bl)<=0.1;

    pgm1 assert:
      always (b:raise_wl_and_not_pgm ->
        eventually! (b:start_prog  and eventually![<=15.0]
          ((a:bl>3.8) until! [3e2:1.5e3] (a:wl>6.0))));

    pgm2 assert:
      always (b:start_prog ->
        (not b:fall_bl until
          (eventually![<=10.0]
            (a:vt > 5.0 and abs(a:id) <= 5e-6))));
  }
```

Informal Description

programming1: When the signal `vt` overcomes the threshold of `5V`, `vt` has to remain above `4.5V` and `id` has to remain above `5e-6A` until `id` falls below `5e-6A`.

programming2: When the wordline `wl` jumps from a value near `0V` above the threshold `3.8V` and the cell has not been programmed yet, ie. the current `id` is above `30e-6A` (`raise_wl_and_not_pgm`), the programming procedure (`start_prog`), characterized by a ramp of the bitline `bl` signal from a value near `0V` to the threshold of `3.8V` in less than `10ms`, has to eventually start and the bitline `bl` has to remain above `3.8V` for at least `300ms` and until the wordline `wl` overcomes `6V`, which has to happen within `1500ms`. This property is expressed by the assertion `pgm1`. Moreover, once the programming procedure starts, the bitline `bl` is not allowed to fall from above `3.8V` to `0V` either until the end of simulation or until the signal `vt` becomes higher than `5V` and the absolute value of `id` smaller than `5e-6`. This requirement is described by the assertion `pgm2`.

Results

The two programming properties are correct with respect to the simulation trace. Figures 2 (a) and (b) show the signals `vt` and `id` used by the assertion `pgm`. We can see in Figure 2 (d) that `abs(id)` is greater than `5e-6V` and `vt` greater than `4.5V` from the moment that `vt` crosses the `5V` threshold (Figure 2 (c)) and until the current `id` falls near `0A` (Figure 2 (e)). Hence the assertion `pgm` holds.

In Figures 3 (a), (b) and (c), we can see the `wl`, `id` and `bl` signals respectively, used in the `pgm1` assertion. The Figures 3 (d) and (e) show that the raise of `wl` is followed by the `start_prog` condition, and from that moment `bl` remains above `3.8V` (Figure 3 (f)) until the wordline `wl` crosses the `6V` threshold as we can see in Figure 3 (g). Hence the assertion `pgm1` holds too.

Figures 4 (a), (b) and (c) show `wl`, `id` and `bl` signals used by the `pgm2` assertion. After the programming mode `start_prog` is triggered (Figure 4 (d)), the bitline `bl` does not fall down before `vt` gets above `5V` and `abs(id)` below `5e-6A`, as we can see in Figures 4 (e) and (f). Hence the assertion `pgm2` holds.

Notes

1. The original specification of the `programming1` property did not correspond to the intended meaning. Hence, it was rewritten by the ST designers using the suffix implication PSL-like syntax as

   ```
   {vt>5} |-> {{{vt>4.5} && {|id|<5e-6[->];|id|<5e-6[+]}}}
   ```

   The final version of the property is an STL/PSL interpretation of the PSL-like property.

2. `programming2` property was split in two assertions in order to enhance the readability of the specification.

