Dynamic Assertions Using TXP

Surrendra Dudani [1] João Geada [1] Grzegorz Jakacki [1]
Daniel Vainer [1]

Abstract

In this paper, we present a new temporal property specification language TXP. The language is designed to support dynamic monitoring of temporal properties at simulation runtime, as well as to provide the input specification for formal property checking. For design verification of hardware systems, hardware description languages (HDL) provide modeling capabilities, but they are inadequate for concise specification of complex assertions where logic relationships involve multi cycle behavior. TXP is a declarative language that provides a rich set of operators based on regular expressions over sequences of values and events. Its key features are to allow multi-cycle behavior, time shift operations in the past or future, conditional matching, repetition of sequences, and restrictions over sequences. The sequences can be constructed with logical connectives such as "and" and "or" to compose more complex assertions. A TXP engine has been developed to monitor the properties at runtime.

1 Introduction

A new temporal property specification language TXP [2] has been developed by Synopsys. Using this language, users can write assertions to verify hardware system properties during simulation at runtime, or feed those assertions to formal or semi-formal verification tools for static checking. Numerous specification languages based on temporal logic [1,2] have been proposed for formal methods such as model checking and theorem proving, but their adoption by the hardware designers have been slow. TXP has been developed with hardware design as the main application, and addresses the problem of usability by providing constructs using concepts familiar to hardware designers.

The TXP language provides a way to accurately and efficiently describe control behaviors that span over multiple cycles and modules of the design under test. For hardware developers, timing and order related functional problems are difficult to detect and most expensive to fix, causing project

[1] Synopsys, Inc.

[2] TXP is part of OpenVera and has been made available as part of Vera Open Source Initiative by Synopsys, Inc.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

delays and cost overruns. Commonly used HDLs such as Verilog [4] and VHDL [5] were designed to model hardware at a Register Tranfer Level (RTL) (i.e, on a single cycle transitions) and provide procedural features to describe the system. Clearly, such an execution model is not adequate to concisely specify multi-cycle functions. One can argue that HDLs are design languages, not specification languages, and as such a new specification methodology is required, particularly in the light of increasing design size and complexity. Using a specification method to easily and intuitively express input/output behavior, bus protocols, and other complex relationships of the design under test, a very efficient dynamic monitoring system can be developed to provide detection of faulty behavior during verification.

The TXP language is a declarative language whose semantics are based on regular expressions [3]. There have been some proposals to extend temporal logic [6] for providing the expressive power of regular expressions. TXP builds upon those concepts to provide a syntax that is intuitive to hardware designers. The decision to base the language on regular expressions was two fold: one, regular expressions fit well with the sequencing operations so well understood by hardware designers; and two, regular languages can be translated easily to a finite state model that can be used by the formal methods to perform formal property checking.

There is also some history of work on temporal logics for specifying properties of software systems. The Bandera project described in [7] has developed a system for JAVA language in which common occuring temporal properties can be specified. These properties are location dependent in the code and get fired when the program execution reaches a control point.

TXP specifications allow a user to express the following sequencing features:

- The basic sequencing construct to specify that an event follows another
- Sequences in the past or future
- Composite sequences using logic connectives(and,or,inv)
- The repetition of sequences
- Conditions to hold during a sequence
- Time bounded sequences
- User specified clock or simulation time as sampling unit of time

In section 2, we describe the theoretical foundations for TXP and a formal semantic structure to define the language. Section 3 shows the run-time verification environment and explains the simulation model. In section 4, we describe the language features and some examples to illustrate the usage of the language constructs. Finally, Section 5 summarizes the current work and outlines the future work.

2 Theoretical Foundation

This section describes the theoretical basis on which TXP semantics are developed. Section 2.1 describes the operators that are defined over the standard regular expressions. Section 2.2 further constructs definitions for negation of operators. The semantics of each TXP construct is later defined in terms of the definitions described in this section.

