Realtime Regular Expressions for Analog and Mixed-Signal Assertions

John Havlicek
Austin, TX 78728, USA
Email: jwhavlicek@yahoo.com

Scott Little
Freescale Semiconductor
Austin, TX 78729, USA
Email: scott.little@freescale.com

***Abstract*—Syntax and semantics are proposed for realtime (i.e., continuous-time) regular expressions, which extend and generalize existing SVA regular expressions. The extensions are motivated by practical needs for AMS circuit verification and were developed as part of the authors' contribution to analog assertions work in the Accellera committee standardizing Verilog-AMS. Given a suitable notion of sampling, we prove that the realtime semantics provided for the existing SVA clocked digital regular expressions is equivalent to the original discrete semantics. As a result, the existing digital operators can intermix freely with the new realtime operators, which is a major contribution of our framework. We also investigate the theoretical relationship between our framework and the timed regular expressions of Asarin, Caspi, and Maler. We provide a semantically faithful embedding of timed regular expressions into our realtime regular expressions, as well as a construction of timed automata recognizers for our realtime regular expressions. These constructions show that our realtime regular expressions are no less expressive than the timed regular expressions of [1] and no more expressive than the generalized timed regular expressions of [2]. The automata recognizers also provide the basis for an implementation strategy for our framework.**

I. Introduction

Over a number of years, assertion-based techniques have been growing in importance as part of functional verification methodologies for industrial semiconductor designs. This growth is evidenced in part by the standardization of industrially focused assertion languages, like SystemVerilog Assertions (SVA) [3] and Property Specification Language (PSL) [4]. SVA and the Foundation Language of PSL are both discrete-time temporal logics, based upon Linear Temporal Logic (LTL) [5] augmented with regular expressions. The reckoning of time in SVA and in clocked formulas of PSL is in terms of discretely occurring events.[1] The use of events to define units of discrete time works well for the majority of applications to digital circuit verification, although complex timing properties can be challenging to write using event-based assertions [6]. Applications to analog/mixed-signal (AMS) circuits require specification of relationships between events and event-based patterns, but also often involve direct timing requirements. For example, the notion of settling time is common in AMS circuits. Settling time is defined as the amount of time required for a signal to stabilize after a specific event. A property to check the settling time of a digital-to-analog converter (DAC) might ensure that the circuit's output has settled for an input pattern of all zeros, then change the input pattern to all ones and verify that the output settles to its new expected value within a specified time. There are many other examples of AMS properties involving direct timing requirements in the literature [7], [8], [9], [10]. For the expression of many AMS properties, a first class notion of time is needed in the assertion language.

Previous work [11], [12] provides a clear roadmap for extending the LTL features of SVA for AMS applications. In this paper, we define realtime extensions to SVA regular expressions.[2] Our extensions are motivated by practical needs for AMS circuit verification and were developed as part of the authors' contribution to the work of the Analog Assertions Subgroup of the Accellera Verilog-AMS Committee [13]. Many AMS properties rely either on continuously varying quantities whose changes are not confined to clock boundaries or on time constraints whose starting and ending points are not clock aligned. Current event-based assertion languages do not facilitate writing these properties carefully and succinctly. The realtime regular expressions presented in this paper take an important step toward enabling accurate verification of AMS properties. Because of its close alignment with SVA, PSL can be extended using the same approach.

Our semantic framework is based on bounded intervals of the real line. The semantics of realtime regular expressions is defined by a matching relation that specifies when a realtime regular expression matches over a bounded realtime interval of a realtime trace. The interval can be empty, open, closed, or half-open. A fundamental characteristic of our definition is that it is indeed an extension of the current SVA regular expressions. For simplicity of exposition, we omit local variables and the `first_match` operator from SVA. Our definition includes realtime semantics for the existing clocked digital regular expressions, and we prove that the new semantics is equivalent, through a suitable notion of sampling, to the original discrete semantics. In this way, the existing digital operators and forms intermix and combine freely with the new realtime operators and forms. The enablement of harmonious

See http://www.async.ece.utah.edu/~little/pubs/realtimeAppendix.pdf for supplementary material and proofs.

[1]An unclocked formula of PSL is also interpreted over discrete time, but the granularity of time is not specified in the formula.

[2]We would like to acknowledge Himyanshu Anand for his insightful comments during the development of the realtime regular expressions and their semantics.

interplay between discrete and realtime elements of the regular expressions is a major contribution of our work; we had to explore several variations on the semantic framework before we discovered one that achieved this goal.

Our definition adds only one basic realtime form (immediate Boolean) and one primitive realtime operator (Boolean smear) to the existing digital regular expressions.[3] We believe that the preponderance of realtime regular expressions of practical interest can be written using our extension. As in the discrete case, the realtime regular expressions are augmented with a number of useful derived operators. These include flexible concatenation, concatenation with realtime delay, and the realtime goto operator.

We provide a semantically faithful mapping from the timed regular expressions of [1] into our realtime regular expressions, which shows that our formulation is no less expressive. We also give a construction of timed automata recognizers for our realtime regular expressions, which shows that they are no more expressive than the generalized timed regular expressions of [2].

The rest of the paper is organized as follows. In Section 2 we introduce some preliminaries and the notation that will be used throughout the paper. In Section 3 we review digital sequences and their discrete-time semantics. We also define realtime semantics for the digital sequences and prove that the realtime semantics for digital sequences is a faithful generalization to realtime of the discrete-time semantics. In Section 4 we define realtime sequences, illustrate their use in some practical examples, and provide the semantically faithful mapping from the timed regular expressions of [1] into our realtime regular expressions. Section 5 describes the construction of timed automata recognizers for our realtime regular expressions. Section 6 discusses relationships with timed regular expressions, and the paper concludes with a brief discussion of future directions.

II. Preliminaries and Notation

As in SVA, we use the term *sequence* as a synonym for regular expression.

$\mathbb{R}$ denotes the set of real numbers, $\mathbb{R}_{\geq 0}$ denotes the set of non-negative real numbers, $\mathbb{B}$ denotes the set $\{0, 1\}$ of Boolean values, and $\mathbb{N}$ denotes the set of non-negative integers. Let A and D be finite sets. A will be understood as the set of *analog variables*, and D will be understood as the set of *discrete variables*. A *state* (*of the variables*) is an assignment of an element of $\mathbb{R}$ to each analog variable and an element of $\mathbb{B}$ to each discrete variable. A state may be identified with an element of the set $\Sigma = \mathbb{R}^A \times \mathbb{B}^D$.