3. Approximations of *raise* and *fall* of signals were introduced where needed.

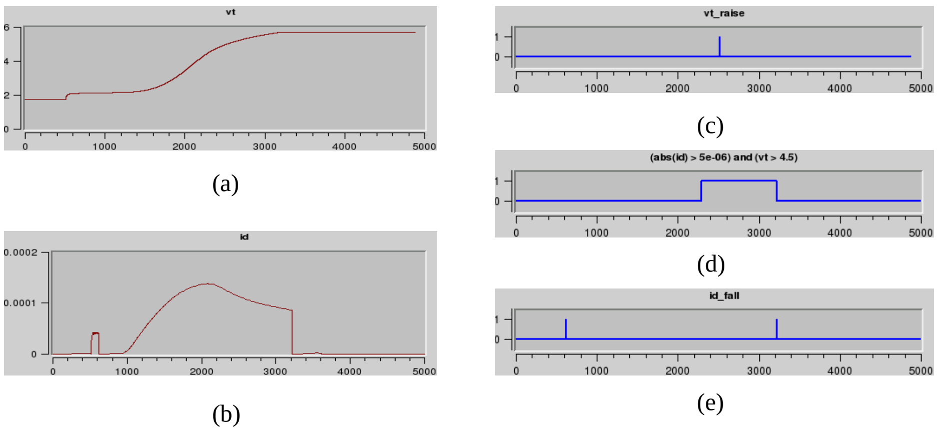


Figure 2: programming1 Property Result: (a) `vt` (b) `id` (c) `raise_vt` (d) `abs(id)>5e-6) and (vt>4.5` (e) `fall_id`

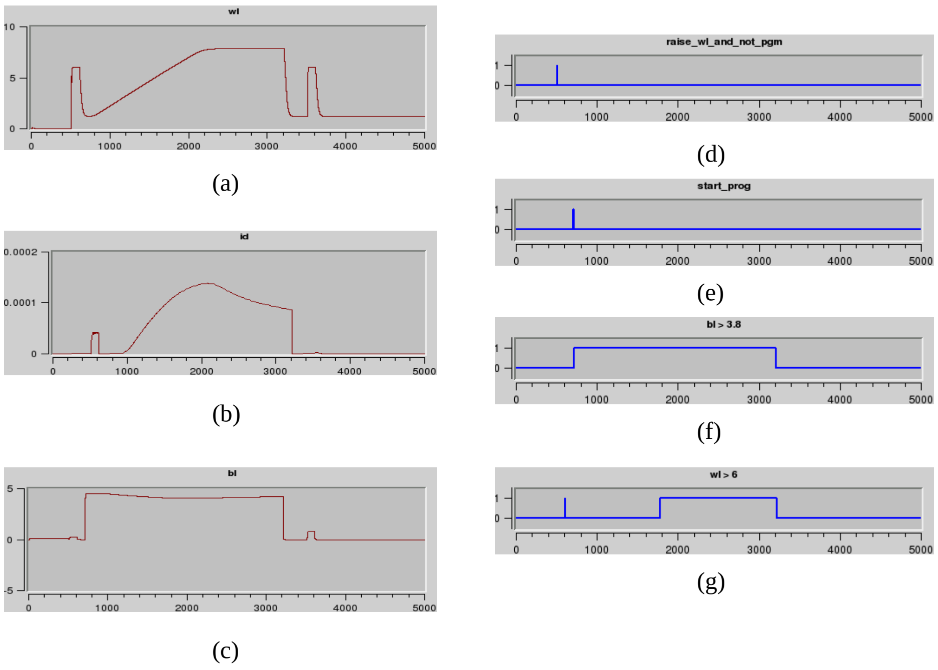


Figure 3: programming2 Property Result: pgm1 (a) `vt` (b) `id` (c) `raise_vt` (d) `abs(id)>5e-6) and (vt>4.5` (e) `fall_id`

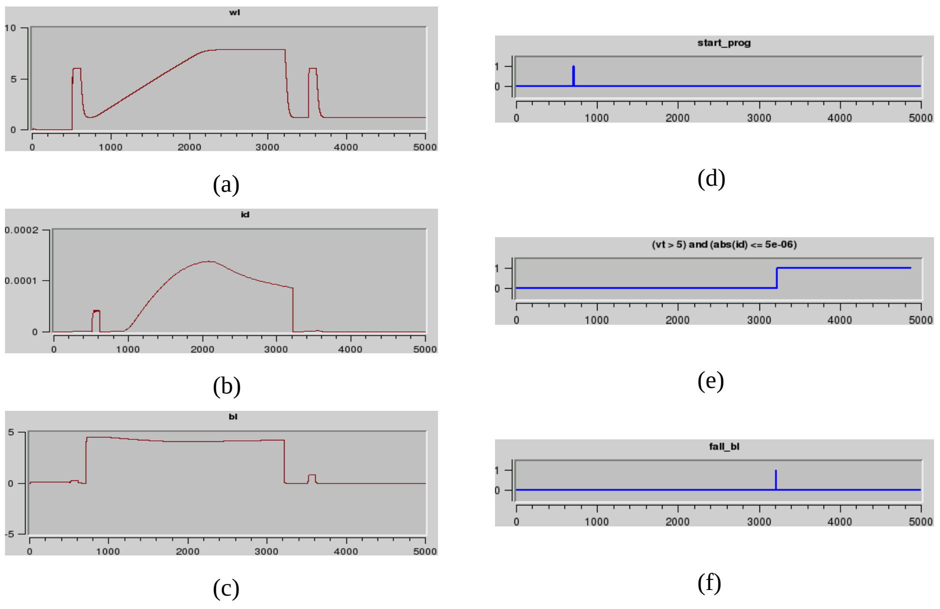


Figure 4: programming2 Property Result: pgm2 (a) vt (b) id (c) raise_vt (d) abs(id)>5e-6) and (vt>4.5 (e) fall_id

2.2 Admissible Bitline Voltage Decay Property

In the programming mode, the bitline has to remain above a certain percentage of its initial value.

