**Center for Embedded and Cyber-Physical Systems**
**University of California, Irvine**

# Using Natural Language Documentation in the Formal Verification of Hardware Designs

Christopher B. Harris and Ian G. Harris

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

Christopher.Harris@uci.edu

CECS Technical Report TR# 14-11
November 7, 2014

# Using Natural Language Documentation in the Formal Verification of Hardware Designs

Christopher B. Harris
Department of Electrical Engineering and Computer Science
University of California – Irvine, USA
Email:Christopher.Harris@uci.edu

Ian G. Harris
Department of Computer Science
University of California – Irvine, USA
Email: harris@ics.uci.edu

***Abstract*—In the modern ASIC design cycle, correctness properties for functional verification are usually created by an engineer whose task is to read the system documentation and manually generate a set of formal statements in the chosen verification language. This process is typical of the reason why up to 60% of engineering effort is spent on verification and test activities. We present a formal attribute grammar as the basis for a Natural Language based translation system which automatically generates syntactically correct verification properties in Computation Tree Logic (CTL) from code comments written in English. The system is evaluated using verification information from an implementation of the PCI bus specification included in the Texas-97 Verification Benchmark suite and successfully translates English to valid CTL in 91% of test cases. The automatically generated CTL properties are compared to CTL properties included in the benchmark and model checking is used to determine the equivalency of generated and benchmark properties. While most generated properties are equivalent, some were found to include additional terms which result in CTL which more closely reflects designer intent.**

## I. Introduction

The rise in digital integrated circuit design complexity combined with the move to System on Chip (SoC) based designs has caused functional verification to become more challenging with each successive technology generation. The verification process begins with a well defined system specification written in a natural language such as English. It is the goal of the designer or verification engineer to use this specification to eventually develop a set of test properties useful in verifying the correct operation of the system.

However, generating a "good" set of such properties is a non-trivial task. Engineers tend to think in terms of natural language. Generating verification properties expressed in a non-intuitive formalism such as Computation Tree Logic (CTL) can be an error prone task for the non-expert. Secondly, the task of deciding exactly which properties to verify relies largely on the expertise of the verification engineer. An uncomprehensive set of verification properties can reduce the quality of the entire process.

In this work we address these increasingly prominent challenges by the presentation of a custom attribute grammar and a methodology for automatically extracting a set of CTL properties from inline code comments written in English. These properties can then be directly used, without any additional processing, in the formal verification of the design via model checking. Although this study focuses on the analysis on inline code comments, our methodology is equally applicable to other natural language design documents such as system specifications.

When implementing a design it is standard practice for engineers to include designer comments written in English (or another natural language) inline with the code. These inline comments often describe what the designer intends to be implemented, thus are good candidates for conversion to verification properties. In addition, these natural language comments tend to be included when implementing a particularly complex or non-intuitive portion of the design. These design modules have a greater likelihood of containing errors and thus are targets for increased verification scrutiny. This work leverages these designer comments to improve the set of test properties and therefore the quality of the verification process.

## II. Related Work

There is an existing body of work in the field of Natural Language processing (NLP) applied to hardware verification. Early work explored using a Natural Language (NL) interface to query circuit simulation results, but the system was never fully realized [1]. Other early work attempted to analyze a restricted set of natural language sentences to encode design information, but only supported a small number of basic verb patterns. The translation procedure resulted in a custom intermediate formalism instead of properties directly useful for model checking [2]. These important first steps helped pave the way for more contemporary work such as investigations into translating natural language specifications into high abstraction system models [3] and SystemVerilog assertions [4]. There is also an existing body of work concerned with translating correctness properties specified in a natural language to various temporal logics. Early work in [5] supports conversion from a NL specification to Action CTL (ACTL). While intended to perform automatic translation the tool instead requires user input in order to resolve language ambiguity. Parallel work in [6] translates parse trees derived from NL into an intermediate representation defined by Discourse Representation Theory [7]. Unfortunately, this intermediate representation is not immediately useful for model checking.

Work in [8] presents a controlled subset of the English language which can be used to describe CTL properties, together with a context-free grammar to recognize the language subset. Our approach is distinguished from this work primarily because research in [8] does not present an algorithm to generate CTL expressions from their proposed English language subset. Our approach presents an algorithm to generate CTL expressions from English and we show results of the application of this algorithm to code comments in the Texas-97 Benchmark suite [9].

## III. Background

### A. Semantic Parsing

Syntactic parsing is a commonly used NLP technique. *Syntactic* parsing is the grammatical analysis of a sentence according to rules defined by a formal grammar in order to show the structure of a sentence. A formal grammar is defined as

$$G = (V_N, V_T, S, P)\,, \tag{1}$$

where $V_N$ is a finite set of nonterminal symbols, $V_T$ is a finite set of terminal symbols where $V_N \cap V_T = \{\}$, $S \in V_N$ is a start symbol, and $P$ is a finite set of mappings from $V_N \rightarrow (V_T \cup V_N)^*$ called productions (where the $*$ operator implies zero or more instances).