A *Boolean expression* (*over the variables*) assigns to each state of the variables an element in $\{0, 1\}$. A Boolean expression may be identified with an element of the Boolean algebra $\mathbb{B}^\Sigma = \mathbb{B}^{\mathbb{R}^A \times \mathbb{B}^D}$. We may think of $\mathbb{B}^\Sigma$ as the set of functions $\Sigma \to \mathbb{B}$ in the usual way, so that if b is a Boolean expression and s is a state, then $b(s) \in \mathbb{B}$. We write $s \models b$ iff $b(s) = 1$. In this case, b is said to *occur* at s. An *event* is a Boolean expression from a designated class.[4] Events are denoted by κ and ζ in the remainder of this paper. In realtime, we require events to occur only at isolated points (see below).

A *discrete trace*, or *word*, is a function $w : \{i \in \mathbb{N} : i \leq n-1\} \to \Sigma$, where $0 \leq n \leq \infty$. n is said to be the *length* of the word, which is also denoted $|w|$. The empty word has length 0 and is denoted ε. Throughout, u, v, and w are used to denote words. The concatenation of u and v is denoted by uv. For $i < |w|$, we use w^i to denote $w(i)$, the $(i+1)^{\text{st}}$ letter of w, and we denote by $w^{i..}$ the suffix of w starting at index i. We denote by $w^{i..j}$ the finite sequence of letters starting from index i and ending in index j. That is, $w^{i..j} = (w^i w^{i+1} \cdots w^j)$. A Boolean expression b is said to *occur* in w at i iff $w^i \models b$. A *sampling* is a strictly increasing function $T : \mathbb{N} \to \mathbb{R}_{\geq 0}$ such that $\lim_{n\to\infty} T(n) = \infty$.

A *realtime trace* is a function $W : \mathbb{R}_{\geq 0} \to \Sigma$. Given a realtime trace W and a sampling T, $W \circ T$ is a discrete trace, where $\circ$ denotes the composition of functions. Given a Boolean expression b and a realtime trace W, we say that b *occurs* in W at t iff $W(t) \models b$. The set $\{t \in \mathbb{R}_{\geq 0} : W(t) \models b\}$ is the set of times at which b occurs in the trace W. If κ is an event, then we require that $\{t \in \mathbb{R}_{\geq 0} : W(t) \models \kappa\}$ have no limit point in $\mathbb{R}$. As a result, the points at which a given event occurs cannot be arbitrarily close together.

Throughout, I,J denote *bounded* intervals in the real line $\mathbb{R}$. They may be open, closed, or half-open.

Definition 1:
(a) $I \leq I'$ *iff* $\forall t \in I\ \forall t' \in I' : t \leq t'$.
(b) $I < I'$ *iff* $\forall t \in I\ \forall t' \in I' : t < t'$.

If I is non-empty, then we write $|I| = \sup I - \inf I$. We write $|\{\}| = 0$.

III. Digital Sequences

Digital sequences are the discrete regular expressions used in this paper. They are generated by the following grammar, where κ denotes an event and b denotes a Boolean expression:

$$\sigma ::= \texttt{@(}\kappa\texttt{)}\ \texttt{(}b\texttt{)} \mid \sigma\ \texttt{\#\#1}\ \sigma \mid \sigma\ \texttt{\#\#0}\ \sigma \mid \sigma\ \texttt{or}\ \sigma$$
$$\mid \sigma\ \texttt{intersect}\ \sigma \mid \sigma\texttt{[*0]} \mid \sigma\texttt{[+]}$$

Intuitively, `@(`κ`)` `(`b`)` specifies that the Boolean expression b occur at the nearest point where event κ occurs; `##1` and `##0` are the non-overlapping and overlapping concatenation operators, respectively; `or` is the union operator; `intersect` is the intersection operator; `[*0]` specifies zero repetitions (i.e., the empty word); and `[+]` specifies one or more repetitions.

Digital sequences mimic SVA syntax and are essentially the same as SVA sequences and PSL SEREs.[5] There are some differences regarding where events can be written. In a digital sequence, the event κ can be attached only to a Boolean expression, as in the form `@(`κ`)` `(`b`)`. This restriction

[3]If an application or implementation restricts Boolean manipulation of events, then a second realtime operator (sequence without an event) may be considered primitive rather than derived.

[4]Events may be treated differently than other Boolean expressions in certain tool flows and verification applications, such as digital and analog simulation.

[5]*SERE* stands for *semi-extended regular expression* and is the regular expression sublanguage of PSL [4].

simplifies reasoning about and defining the semantics of digital sequences. SVA and PSL are less restrictive: they allow an event to be specified in more general positions and provide rules to determine the scope of an event. Let's say that a sequence is *basic* if it does not employ any of the following constructs: local variables, `first_match` (SVA only), or endpoint query methods (`triggered` and `matched` in SVA; `ended` in PSL). Any SVA sequence or PSL SERE that is basic can be rewritten as an equivalent digital sequence by eliminating derived operators ([3], Annex F.3.4; [4], Annex B.4); elaborating instances (cf. [3], Annex F.4.1); and eliminating reliance on the scoping rules for events (cf. [3], Annex F.5.1; [4], Annex B.5). For example, the SVA sequence $\texttt{@(}\kappa\texttt{)}\ x\ \texttt{##0}\ y\texttt{[+]}\ \texttt{##1}\ \texttt{@(}\zeta\texttt{)}\ z$ is equivalent to the digital sequence $\texttt{@(}\kappa\texttt{)(}x\texttt{)}\ \texttt{##0}\ \texttt{(@(}\kappa\texttt{)(}y\texttt{))[+]}\ \texttt{##1}\ \texttt{@(}\zeta\texttt{)(}z\texttt{)}$.

In a different sense, digital sequences are less restrictive than SVA sequences. SVA allows multiply clocked sequences to be joined only with `##1` or `##0`, while digital sequences can be combined freely, without regard to how events appear within them. For example, the digital sequence $\texttt{(@(}\kappa\texttt{)(}x\texttt{)}\ \texttt{or}\ \texttt{@(}\zeta\texttt{)(}y\texttt{))[+]}$ is not a legal sequence in SVA.[6] Every digital sequence can be regarded as a PSL SERE by straightforward syntactic translation.