STL/PSL Specification

```
vprop decay {
  define b:start_prog :=
    ((a:bl <= 0.1) and eventually! [<=10.0] (a:bl > 3.8));

  define b:freeze_cond :=
    b:start_prog and a:pw <= -0.8;

  define b:derivative_saturates_wl :=
    distance (shift(a:wl, 50.0), a:wl, 0.05);

  define a:diff_bl_frozen_bl :=
    a:bl - (freeze (a:bl, b:freeze_cond, 50.0) * 0.85);

  adm_volt_decay assert:
    always (b:freeze_cond ->
      eventually![<=10.0](((a:diff_bl_frozen_bl > 0.0)
        until! b:derivative_saturates_wl)));
}
```

Informal Description

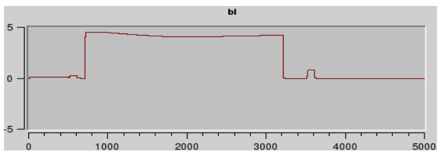
The condition to take a "snapshot" of the value of the bitline `freeze_cond` is determined by the beginning of the programming mode (`start_prog`) and the p-well voltage being below `-0.8V`. When this condition is met, the value of the bitline `bl` is frozen in the next `50ms` and has to remain at least over `85%` of that value until the wordline `wl` stabilizes (`derivative_saturates_wl`). The stabilization of the wordline is met when `wl` does not change more than `0.05V` in `50ms`.

Results

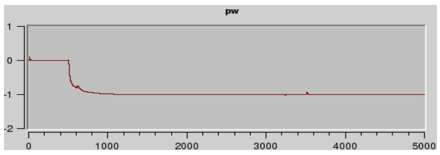
This property is shown to be correct with respect to the simulation trace. The freeze condition `freeze_cond` that we can see in Figure 5 (g) is triggered at $\sim$`700ms` by a ramp in the bitline `bl` (Figure 5 (a)) and a requirement on the value of the p-well voltage `pw` (Figure 5 (b)). The frozen value of `bl` (mutiplied by `85%`) is shown in Figure 5 (d) and the difference between `bl` and frozen `bl` `diff_bl_frozen_bl` in Figure 5 (e). From Figure 5 (h) we can see that `diff_bl_frozen_bl` remains higher than `0` between `700` and `3200ms`. Since the wordline `wl` (Figure 5 (c)) stabilizes its value at $\sim$`2200ms` (Figure 5 (f) and (i)), which is before `diff_bl_frozen_bl` goes below `0`, we can conlude that the property holds.

Notes

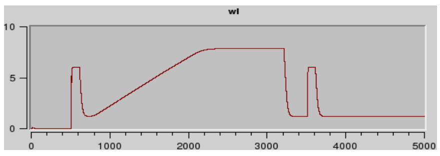
1. The original property required the *freeze* operator to be applied to `bl` as soon as it crossed `3.8V` threshold. However, this trivially implies that the frozen value of `bl` would be equal to `3.8V`. The intended meaning was to freeze the value of `bl` with some small delay, in order to have `bl` stabilized. That is the reason that we added `50ms` delay for the freeze operator to be applied.