Grammars which are defined to have a single nonterminal symbol for the left side of each production rule are known as context-free grammars (CFG). Although natural language is not context free the use of CFGs is well accepted in NLP to adequately model syntax in practice. In syntactic parsing a sentence consisting of terminal symbols is converted to a constituent (or parse) tree consisting of both terminal and nonterminal symbols. This parse tree displays the constituents and syntactic structure of a sentence.

Let $G$ be a CFG defined as the sets in (2) and (3) combined with the set of productions shown in Figure 1a.

$$V_T = \{\text{“a”,“the”,“dog”,“cat”,“chased”,“sat”,“on”,“in”}\} \tag{2}$$

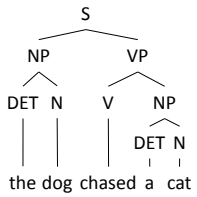
$$V_N = \{\text{S, DET, N, V, P, NP, VP, PP}\} \tag{3}$$

We can now present an example of syntactic parsing using the sentence “the dog chased a cat”. Figure 1b shows the parse tree for our example sentence. Each node in the tree represents a production from set $P$ where the node closest to the root of the tree is the left side of the production and its child nodes represent the right side.

A *semantic* grammar differs from a merely *syntactic* grammar in that a semantic grammar associates domain-specific meanings with the symbols as opposed to syntactic categories. Consider the semantic grammar $G_{SEM}$ defined in (4)-(11). The nonterminal symbols in semantic grammar $G_{SEM}$ no longer represent syntactic categories such as noun or verb but semantic categories such as “signal name” or “signal level”. Using $G_{SEM}$ we can now perform a semantic parse of the sentence “reset should be asserted” resulting in the parse tree shown in Figure 2a.

```
S   → NP VP
PP  → P NP
NP  → DET N|NP PP
VP  → V NP|VP PP
N   → "dog"|"cat"
V   → "chased"|"sat"
P   → "on"|"in"
DET → "a"|"the"
```

(a) Sample Grammar




(b) Parse Tree

Fig. 1. Simple Parsing Example

$$G_{SEM} = (V_{TSEM}, V_{NSEM}, S, P_{SEM}) \tag{4}$$

$$V_{TSEM} = \left\{ \begin{array}{c} \text{“asserted”, “deasserted”,} \\ \text{“be”, “should”, “reset”} \end{array} \right\} \tag{5}$$

$$V_{NSEM} = \{\text{S, SETSIG, SIGNAME, VAL}\} \tag{6}$$

$$P_{SEM} = \{P_0, P_1, P_2, P_3\} \tag{7}$$

$$P_0 \equiv \text{S} \rightarrow \text{SETSIG} \tag{8}$$

$$P_1 \equiv \text{SETSIG} \rightarrow \text{SIGNAME “should” “be” VAL} \tag{9}$$

$$P_2 \equiv \text{SIGNAME} \rightarrow \text{“reset”} \tag{10}$$

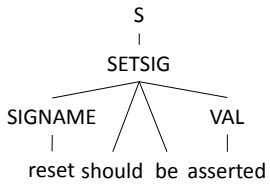
$$P_3 \equiv \text{VAL} \rightarrow \text{“asserted” | “deasserted”} \tag{11}$$

### B. Attribute Grammars

An attribute grammar is a formalism which extends that of context free grammars. First developed by Knuth, attribute grammars were originally used for evaluating the semantics of programming languages and are used extensively in compiler writing [10]. Attribute grammars allow grammatical symbols present in a parse tree to be replaced by an attribute which is then evaluated in terms of the attributes of other symbols. This is similar to the way a software function is evaluated in terms of one or more arguments, where those arguments are in turn the return values of other functions.

More formally defined, an attribute grammar associates a finite set of attributes $A(X)$ with each symbol $X \in (V_T \cup V_N)$. Each attribute $a \in A(X)$ represents a specific property of symbol $X$ and is denoted $X.a$. For each property $X.a$ associated with a symbol $X$, the value of $X.a$ is determined by a set of attribution rules. Given a set of symbols $\{X_1, \ldots, X_n\}$ each production $p = X_i \rightarrow (X_{j=1\ldots n})^*$ has one attribution rule with which it is associated. The rule is applied every time that production appears in a parse tree. Each attribution rule is of the form $X_i.a = f(X_j.a, \ldots, X_n.c)$. This implies that the value of an attribute $X_i.a$ is a function of the values of other attributes in the grammar. When the value of an attribute $X_i.a$ for a symbol $X_i$ is dependent only on nodes which are a descendent of $X_i$ in the parse tree then $X_i.a$ is known as a *synthesized attribute* (because the attribute value is iteratively synthesized from the values of child nodes). An attribute grammar where all attributes are synthesized attributes is known as an *S-attributed grammar*. We only make use of S-attributed grammars in this work.

