# Efficient Monitoring of Loose-Ordering Properties for SystemC/TLM

Yuliia Romenska and Florence Maraninchi
Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
CNRS, VERIMAG, F-38000 Grenoble, France

***Abstract*—SystemC Transaction-level modeling (TLM) provides high-level component-based models for SoCs, for which assertion-Based-Verification (ABV) allows property checking early in the design cycle. We introduce the notion of *loose-ordering* to specify when components interact with each other and we propose a set of patterns to capture this notion in assertions.**

**This new notion can already be expressed in languages like PSL, for which there exist tools to generate ABV monitors. But the definition of dedicated patterns makes it easier to write the properties. Moreover we define a direct translation of these patterns into SystemC monitors, and we show that it avoids the combinatorial explosion that would occur during a prior translation into PSL.**

## I. Introduction

SystemC-based Transaction-level modeling (TLM) [8] has been very successful in providing high-level executable component-based models for systems-on-chip (SoCs). The rationale has been to raise the level of abstraction by removing details of lower models like RTL models, especially on timing aspects. The notion of *loose-timing* is very interesting in that perspective. Exact delays in SystemC models (e.g., `wait(100, SC_NS);`) have been identified as a source of over-constraints and spurious synchronizations in models. The *Loose-timing* principle allows to write `wait (90, 110, SC_NS)`, to specify a non-deterministic delay.

*a) Introducing Loose-Ordering:* In this paper, we identify another source of over-constraints. Typically, when a component needs several input data (e.g., the address of an image, the size of it, etc.) before one of the functions it provides (e.g., some transformation of the image) can be started, the *order* in which the input data elements are provided is usually irrelevant. The same is true for data and control outputs. Any specification in which the order is imposed is over-constrained.

This type of property can already be expressed in languages like the Property Specification Language (PSL) [6], [3] (in the whole paper, PSL stands for PSL 1.1). But even simple loose-ordering properties require complex formulas, hence dedicated constructs are helpful.

*b) Efficient Monitoring for Loose-ordering properties:* The first idea is to translate automatically our new properties into PSL, for which there exists monitor generation techniques. However, we will show that this produces complex formulas. Then, even the efficient techniques for exploiting such logics (e.g., the automatic modular generation of monitors described in [10], [13]) cannot do better than producing complex monitors from the obtained complex formulas.

*c) The contributions are:* (i) a new notion called *loose-ordering*, allowing to remove the sources of over-constraints due to the order of interactions between components in a TL model; (ii) a set of patterns to capture these properties; (iii) a translation into PSL, for comparison purposes; (iv) a direct translation into efficient SystemC monitors.

This work has been partially supported by the European CATRENE project CA703 and by the LabEx PERSYVAL-Lab (ANR–11-LABX-0025).
F. Maraninchi is orcid.org/0000-0003-0783-9178

## II. Background and Related Work: Testing and Monitoring Hardware Designs

Figure 1 illustrates the general method for testing a *design-under-verification* (DUV), as presented in the Universal Verification Methodology (UVM) [4], SystemC/SCV [12], eRM [9], [1], SVM [11], etc. A stimuli generator is in charge of producing inputs for the DUV; an assertion checker is in charge of deciding whether the test passes. An additional element of the method is a tool capable of evaluating the coverage of the design obtained with a set of stimuli.

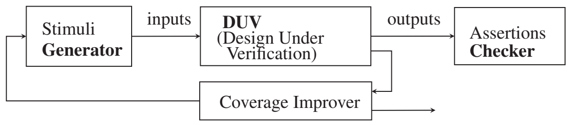

Fig. 1. The verification framework

Assertions can be expressed with languages providing *temporal logic constructs*, i.e., ways of expressing what happens on a (finite or infinite) sequence of states. For instance PSL is built on the Linear Temporal Logic (LTL). The general method can be applied at various levels of abstraction. At the RT level, specifications are synchronous, and all the properties of the DUV can be expressed by specifying what happens on a unique discrete time scale.For TL models, the semantics of specification languages has to be adapted [13]. The idea is to recover the notion of a unique discrete scale on which to interpret the temporal logic constructs, by using the sequence of *events* instead of a unique clock.