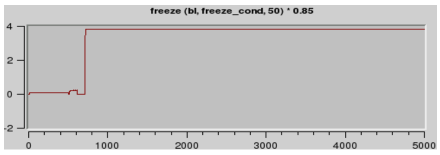
(a)



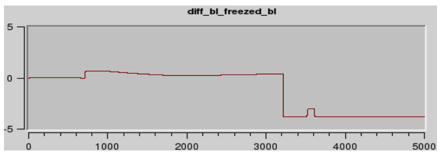
(b)



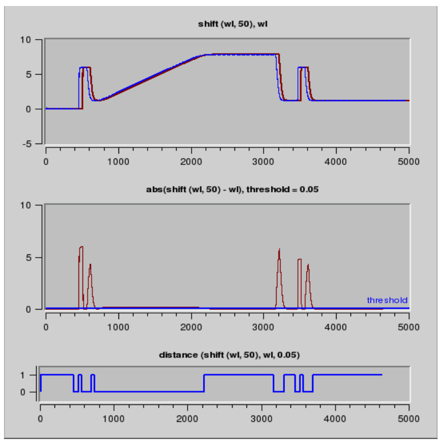
(c)



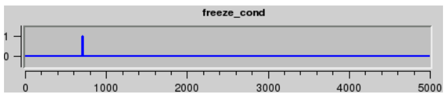
(d)



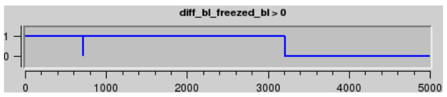
(e)



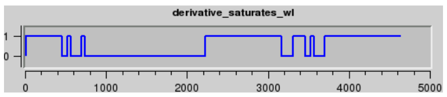
(f)



(g)



(h)



(i)

Figure 5: decay Property Result: (a) `bl` (b) `pw` (c) `wl` (d) `freeze(bl,freeze_cond,50)*0.85` (e) `diff_bl_frozen_bl` (f) `distance(shift(wl,50),wl,0.05)` (g) `freeze_cond` (h) `diff_bl_frozen_bl>0` (i) `derivative_saturates_wl`

2.3 P-Well Driving During Programming Property

In the programming mode, p-well must always be negative.

STL/PSL Specification

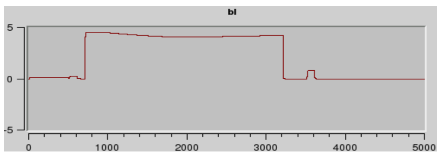
```
vprop pwell {
  pw_during_prog assert:
    always ((a:bl > 2.5 and a:wl > 2.5) ->
        a:pw <= -0.5);
}
```

Informal Description

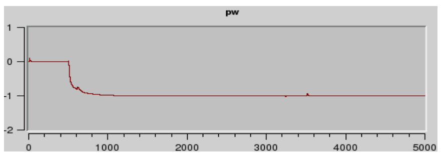
Whenever the programming mode, characterized by the bitline `bl` and the wordline `wl` being both above `2.5`V, is enabled, the p-well `pw` has to be below `-0.5`V.

Results

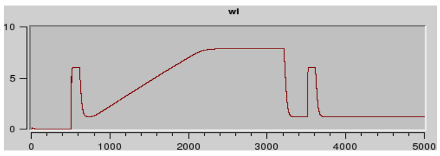
This property is shown to be correct with respect to the simulation trace. From the Figures 6 (a) and (b) showing the `bl` and `wl` respectively, we can see that the left-hand side of the implication `bl > 2.5 and wl > 2.5` is `true` roughly in the interval `[1050,3200)` (Figure 6 (c)). Since the signal `pw` shown in Figure 6 (d) is below `-0.5` from `550` until the end of the simulation (Figure 6 (e)), we can conclude that the property holds.