A. Discrete-Time Semantics

Let $w = w^0 w^1 \cdots w^{|w|-1}$ be a finite word. The discrete-time semantics of a digital sequence σ is defined by the matching relation $\models_d$, which is given recursively as follows:

- $w \models_d \texttt{@(}\kappa\texttt{)}\ \texttt{(}b\texttt{)}$ iff $|w| > 0$ and b and κ occur at $w^{|w|-1}$ and κ does not occur at any earlier position of w.
- $w \models_d \sigma\ \texttt{##1}\ \sigma'$ iff there exist u, u' such that $uu' = w$ and $u \models_d \sigma$ and $u' \models_d \sigma'$.
- $w \models_d \sigma\ \texttt{##0}\ \sigma'$ iff there exist u, v, u' such that $uvu' = w$ and $|v| = 1$ and $uv \models_d \sigma$ and $vu' \models_d \sigma'$.
- $w \models_d \sigma\ \texttt{or}\ \sigma'$ iff either $w \models_d \sigma$ or $w \models_d \sigma'$.
- $w \models_d \sigma\ \texttt{intersect}\ \sigma'$ iff both $w \models_d \sigma$ and $w \models_d \sigma'$.
- $w \models_d \sigma\texttt{[*0]}$ iff w is empty.
- $w \models_d \sigma\texttt{[+]}$ iff there exist $n \geq 1$ and $u_1, \ldots, u_n$ such that $w = u_1 \cdots u_n$ and $u_i \models_d \sigma$ for all $1 \leq i \leq n$.

B. Realtime Semantics

This section defines our interval-based realtime semantics for digital sequences and presents a correspondence theorem between the realtime and the discrete-time semantics. Let W be a realtime trace and I be a bounded interval. The realtime semantics of digital sequence σ is defined by the relation $\models_r$, given recursively as follows:

- $W, I \models_r \texttt{@(}\kappa\texttt{)}\ \texttt{(}b\texttt{)}$ iff $\{t \in I \colon W(t) \models \kappa\} = \{\sup I\}$ and $W(\sup I) \models b$.
- $W, I \models_r \sigma\ \texttt{##1}\ \sigma'$ iff there exist J, J' such that $I = J \cup J'$ and $J < J'$ and $W, J \models_r \sigma$ and $W, J' \models_r \sigma'$.
- $W, I \models_r \sigma\ \texttt{##0}\ \sigma'$ iff there exist J, t, J' such that $I = J \cup J'$ and $\{t\} = J \cap J'$ and $J \leq \{t\}$ and $\{t\} \leq J'$ and $W, J \models_r \sigma$ and $W, J' \models_r \sigma'$.
- $W, I \models_r \sigma\ \texttt{or}\ \sigma'$ iff either $W, I \models_r \sigma$ or $W, I \models_r \sigma'$.
- $W, I \models_r \sigma\ \texttt{intersect}\ \sigma'$ iff both $W, I \models_r \sigma$ and $W, I \models_r \sigma'$.
- $W, I \models_r \sigma\texttt{[*0]}$ iff I is empty.
- $W, I \models_r \sigma\texttt{[+]}$ iff there exist $n \geq 1$ and $J_1, \ldots, J_n$ such that $J_i < J_j$ for all $1 \leq i < j \leq n$ and $I = J_1 \cup \cdots \cup J_n$ and $W, J_i \models_r \sigma$ for all $1 \leq i \leq n$.

If σ is a digital sequence and if $\texttt{@(}\kappa\texttt{)}\ \texttt{(}b\texttt{)}$ appears as a subsequence of σ, then we say that κ *is an event of* σ. The following theorem establishes the correspondence between the discrete-time and realtime semantics for digital sequences.

Theorem 1: *Let σ be a digital sequence, let W be a realtime trace, and let $T \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a sampling such that $T(\mathbb{N})$ contains all points of $\mathbb{R}$ at which any event of σ occurs in W. Let $w = W \circ T$. Let I be a bounded interval. If I is empty, then let v be the empty word. Otherwise, assume that I is right-closed with $\sup I \in T(\mathbb{N})$ and let $v = w^{i..j}$, where $i = \min T^{-1}(I)$ and $j = \max T^{-1}(I)$. Then $W, I \models_r \sigma$ iff $v \models_d \sigma$.* □

According to Theorem 1, the realtime semantics for digital sequences is a faithful generalization to realtime of the discrete-time semantics. The proof is by induction and makes use of the following

Lemma 2: *Let σ be a digital sequence. If $W, I \models_r \sigma$ and I is non-empty, then I is right-closed and at least one of the events of σ occurs in W at the right endpoint of I.* □

IV. Realtime Sequences

This section generalizes digital sequences by adding one new basic form and one new primitive operator. The new constructs are motivated by assertion-based applications to AMS verification. Realtime sequences are generated by the following grammar, where κ denotes an event, b denotes a Boolean expression, α denotes a non-negative rational constant, and β denotes either a non-negative rational constant or the special symbol `$`, representing ∞:

$$
\begin{aligned}
R ::= {} & \texttt{@(}\kappa\texttt{)}\ \texttt{(}b\texttt{)} \mid R\ \texttt{##1}\ R \mid R\ \texttt{##0}\ R \mid R\ \texttt{or}\ R \\
& \mid R\ \texttt{intersect}\ R \mid R\texttt{[*0]} \mid R\texttt{[+]} \\
& \mid b \mid b\texttt{[*}\alpha\texttt{[+]:}\beta\texttt{[-]]}
\end{aligned}
$$

The realtime semantics of the digital sequence forms and operators remains as before, while the semantics of the new constructs is given as follows:

- $W, I \models_r b$ iff there exists t such that $I = \{t\}$ and $W(t) \models b$.
- $W, I \models_r b\texttt{[*}\alpha\texttt{[+]:}\beta\texttt{[-]]}$ iff $\alpha \leq [<]\ |I| \leq [<]\ \beta$ and $W(t) \models b$ for all $t \in I$.

A. Derived Realtime Forms

Assertion languages for industrial use typically provide numerous derived forms, with the goal of improving the time efficiency of the engineers deploying them. Both SVA and PSL have many derived sequence forms, all of which can be thought of as extending the present digital and realtime sequences, with syntax adapted as necessary. For example,

[6] An equivalent non-basic SVA sequence can be written using method `triggered` [14].