In this paper, we focus on TL models. We follow the ideas of [13] for generating efficient monitors from our patterns, in a modular way. We compare with PSL, often used in assertion-based verification. Logics like the duration calculus or its discrete version [7], which offer a “chop” operator to express sequences, could be more appropriate for the expression of our patterns, but they are not usual in the domain.

## III. The Case-Study

Our case-study is an access-control device based on face recognition analysis. A virtual prototype of the system has been implemented using standardized SystemC/TLM (see Fig. 2).

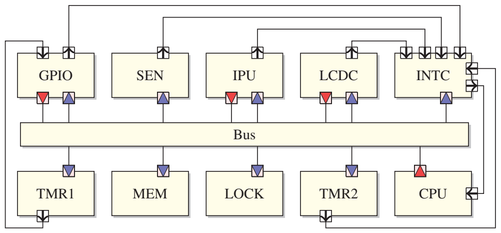

Fig. 2. The TLM platform of the device. It includes: a component to handle buttons (GPIO); an image sensor (SEN); an image processing unit (IPU); an LCD controller (LCDC); an interrupt controller (INTC); two timers (TMR1, TMR2); the system's memory (MEM); a door lock actuator (LOCK); a bus (Bus); and a central processing unit (CPU); the embedded software controls the face recognition process.

We focus on the Image Processing Unit (IPU), which performs face recognition. It contains read/write registers to configure the component; after termination of recognition, the IPU sends an interrupt through its channel *irq*. Its input/output interface is defined as follows: an *input* of the IPU is any action of the other components that affects the IPU (e.g. *set_X* standing for 'write' operation on the register *X*, *start* launching recognition); *output* is any activity performed by the IPU that affects other components (e.g. *read_img* standing for 'read image', *set_irq* for the interrupt signal). The specification of interactions is: (i) the input *start* must be preceded by at least one occurrence of each of the inputs *set_X* for all registers *X*, but the order does not matter (see Example 2 below); (ii) after face recognition has been requested (input *start*), the IPU reads images from the gallery (i.e., repeats the output *read_img* several times in a row) and then sends an interrupt (i.e., produces the output *set_irq*, see Example 3).

## IV. Specification Patterns for Loose-Orderings

We generalize the case-study properties with two patterns for: **antecedent requirements** and **timed implication constraints**. Both patterns are written on the vocabulary of the *input/output interface* $(I, O)$ of the component. The syntax is given by the abstract grammar shown in Fig. 3. The right column of the table is the set of additional constraints defining well-formed formulas of this grammar. They are expressed using $\alpha$, which denotes the set of interface names (inputs or outputs) that appear in the formula. The constraints mainly state that we should not reuse the same interface names in two ranges, or fragments, of the same property. All well-formed formulas are interpreted on sequences where only the names of the root pattern appear; only one name at a time can occur due to asynchrony of considered models. The notion of time used in the timed implication constraint will be mapped directly to the simulation time of the SystemC simulation kernel.

| **Grammar Rule** | **Constraints** |
|---|---|
| a range $\mathcal{R} = n^{[u,v]}$ | $\alpha(\mathcal{R}) = \{n\} \subseteq I \cup O$, $u, v \in \mathbb{N}$ |
| a fragment $\mathcal{F} = (\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}, \sharp)$ | $i \neq j \Longrightarrow \alpha(\mathcal{R}_i) \cap \alpha(\mathcal{R}_j) = \emptyset$, $\sharp \in \{\wedge, \vee\}$ |
| a loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_q$ | $i \neq j \Longrightarrow \alpha(\mathcal{F}_i) \cap \alpha(\mathcal{F}_j) = \emptyset$ |
| $\mathcal{P} = \mathcal{L}$ | $\alpha(\mathcal{P}) \subseteq I \cup O$ |
| $\mathcal{Q} = \mathcal{L}$ | $\alpha(\mathcal{Q}) \subseteq O$ |
| an antecedent requirement $\mathcal{A} = (\mathcal{P} << i, b)$ | $\alpha(\mathcal{P}) \cap \{i\} = \emptyset$, $i \in I, \; b \in \mathbb{B}$ |
| a timed implication constraint $\mathcal{T} = (\mathcal{P} \Rightarrow \mathcal{Q}, t)$ | $t \in \mathbb{N}$ |