As an example, let us extend the semantic grammar $G_{SEM}$

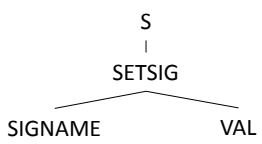



(a) Semantic Parse Tree

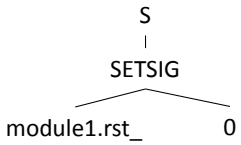
```
S.v       = SETSIG.v
          = "module1.rst_ = 0"
SETSIG.v  = SIGNAME.v + "=" + VAL.v + ";"
SIGNAME.v = "module1.rst_"
VAL.v     = "0"
```

(b) Produciton Attribute Values

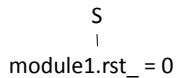
Fig. 2. Attribute Grammar Parse




(a) Initial Parse Tree




(b) Leaves Evaluated




(c) SETSIG Evaluated

module1.rst_ = 0;

(d) Final CTL

Fig. 3. Attribute Evaluation Example

from (4) to form a semantic attribute grammar with productions $P_0$, $P_1$, $P_2$, and $P_3$ (8)–(11) associated with rules $R_0$, $R_1$, $R_2$, and $R_3$ (12)–(15) respectively.

$$R_0 \equiv \text{S.v} = \text{SETSIG.v} + \text{“;”} \tag{12}$$

$$R_1 \equiv \text{SETSIG.v} = \text{SIGNAME.v} + \text{“=”} + \text{VAL.v} \tag{13}$$

$$R_2 \equiv \text{SIGNAME.v} = \text{“module1.rst\_”} \tag{14}$$

$$R_3 \equiv \text{VAL.v} = \text{“0”} \tag{15}$$

This attribute grammar has a single attribute named “value” (denoted with a lowercase v). When we revisit the sentence “reset should be asserted” we can now associate the attribute rules in Figure 2b with the productions from the parse tree in Figure 2a. To apply our attribute grammar to this parse tree the first step is to evaluate the values of the symbols at the leaf nodes. That is, we replace the symbols $SIGNAME$ and $VAL$ with their attribute values $SIGNAME.v$ and $VAL.v$ from (14) and (15) respectively. This step is shown in Figure 3b. The values of these leaf nodes are then used to evaluate the attribute value of their parent node $SETSIG$. In Figure 3c we replace the symbol $SETSIG$ with its attribute value $SETSIG.v$ evaluated according to (13). Finally, in Figure 3d we evaluate the final symbol $S$ which is defined in (12) to simply be the value of the $SETSIG$ attribute terminated by a semicolon. The use of attribute grammars in this manner allows a NL sentence represented by a semantic parse tree, with the appropriate choice of attributes, to be translated to a different formalization.

## IV. Methodology

### A. System Overview

Figure 4 depicts a high level block diagram of the translation system. In this description English language sentences are harvested from inline code comments in design or verification code. These sentences are then semantically parsed using a recursive descent parser and an appropriate attribute grammar to generate a parse tree. The resulting parse tree undergoes attribute evaluation in the translation engine where CTL is directly generated by evaluating the attributes of the productions used in the parse. The resulting CTL property is then checked against the system model using the VIS formal verification tool [11].

### B. Translation Engine

The system utilizes a stock recursive descent parser distributed as part of the Natural Language Toolkit (NLTK) [12]. NLTK is a general NLP environment written in Python which supports sentence parsing, text tokenization and classification, and other syntactic analysis [12]. However, the heart of the system is the translation engine which utilizes our custom attribute grammar.

A new grammar is commonly developed through the analysis of a large collection of discipline specific text samples. This is known in linguistics as a *corpus*. Lacking such a corpus of digital system specifications and code comments we utilized a subset of NL comments from the PCI Local Bus implementation in the Texas-97 Verification Benchmark suite [9] with additional information from the PCI Local Bus specification [13]. The use of specification information in the generation of the grammar allows the ability to correctly translate design abstractions such as “bus transaction”, “mem-

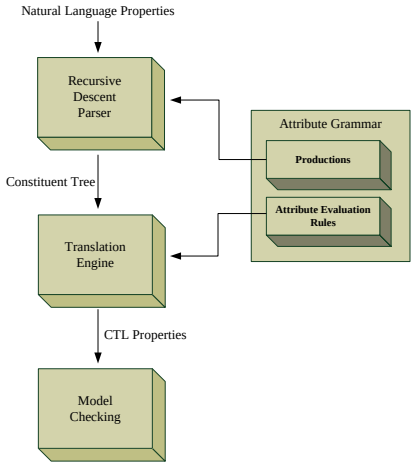



Fig. 4. System Block Diagram

```
S  → PP CTLS | NP VP
   [S.v = LAST_CHILD_NODE.v]

CTLS  → IMPLI PERIOD | DELAYED_IMPLI  PERIOD
   [CTLS.v = AG(FIRST_CHILD_NODE.v)]
```