R`[*]` $\equiv$ R`[*0] or `R`[+]` is the usual Kleene operator. There are also several derived realtime sequence forms that are useful for AMS assertion applications. These are defined as follows, where `!` denotes Boolean negation and the other notational conventions are as before:

- b`[*`α`]` $\equiv b$`[*`α : α`]` [exact-length smear].
- b`[~>1]` $\equiv$ `!`b`[*0.0 : $] ##1` b [realtime goto].
- R `without @(`κ`)` $\equiv$ R `intersect !`κ`[*0.0:$]` [sequence without an event].
- R `#0` R' $\equiv$ `(`R `##0` R'`) or (`R `##1` R'`)` [flexible concatenation].[7]
- R `#[`α`[+]:`β`[-]]` R' $\equiv$ R `#0 1[*`α`[+]:`β`[-]] #0` R' [concatenation with realtime delay].
- R `#[`α`]` $R' \equiv R$ `#[`$\alpha : \alpha$`]` R' [concatenation with exact-length delay].
- R`[*]` $\equiv$ R`[*0] or `R`[+]` [repetition]
- R `and` $R' \equiv$ `((`R `#0 1[*0.0 : $]) intersect` R'`) or (`R `intersect (`R' `#0 1[*0.0 : $]))` [flexible intersection].

In SVA and PSL, the discrete-time goto b`[->1]` is governed by an event and only checks the Boolean condition at occurrences of that event. It can be derived according to `@(`κ`) (`b`[->1])` $\equiv$ `@(`κ`) (!`b`)[*] ##1 @(`κ`) (`b`)`. The realtime goto checks the Boolean condition continuously and advances to the nearest point in time at which the condition is true. Its direct semantics is $W, I \models_r b$`[~>1]` iff $\{t \in I : W(t) \models b\} = \{\sup I\}$.

The flexible concatenation is an important realtime operator. Its direct semantics is the following: $W, I \models_r R$ `#0` R' iff there exist J, J' such that $I = J \cup J'$, $J \leq J'$, $W, J \models_r R$, and $W, J' \models_r R'$. The intervals J and J' being joined must leave no gap and can overlap at most in a shared point. This capability is often needed because the intervals over which realtime sequences match can be open, closed, or half-open.

Consider, for example, `@(`κ`) (`b`) ##1` R. `@(`κ`) (`b`)` matches only a right-closed interval, and `##1` requires the interval over which R matches to abut but not overlap with this interval. If $R = b'$, then the overall sequence cannot match since the realtime Boolean b' matches only over a single point. This incompatibility can be avoided with the flexible concatenation: `@(`κ`) (`b`) #0` R.

Flexibility in matching is important to our semantics because it allows the user to be careful about including or excluding endpoints when needed and not to worry about accounting for endpoints when it is not important. The semantics for `##0` requires that it join a right-closed with a left-closed interval, while `##1` joins a right-closed (resp., -open) interval with a left-open (resp., -closed) interval. Digital sequences and smear-free realtime sequences match only over empty and right-closed intervals. The smear operator introduces the possibility of matching right-open intervals, but whether a right-open interval is actually matched depends on the trace. This flexibility is built into the smear operator, similar to the flexible concatenation operator.

[7]We are grateful to Dejan Nickovic for pointing out that flexible concatenation can be derived in this way.

B. Realtime Sequence Examples

To illustrate the use of realtime sequences we discuss two representative examples. The first illustrates the utility of intermingling digital and realtime sequences. The second illustrates the inadequacy of discrete approximations for realtime specifications.

Let's examine how the settling time specification mentioned in the introduction can be written using realtime sequences. We will make the specification more concrete by specializing it to an 8-bit DAC. The 8-bit DAC input, *in*, is latched on the rising edge of its clock, *clk*. Settling time measurement begins when *in* equals `8'h00`[8] on the input for five cycles, followed by a change to `8'hff` in the next clock cycle. The input is then required to remain `8'hff` throughout the remainder of the measurement. The DAC output, *out*, should then settle to $5\ V \pm 25\ mV$ after 50 ns of latching the `8'hff` input. We understand *settled* to mean that the output remains within the specified voltage range for 25 ns after the initial 50 ns period has passed. The following sequence captures this behavior:

```
@(posedge clk) (in == 8'h00)[*5] ##1
@(posedge clk) (in == 8'hff) #0
( (in == 8'hff)[*0.0:$] intersect
  1 #[50.0n](out < 5.25 && out > 4.75)[*25.0n])
```

The sequence begins by matching the Boolean expression *in* `== 8'h00` for five cycles followed by *in* `== 8'hff`, sampled at posedges of *clk*. The sequence then switches to matching a realtime subsequence where the input remains constant and the output stays within the specified range for 25 ns after the initial 50 ns period. This is an example of the usefulness of the intermixing of realtime and clocked operators within a single sequence.

Many common idioms for AMS circuit verification can be approximated using digital sequences. These approximations typically involve user management of the sampling clock and of auxiliary signals to represent inequalities involving continuously varying quantities. These approximations result in both imprecise assertions and usability challenges. Matching a glitch is an example of one commonly encountered idiom that illustrates user management of the sampling clock. Assume we want to write a sequence to match glitches of 25 ns or less on a signal a. For our purposes a glitch is a short positive pulse of a Boolean. The Boolean may represent a digital signal or a threshold crossing of a realtime signal. A digital sequence to capture these glitches is shown below.

```
@(posedge a) (1) ##1
@(posedge s) (a)[*0:25] ##1
@(posedge s) (!a)
```

s is a clock with a period of 1 ns, which functions as a sampling clock. This sequence provides a reasonable approximation to the specification, but it may miss glitches, as, for

[8]The syntax `8'h00` represents the 8-bit number whose hex value is 00. Similarly, `8'hff` is the 8-bit number whose hex value is ff.

example, in the case that the glitch is less than 1 ns in length and does not stay high across a posedge of s. In fact, the choice of sampling clock is a key decision that must be made by the user when coding the sequence. In this encoding, the sequence is accurate to a precision of 1 ns. Glitches less than 1 ns may be missed, and glitches nearly 27 ns long may result in an undesired successful match. Also, the user must provide the sampling clock, which may add additional complication to the verification environment.

A realtime sequence to match the same types of glitches is shown below.

```
@(posedge a) (1) #0 (!a[~>1] intersect
      1 [*0.0:25.0n])
```

The most notable difference is the time accounting. In the realtime sequence, no sampling clock is needed because the ability to describe time is provided in the language. This sequence matches all of the expected glitches. In a realistic simulation sampling will occur, but it is likely that the user will not directly manage the sampling. Simulator controls that manage sampling should be adequate for most AMS verification applications.