Fig. 3. Abstract grammar and constraints for well-formed formulas

Definition 1: Range — A range $\mathcal{R} = n^{[u,v]}$ denotes any sequence of $k$ occurrences of $n$, $n \in I \cup O$ and $k \in [u, v]$.

Definition 2: Fragment — A fragment $\mathcal{F} = (\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}, \sharp)$, $\sharp \in \{\wedge, \vee\}$ is made of sequences $s_1, \ldots s_n$ matching the corresponding ranges. If $\sharp = \wedge$ then all these $s_i$s should appear, concatenated in *any order*; if $\sharp = \vee$, at least one of these $s_i$s should appear, and possibly several of them, concatenated in any order.

Definition 3: Loose-Ordering — A loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_q$ is made of sequences $s_1, \ldots s_n$ matching the corresponding fragments. All the $s_i$s should appear, concatenated in this exact order. *Notice we call it loose-ordering because the order in fragments is free.*

Example 1: Loose-Ordering — Consider the loose-ordering $\ell = n_1^{[2,8]} < (\{n_2, n_3\}, \vee)$. It defines sequences such that: first we have several $n_1$ in a row (the number of occurrences of $n_1$ is in $[2, 8]$); then we have either $n_2$ **or** $n_3$, **or** both in any order.

Definition 4: Antecedent requirement — An antecedent requirement $\mathcal{A} = (\mathcal{P} << i, b)$, $b \in \mathbb{B}$, means that $i$ can occur only if $\mathcal{P}$ has been observed before. When $b$ is true the condition has to be repeated: each occurrence of $i$ should be preceded by its "own" occurrence of $\mathcal{P}$, i.e. an occurrence that happened since the last $i$. When $b$ is false one occurrence of $\mathcal{P}$ is enough to validate all the further occurrences of $i$.

Example 2: Non-repeated requirement with a conjunction — Before starting face recognition the environment of the IPU has to provide values of the image to be analyzed, the address of the image gallery, and the size of the gallery: $((\{$*set_imgAddr*, *set_glAddr*, *set_glSize*$\}, \wedge) <<$ *start*, false$)$.

Definition 5: Timed implication constraint — A timed implication constraint $\mathcal{T} = (\mathcal{P} \Rightarrow \mathcal{Q}, t)$ means that, whenever $\mathcal{P}$ has been observed, $\mathcal{Q}$ should occur, and should have finished before $t$ time units have elapsed since the end of $\mathcal{P}$. This pattern is implicitly of the "repeated" kind: when $\mathcal{P}$ has been observed, $\mathcal{Q}$ should occur, and if a new occurrence of $\mathcal{P}$ is observed, a new occurrence of $\mathcal{Q}$ should occur.

Example 3: Timed implication constraint — If the recognition starts, the IPU reads images from the gallery (i.e., produces several times the output *read_img*) and sends an interrupt (the output *set_irq*). All outputs must be produced during some time interval *T*, which models the duration taken by the face recognition: $(\mathit{start} \Rightarrow \mathit{read\_img}^{[100,60000]} < \mathit{set\_irq}, T)$

## V. Translation into PSL

The translation has been validated with the SPOT tool [2] which translates LTL or PSL formulas into Büchi automata.