Fig. 5. Sample Top Level Productions

```
IMPLI   → 'if' CTL_TRANS 'then' CTL_SET_NEXT
   [IMPLI.v = "(" + FIRST_CHILD_NODE.v + ") →
              (" + SECOND_CHILD_NODE.v + ")"]

DELAYED_IMPLI → CTL_TRANS COMMA CTL_SET |
                SBAR COMMA CTL_UNTIL
   [DELAYED_IMPLI.v = "(" + FIRST_CHILD_NODE.v + ") →
                      AX(" + THIRD_CHILD_NODE.v + ")"]
```

Fig. 6. Sample Implication Productions

ory read", "initialize", or "assert". A more general grammar could be derived from a larger digital system specification corpus if one were to exist. However, specific features can still be extracted from inline code comments as word usage and writing style tends to be consistent from a single designer.

The process of generating our grammar entailed an analysis of the target natural language comments in order to identify common syntactic patterns and combined to form symbols $X_i$. This iterative process resulted in the appearance of a set of emergent syntactic structures and associated symbols. Although the process was conducted manually for this study, there is hope that it can be refined and automated in future work.

The 130 unique productions in our grammar can be divided into 9 distinct categories. We will present a brief characterization of each grammatical category and present selected examples of some productions.

1) *Top Level:* Symbols in this category include the start symbol $S$ and symbols found directly below the start symbol in a parse tree. The attribute value for the symbol $S$ is the value of the last child node listed in the production associated with $S$. For example, in the production $S \rightarrow PP\ CTLS$ the value of the attribute $S.v$ will be evaluated to be $S.v = CTLS.v$. If the symbol $PP$ parses the phrase "In the clock cycle that" and the symbol $CTLS$ parses the fragment "*SIGNAL1* is low, *SIGNAL2* must be high", then production $S \rightarrow PP\ CTLS$ would successfully parse the NL sentence "In the clock cycle that REQ_ is low, GNT must be high". Examples productions from the top level category in Figure 5 show that a $CTLS$ symbol can consist of an implication symbol $IMPLI$ followed by the punctuation symbol $PERIOD$. Attribute values are denoted in square brackets.
2) *Implication:* These symbols represent syntactic patterns that are mapped to various types of implications when translated into CTL. These can be thought of as simple *if-then* statements. Three types of CTL implications are considered. The first type simply evaluates to an antecedent which implies a consequence. The second type of implication is one where the consequence is delayed one cycle, while the third type of implication maps to a consequence realized at some unspecified point in the future. The sentence, "If REQ_ is asserted then it must eventually be acknowledged" prominently features an implication structure. Figure 4 shows examples of productions in this category along with their value attributes.
3) *Wait-Until:* Symbols in Figure 5 implement the wait-until semantic in CTL. This wait-until semantic captures the idea that a property holds true for a period of time "until" another condition is satisfied.
4) *Signal Value:* This category contains the symbols which denote a logical one or zero.
5) *Signal Assignment and Edge Transitions:* The symbols in Figure 6 cover signal assignment as well as rising or falling edges.
6) *Distributive Rules:* Symbols in this category cover cases when a value or condition is distributed between two or more signals.
7) *Design Abstractions:* Design Abstraction symbols capture design specific abstractions such as the beginning or end of a transaction. These properties are generally pulled directly from the specification or from predefined values within the design and thus are not necessarily functions of child nodes.
8) *Signal Names and Storage Elements:* This category simply contains symbols that capture various types of signals and register storage elements in a design.
9) *Null Strings:* The final category of symbols are those which are useful in parsing but whose attributes return a null value. In practice, these symbols return an empty string.

```
CTL_UNTIL  → WAIT_ACTION "until" WAIT_COND
   [CTL_UNTIL.v = "A(" + WAIT_ACTION.v + " U " + WAIT_COND.v + ")"]

WAIT_ACTION → CTL_SET | DIST2
   [WAIT_ACTION.v = CHILD_NODE.v]

WAIT_COND → CTL_SET | CTL_ANP
   [WAIT_COND.v = CHILD_NODE.v]
```

