GLAsT: Learning Formal Grammars to Translate Natural Language Specifications into Hardware Assertions

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

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

***Abstract*—The purpose of functional verification is to ensure that a design conforms to its specification. However, large written specifications can contain hundreds of statements describing correct operation which an engineer must use to create sets of correctness properties. This laborious manual process increases both verification time and cost. In this work we present GLAsT, a new learning algorithm which accepts a small set of sentences describing correctness properties and corresponding SystemVerilog Assertions (SVAs). GLAsT creates a custom formal grammar which captures the writing style and sentence structure of a specification and facilitates the automatic translation of English specification sentences into formal SystemVerilog Assertions. We evaluate GLAsT on English sentences from two ARM AMBA bus protocols. Results show that a translation system using the formal grammar generated by GLAsT automatically generates correctly formed SVAs from the targeted AMBA specification as well as from a second, different AMBA bus specification.**

***Index Terms*—hardware design, verification, machine learning, natural language processing**

I. INTRODUCTION

With the advent of Systems-on-Chip (SoCs) the complexity of digital systems is rapidly increasing. In modern engineering practice, verification engineers read system requirements from a specification document and write verification code to check each requirement, one by one. This is a time consuming and labor intensive process and contributes substantially to the increasing percentage of the design cycle spent on verification.

Historically, advances in design automation have primarily targeted digital design as opposed to verification and testing. As a result of this asymmetric focus, up to 60% of the design cycle of a modern SoC consists of verification activities. Recently, some researchers have begun to focus on increasing automation during functional verification. In [1] and [2] researchers propose generating hardware assertions from analysis of register transfer level (RTL) code. Assertion generation from simulation traces is proposed in [3]. These methods, while useful, generate assertions based on an existing implementation of a design as opposed to previously specified design requirements.

Many formalisms for expressing correctness properties exist. A partial list includes the *e* verification language, Property Specification Language (PSL), SystemVerilog Assertions (SVAs), Computation Tree Logic (CTL), Linear Temporal Logic (LTL), and SystemC. These formalisms vary in their expressivity and ease of use. People, however, tend to express themselves best in natural languages such as English.

In the software community artifact generation from natural language is explored in several works. Unified Modeling Language (UML) models are generated from natural language in [4]–[6]. Work in [7], [8] explores the generation of software test cases from natural language. Natural language assisted design has also been explored, to a lesser extent, in the hardware community. In [9] an intermediate level specification is proposed as a bridge between a natural language specification and and executable, high level design implementation. In [10] a method to generate SVAs from natural language based on assertion templates is presented.

Previous work in [11]–[13] demonstrates that a carefully constructed formal attribute grammar can be used in a Natural Language Processing (NLP) based automatic translation system to map English language requirements to hardware verification invariants such as SVAs or CTL. However, these translation systems require that the grammar be constrained to a domain specific subset of English. This reduces ambiguity in requirement descriptions while maintaining the expressive power of natural language. These types of grammar based translation systems work best when the formal grammar used in translation is customized to capture the specific writing style used in the target specification. However, the creation of such grammars generally requires an expert in both NLP and digital design. Experts of this type are in exceedingly short supply.

The main contribution of this work is the GLAsT algorithm. **GLAsT** stands for **G**rammar **L**earning for **As**sertion **T**ranslation. GLAsT is a *grammar learning* algorithm which accepts a small example set of natural language requirements from a specification and SVAs describing these requirements. GLAsT then infers a formal attribute grammar from these examples which captures the specific writing style used in the specification. This custom grammar is then used in a translation system which maps sentences in a specification to fully specified hardware assertions. These assertions, which have been translated from English to SystemVerilog, can then

978-3-9815370-6-2/DATE16/©2016 EDAA

be used in a standard verification flow.

The remainder of this paper will outline the development and evaluation of the GLAsT algorithm. Section 2 presents necessary background on attribute grammars, grammatical inference, and NLP based translation systems. In Section 3 we present the details of GLAsT. In Section 4 we present implementation details and highlight our experimental results. Finally, in Section 5 we summarize our conclusions.

II. Background

A. Attribute Grammars

Attribute grammars, originally developed by Knuth, are a type of formal grammar which assign specific attributes to the productions of a *context free grammar* (CFG) [14]. Attribute grammars allow grammatical symbols in a parse tree to be replaced by attributes associated with the production corresponding to the grammatical symbol.

Attribute grammars used in *semantic parsing* can generate parse trees which not only represent the syntactic structure of a sentence but also the semantic meaning. Equations (1)–(4) are productions from a simple attribute grammar. These four productions and associated attributes (denoted by the symbol name and a trailing **.a**) are used in parsing the natural language sentence "Reset should be high". The semantic parse tree generated from the application of the sample grammar to the target sentence is shown in Fig.1a. This parse tree represents the semantics of setting a signal in a digital system.