*a) Dealing with Ranges:* Temporal logics lack counting facilities. One way to encode ranges is to use a big disjunction of nested $next$ operators encoding all sequences defined by ranges. This encoding is explosive. The alternative approach to encode a range (e.g. $n^{[1,2]}$) is to treat sequences of consecutive occurrences of a range's name (e.g. $n$ and $nn$) as new elements (e.g. $n^1$ and $n^2$). The new vocabulary of $n^{[1,2]}$ is $\alpha = \{n^1, n^2\}$ (instead of $\alpha = \{n\}$). Implementation of this approach includes a lexical analyzer to define new elements.

*b) Encoding of Antecedent Requirements:* The encoding of $\mathcal{A} = (\mathcal{P} << i, \text{true})$ is a big conjunction of the expressions: Asynch – $\bigwedge always(\ not(n_x \wedge n_y))$ for all $n_x, n_y \in \alpha(\mathcal{A})$; MaxOne – before the occurrence of $i$ each name of $\mathcal{P}$ can occur at most once, $\bigwedge always(n_x \rightarrow\ next(not\ n_x\ until!\ i))$ for all $n_x \in \alpha(\mathcal{P})$; Range – before the occurrence of $i$ at most one name per range $\mathcal{R}_k$ of the loose-ordering $\mathcal{P}$ can occur, $\bigwedge always(n_k^x \rightarrow (not\ n_k^y\ until!\ i))$ for all $n_k^x, n_k^y \in \alpha(\mathcal{R}_k)$, for all ranges $\mathcal{R}_k$s; Order – if any of the names of the fragment $\mathcal{F}_k$ occurs, all names of the preceding fragment $\mathcal{F}_{k-1}$ have lost their turn, $\bigwedge always(n_x \rightarrow (not(m_y)\ until!\ i))$ for all $n_x \in \alpha(\mathcal{F}_k)$, for all $m_y \in \alpha(\mathcal{F}_{k-1})$; BeforeI – the name $i$ can occur only after $\mathcal{P}$ has been observed, $\bigwedge(not(i)\ until!\ n_x)$ for all $n_x \in \alpha(\mathcal{P})$; AfterI – the loose-ordering $\mathcal{P}$ should be observed before each occurrence of the name $i$ (i.e., $i$ plays the role of a *reset point*), $\bigwedge always(i \rightarrow next(\bigvee(not(i)\ until!\ n_k^x)))$ where $\bigvee$ stands for all $n_k^x \in \alpha(\mathcal{R}_k)$, for all the ranges $\mathcal{R}_k$s of $\mathcal{P}$ and for all fragments $\mathcal{F}$s of $\mathcal{P}$ with $\sharp = \vee$.

*c) Encoding of Timed Implication Constraints:* We can translate $\mathcal{G} = (\mathcal{P} \Rightarrow \mathcal{Q}, t)$ into PSL applying the encoding proposed above. To do that we need to (i) "concatenate" $\mathcal{P}$ and $\mathcal{Q}$ giving $\mathcal{F}_1^p < \cdots < \mathcal{F}_k^p < \mathcal{F}_1^q < \cdots < \mathcal{F}_\ell^q$, (ii) consider the end of $\mathcal{Q}$ (i.e., the fragment $\mathcal{F}_\ell^q$) as the reset point.

## VI. Modular Monitors in SystemC

All the monitors are expressed as synchronous parallel compositions of elementary recognizers for ranges (Fig. 5). The proposed constructions have been programmed in Lustre [5]; it allows to check their correctness with respect to the intuitive semantics given in section IV using automatic testing tools.

*a) The Recognition Context:* A recognizer for a range works in a recognition context, depending on where the range appears in the syntax tree of $\mathcal{A}$ or $\mathcal{G}$. Consider for example the property of Figure 4. While recognizing $n_3^{[2,8]}$: (i) $n_1$ and $n_2$ are forbidden, since they are supposed to have happened before, the set of such names is denoted as $B$; (ii) $n_4$ is forbidden unless $n_3$ has occurred at least twice (notice that $n_4$ can occur both before and after the range $n_3^{[2,8]}$, because it belongs to the same parent fragment $\mathcal{F}_2$), this set of names is denoted as $C$; (iii) $n_5$ is forbidden until $n_3$ has been observed at least twice, in which case it stops the recognition of the range $n_3^{[2,8]}$ (and starts the recognition of the appropriate range), the appropriate set of names is $A_c$; (iii) $i$ is forbidden since it must occur after $n_3$ and it may not act as a stopping condition, the set of names is $A_f$. Moreover, the recognizer for a range depends on whether its parent fragment has a conjunctive ($\wedge$) or disjunctive ($\vee$) semantics. Therefor, the context for a range recognizer is a tuple $(B, C, A_c, A_f, s)$, where $s \in \{\wedge, \vee\}$ stands for the semantics of the parent fragment.