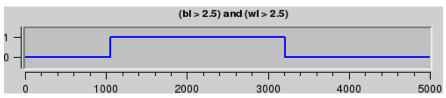
(a)



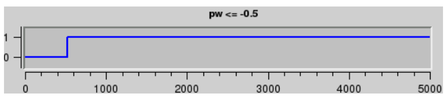
(d)



(b)



(c)



(e)

Figure 6: pwell Property Result: (a) `bl` (b) `wl` (c) `bl>2.5 and wl>2.5` (d) `pw` (e) `pw<=-0.5`

2.4 Erasing Property

In the erasing mode, the bitline must follow the p-well signal.

STL/PSL Specification

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

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

Informal Description

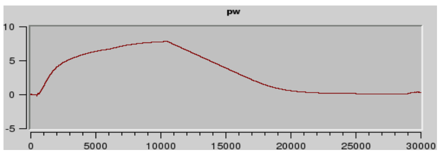
The erasing mode `erasing_cond` is characterized by the wordline signal being lower than `-6V` and p-well `pw` above `5V`. Whenever the erasing condition is enabled the pointwise distance between the source `s` and p-well `pw` voltages has to be smaller than `0.1V` and the value of p-well `pw` cannot be greater than `0.83V` from the value of the bitline `bl`.

Results

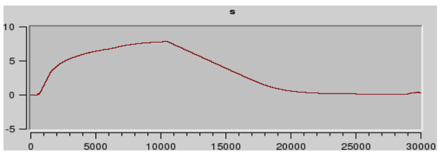
This property is shown to be correct with respect to the simulation trace. The erasing mode `erasing_cond` depending from values of the signals `wl` and `bl` (Figure 7 (c) and (d) respectively) holds between `5000` and `14000ms` roughly. Figures 7 (f) and (g) show that the difference between `bl` and `pw` becomes greater than `-0.83` from `5000ms` onward. Finally, we can see in Figure 7 (h) that the pointwise distance between `s` and `pw` is smaller than `0.1V` from $\sim$`3000ms` until the end of the simulation. Hence, the property holds on these simulation traces.

Notes

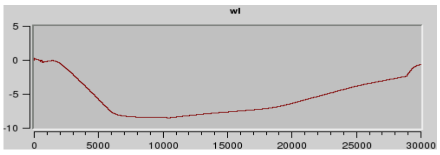
1. The requirement that the bitline `bl` must follow the p-well `pw` signal (with some tolerance allowed) was identified to correspond exactly to the point-wise threshold distance between `bl` and `pw`, and the corresponding STL/PSL distance operator replaced the original specification.
2. The original property specification had the requirement `(bl-pw)>-0.7`. The property was found to be incorrect with the `-0.7` constant, as it represented a too strong constraint. The final specification was relaxed and the original requirement was replaced with `(bl-pw)>-0.83`.



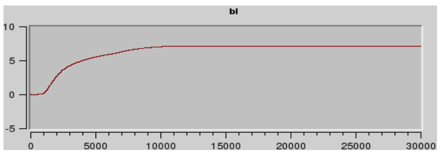
(a)



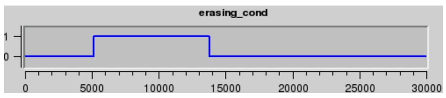
(b)



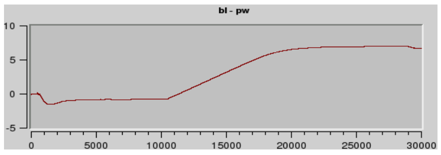
(c)



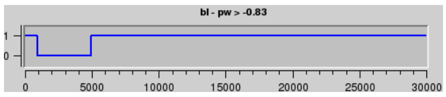
(d)



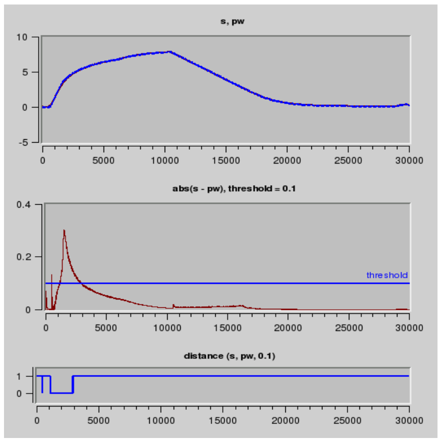
(e)



(f)



(g)



(h)

Figure 7: erase Property Result: (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)`

3 Evaluation

This case study evaluates both the STL/PSL specification language [NM$^+$06] and the performance of the STL/PSL Monitor tool [NM06]. The STL/PSL language evaluation is mainly qualitative, and analysis the expressiveness of the formalism as well as certain aspects observed about the writing of STL/PSL specifications. The evaluation of the STL/PSL Monitor tool is more quantitative and provides the time and memory requirements recorded during the case study.