2.1 Regular expressions over Valuations

We deal with *temporal sequences* (*t-sequences*) over the alphabet of valuations. Valuation is complete description of the state of the design at a particular moment. To the standard signature of algebra of sequences, we add an operator $\_^\flat$. The carrier of the new algebra **VTSeq** is $\mathbf{Val}^\star \times \mathbf{Nat} \times \mathbf{Nat}$, where **Val** is the set of design valuations and **Nat** is the set of natural numbers. In the t-sequence (v, n, m), n and m represent the placement of "virtual beginning" and "virtual end" of the t-sequence with respect to the beginning of v. We define:

- $\varepsilon = (\varepsilon, 0, 0)$
- $(u, n', n'')^\flat = (u, n', n')$

The interpretation of

$$(u, n', n'') \cdot (v, m', m'')$$

is defined as $(w, n' + i, m'' + j)$ if w is the shortest sequence such that (1) u is a subsequence of w at position i, (2) v is a subsequence of w at position j and (3) $n'' + i = m' + j$. The concatenation is undefined if such i, j, w does not exist.

Observe, that the interpretation of concatenation is a partial function. The t-sequences of the form $(v, 0, |v|)$ are called *nonextended*. Observe, that the sub-algebra of all nonextended t-sequences is isomorphic to the standard algebra of sequences.

The algebra of temporal regular languages (t-languages) **VTReg** is built over the algebra of t-sequences. To the standard signature of algebra of regular languages, we add operators $\_^\flat$, $\_ \cap\!\!\llcorner \_$, $\mathbf{first}\_$, $\mathbf{pref}\_$, $\overline{\_}$ and $\mathbf{neg}\_$. The carrier is the powerset of the carrier of the algebra of t-sequences. The interpretations are defined as

- $U^\flat = \{u^\flat : u \in U\}$
- $U \cap\!\!\llcorner V = ((U \cdot \Sigma^\star) \cap V) \cup (U \cap (V \cdot \Sigma^\star))$

We say, that the t-language U is *nonextended* iff all sequences in U are nonextended. Observe, that the sub-algebra of all nonextended t-languages is isomorphic to the standard algebra of regular languages. The remaining operations are defined only for nonextended arguments. Let us assume **NExtend**, where **NExtend** denotes the set of all nonextended t-sequences. Additionally we will write $\langle e \rangle$ to denote the set of all t-sequences $([v], 0, 1)$ such that Verilog expression e evaluates to *true* in the valuation v.

- $\mathbf{first}\, U = \{u \in U : \forall u' \in \mathbf{NExtend}\ .\ u' \neq \varepsilon \Rightarrow (u \cdot u') \notin U\}$
- $\mathbf{pref}\, U = \{u' \in \mathbf{NExtend} : \exists u'' \in \mathbf{NExtend}\ .\ u = u' \cdot u''\}$
- $\overline{U} = \{u' \in \mathbf{NExtend} : u' \notin U\}$
- $\mathbf{neg}\, U = \mathbf{first}(\overline{\mathbf{pref}\, U}) \cap \overline{U \cdot \Sigma}$

2.2 Temporal regular expressions with explicit failure

In this section we show how to construct the algebra **VTFReg** of temporal regular expressions over valuations with explicit failure. The elements of its carrier are defined to be pairs, which for convenience we write as ratios:

$$(U, U') = \frac{U}{U'}$$

where the "numerator" represents success of the assertion, while "denominator" represents failure. The signature and operations of this algebra are defined as follows in terms of operations in **VTReg**:

- $\dfrac{U}{U'} \cdot \dfrac{V}{V'} = \dfrac{U \cdot V}{U' \cup (U \cdot V')}$
- $\dfrac{U}{U'} \cup \dfrac{V}{V'} = \dfrac{U \cup V}{U' \cap\!\!\!\!\_\ V'}$
- $\dfrac{U}{U'} \cap\!\!\!\!\_\ \dfrac{V}{V'} = \dfrac{U \cap\!\!\!\!\_\ V}{U' \cup V'}$
- $\varepsilon = \dfrac{\varepsilon}{\emptyset}$
- $\Sigma = \dfrac{\Sigma}{\emptyset}$
- $\emptyset = \dfrac{\emptyset}{\Sigma^\star}$
- $\langle e \rangle = \dfrac{e}{\neg e}$ for $e \in Vexp$
- $\left(\dfrac{U}{U'}\right)^\star = \dfrac{U^\star}{\emptyset}$
- $\mathbf{if}\, \dfrac{U}{U'}\, \mathbf{next}\, \dfrac{V}{V'}\, \mathbf{else}\, \dfrac{W}{W'} = \dfrac{(U \cdot V) \cup ((\mathbf{neg}\, U) \cdot W)}{(U \cdot V') \cup ((\mathbf{neg}\, U) \cdot W')}$
- $\mathbf{inv}\, \dfrac{U}{U'} = \dfrac{U'}{U}$
- $\mathbf{istrue}\, \dfrac{U}{U'}\, \mathbf{in}\, \dfrac{V}{V'} = \dfrac{V \cap U^\star}{V' \cup ((U^\star \cdot U') \cap \mathbf{pref}(V))}$
  if U and U' contain only sequences of the form $([v], 0, 1)$; undefined otherwise
- $\mathbf{first}\, \dfrac{U}{U'} = \dfrac{\mathbf{first}\, U}{\mathbf{first}\, U'}$

- $\mathbf{len}_{[n..m]} \dfrac{U}{U'} = \dfrac{U \cap \Sigma^{[n..m]}}{U' \cap \Sigma^{[n..m]}}$ if $U \cap \Sigma^{[n..m]} \neq \emptyset$
- $\mathbf{len}_{[n..m]} \dfrac{U}{U'} = \dfrac{\emptyset}{(U' \cap \Sigma^{[n..m]}) \cup \Sigma^{m}}$ if $U \cap \Sigma^{[n..m]} = \emptyset$

The carrier of algebra **VTFReg** is used as the semantic domain in formal definition of TXP semantics later in this paper.

3 Run-time Environment

Although TXP can support any HDL, we will describe the run-time environment for Verilog HDL. Verilog is an event driven language used by many hardware designers to model their systems for design and verification using simulation. To simulate the design, users write Verilog descriptions and compile to generate a model for simulation. For writing application code that uses the run-time information from the simulation, Verilog provides a software interface called Programming Language Interface(PLI).

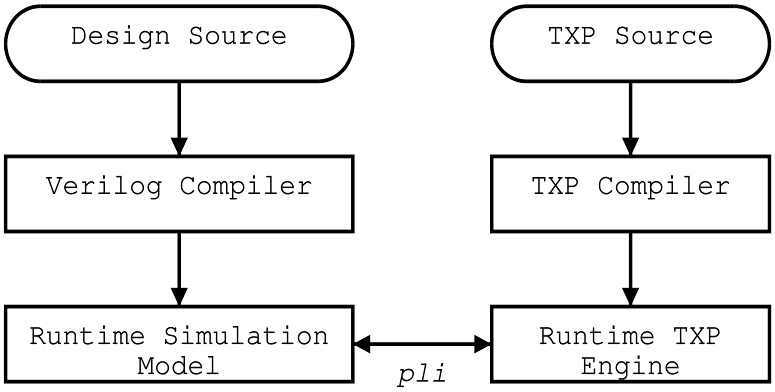

Fig. 1. TXP Run-time Environment

As shown in Figure 1, TXP specifications are compiled separately and provided to the TXP monitoring engine. The TXP run-time engine interacts with Verilog simulation using the Programming Language Interface(PLI)[3], mainly to obtain design variable information such as values and to obtain notification upon value change. While the simulation is running, TXP monitors and keeps track of the state of the assertions described in the TXP file. When a failure of an assertion is detected, the reporting provides debugging information that reports the starting time of the assertion and the time when failure is detected.

4 Language Basics

This section explains the basics of the language. The description includes the timing model, a match of a sequence, assertions, time shift operations, and

[3] Although the prototype implementation used PLI interface, there are more efficient ways of interacting with the simulator such as VPI and direct kernel interface.

construction of sequences.

4.1 Timing Model And Events

The timing model employed in TXP is based on clock ticks (cycles), and uses a generalized notion of clock cycles. A clock tick is an atomic moment in time and implies that there is no duration of time in a clock tick. The value of a variable in expression at a clock tick is sampled precisely at that moment in time. The sampled value is the only valid value of a variable at a clock tick. The definition of a clock is explicitly specified by the user, and can vary from one expression to another. In addition, a user can choose to use the simulation time as a clock to express asynchronous events.