C. Mapping from Timed Regular Expressions

In this section, we show how to map from the timed regular expressions of [1] into our realtime sequences. Further relationships with timed regular expressions are discussed in Section VI. The definition of timed regular expressions in [1] uses the following grammar, where b denotes a Boolean expression and Z denotes an integer bounded interval:

$$\varphi ::= b \mid \varphi \cdot \varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi^* \mid \langle\varphi\rangle_Z$$

The semantics from [1] can be rendered in our notation by the satisfaction relation $\models_{tre}$ defined as follows:

- $W, I \models_{tre} b$ iff $|I| > 0$ and $W(t) \models b$ for every $t \in I$.
- $W, I \models_{tre} \varphi \cdot \varphi'$ iff there exist J, J' such that $J < J'$, $I = J \cup J'$, $W, J \models_{tre} \varphi$, and $W, J' \models_{tre} \varphi'$.
- $W, I \models_{tre} \varphi \vee \varphi'$ iff either $W, I \models_{tre} \varphi$ or $W, I \models_{tre} \varphi'$.
- $W, I \models_{tre} \varphi \wedge \varphi'$ iff both $W, I \models_{tre} \varphi$ and $W, I \models_{tre} \varphi'$.
- $W, I \models_{tre} \varphi^*$ iff there exist $n \geq 0$ and $J_1, \ldots, J_n$ such that $J_i < J_j$ for all $1 \leq i < j \leq n$, $I = J_1 \cup \cdots \cup J_n$, and $W, J_i \models_{tre} \varphi$ for all $1 \leq i \leq n$.
- $W, I \models_{tre} \langle\varphi\rangle_Z$ iff $W, I \models_{tre} \varphi$ and $|I| \in Z$.

This casting of the semantics of timed regular expressions leads directly to a linear syntactic map $\mathfrak{M}$ into realtime sequences:

- $\mathfrak{M}(b) =$ b`[*0.0+:$]`.
- $\mathfrak{M}(\varphi \cdot \varphi') = \mathfrak{M}(\varphi)$ `##1` $\mathfrak{M}(\varphi')$.
- $\mathfrak{M}(\varphi \vee \varphi') = \mathfrak{M}(\varphi)$ `or` $\mathfrak{M}(\varphi')$.
- $\mathfrak{M}(\varphi \wedge \varphi') = \mathfrak{M}(\varphi)$ `intersect` $\mathfrak{M}(\varphi')$.
- $\mathfrak{M}(\varphi^*) = \mathfrak{M}(\varphi)$`[*]`.
- $\mathfrak{M}(\langle\varphi\rangle_Z) = \mathfrak{M}(\varphi)$ `intersect` $\mathfrak{M}(Z)$, where
  - $\mathfrak{M}([\alpha, \beta]) =$ `1[*`α`:`β`]`
  - $\mathfrak{M}((\alpha, \beta]) =$ `1[*`α`+:`β`]`
  - $\mathfrak{M}([\alpha, \beta)) =$ `1[*`α`:`β`-]`
  - $\mathfrak{M}((\alpha, \beta)) =$ `1[*`α`+:`β`-]`

Semantic faithfulness of $\mathfrak{M}$ is given by the following

Proposition 3: *$W, I \models_{tre} \varphi$ iff $W, I \models_r \mathfrak{M}(\varphi)$.* □

Proposition 3 holds for any trace W and bounded interval I. It should be noted that in [1], timed regular expressions are interpreted over a restricted class of realtime traces, called *signals*, that are piecewise constant and left continuous. The piecewise constant condition requires the set of discontinuities of the trace to have no limit point in $\mathbb{R}$, so, in particular, there can be at most finitely many discontinuities in any bounded interval of the trace. The condition of left continuity implies that no event can occur in a signal. Furthermore, [1] restricts the domain of a signal to be a bounded interval that either is empty or is left open and right closed.

V. Automata Construction

This section provides a construction for timed automata recognizers for our realtime regular expressions. The automaton $\mathcal{A}$ constructed for sequence R recognizes R in the sense that for all W and I, $W, I \models_r R$ iff $\mathcal{A}$ has an accepting run whose trace is satisfied by W over the interval I. These notions will be made precise below. The construction provides the basis for an implementation strategy for our framework. The definition of timed automaton below is based on that in [1].

A. Definition of Timed Automaton

A timed automaton is a tuple $\mathcal{A} = (Q, C, \Delta, \Gamma, L, S, F)$ where Q is a finite set of states, C is a finite set of clocks, Δ is a transition relation (see below), Γ is an alphabet that we assume to be a Boolean algebra with multiplicative unit 1, $L : Q \to \Gamma$ is the state labeling, $S \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of final (accepting) states. The transition relation, Δ, consists of tuples of the form (q, ϕ, ρ, q') where $q, q' \in Q$, $\rho \subseteq C$ (the set of clocks to be reset), and ϕ is a Boolean combination of terms of the form $(c \in I)$, where $c \in C$ and I is an interval of $\mathbb{R}_{\geq 0}$ whose endpoints are rational numbers or ∞.

A *clock valuation* is a function $\mathbf{v} : C \to \mathbb{R}_{\geq 0}$. We denote the space of all clock valuations by $\mathcal{H}$. By using clocks as coordinates, a clock valuation can be identified with a vector in $\mathbb{R}^C$. $\mathbf{0}$ denotes the vector $(0, 0, \ldots, 0)$ and $\mathbf{1}$ denotes the vector $(1, 1, \ldots, 1)$. A *configuration* of the automaton is a pair $(q, \mathbf{v}) \in Q \times \mathcal{H}$. Every ρ creates a reset function $Reset_\rho : \mathcal{H} \to \mathcal{H}$ defined by

$$Reset_\rho(\mathbf{v})(c) = \begin{cases} 0 & \text{if } c \in \rho \\ \mathbf{v}(c) & \text{if } c \notin \rho \end{cases}$$

A *finite run* of the automaton on $[a, b]$ is a sequence

$$t_0, (q_0, \mathbf{v}_0) \xrightarrow[t_1]{d_1} (q_1, \mathbf{v}_1) \xrightarrow[t_2]{d_2} \cdots \xrightarrow[t_n]{d_n} (q_n, \mathbf{v}_n),$$

where $q_i \in Q$, $\mathbf{v}_i \in \mathcal{H}$, $d_i \in \Delta$, $t_i \in \mathbb{R}_{\geq 0}$ are such that

- $0 \leq a = t_0 \leq t_1 \leq t_2 \leq \cdots \leq t_n = b$ and $\mathbf{v}_0 = \mathbf{0}$.
- $d_i = (q_{i-1}, \varphi_i, \rho_i, q_i)$, where $\mathbf{v}_i = Reset_{\rho_i}(\mathbf{v}_{i-1} + (t_i - t_{i-1}) \cdot \mathbf{1})$ and φ_i is satisfied on $\mathbf{v}_{i-1} + (t_i - t_{i-1}) \cdot \mathbf{1}$.