3.1 Expressiveness of STL/PSL Specifications

In this Section we discuss the expressiveness of the STL/PSL language. We mainly describe the classes of properties that can and cannot be specified in STL/PSL.

Threshold cross detection: This is the basic property of the analog extension of PSL. It allows abstracting the values of a continuous signal to the Boolean domain, thus making the monitoring procedure efficient. The threshold cross detection is used in all the properties of the case study.

Ramp Detection: The signal jumping from one value to another in a short time can be expressed in STL/PSL and efficiently observed by the monitoring tool. For example, `x<=0.1 and eventually![<=10](x>5)` is a property that detects the signal `x` going from below `0.1` to more than `5` in less then `10` time units. The detection of signal ramping is used in properties `decay` and `programming2`.

Distance-based comparisons: Two continuous signals can be compared with respect to some distance metrics. This is especially useful for comparing simulation traces to a perfect reference signal with some degree of tolerance allowed. The pointwise threshold distance-based operator is used in the `decay` and `erasing` properties.

Signal stabilization: STL/PSL can express the stabilization of a continuous signal around an arbitrary value. This can be done by comparing the signal's current value to the value of the same signal shifted by a certain amount of time. The signal stabilization detection is used in the `decay` property.

Raise and fall of Boolean signals: The dense semantics of STL/PSL do not allow pointwise intervals, and hence we cannot detect directly the *raise* and *fall* of a Boolean signal. However, the *raise* and *fall* operators can be approximated to any precision by a combination of existing Boolean and temporal operators. As an example, consider the formula `not p and eventually! [<=m] p`, which is true from `m` time units before `p` raises until its raise. By choosing `m` sufficiently small, we can approach the exact point of `p` raising.

The main class of properties that cannot be expressed in STL/PSL are the ones reasoning about the frequency spectrum of the 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". The quantitative measures are not directly available to the user either, as an STL/PSL property either holds or does not hold on a set of traces. However, a smart usage of the *analog layer* of STL/PSL can reveal many interesting quantitative properties of the continuous signals. This fact has been used in designing distance-based temporal operators, which we were able to reduce to a combination of analog and temporal basic operators.

3.2 Writing STL/PSL Properties

STL/PSL language is a novel but new extension of PSL, and as such it still lacks maturity, making it difficult to evaluate the easiness of specifying properties of continuous signals, as well for the analog designers as for the people from digital verification world that do have a knowledge of PSL. With the current techniques, the evaluation of the properties would require a visual observation of the signals, with manual tracking of interesting events. This process can be automated using STL/PSL specifications. However, we have observed that expressing such properties in STL/PSL is still error prone, mainly due to the inexperience of analog designers with formal languages and logics. Hence, the specification of such properties requires multiple iterations, in order to be sure that they describe exactly the desired behavior of the continuous signals. It is therefore early to quantify the effort that can be saved by using STL/PSL specifications.

We have identified two types of common specification errors that we have encountered during the work on the case study:

Parameter based errors: The threshold value or the time bounds of the temporal operators are not chosen properly. We suggest the property to be first specified with a larger tolerance on time and threshold bounds, in order to get an idea whether the general temporal pattern of the property holds. Afterwards, the parameters can be made tighter, so that the particular thresholds are respected.