A temporal expression is always tied to a clock definition. The values of variables involved in the expression are sampled only at clock ticks. These values are used to evaluate events or logical sub-expressions that are required to determine a match with respect to a temporal expression.

An event at a clock tick is defined as a change in the value of an expression from the value of that expression at the previous clock tick. For example, when a signal changes its value from low to high (a rising edge), it is considered an event. An event evaluates to true if the event occurs, and to false if the event does not occur.

4.2 Matching A Sequence

A sequence, described by a temporal expression, consists of checkpoints, dispersed in time from the beginning to the end of evaluation time. A checkpoint is the evaluation of a logical expression or an event, resulting in a true/false value. Generally, a sequence is written as a series of checkpoints, where a checkpoint is followed by the next checkpoint. To determine a match of a sequence, checkpoints are evaluated at appropriate times to satisfy the expression. If all the checkpoints are satisfied, then a match of a sequence is said to occur.

An assertion is tried at every clock tick to see if it is violated. To test the assertion at a clock tick, a new evaluation attempt for the expression is carried out, independent of any attempt at a previous clock tick. The results of each attempt are also reported separately.

A more complex scenario arises when the expression evaluation branches out to compute all alternatives sequences implied by a construct. In such cases, a match is determined for every sequence independent of each other. The expression can result in multiple matches or failed matches. If such a temporal expression is a sub-expression of a larger expression, then the resulting matches are used to determine matches of the enclosing expression.

Example 4.1

```
e1 #[1..3] (ack==0)
```

This statement says that signal `ack` *must be low after the occurrence of event e1 at either clock tick 1, 2, or 3. To determine a match for each of these three cases, three separate evaluations are started. The three sequences are:*

```
e1 #1 (ack==0)
e1 #2 (ack==0)
e1 #3 (ack==0)
```

4.3 Assertions

Assertions, also called checkers, are expressed as `check` or `forbid` clauses. An assertion defines a property of a system that is monitored to provide the user with a functional validation capability. An assertion is written as a temporal expression and can express complex timing and functional relationships between values and events of the system.

A `check` clause results in a success for an attempt if the associated temporal expression has at least one match for that attempt. Otherwise, the `check` clause fails. Generally, a `check` clause is specified with the expectation that the temporal expression will hold true for all attempts.

On the other hand, a `forbid` clause results in a success for an attempt if the associated temporal expression has no match. Otherwise, the `forbid` clause fails. Contrary to a `check` clause, a `forbid` clause is specified to ensure that a certain condition never occurs.

A `check` or `forbid` is declared with an identifier to name the assertion, a clock to specify clock ticks for sampling values/events, and a temporal expression to specify the relationship for monitoring.

```
clock event_spec {
      check name : temporal_expr ;
      forbid name: temporal_expr ;
  }
```

The clock determines the sampling times for values and events. A clock tick occurs whenever the event described by *event_spec* occurs during simulation. The *event_spec* can be a rising edge, falling edge, an edge (falling or rising), or simulation clock itself.

Example 4.2

```
clock posedge clk {
  check  rule1 :  if (pipelined) stage1 #1 stage2 #1 stage3;
  forbid rule2 : if (mem_op) mem_instr #2 mem_fetch;
}
```

4.4 Time Shift Operations

Formal Semantics

- $[\![\texttt{\#[}n\texttt{..}m\texttt{]}\ A]\!] = \Sigma^{[n..m]} \cdot [\![A]\!]$ for $1 \le n \le m$
- $[\![\texttt{\#[}n\texttt{..}m\texttt{]}\ A]\!] = \lhd^{[-m..-n]} \cdot [\![A]\!]$ for $n \le m < 0$
- $[\![\texttt{\#[}n\texttt{..}m\texttt{]}\ A]\!] = (\lhd^{[0..-n]} \cup \Sigma^{[0..m]}) \cdot [\![A]\!]$ for $n < 0 \le m$
- $[\![\texttt{\#}n\ A]\!] = [\![\texttt{\#[}n\texttt{..}n\texttt{]}\ A]\!]$ if $n > 0$
- $[\![A\ \texttt{\#}n\ B]\!] = [\![A]\!] \cdot [\![\texttt{\#}(n-1)\ B]\!]$
- $[\![A\texttt{->>}B]\!] = [\![A]\!] \cdot \Sigma^\star \cdot [\![B]\!]$