Fig. 7. Sample Wait-Until Productions

```
CTL_SET → CTL_NP "has" "to" "be" CTL_VAL | CTL_NP MD "be" CTL_VAL
   [CTL_SET.v = "(" + FIRST_CHILD_NODE.v + " = " + LAST_CHILD_NODE.v + ")"]

SET_FUTURE → CTL_NP "must" "be" CTL_VAL | CTL_NP "should" CTL_RB "be" CTL_VAL
   [SET_FUTURE.v = "AF(" + CTL_NP.v + " = " + CTL_VAL.v + ")"]

CTL_TRANS → CTL_NP "has" "been" CTL_VAL | CTL_NP "is" CTL_VAL
   ["(!(" + CTL_NP.v + " = " + CTL_VAL.v + ") * AX(" + CTL_NP.v + " = " + CTL_VAL.v + "))"]
```

Fig. 8. Sample Assignment and Transition Productions

### C. CTL Property Equivalence

The ready availability of the Texas-97 Benchmark suite [9] informed the choice of CTL as the target formalism for this study. In addition to CTL verification properties the Texas-97 suite [9] also contains in the verification code natural language descriptions of what each CTL property is

intended to verify. This provides the opportunity to not only show that our automatically generated CTL properties can be successfully used in model checking, but to evaluate the quality of our CTL translations by comparison to the CTL generated by human designers. However, to effectively do so we must establish a criteria for equivalence between CTL formulae. This is accomplished by restricting the discussion of equivalence to safety properties which utilize the same set of variables in each of the properties to be compared. Properties which do not utilize the same variables are automatically declared non-equivalent. Given two CTL formulae $f$ and $g$, we show equivalence by constructing a composite formula which is the boolean equivalent of $f$ XNOR $g$. By performing model checking using the composite formula we can show the equivalence of the component CTL formulae, but only for the model under investigation. However, even with these limitations we can distinguish when the natural language generated CTL differs from the benchmark CTL and if the natural language CTL allows the model to be successfully verified.

## V. Experimental Results

The natural language based formal verification system outlined above was implemented in Python using Natural Language Toolkit 2.0 [12]. The Texas-97 Verification Benchmark suite [9], which was used in the construction of the attribute grammar, was also used to generate test data for our system. Model checking was performed using VIS version 2.4. Our experimental setup allowed us to answer the following three questions:

- Can the system generate valid CTL properties from English language statements?
- Can these automatically generated CTL properties be used to verify the target design?
- Are the automatically generated CTL statements logically equivalent to the control CTL properties included in the benchmark for the design under verification?

### A. CTL Property Generation

A limited pre-processing step was performed on the natural language comments. This pre-procssing only corrected obvious spelling errors such as the word "till" corrected to be the word "until". In the PCI Local Bus design example of the Texas-97 Benchmark suite [9] a total of 22 CTL properties were identified which contained inline descriptions in English. Two of these code comments used multiple sentences to describe the verification property. The current implementation of our system performs semantic analysis on a per sentence basis. Thus, a verification requirement which requires a multi-sentence description to cannot currently be processed by our system. While semantic analysis across sentence boundaries can be facilitated though the use of Discourse Representation Structures [7] this capability was outside the parameters of our investigation and so was not implemented. We do not view this as a limitation to the utility of our approach because in practice many complex sentences can be reduced to a sequence of simpler sentences which convey the same idea. All results were generated on a 2.2 MHz AMD Opteron processor with 8 GB of RAM.

In a transaction if STOP and IRDY are asserted, FRAME must be deasserted.

(a) Natural Laguage Comment

$$\mathrm{AG}((\mathrm{t1.STOP\_} = 0) * (\mathrm{m1.rIRDY\_} = 0) \rightarrow \mathrm{AX}(\mathrm{m1.rFRAME\_} = 1)$$

(b) Benchmark CTL

$$\mathrm{AG}(((\mathrm{m1.rFRAME\_}{=}0) * ((\mathrm{t1.STOP\_} = 0) * (\mathrm{m1.rIRDY\_} = 0))) \rightarrow (\mathrm{AF}(\mathrm{m1.rFRAME\_} = 1)))$$

(c) Autogenerated CTL

Fig. 9. Nonequivalent CTL Properties

The results of the experiment are presented in Table I. Of our 22 natural language comments 20 yielded properly formed CTL properties after processing, a successful translation rate of 91%. All 20 of these CTL properties also successfully passed model checking (where success is defined as the generated property yielding the same model checking result as the benchmark property). However, in three instances the generated CTL property was found not to be logically equivalent to the control CTL. We will briefly analyze one such representative case.

The natural language comment labeled PCI5 is shown in Figure 9a. The benchmark CTL property in Figure 9b verifies that when both the STOP_ and IRDY_ signals are logic 0 there is an implication that in the next cycle the FRAME_ signal will be logic 1. However, if we read the natural language comment closely we see that this should only be true "in a transaction". A "transaction" in this context is a data transfer abstraction specific to the design. As such, our attribute grammar makes use of the PCI specification to incorporate the semantics of this abstraction. In this design, the FRAME_ signal equal to logic 0 is indicative of an ongoing transaction. As a result, when our grammar parses the phrase at the beginning of the comment it assigns an appropriate semantic meaning in the CTL property as shown in Figure 9c. In this respect, the generated CTL more closely reflects the comment intent than the benchmark control CTL. The natural language comment clearly stated what the designer intended, but some of the information was lost in translation and did not appear in the control CTL property.