The run is *accepting* if $q_0 \in S$ and $q_n \in F$. The *full trace* of the run is the function $T : [t_0, t_n] \to \Gamma$ defined by the following rules:

- If $t_i < t_{i+1}$, then for $t \in (t_i, t_{i+1})$, $T(t) = L(q_i)$.
- If $t_{i-1} < t_i = t_{i+1} = \cdots = t_{j-1} = t_j < t_{j+1}$, then $T(t_i) = L(q_i) \wedge \cdots \wedge L(q_{j-1}) \in \Gamma$.
- If $T(t)$ is not defined by the preceding conditions, then $T(t) = 1 \in \Gamma$.

We assume that each initial and final state is classified either as *inclusive* or *exclusive*. The *restricted trace* of a run is the function $T|_I$, where $T : [t_0, t_n] \to \Gamma$ is the full trace and $I \subseteq [t_0, t_n]$ is the interval obtained from $[t_0, t_n]$ by including or excluding each of the endpoints t_0 and t_n in accordance with the kind, inclusive or exclusive, of the initial and final states of the run, respectively.[9] If the initial and final states are both exclusive and $t_0 = t_n$, then $I = \{\}$.

Let W be a realtime trace, $\mathcal{A}$ be a timed automaton, and I be a bounded interval. For I non-empty, we define $W, I \models \mathcal{A}$ iff there exists a run of $\mathcal{A}$ on $[\inf I, \sup I]$ such that I is the domain of the restricted trace of the run and $W(t) \models T(t)$ for all $t \in I$, where T is the trace (full or restricted) of the run. We define $W, \{\} \models \mathcal{A}$ iff there exists $t_0 \in \mathbb{R}_{\geq 0}$ and a run of $\mathcal{A}$ on $[t_0, t_0]$ such that the domain of the restricted trace of the run is $\{\}$. We say that $\mathcal{A}$ *recognizes* the sequence R if for all W and I, $W, I \models_r R$ *iff* $W, I \models \mathcal{A}$.

B. Automata Convenience Features

To simplify the exposition of our automata construction we use additional features and notations described below. By definition, clock constraints ϕ appear only on transitions of a timed automaton. It can be convenient to specify a timing condition on a state, where the timing condition restricts the amount of time that a run may spend in a single visit to the state (distinct visits are treated independently). Such conditions can be implemented by (at worst) adding a single *state clock* η that is reset on every transition and such that the timing condition of a state is added to each outgoing transition.[10] The state timing condition "0" abbreviates "$\eta = 0$", i.e., no time elapse in the state. Such a state is called a 0-time state. The state timing condition "+" abbreviates "$\eta > 0$", i.e., positive time must elapse in each visit to the state. Such a state is called a +-time state and is annotated by a "+" in the lower half of the state in figures.

The label of a state conditions the trace of a run for the times that the run is in the state. It can be useful also to condition with a label the times at which a transition is taken. A transition label can be implemented by inserting a 0-time state and placing the label on the new state.

Ingresses and egresses provide a graphical notation for simplified initial and final states and their classification as inclusive or exclusive. An *ingress* is an initial state with no incoming and a single outgoing transition, while an *egress* is a final state with no outgoing and a single incoming transition. The ingress and egress states are 0-time and their state label is understood to be $1 \in \Gamma$. If needed, a labeling condition may be placed on the incident transition. Inclusive states are denoted by small closed circles, while exclusive states are denoted by small open circles. For example, the automaton in Fig. 1 has three ingresses, one exclusive and two inclusive. The transitions from the inclusive ingresses are labeled "$\neg\kappa$" and "$\kappa \wedge b$", respectively. The automaton has one egress, which is inclusive and has no label on its incident transition.

C. Automata Construction

The automata are built by induction on the structure of the sequences.

- The automaton for `@(`κ`)` `(`b`)` is shown in Fig. 1.

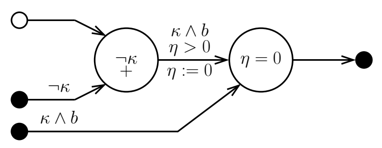

Fig. 1. Automaton for a clocked Boolean, `@(`κ`)` `(`b`)`.

- The automaton for R `##1` R' is created by connecting the automata for R and R'. The rule for connection requires that an inclusive ingress/egress be connected to an exclusive egress/ingress. This rule ensures that there is no overlap and no gap in the interval matched. When an ingress/egress is connected it is no longer an ingress/egress and consequently is no longer an initial/final state, respectively. An example of how the connection works is shown in Fig. 2. This connection rule does not apply to a subautomaton for empty, as empty is an identity for `##1`. Instead, the subautomaton is combined as though it were an identity.
- The automaton for R `##0` R' is created by connecting the automata for R and R'. The rule for connection requires that inclusive egresses of R be connected to inclusive ingresses of R'. This ensures that there is a single point overlap between the matches for R and R', which is required by `##0`. The connection works in a manner similar to R `##1` R'.
- The automaton for R `or` R' is created using a standard union of the automata for R and R'.
- The automaton for R `intersect` R' is created as a product automaton for R and R'. The rules for the product construction are as follows:
  - (p, q) is an ingress (resp., egress) iff both p and q are ingresses (resp. egresses) and either both p and q are inclusive or both p and q are exclusive.
  - (p, q) is a +-time state iff both p and q are +-time states.
  - (p, q) is a 0-time state iff either p or q is a 0-time state.
  - *Parallel transitions* are transitions where both of the factor automata are changing state. If $p \xrightarrow{\alpha} p'$ is a

[9] Inclusion is understood to take precedence over exclusion in the case where $t_0 = t_n$ and the classifications of the initial and final states do not agree.

[10] A final state with a timing condition can be implemented by rendering the state non-final and adding a companion final state to which it transitions. The companion state matches the original in its label and classification.