Time is expressed in terms of clock tick delays and is specified using `#` notation. $e1$ $\#t$ e2 means that $e1$ should occur, followed by $(t-1)$ clock ticks, followed by $e2$. In other words, $e1$ must occur, followed by $e2$ on the t clock tick [4] .

This basic time notation is used to express temporal relationships between expressions, and provides the building blocks for sequences. Examples of specification are: delays between sequences, specific clock tick when a sequence is expected to occur, time period during which a sequence is expected to complete, and eventuality of occurrence of a sequence.

Time can be expressed in a variety of ways, such as

`#`t, to denote t clock ticks,

`#[`$t1..t2$`]`, to denote a variable time delay between $t1$ and $t2$, and

`#[1..simend]`, to denote eventually before the end of simulation. (A shortcut symbol `->>` can be used to denote the same)

Example 4.3

Consider three expressions, showing positive shift, negative shift, and eventuality.

```
txp expr1 = te1 #2 te2;
```

For expr1, te2 is expected to occur 2 clock ticks after te1.

```
txp expr2 = te1 # -1 te2;
```

For expr2, te2 is expected to occur 1 clock tick prior to te1.

```
txp expr3 = te1 ->> te2;
```

For expr3, te2 is expected to occur sometime after te1 but before the simulation ends.

4.5 Composite Sequences

Formal Semantics

- $[\![A\ \texttt{or}\ B]\!] = [\![A]\!] \cup [\![B]\!]$
- $[\![A\ \texttt{and}\ B]\!] = [\![A]\!] \cap\!\llcorner\ [\![B]\!]$
- $[\![\texttt{inv}\ A]\!] = \textbf{inv}\ [\![A]\!]$

Three operators are provided to construct sequences: `and`, `or`, and `inv`.

[4] TXP allows the time shift to be negative. However, in this paper we will not detail the formal semantics of this feature.

The binary operator `and` is used when both operand expressions are expected to succeed.

temporal_expr `and` *temporal_expr*

The two operands of `and` are temporal expressions. The requirement for the match of the and operation is that both the operand expressions must match. When one of the operand expressions matches, it waits for the other to match. The end time of the composite expression is the end time of the operand expression that completes the last.

Example 4.4

Assuming that ve1, ve2, ve3, ve4 and ve5 are logical Verilog expressions, then the second operand is longer, so the end time is 4 clock ticks from the current time.

(ve1 #2 ve2) and (ve3 #2 ve4 #2 ve5)

The operator `or` is used when at least one of the two operand sequences is expected to succeed.

temporal_expr `or` *temporal_expr*

The requirement for the match of the or operation is that at least one of the two operand expressions succeed. The end time of a match is the end time of any temporal expression that matched.

To obtain the inverse of a match, `inv` operator is used.

`inv` *temporal_expr*

The operator `inv` complements[5] the matches. Every successful match is converted to a failed match, and vice versa. Generally, this feature is used when a user is interested in detecting failure of an expression. A failure occurs when all possibilities of matching the expression for a success are exhausted.

4.6 Conditional Matching

Formal Semantics

- $[\![\texttt{if}\ A\ \texttt{next}\ B\ \texttt{nextelse}\ C]\!] = \textbf{if}\ [\![A]\!]\ \textbf{next}\ [\![B]\!]\ \textbf{else}\ [\![C]\!]$
- $[\![\texttt{if}\ A\ \texttt{then}\ B\ \texttt{else}\ C]\!] = \textbf{if}\ [\![A]\!]^\flat\ \textbf{next}\ [\![B]\!]\ \textbf{else}\ [\![C]\!]$
- $[\![\texttt{if}\ A\ \texttt{next}\ B]\!] = \textbf{if}\ [\![A]\!]\ \textbf{next}\ [\![B]\!]\ \textbf{else}\ \varepsilon$
- $[\![\texttt{if}\ A\ \texttt{then}\ B]\!] = \textbf{if}\ [\![A]\!]^\flat\ \textbf{next}\ [\![B]\!]\ \textbf{else}\ \varepsilon$

These constructs allow a user to monitor sequences based on satisfying some criteria. Most common uses are to attach a precondition to a sequence, and to select a sequence between two alternatives, where the selection is made based on the success of a condition. Two kinds of clauses are provided.