There is, however, one additional difference between the two equations. In the consequence of the control CTL the use of $AX$ indicates that the FRAME_ signal is required to be logic 1 during the next cycle. In the generated CTL an $AF$ is used instead, indicating that less strict requirement that FRAME_ is required to be logic 1 at some point in the future. These two conditions are not logically equivalent.

Looking again at the natural language comment, the phrase "FRAME_ must be deasserted" shows no obvious temporal reference. However, upon further investigation it was discovered during the analysis of the natural language text used to

TABLE I
EXPERIMENTAL RESULTS

| TextID | Parseable | CTL Equivalent to Benchmark | Natural Language Comment<br>Autogenerated CTL Property (If Parseable) |
|---|---|---|---|
| PCI0 | Y | Y | Once FRAME_ has been asserted, the CBE_ lines should be driven until the end of the transaction.<br>$AG(((!(m1.rFRAME\_ = 0) * AX(m1.rFRAME\_ = 0))) \rightarrow AX(A((m1.OE\_CBE\_ = 1) \cup ((m1.rIRDY\_ = 0) * (t1.rTRDY\_ = 0) + (m1.rIRDY\_ = 0) * (t1.STOP\_ = 0)))));$ |
| PCI1 | Y | Y | In the clock cycle that FRAME_ is deasserted, IRDY_ has to be asserted.<br>$AG(((!(m1.rFRAME\_ = 1) * AX(m1.rFRAME\_ = 1))) \rightarrow AX((m1.rIRDY\_ = 0)));$ |
| PCI2 | Y | N | If DEVSEL_ is asserted, ultimately TRDY_ should be asserted.<br>$AG(((!(t1.DEVSEL\_ = 0) * AX(t1.DEVSEL\_ = 0))) \rightarrow (AF(t1.rTRDY\_ = 0)));$ |
| PCI3 | Y | Y | Whenever Trigger is set, a transaction must be initiated by the master.<br>$AG(((!(m1.Trigger = 1) * AX(m1.Trigger = 1))) \rightarrow (AF(m1.rFRAME\_ = 0)));$ |
| PCI4 | Y | Y | Once FRAME_ has been asserted, it should eventually be deasserted.<br>$AG(((!(m1.rFRAME\_ = 0) * AX(m1.rFRAME\_ = 0))) \rightarrow (AF(m1.rFRAME\_ = 1)));$ |
| PCI5 | Y | N | In a transaction if STOP_ and IRDY_ are asserted, FRAME_ must be deasserted.<br>$AG(((m1.rFRAME\_ = 0) * ((t1.STOP\_ = 0) * (m1.rIRDY\_ = 0))) \rightarrow (AF(m1.rFRAME\_ = 1)));$ |
| PCI6 | Y | N | This is achieved by ensuring that TRDY_ is asserted at least one clock cycle after FRAME_ is asserted.<br>$AG((!(m1.rFRAME\_ = 0) * AX(m1.rFRAME\_ = 0)) \rightarrow AF((!(t1.rTRDY\_ = 0) * AX(t1.rTRDY\_ = 0))));$ |
| PCI7 | Y | Y | Once TRDY_ has been asserted, TRDY_ and STOP_ must remain unchanged until the end of the current data phase.<br>$AG(((!(t1.rTRDY\_ = 0) * AX(t1.rTRDY\_ = 0))) \rightarrow AX(A((((t1.rTRDY\_ = 0) * AX(t1.rTRDY\_ = 0)) * ((t1.STOP\_ = 0) * AX(t1.STOP\_ = 0))) \cup (((t1.rTRDY\_ = 0) * (m1.rIRDY\_ = 0)) + ((m1.rIRDY\_ = 0) * (t1.STOP\_ = 0))))));$ |
| PCI8 | Y | Y | Once STOP_ has been asserted, STOP_ and DEVSEL_ must remain unchanged until the end of the transaction.<br>$AG(((!(t1.STOP_{=}0) * AX(t1.STOP_{=}0))) \rightarrow AX(A((((t1.STOP_{=}0) * AX(t1.STOP_{=}0)) * ((t1.DEVSEL_{=}0) * AX(t1.DEVSEL_{=}0)))U((m1.rIRDY_{=}0) * (t1.rTRDY_{=}0) + (m1.rIRDY_{=}0) * (t1.STOP_{=}0)))));$ |
| PCI9 | Y | Y | Once STOP_ has been asserted, TRDY_ and STOP_ must remain unchanged till the end of the transaction.<br>$AG(((!(t1.STOP\_ = 0) * AX(t1.STOP\_ = 0))) \rightarrow AX(A((((t1.rTRDY\_ = 0) * AX(t1.rTRDY\_ = 0)) * ((t1.STOP\_ = 0) * AX(t1.STOP\_ = 0))) \cup ((m1.rIRDY\_ = 0) * (t1.rTRDY\_ = 0) + (m1.rIRDY\_ = 0) * (t1.STOP\_ = 0)))));$ |
| PCI10 | Y | Y | If REQ_ is asserted, GNT_ is eventually asserted.<br>$AG(((!(m1.REQ\_ = 0) * AX(m1.REQ\_ = 0))) \rightarrow (AF(m1.GNT\_ = 0)));$ |
| PCI11 | Y | Y | Observed that if devsel is removed then FRAME_ gets deasserted.<br>$AG(((!(t1.DEVSEL\_ = 1) * AX(t1.DEVSEL\_ = 1))) \rightarrow (AX(m1.rFRAME\_ = 1)));$ |
| PCI12 | N | N/A | This formula checks for the turnaround cycle necessary in READ operations. It uses the OE_AD signal. Compare with 9.ctl which uses the TRDY_ signal. |
| PCI13 | N | N/A | This formula checks for the timing of DEVSEL_ followed by TRDY_ first we had placed an AG condition before DEVSEL_ which was causing formula failure when GNT_ got removed (leading to deassertion of DEVSEL#). 1. The Starting address was set to 0 so the Target must decode it. 2. DecodeWait register is fixed to medium decode so DEVSEL_ should get asserted 2 clocks after frame is asserted. 3. Target Wait is fixed at 3 so TRDY_ should be asserted 2 clocks after DEVSEL_ (Note that all abnormal terminations are disabled, so TRDY_ will be asserted once Decode is done). |
| PCI14 | Y | N | Whether any transaction completes successfully?<br>$EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 0 * m1.TransStatus[1] = 0 * m1.TransStatus[0] = 0);$ |
| PCI15 | Y | N | Whether any transaction completes with Retry?<br>$EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 0 * m1.TransStatus[1] = 0 * m1.TransStatus[0] = 1);$ |
| PCI16 | Y | N | Whether any transaction completes as a disconnect?<br>$EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 0 * m1.TransStatus[1] = 1 * m1.TransStatus[0] = 1));$ |
| PCI17 | Y | N | Whether any transaction completes as target Abort?<br>$EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 0 * m1.TransStatus[1] = 1 * m1.TransStatus[0] = 0);$ |
| PCI18 | Y | N | Whether any transaction sees the master to be busy?<br>$EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 1 * m1.TransStatus[1] = 1 * m1.TransStatus[0] = 0);$ |
| PCI19 | Y | N | Whether any transaction completes with master preempted?<br>$EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 1 * m1.TransStatus[1] = 0 * m1.TransStatus[0] = 1);$ |
| PCI20 | Y | N | Whether transaction status becomes Incomplete?<br>$EF(m1.TransStatus[3] = 1 * m1.TransStatus[2] = 0 * m1.TransStatus[1] = 0 * m1.TransStatus[0] = 0);$ |
| PCI21 | Y | N | Whether any transaction completes with Device Time Out?<br>$EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 1 * m1.TransStatus[1] = 0 * m1.TransStatus[0] = 0);$ |