*b) Elementary Recognizers for Ranges:* The elementary recognizer for a range with context is shown in Figure 5. It is started with the input $start$, termination is signaled by the outputs $ok$ or $nok$. In $s_0$ it is idle and waits to be started; $s_5$ is the error state; in $s_1$ it is started and waits for the first occurrence of its name $n$; in $s_3$ it is counting the occurrences with `cpt`; in $s_2$ it is started and waits for the first occurrence of its name $n$, another range of the same fragment has started; in $s_4$ the minimum number of occurrences of $n$ have been recognized, and another range of the same fragment has started.

*c) Recognizers for Fragments:* The recognizer for a fragment $\mathcal{F} = (\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}, \sharp)$ with $\sharp \in \{\wedge, \vee\}$ is the synchronous parallel composition of the recognizers for the $\mathcal{R}_i$s. At any time these recognizers can be in one of the following global states: (i) all are idle (state $s_0$); (ii) all are waiting in state $s_1$; (iii) exactly one recognizer is counting (state $s_3$) and all others are either still waiting for their names to come

$$\begin{tikzcd}
 & & (\mathcal{P} << i, \text{false}) \arrow[d] & & \\
 & & \mathcal{F}_1 < \mathcal{F}_2 < \mathcal{F}_3 \arrow[dll] \arrow[d] \arrow[drr] & & \\
(\{\mathcal{R}_1, \mathcal{R}_2\}, \wedge) \arrow[d] \arrow[dr] & & (\{\mathcal{R}_1, \mathcal{R}_2\}, \vee) \arrow[d] \arrow[dr] & & (\{\mathcal{R}_1\}, \wedge) \arrow[d] \\
n_1^{[1,1]} & n_2^{[1,1]} & n_3^{[2,8]} & n_4^{[1,1]} & n_5^{[1,1]}
\end{tikzcd}$$

**Attributes** (for each node except the root):
- inherited:
  s, $B$, $C$, $A_c$, $A_f$
- synthesized: $\alpha$

**Example** for $n_3^{[2,8]}$:
s$= \vee$ $B = \{n_1, n_2\}$
$C = \{n_4\}$ $A_c = \{n_5\}$
$A_f = \{i\}$ $\alpha = \{n_3\}$

Fig. 4. The property $((\{n_1, n_2\}, \wedge) < (\{n_3^{[2,8]}, n_4\}, \vee) < n_5 << i, \text{false})$

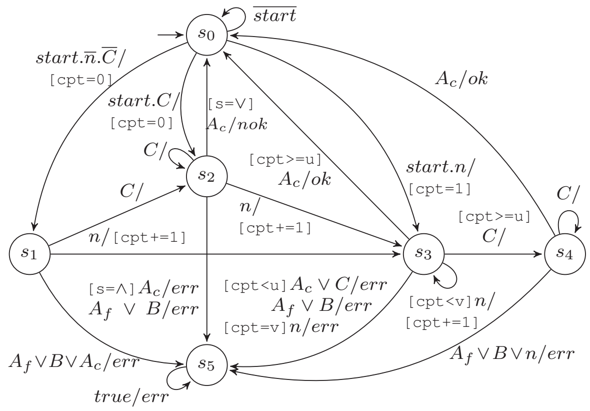

Fig. 5. Elementary recognizer for a range $\mathcal{R} = n^{[\mathtt{u},\mathtt{v}]}$: `cpt` is a counter; $\{start, n, B, A_c, A_f, C\}$ are inputs; $\{err, ok, nok\}$ are outputs. Each transition is of the form `[condition]`$input/output$`[action]` where $input$ is a Boolean formula and $output$ is a set.