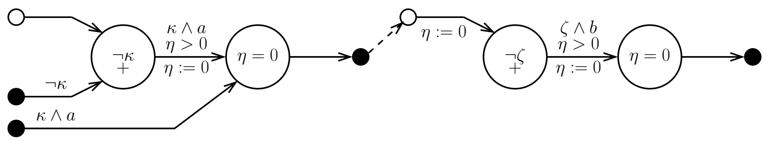

Fig. 2. An automaton for `@(`κ`)` `(`a`)` `##1` `@(`ζ`)` `(`b`)`. This demonstrates the connection rules for `##1`.

transition of R and $q \xrightarrow{\beta} q'$ is a transition of R', then $(p, q) \xrightarrow{\alpha\wedge\beta} (p', q')$ is a parallel transition of the product.

– *Stutter transitions* are transitions where only one of the factor automata is changing state. The non-changing state is said to be *stuttering*. There is a subtlety when the stuttering state is +-time. The stuttering transition can lead to gaps in the state label for this factor because a trace associates a state label with the interior of a positive length time interval spent in that state. To avoid gaps, the state label needs to be added to the stutter transition in certain circumstances. Intuitively, the case where the state label is added occurs when the changing state is transitioning from a 0-time state to a +-time state and the stuttering state has already been occupied for positive time in this visit. A complete definition of the rules is found below. Each factor state is either 0-time, denoted, e.g., ${}^0q$, or +-time, denoted, e.g., ${}^+q$. A mixed product state is one with one factor state 0-time and one factor state +-time. It is annotated with a *kind* 0 or 1. A kind 0 state indicates that the +-time factor has not yet been occupied for positive time in this visit, while a kind 1 state indicates that the +-time factor has been occupied for positive time in this visit. A parallel transition is denoted by two solid arrows, while a stutter transition is denoted by one solid and one dashed arrow. Rule g shows the addition of a state label to a stutter transition. Rules a-h below describe all the transition forms involving mixed product states of various kinds, up to swapping the factors of the tuples. Each rule has a dual which is also valid. In the description, * allows any possible qualifier and e is a variable used to represent kind 0 or 1.

a. $\begin{pmatrix} {}^*p \\ {}^*q \end{pmatrix}_* \begin{matrix}\longrightarrow \\ \longrightarrow\end{matrix} \begin{pmatrix} {}^0p' \\ {}^+q' \end{pmatrix}_0$ **b.** $\begin{pmatrix} {}^0p \\ {}^+q \end{pmatrix}_1 \begin{matrix}\longrightarrow \\ \longrightarrow\end{matrix} \begin{pmatrix} {}^*p' \\ {}^*q' \end{pmatrix}_*$

c. $\begin{pmatrix} {}^0p \\ {}^+q \end{pmatrix}_1 \begin{matrix}\dashrightarrow \\ \longrightarrow\end{matrix} \begin{pmatrix} {}^0p \\ {}^*q' \end{pmatrix}_*$ **d.** $\begin{pmatrix} {}^0p \\ {}^*q \end{pmatrix}_* \begin{matrix}\dashrightarrow \\ \longrightarrow\end{matrix} \begin{pmatrix} {}^0p \\ {}^+q' \end{pmatrix}_0$

e. $\begin{pmatrix} {}^0p \\ {}^+q \end{pmatrix}_e \begin{matrix}\longrightarrow \\ \dashrightarrow\end{matrix} \begin{pmatrix} {}^0p' \\ {}^+q \end{pmatrix}_e$ **f.** $\begin{pmatrix} {}^0p \\ {}^+q \end{pmatrix}_0 \begin{matrix}\longrightarrow \\ \dashrightarrow\end{matrix} \begin{pmatrix} {}^+p' \\ {}^+q \end{pmatrix}$

g. $\begin{pmatrix} {}^0p \\ {}^+q \end{pmatrix}_1 \begin{matrix}\longrightarrow \\ \underset{L(q)}{\dashrightarrow}\end{matrix} \begin{pmatrix} {}^+p' \\ {}^+q \end{pmatrix}$ **h.** $\begin{pmatrix} {}^+p \\ {}^+q \end{pmatrix} \begin{matrix}\longrightarrow \\ \dashrightarrow\end{matrix} \begin{pmatrix} {}^0p' \\ {}^+q \end{pmatrix}_1$

.

- The automaton for R`[*0]` is shown in Fig. 3(a).
- The automaton for R`[+]` follows the connection rule for R `##1` R'. A connected ingress/egress no longer functions as ingress/egress. In a repetition the initial/final state behavior must be maintained, so ingresses/egresses for the repetition must be duplicated for use in the connection. If R has an empty subautomaton, then so does R`[+]`, but the empty subautomaton does not play a role in the connections. See Fig. 3(b) for an example.
- The automaton for b is shown in Fig. 3(c).

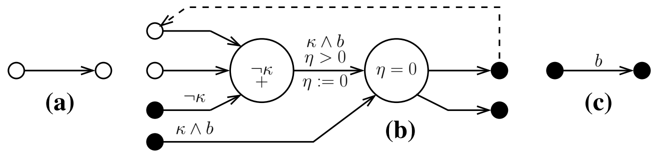

Fig. 3. Automata for (a) R`[*0]`, (b) `(@(`κ`)` `(`b`))[+]`, and (c) realtime Boolean, b.

- The automata for b`[*`α`[+]` `:` β`[-]]` are shown in Fig. 4.

VI. Further Relationships with Timed Regular Expressions

In [1], [2], Asarin, Caspi, and Maler define *timed regular expressions* and study their relationship to timed automata. The main results show that timed regular expressions, generalized to support renaming, have the same expressive power as timed automata. This section discusses relationships between our realtime sequences and timed regular expressions.

The syntax for timed regular expressions is essentially the same in [1] and [2] and has been given in Section IV-C. [2] adds syntax for the empty word, analogous to our form R`[*0]`. [2] also adds syntax to specify renamings, which have no analog in our framework.

Several different semantic models are presented in [1] and [2]. The piecewise constant, left continuous signals of [1] are, in many regards, the closest to our realtime traces. They underly the relation $\models_{tre}$ and provide the basis for the semantically faithful embedding $\mathfrak{M}$ presented in Section IV-C. Signals do not, however, support expression that a condition hold or an event occur at a specific point in time. These capabilities are important for applications to AMS verification and are supported in realtime sequences by the immediate Boolean (b), clocked Boolean (`@(`κ`)` `(`b`)`), and concatenations with overlap (`##0`, `#0`, etc.).