[5] There are some exceptions when a `inv` can result in a match and a failure at the same time. Note the difference between `inv` and `neg` defined in section 2.1.

`if` *cond* `then` *seq*

`if` *cond* `then` seq_1 *`else`* seq_2

The first clause is used to precondition monitoring of a temporal expression (The functionality provided here is the same as obtained by an implication operator in temporal languages). If the condition *cond* fails then *seq* is skipped for monitoring. Note that, *cond* and *seq*, both can be temporal expressions. These clauses work well when the monitoring of *cond* is to be overlapped with monitoring of *seq*, seq_1 or seq_2.

Example 4.5

Consider a bus operation for data transfer from a master to a target device. When the bus enters a data transfer phase, multiple data phases can occur to transfer a block of data. During the data transfer phase, a data phase completes on any rising clock edge on which `irdy` is deasserted and either `trdy` or `stop` is deasserted. Note that a deasserted signal here implies a value of low and negedge means a transition from a high to low value. The end of a data phase can be expressed as

```
clock posedge mclk {
  txp data_end =
    if (data_phase) then ((negedge irdy &&(negedge trdy || negedge stop));
}
```

Another variation is to offer the concatenation effect between the condition and the sequence. These constructs are provided as

`if` cond `next` seq

`if` cond `next` seq_1 `nextelse` seq_2

Unlike the previous clauses, there is no overlapping between *cond* and *seq*, seq_1 or $seq\ _2$. Upon the success of condition *cond*, monitoring for temporal expression *seq* begins at the next clock tick, i.e., there is one clock tick difference between the end of *cond* and the beginning of *seq*. Note that if *cond* spawns multiple sequences, then the earliest match is considered.

4.7 Repetition of Sequences

Formal Semantics

- $[\![\texttt{rep[}n\texttt{..}m\texttt{]}A]\!] = [\![A]\!]^{[n..m]}$ for $0 \leq n \leq m$
- $[\![\texttt{rep[}n\texttt{..]}A]\!] = [\![A]\!]^{n} \cdot [\![A]\!]^{\star}$ for $0 \leq n$ and nonextended $[\![A]\!]$

There are situations when a temporal expression is monitored repeated times in succession. In such cases, monitoring is performed for a specified number of times, and each time a success is expected to result from evaluating the temporal expression. In other words, repetition is same as concatenation of the same temporal expression for the specified number of times. Repetition is expressed with a repetition parameter to specify the number of times an expression needs to be monitored. This parameter can be a number or a range of values.

`rep` *n seq*

n can be any non-zero positive integer constant, and *seq* can be a temporal expression.

The above expression is semantically equivalent to the following expression,

$\underbrace{seq \ \#1 \ seq \ \#1 \ seq \ ...}_{n \ times}$

Example 4.6

```
rep 3 (ev1 #1 ev2)
```

says "sequence (ev1 #1 ev2) must occur three times in a row". It is equivalent to writing

```
(ev1 #1 ev2) #1 (ev1 #1 ev2) #1 (ev1 #1 ev2)
```

Example 4.7

A bus read transaction in burst mode is expected to read data in 8 data phases. Each data phase follows the next, where a data phase is said to occur when signals `irdy` *and* `trdy` *are deasserted at the same time.*

```
clock posedge mclk {
  check burst_rule:
    if (negedge burst_mode) then #2 rep 8 (negedge trdy)&&(negedge irdy);
}
```

The assertion burst_rule says " when a falling edge of `burst_mode` *is detected, two clock ticks later, data transfer begins (*`trdy` *and* `irdy` *both deasserted) and continues for 8 times".*