(state $s_2$) or have already recognized their ranges (state $s_4$). If the semantics s inherited from the parent fragment $\mathcal{F}$ is disjunctive (i.e. s=$\vee$), each recognizer can be stopped before it has even started counting, provided that at least one of the other ranges of $\mathcal{F}$ has started recognizing a sequence of its name. The recognizer of $\mathcal{F}$ signals termination with the output $ok$ if all the recognizers for the $\mathcal{R}_i$s have signaled termination.

*d) Recognizers for Loose-Orderings:* Consider a loose-ordering $\mathcal{L} = \mathcal{F}_1 < \cdots < \mathcal{F}_q$. The recognizer of $\mathcal{L}$ is made by composing *sequentially* the recognizers of the $\mathcal{F}_i$s: to start recognizing $\mathcal{L}$, we have to send $start$ to the recognizer of $\mathcal{F}_1$; the output $ok$ of the recognizer of $\mathcal{F}_i$ is connected to the input $start$ of the recognizer of $\mathcal{F}_{i+1}$. The output $ok$ of the last fragment signals the stop of the recognizer of $\mathcal{L}$.

*e) SystemC Implementation:* Each node of the syntax tree of the formula is translated into a SystemC monitor. The monitor of a range encodes the state machine shown in Figure 5. Activation of the root monitor is propagated to monitors at lower levels. If any of the range monitors detects an error, the composite monitor of the property reports an error. The monitor of a timed implication constraint $\mathcal{T} = (\mathcal{P} \Rightarrow \mathcal{Q}, t)$ has two SystemC specific variables `sc_core::sc_time start,stop`. `start` is set to the current simulation time when $\mathcal{P}$ is recognized; `stop` is set to the current simulation time when the recognition of $\mathcal{Q}$ is finished; their difference should not be greater than $T$.

## VII. Experiments and Evaluation Results

*a) Experimental Setting:* We consider two strategies to obtain monitors from loose-ordering properties: (i) Drct is the direct translation into SystemC (section VI); (ii) ViaPSL first translates the properties into PSL (section V); then the built PSL encodings are translated into SystemC monitors as described in [13]. The time and memory complexities of the obtained monitors are compared: the former is measured in the number of operations executed by the monitors for each event observed, the latter is defined by the number of bits needed to store the Boolean and bounded Integer variables.

| Configurations | Drct | | ViaPSL | |
|---|---|---|---|---|
| | time (ops) | space (bits) | time (ops) | space (bits) |
| $(n << i, \text{true})$ | 80 | 192 | 238+△ | 896+△ |
| $(n^{[100,60K]} << i, \text{true})$ | 80 | 192 | $4\times10^{11}$+△ | $2\times10^{12}$+△ |
| $((\{n_1,\ldots,n_4\},\wedge) << i, \text{false})$ | 230 | 1132 | 1785+△ | 6720+△ |
| $((\{n_1,\ldots,n_5\},\wedge) << i, \text{false})$ | 280 | 1568 | 2142+△ | 8064+△ |
| $(n_1 \Rightarrow n_2 < n_3 < n_4, T)$ | 296 | 1051 | 1428+△ | 5376+△ |
| $(n_1 \Rightarrow n_2^{[100,60K]} < n_3 < n_4, T)$ | 296 | 1051 | $4\times10^{11}$+△ | $2\times10^{12}$+△ |

Fig. 6. Comparison of Drct and ViaPSL strategies

*b) Comparison:* According to [13] the time and memory complexities of the monitors generated with the ViaPSL strategy are linear in the size of the formula; therefore they are equal to $\Theta(\Delta + \sum_{i=1}^{p}(v_i - u_i + 1)^2 + \sum_{j=2}^{q} \mid \alpha(\mathcal{F}_j) \mid \times \mid \alpha(\mathcal{F}_{j-1}) \mid)$ with $i$ (resp. $j$) ranging over all ranges $\mathcal{R}_i = n^{[u_i,v_i]}$ (resp. fragments $\mathcal{F}_j$) of $\mathcal{A}$ or $\mathcal{G}$, $\Delta$ is a cost of translating ranges into new names.