generate our attribute grammar that the auxiliary verb "must" was used multiple times in conjunction with the word "be". In these initial occurrences there was a clear temporal reference to a future time. As a result, the phrase "must be" was incorporated into the production for the symbol with a semantic mapping to the CTL structure $AF$. In short, due to the phrase "must be" meaning "in the future" in the text that "trained" our grammar, this association was incorporated into the grammar

rules. Explained another way, the ambiguity associated with the phrase "must be" was resolved by looking at past usage of the phrase where the ambiguity was not present. While there is no way to determine if the ambiguity was resolved correctly we believe that it is a reasonable simplification. However, it does indicate that future work must give careful attention to the attribute grammar generation process.

The most obvious area for improvement in this work is in the generation of the attribute grammar. In our implementation the grammar was manually generated by inspecting the natural language text. This a moderately labor intensive process and thus mitigates the benefit realized from the automatic verification property generation. However, context free grammars can be generated from sample text using a process known as *grammatical inference* [14]. In this work we have shown that an appropriately generated attribute grammar can successfully translate natural language text into syntactically and semantically correct CTL, which can then be used without further modification for model checking a design. The natural next step is to adapt an appropriate grammatical inference technique to automatically generate our restricted attribute grammars from sample text. Examples of the grammatical inference of attribute grammars are detailed in [15] and [16]. By adapting a grammar inference technique for our methodology one can envision an improved system where an attribute grammar automatically generated from a specific set of documents (originating from a specific designer or design team) is used to customize a natural language translation system. Because writing style tends to be consistent from fixed individual(s), such a translation system could be customized once and subsequently used for for all code comments written by the target individual(s).