4.8 Specification of Conditions On Sequences

Formal Semantics

- $[\![\texttt{istrue}\ v\ \texttt{in}\ A]\!] = \textbf{istrue}\ \langle v \rangle\ \textbf{in}\ [\![A]\!]$
- $[\![\texttt{len[}n\texttt{..}m\texttt{]in}\ A]\!] = \textbf{len}_{[n..m]}\ [\![A]\!]$
- $[\![\texttt{len}\ n\ \texttt{in}\ A]\!] = [\![\texttt{len[}n\texttt{..}n\texttt{]in}\ A]\!]$

Sequences of events often occur under the assumptions of some restrictions for correct behavior. A logical condition must hold true, for instance, while processing a transaction. Or, a transaction must complete within a given period of time, no matter what variation of commands are issued in the transaction to be processed. Also frequently, occurrence of certain events is prohibited while processing a transaction. These situations can be expressed directly using the following two constructs:

`istrue` *logical_expr* `in` *temporal_expr*

logical_expr is a logical expressions which must result to true at every clock tick during the monitoring of *temporal_expr*. If *temporal_expr* starts at time $t1$ and ends at time $t2$, then *logical_expr* must hold true from time $t1$ to $t2$.

`len` *time_shift* `in` *temporal_expr*

time_shift specifies the length of the duration for the *temporal_expr*. All variations of the time shift specification are allowed. If a single number is specified for *time_shift*, then it represents a fixed time duration. If *time_shift* specifies a range of numbers, then the *temporal_expr* may terminate anytime within a time period determined by the minimum and maximum numbers.

Example 4.8

```
clock posedge mclk {
    check burst_rule1:
      if (negedge burst_mode) then
         istrue (!burst_mode) in (#1 rep 8 (negedge trdy && negedge irdy));
  }
```

In the above expression, the value of signal `burst_mode` is required to be low, and is checked at every clock tick during the expression `(#1 rep 8 (trdy ==0) && (irdy==0)`.

```
clock posedge mclk {
    check burst_rule3:
      if (negedge burst_mode) then
         len #[9..11] in (#[1..4] rep 8 (negedge trdy && negedge irdy));
  }
```

In the above expression, the total duration of the entire repeated sequence must not be less than 9 clock ticks and greater than 11 clock ticks. This restriction is expressed by `len #[9..11] in` expression.

5 Conclusions

We have presented a new language for specifying properties that are useful for hardware designers. We believe that the sequencing operations are the most common considerations in verifying a hardware design. TXP is well suited for such applications, as it presents a convenient syntax to write common protocols such as PCI and Infiniband. TXP, based on regular expression semantics, has been implemented to monitor temporal expressions at simulation run time. TXP is also being experimented to drive a formal property checker. As our model of TXP assertions is based on finite state automata, an intersetting area for investigation is the automatic stimulus generation for exercising the TXP assertions. Our work continues to examine all aspects of hardware design verification and investigate future extensions to the language.

References

[1] T. Kropf. Introduction to Formal Hardware Verification. *Springer-Verlag*, 1999

[2] E. Clarke, O. Grumberg, and D. Peled. Model Checking, *MIT Press*, 1999

[3] J. Hopcroft and J. Ullman. Introduction to Automata Theory, Languages and Computation, *Addison Wesley*, 1979

[4] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, *IEEE Computer Society*, *IEEE Std 1364-1995*,New York, 1996

[5] IEEE Standard Hardware Description Language Based on VHDL Hardware Description Language, *IEEE Computer Society*, *IEEE Std 1076-199*3,New York, 1996

[6] H. Hiraishi. Design Verification of Sequential Machines Based on a Model Checking Algorithm of ε-free Regular Temporal Logic, *CMU-CS-88-1953, Carnegie-Mellon University*, 1988

[7] J. Corbett, M. Dwyer, J. Hatcliff, and Robby. A Language Framework For Expressing Checkable Properties of Dynamic Software, Proceedings of the SPIN Software Model Checking Workshop, Lecture Notes in Computer Science *Springer-Verlog*, Aug. 2000