Attributes are evaluated in a top-down manner. The attribute for the sentence start symbol S is defined as the attribute value for the SETSIG symbol followed by a terminating semicolon. The attribute value SETSIG.a is a synthesized attribute which consists of the body of the assertion statement defined in terms of the attributes of SETSIG's child nodes. These child nodes are leaf nodes and so their attribute values (SIGNAME.a and VAL.a) can be evaluated directly. With the values of the leaf nodes known, SETSIG.a is evaluated to be "assert property (@(posedge clk) module1.rst == 1)". This, in turn, is used to resolve the attribute value at the sentence level resulting in the fully defined SVA: "assert property (@(posedge clk) module1.rst == 1);".

$$P_0 \begin{cases} \text{S} \rightarrow \text{SETSIG} \\ \text{S.a} \equiv SETSIG.a \text{ ;} \end{cases} \tag{1}$$

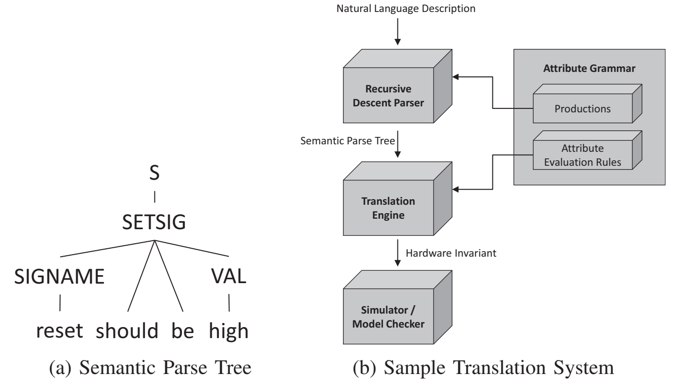
$$P_1 \begin{cases} \text{SETSIG} \rightarrow \text{SIGNAME “should” “be” VAL} \\ \text{SETSIG.a} \equiv \text{assert property (@(posedge clk)} \\ \qquad\qquad SIGNAME.a \text{ == } VAL.a) \end{cases} \tag{2}$$

$$P_2 \begin{cases} \text{SIGNAME} \rightarrow \text{“Reset”} \\ \text{SIGNAME.a} \equiv \text{module1.rst} \end{cases} \tag{3}$$

$$P_3 \begin{cases} \text{VAL} \rightarrow \text{“high”} \\ \text{VAL.a} \equiv 1 \end{cases} \tag{4}$$

B. NLP Translation Systems

A typical attribute grammar based translation system used to generate hardware verification properties is shown in Figure 1b. In a system of this type natural language system requirements are taken from a specification or other design document. Each sentence is then parsed using the productions in the formal grammar to generate a semantic parse tree.



(a) Semantic Parse Tree (b) Sample Translation System

Fig. 1. Semantic Parsing Example

The semantic parse is sent to the translation engine where attribute evaluation rules from the formal grammar are used to convert individual grammatical symbols into verification code snippets. These code snippets are assembled according to the structure defined by the semantic parse tree, resulting in a fully defined hardware invariant. One hardware invariant is output for each natural language requirement input to the system.

The type of invariant created is defined by the attribute evaluation rules in the formal grammar, but is typically an assertion (SVA) or formal logic property (CTL). The set of hardware invariants generated by the translation engine can be used without any further processing in a simulator or model checker to verify the design. For a more detailed overview of translation systems for hardware invariants see [13].

C. Grammatical Inference

Grammatical inference is also known in the literature as language learning, grammar learning, or grammar induction. It is the process of inferring a set of grammatical rules forming a grammar G, given some information about the language to be learned. This information usually takes the form of a set of sample sentences from the unknown language [15].

The language $L(G)$ is the set of all sentences recognizable (or able to be generated) by the learned grammar G. $L(G)$ should model the language to be learned as closely as possible. Ideally, $L(G)$ is equal to the set of all possible sentences in the unknown language but in practice it is either larger or smaller. Larger induced languages are the result of grammars that are too general (a result of underfitting) while too small languages are too specific (due to overfitting).

Language learning begins with a set of example sentences. From this learning set an initial set of production rules are generated. These production rules are iteratively generalized and constrained until a satisfactory set of rules is reached which can recognize (or generate) a maximum number of sentences in the unknown target language.

Grammatical inference algorithms can be either *supervised* or *unsupervised*. *Supervised* algorithms have some knowl-

edge of the structure of a "valid" grammar. Thus when a set of potential production rules is generated they can be evaluated against a known good set of rules. As a result, supervised methods tend to generate more accurate results than unsupervised methods. However, for many interesting problems a known good grammar is not available. *Unsupervised* algorithms have no knowledge of what a valid grammar looks like, only unstructured sentences from the language to be learned (positive examples). The majority of grammatical inference algorithms found in the literature in recent years are unsupervised algorithms using positive learning examples only [16]. In this work we also utilize an unsupervised algorithm using positive examples, which will be detailed in the next section.