Raise and falls of signals: A common pattern in the specification of temporal properties is that enabling a certain condition `p` should trigger an obligation `phi` (where `phi` is a temporal subformula). We have observed that the usual way to specify such a requirement is to use the formula `p->phi`. This is not the correct way to do it, since in this example whenever `p` is true, an obligation `phi` is triggered. We should rather write such a requirement using `rose(p)->phi`, where the obligation `phi` is triggered only at the moment that `p` becomes true. STL/PSL does not allow precise detection of raise and fall of signals, however they can be approximated at any precision.

3.3 Time and Memory Requirements

In this Section we present the time and memory requirement of the STL/PSL Monitor tool that was required for this case study. The tool was tested with the two lightweight-verification algorithms, *offline* and *incremental*. The complexity of the algorithm used in STL/PSL Monitor tool is shown to be $O(k*m)$ 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 (the number of intervals that they have). We can see that the signals generated by the simulation of the erasing mode are about 10 times larger than those generated during the programming mode simulation.

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

Table 2 shows the evaluation results for the *offline* procedure of the tool. The properties monitored over the programming mode simulation required less than half a second. Only the `erasing` property took more than 2 seconds, which is an expected result, given that it is the only property evaluated over a much longer erasing mode simulation. We can also see that the evaluation time is linear in the

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

number of intervals generated by the procedure. We can deduce that the procedure evaluates about 1.000.000 intervals per second.

The time complexity of the *incremental* algorithm is not interesting per se, as the procedure is applied in parallel with the simulation, and hence includes many overheads, such as the processing time of the simulator and the communication costs. On the other hand, the attractiveness of the *incremental* procedure lies mainly in the fact that it does not need to store in memory the entire simulation and monitoring result, ie. the parts of the simulation that have already been evaluated can be discarded. Table 3 compares the memory requirements of the *offline* and *incremental* procedure. For the *offline* procedure we take the total number of intervals generated by the tool after the evaluation of the property. The memory requirements of the *incremental* procedure change dynamically during the evaluation of the property. Hence, we take the maximum number of intervals that was needed to keep in memory by the procedure during the evaluation. As we can see from the fourth column of the Table 3, the memory required by the *incremental* procedure with respect to what is needed in the *offline* mode, varies a lot from one property to another. When the property compares signals in the pointwise fashion, the *incremental* procedure is very efficient, as it can immediately update new values and discard the rest. Hence, the evaluation of the `p-well` property using the *incremental* procedure needs only 0.01% of the memory required by the *offline* algorithm. On the other hand, specifications with nesting of temporal properties require much more memory in the *incremental* mode. For example, the evaluation of the `programming1` property in the *incremental* mode requires almost 70% of the memory needed by the *offline* procedure. This can be explained by the fact that temporal subproperties cannot be updated very often with the available information, so input data cannot be descarded before receiving more information.

property	offline t = total # intervals	online 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

4 Conclusion

The main objective of the analog case study was to validate the concept developed in the PROSYD project of exporting the property-based verification framework to the analog world. The STL/PSL specification and monitoring methods were applied to an industrial Flash memory simulation in order to evaluate certain qualitative and quantitative criteria. We have identified a number of useful classes of analog properties that can be expressed by combining simple *analog* and *temporal* STL/PSL operators. We believe that the STL/PSL language can be extended further in the future to richer analog operations on continuous signals. Writing analog properties in STL/PSL can be error prone for the beginners, and we have identified some common specification mistakes. Finally, the STL/PSL Monitor tool is shown to be an efficient application for automated checking of STL/PSL property correctness with respect to simulation traces.

5 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.

[AFH96] R. Alur, T. Feder, and T.A. Henzinger. The Benefits of Relaxing Punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

[AH92] R. Alur and T.A. Henzinger. Logics and Models of Real-Time: A Survey. In *Proc. REX Workshop, Real-time: Theory in Practice*, pages 74–106. LNCS 600, Springer, 1992.

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

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

[Hen98] T.A. Henzinger. It's about Time: Real-time Logics Reviewed. In *Proc. CONCUR'98*, pages 439–454. LNCS 1466, Springer, 1998.

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

[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.

[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.

[NM06] D. Nickovic, O. Maler, *Manual for Property-based automatic generation of simulation monitors for digital, timed, and analog designs*, PROSYD Deliverable D3.2/13, 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.

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