## VI. Conclusion

We have defined and implemented a methodology to automatically generate CTL verification properties from natural language text, characterized our system using a verification benchmark suite, and verified that our automatically generated CTL properties successfully verify the system model. We have also compared the quality of the CTL generated by our method to CTL included in the benchmark.

Human designers most naturally express themselves in a native language. As highlighted in our results analysis, taking ideas that are easily expressed in a natural language and expressing them in a verification language can be an error prone process. While previous attempts at automatic CTL generation attempt to parse unrestricted language our approach is unique in that it uses a customized grammar designed to capture a language subset intended to streamline translation while maintaining expressive power. In addition, our translation process is fully automatic and yields fully defined CTL properties which can be immediately used in model checking or other formal verification processes. Methodologies such as ours can aid in the reduction of verification effort while better leveraging the intelligence of a human engineer, thereby improving the quality of hardware verification results while simultaneously reducing cost.

## References


[1] T. Samad and S. Director, "Towards a natural language interface for cad," in *Design Automation, 1985. 22nd Conference on*, 1985, pp. 2–8.

[2] J. Granacki and A. Parker, "Phran-span: A natural language interface for system specifications," in *Design Automation, 1987. 24th Conference on*, 1987, pp. 416–422.

[3] R. Drechsler, M. Soeken, and R. Wille, "Formal specification level: Towards verification-driven design based on natural language processing," in *Specification and Design Languages (FDL), 2012 Forum on*, Sept 2012, pp. 53–58.

[4] I. Harris, "Capturing assertions from natural language descriptions," in *Natural Language Analysis in Software Engineering (NaturaLiSE), 2013 1st International Workshop on*, May 2013, pp. 17–24.

[5] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini, "Assisting requirement formalization by means of natural language translation," *Form. Methods Syst. Des.*, vol. 4, no. 3, pp. 243–263, May 1994.

[6] R. Nelken and N. Francez, "Automatic translation of natural language system specifications into temporal logic," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, R. Alur and T. Henzinger, Eds. Springer Berlin Heidelberg, 1996, vol. 1102, pp. 360–371.

[7] H. Kamp, "A theory of truth and semantic representation," in *Formal Methods in the Study of Language*, J. Groenendijk, T. Janssen, and M. Stokhof, Eds., 1981.

[8] A. Holt and E. Klein, "A semantically-derived subset of english for hardware verification," in *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, ser. ACL '99. Stroudsburg, PA, USA: Association for Computational Linguistics, 1999, pp. 451–456.

[9] A. Aziz, A. Jas, A. Sen, A. Ramachandran, C. Akturan, C. Liu, D. Das, I.-M. Liu, J. Bhadra, J. R. Denison, K. Das, M. K. Ganai, P. Gopalakrishnan, P. S. Chhabra, P. K. Jaini, R. Chaudhry, R. Narayan, R. Chaba, S. Padmanabhan, S. Srinivasan, W. U. Quddus, and Z. Zhe, "Examples of hw verification using vis," 1997. [Online]. Available: http://vlsi.colorado.edu/ vis/texas-97/

[10] D. E. Knuth, "Semantics of context-free languages," *Theory of Computing Systems*, vol. 2, no. 2, pp. 127–145, Jun. 1968.

[11] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Staple, G. Swamy, and T. Villa, "Vis: A system for verification and synthesis," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, R. Alur and T. Henzinger, Eds. Springer Berlin Heidelberg, 1996, vol. 1102, pp. 428–432.

[12] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, 1st ed. O'Reilly Media, Inc., 2009.

[13] T. Shanley and D. Anderson, *PCI system architecture (3. ed.).*, ser. PC system architecture series. Addison-Wesley, 1995.

[14] A. D'Ulizia, F. Ferri, and P. Grifoni, "A survey of grammatical inference methods for natural language learning," *Artif. Intell. Rev.*, vol. 36, no. 1, pp. 1–27, Jun. 2011.

[15] B. Starkie, "Inferring attribute grammars with structured data for natural language processing," in *Grammatical Inference: Algorithms and Applications*, ser. Lecture Notes in Computer Science, P. Adriaans, H. Fernau, and M. Zaanen, Eds. Springer Berlin Heidelberg, 2002, vol. 2484, pp. 237–248.

[16] S. Zvada and T. Gyimáthy, "Using decision trees to infer semantic functions of attribute grammars," *Acta Cybern.*, vol. 15, no. 2, pp. 279–304, Dec. 2001.