III. Algorithm Design

GLAsT extends the E-GRIDS algorithm presented in [17]. E-GRIDS performs grammatical inference for context free grammars. GLAsT extends E-GRIDS to support attributed grammars which allows English to SVA translation. An overview of the GLAsT architecture is shown in Fig.2. The user must supply a learning set of natural language sentences from the specification describing correctness properties, an SVA for each learning set sentence, and a list of signal names used in the design. In the initial Grammar Generation phase, a token symbol is created for each unique word in the set of sample sentences. Attribute values for this set of symbols are assigned based on an initial mapping. An example would be the value *module1.rst* which was assigned as the attribute value for the terminal symbol SIGNAME in equation (3). These initial mappings are provided by the user. After each unique word in the learning set of example sentences has been assigned a token (and attribute if appropriate), a top level production for each sentence is created by replacing each word in the sentence with its previously defined token. Finally, each top level production is assigned an attribute. The attribute is the SVA which matches the sentence from the learning set. Words or symbols in the SVA which match the attribute value of a tokenized word are replaced with the appropriate attribute references.

Individual words in the learning set are called *terminal* symbols and can only appear on the right hand side of productions. All symbols which are not terminal symbols are called *non-terminal* symbols (denoted S#). A special non-terminal symbol, called the *start* symbol (denoted S0), begins each full sentence. Non-terminal symbols may appear anywhere in a production. A small learning set and an initial grammar of 12 productions are shown in Tables I and II. Note that in the attribute value $S0.a$ of production $P1$ the references $S1\text{-}1.a$ and $S1\text{-}2.a$ refer to the first and second occurrences of the $S1$ symbol on the right hand side of the relation in $P1$.

Once an initial grammar has been created the search space of potential grammars is explored using a beam search technique. A beam search is a best first search where, due to limited memory, the number of states in the search tree is pruned using a heuristic [18]. A fixed number of search states are retained at any given time. These states are stored in a data structure called a "beam". In GLAsT, the beam retains between 3 and 5 grammars at a time and the heuristic used for pruning is the *minimum description length* (MDL) of a grammar. MDL, first proposed by Rissanen [19], formalizes the idea that the "best" configuration for a set of data is the one that leads to the minimum length representation of the data. With respect to formal grammars, a good grammar is considered one that can not only parse the learning set, but has a small number of productions where each production has a small average length. Using MDL as a pruning heuristic biases the algorithm towards more compact grammars. Calculation of the MDL for formal grammars is outlined in [17].

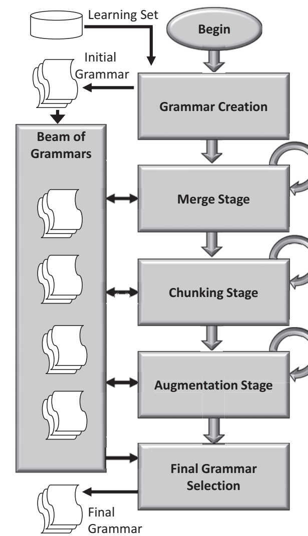


Fig. 2. GLAsT Architecture

GLAsT explores the search space of grammars by proceeding sequentially in three main stages. At each stage grammars in the beam are modified using one of three binary operators: *merge*, *chunk*, or *augment*. Each operator accepts a pair of non-terminal symbols from the grammar. During the corresponding stage, each operator is repeatedly applied to the grammars in the beam. Every application of a primary operator is performed with a different pair of non-terminal symbols, generating a new candidate grammar. When all grammars in the beam have been permuted using a primary operator, the candidate grammars are evaluated by MDL. Candidates which have a smaller MDL than a grammar currently in the beam replace the beam grammar with the highest MDL. If any new grammars enter the beam in this manner then the algorithm remains in the same stage and begins another pass, further exploring the search space by again permuting and evaluating grammars in the beam. Only when no new grammars enter the beam during a pass does GLAsT determine that no additional improvement is possible with the current operator it advances to the next stage. After completion of the augment stage the grammar in the beam with the lowest MDL is selected as the best solution. We will now briefly outline each of the three primary operators.