Time-event sequences are the primary semantic model in [2]. A *time-event sequence* is a sequence of terms, each of which is either an event or a non-negative real number, specifying a time elapse. For example, the time-event sequence $r \cdot \kappa \cdot s \cdot \zeta$ represents that event κ occur after r units of time and that event ζ occur after another s units of time. Simultaneity of multiple events is expressible, as in $r \cdot \kappa \cdot \zeta$,

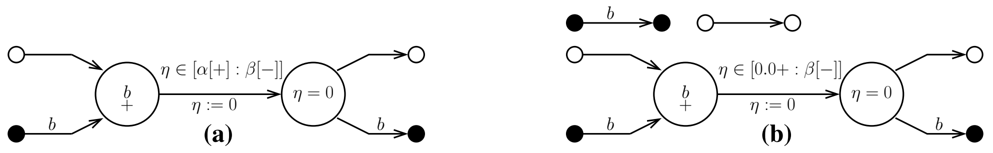
b
+
$\eta \in [\alpha[+] : \beta[-]]$
$\eta := 0$
$\eta = 0$
b
b
(a)
b
b
+
$\eta \in [0.0+ : \beta[-]]$
$\eta := 0$
$\eta = 0$
b
b
(b)

Fig. 4. Automata for (a) the Boolean smear, `b[*`α`[+] : `β`[-]]` with a positive α and (b) the Boolean smear, `b[*0.0 : `$\beta[-]$`]` with α equal to zero.

$r \cdot \zeta \cdot \kappa$, $r \cdot \kappa \cdot \kappa$, which are all distinct. In [2], timed regular expressions can specify unbounded regular patterns of events, all occurring at the same time but with discrete ordering amongst themselves. For applications to AMS verification, we do not believe that such capabilities are needed. For example, we see no practical use for expressing the condition that the value of a particular analog variable cross a particular threshold five times, say, at a single instant of time. Our realtime traces do not admit this granularity of ordering at a single time. In a realtime trace, an event either occurs or does not at any particular time. There is no notion of multiplicity, and if two events occur at the same time, there is no distinction of their order. Our realtime sequences can express the condition that a fixed set of events occur simultaneously, as in `(@(`κ`)(1) ##0 @(`ζ`)(1)) intersect 1`. This form also shows a syntactic order of κ before ζ, but our realtime trace models do not resolve this order.

In Section V we showed a construction of timed automata recognizers for our realtime sequences. Assuming that suitable translation conventions are fixed to convert between the differing semantic models, this construction shows that our realtime regular expressions are no more expressive than the generalized extended timed regular expressions of [2]. Definitive comparison of the two regular expression languages seems to depend on precise reconciliation of the semantic models. In mapping from time-event sequences to realtime traces, multiplicity and ordering of simultaneous events need to be encoded using analog and discrete variables. In mapping the other direction, behaviors of analog and discrete variables to which the relevant Boolean expressions and events are sensitive need to be represented by regular patterns of events. The details of such analysis appear non-trivial and merit consideration in future work.

VII. Conclusion

Verification of AMS systems is becoming increasingly important as AMS designs become more popular and complex. To meet the needs of AMS verification, we must develop verification techniques and languages that support both the clocked and realtime domains. We have proposed syntax and semantics for realtime regular expressions. This has been done before [1], [2], but the key feature of our framework is that it generalizes the framework of the existing SVA regular expressions. This feature allows free intermingling of realtime and digital sequences, which enables our realtime regular expressions conveniently to represent complex properties that specify both clocked and realtime requirements. We have investigated how the new syntax and semantics relate to existing definitions of realtime regular expressions. We provide a semantically faithful mapping from the timed regular expressions of [1], which demonstrates that our formalism is not less expressive. We also provide a construction of timed automata recognizers for our realtime regular expressions. This construction demonstrates that our realtime regular expressions are no more expressive than the generalized timed regular expressions of [2] and provides a basis for an implementation strategy.

In the future, we plan to demonstrate how this semantics can be extended to local variables and the `first_match` operator. We also plan to develop similarly compatible semantics for the SVA property operators. When completed, these pieces will constitute a realtime extension to the full SVA language. This new realtime SVA language will provide engineers the ability to specify complex AMS properties accurately and in a single assertion language.

References

[1] E. Asarin, P. Caspi, and O. Maler, "A Kleene theorem for timed automata," in *IEEE Symposium on Logic in Computer Science*. IEEE Press, 1997, pp. 160–171.

[2] ——, "Timed regular expressions," *Journal of the ACM (JACM)*, vol. 49, no. 2, pp. 172–206, 2002.

[3] *IEEE Standard for SystemVerilog (1800-2009)*, IEEE Computer Society, Dec. 2009.

[4] *IEEE Standard for Property Specification Language (PSL) (1850-2010)*, IEEE Computer Society, Jun. 2010.

[5] A. Pnueli, "The temporal semantics of concurrent programs," *Theor. Comput. Sci.*, vol. 13, pp. 45–60, 1981.

[6] D. Smith, "Asynchronous behaviors meet their match with SystemVerilog Assertions," in *Design and Verification Conference (DVCON)*, 2010.

[7] D. Nickovic, O. Maler, A. Fedeli, P. Daglio, and D. Lena. (2007) Analog case study : PROSYD deliverable 3.4/2. [Online]. Available: http://www.prosyd.org/twiki/pub/Public/DeliverablePageWP3/prosyd3.4.2.pdf

[8] K. D. Jones, V. Konrad, and D. Nickovic, "Analog property checkers: a DDR2 case study," *Formal Methods in System Design*, vol. 36, no. 2, pp. 114–130, 2010.

[9] R. Mukhopadhyay, S. K. Panda, P. Dasgupta, and J. Gough, "Instrumenting AMS assertion verification on commercial platforms," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, pp. 21:1–21:47, April 2009.

[10] S. Mukherjee and P. Dasgupta, "Incorporating local variables in mixed-signal assertions," in *TENCON*, 2009, pp. 1–5.

[11] R. Alur, T. Feder, and T. A. Henzinger, "The benefits of relaxing punctuality," *Journal of the ACM*, vol. 43, no. 1, pp. 116–146, 1996.

[12] D. Nickovic, "Checking timed and hybrid properties: Theory and applications," Ph.D. dissertation, Université Joseph Fourier, 2009.

[13] H. Anand, J. Havlicek, and S. Little. (2010) Some notes on realtime semantics. [Online]. Available: http://www.vhdl.org/twiki/pub/VerilogAMS/RequirementsGatheringGroup/semantics.pdf

[14] E. Cerny, S. Dudani, J. Havlicek, and D. Korchemny, *The Power of Assertions in SystemVerilog*. Springer, 2010.