The *time* complexity of the monitors generated with the Drct strategy is $\Theta(\max_{i\in[1..q]} \mid \alpha(\mathcal{F}_i) \mid)$, $i$ ranging over all the fragments of $\mathcal{A}$ (resp. $\mathcal{G}$); "max" is due to the fact that only monitors of the active fragment work while scanning a sequence. The *space* complexity of the monitor is $\Theta(\sum_{i=1}^{q} \mid \alpha(\mathcal{F}_i) \mid)$ for both Boolean and bounded unsigned Integer variables. The maximum value which can be assigned to any of these Integers is equal to $\max v_i$ for all ranges $\mathcal{R}_i = n_i^{[u_i,v_i]}$ of the pattern.

Figure 6 lists different configurations of the loose-ordering patterns used in the specification of our case-study (see section III). The provided results show that our monitors have always smaller time/space complexities than the monitors obtained from PSL encodings. The presence of non-trivial ranges has no effect on the complexities of our Drct monitors, however their impact on the complexities of the ViaPSL monitors is huge.

## VIII. Conclusion and Further Work

We defined the notion of *loose-ordering* for specifying the interactions between components. We proposed patterns to capture these properties and an encoding into SystemC monitors which is direct, and modular. The encoding is very efficient because it avoids size explosion, and fully exploits the fact that sub-formulas do not share names; when translating into PSL this structural information is lost. Future work will be devoted to a translation of the patterns into some code for generating random sequences. This will provide a full integration of loose-orderings in an ABV framework.

## References

[1] Verisity Design e Reuse Methodology Developer Manual, 2002-2004.

[2] Spot's Online LTL-to-TGBA Translator `spot.lrde.epita.fr/trans.html`, 2015.

[3] I. S. 1850-2005. IEEE Standard for Property Specification Language (PSL), 2005.

[4] Universal Verification Methodology - `www.accellera.org`, 2012.

[5] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. Lustre, a Declarative Language for Programming Synchronous Systems. In *14th Symposium on Principles of Programming Languages*, Munich, Jan. 1987.

[6] B. Cohen, S. Venkataramanan, and A. Kumari. *Using PSL/Sugar for formal and dynamic verification: Guide to Property Specification Language for Assertion-based Verification.* Cohen Publishing, 2004.

[7] M. Hansen and Z. Chaochen. Duration calculus: Logical foundations. *Formal Aspects of Computing*, 9(3):283–330, 1997.

[8] IEEE standard for SystemC language manual, 2011. Computer Society Std.

[9] S. Iman and S. Joshi. *The e hardware verification language.* Dordrecht: Kluwer Academic Publishers. xxii, 349 p., 2004.

[10] K. Morin-Allory and D. Borrione. On-line monitoring of properties built on regular expressions sequences. In S. A. Huss, editor, *Advances in Design and Specification Languages for Embedded Systems (Selected Contributions from FDL'06)*, ISBN :978-1-4020-6147-9, pages 197–207. Springer, 2007.

[11] M. F. Oliveira, C. Kuznik, H. M. Le, D. Große, F. Haedicke, W. Mueller, R. Drechsler, W. Ecker, and V. Esen. The System Verification Methodology for Advanced TLM Verification. In *Proceedings of the Eighth IEEE/ACM/IFIP CODES+ISSS*, pages 313–322, New York, NY, USA, 2012. ACM.

[12] Open SystemC Initiative, SystemC Verification Library, 2014. v2.

[13] L. Pierre and L. Ferro. A Tractable and Fast Method for Monitoring SystemC TL Specifications. *IEEE Transactions on Computers*, 57(10):1346–1356, 2008.