TABLE I
INITIAL LEARNING SET

Natural Language Description	SystemVerilog Assertion
Reset should be high	assert property(@(posedge clk) module1.rst == 1);
AWBURST remains stable when AWVALID is asserted	assert property (@(posedge clk) (AWVALID == 1) |→ $stable(AWBURST_));

TABLE II
INITIAL GRAMMAR CREATION

Initial Grammar					
$P_0 \equiv$	S0 → S1 S2 S3 S4 S0.a = [assert property(@(posedge clk) $S1.a$ $S3.a$ $S4.a$;]			$P_2 \equiv$	S1 → Reset S1.a = [module1.rst]
$P_1 \equiv$	S0 → S1 S5 S6 S7 S1 S8 S9 S0.a = [assert property(@(posedge clk) ($S1$-2.a $S8.a$ $S9.a$) |→ $S6.a$ ($S1$-1.a));]			$P_3 \equiv$	S1 → AWBURST S1.a = [AWBURST_]
$P_4 \equiv$	S1 → AWVALID S1.a = [AWVALID]	$P_5 \equiv$	S2 → should S2.a = [ ]	$P_6 \equiv$	S3 → be S3.a = [==]
$P_7 \equiv$	S4 → high S4.a = ["1"]	$P_8 \equiv$	S5 → remains S5.a = [ ]	$P_9 \equiv$	S6 → stable S6.a = [$stable]
$P_{10} \equiv$	S7 → when S7.a = [ ]	$P_{11} \equiv$	S8 → is S8.a = [==]	$P_{12} \equiv$	S9 → asserted S9.a = ["1"]

The *merge* operator combines two existing non-terminal symbols into a new non-terminal symbol. This new non-terminal symbol replaces all instances of the two existing non-terminal symbols in grammar. The merge operator preserves attribute assignments associated with a production. However, attribute values that appear in the definition(s) of higher level attributes may need to be updated. The merge operator has the effect of generalizing the grammar and capturing words or phrases which are used in an equivalent manner in a specification. This is shown in Table III where the symbols S3 and S8 are merged into a single symbol as the words "be" and "is" are used interchangeably in the sentences of the learning set to denote signal assignments.

TABLE III
MERGE OPERATOR FUNCTION

Before Merge			
$P_0 \equiv$	S0 → S1 S2 S3 S4 S0.a = [assert property(@(posedge clk) $S1.a$ $S3.a$ $S4.a$;]		
$P_6 \equiv$	S3 → be S3.a = [==]	$P_{11} \equiv$	S8 → is S8.a = [==]
After Merging Symbols S3 and S8			
$P_0 \equiv$	S0 → S1 S2 S3_S8 S4 S0.a = [assert property(@(posedge clk) $S1.a$ $S3_S8.a$ $S4.a$;]		
$P_6 \equiv$	S3_S8 → be S3_S8.a = [==]	$P_{11} \equiv$	S3_S8 → is S3_S8.a = [==]

The second primary operator, the *chunk* operator, could also appropriately been named the concatenation operator. It takes two existing non-terminal symbols and creates a new production where the two existing non-terminals appear sequentially. The operator then searches all existing productions and replaces each sequence of the two existing non-terminals (the "chunk") in a production with the symbol for the new production. This operator reduces the generality of the grammar and captures commonly used sequences of words (phrases) in the specification. The attribute value of a new "chunk" is the concatenation of the attribute values of its constituent symbols. Table IV shows how the chunk operator is used to capture the phrase "should be". The attribute for this new symbol is the attribute value of "should" (empty) followed by the attribute value of "be" (=), thus the phrase "should be" also has an attribute value denoting a signal assignment.

TABLE IV
CHUNK OPERATOR FUNCTION

Before Chunking			
$P_0 \equiv$	S0 → S1 S2 S3 S4 S0.a = [assert property(@(posedge clk) $S1.a$ $S3.a$ $S4.a$;]		
$P_5 \equiv$	S2 → should S2.a = [ ]	$P_6 \equiv$	S3 → be S3.a = [==]
After Chunking			
$P_0 \equiv$	S0 → S1 S2__S3 S4 S0.a = [assert property(@(posedge clk) $S1.a$ $S2__S3.a$ $S4.a$;]		
$P_5 \equiv$	S2 → should S2.a = [ ]	$P_6 \equiv$	S3 → be S3.a = [==]
$P_{13} \equiv$	S2__S3 → S2 S3 S2__S3.a = [$S2.a$ $S3.a$]		

The third and final major stage in GLAsT is the aug-

TABLE V
AUGMENT OPERATOR FUNCTION

Before Augmentation		After Augmentation of S2__S3 with S4	
$P_0 \equiv$ { S0 → S1 S2__S3 S4 S0.a = [assert property(@(posedge clk) $S1.a$ $S2__S3.a$ $S4.a$;] }		$P_0 \equiv$ { S0 → S1 S2__S3 S0.a = [assert property(@(posedge clk) $S1.a$ $S2__S3.a$;] }	
$P_5 \equiv$ { S2 → should S2.a = [ ] }	$P_{16} \equiv$ { S0 → ... S11 S4 ... S0.a = [...] }	$P_5 \equiv$ { S2 → should S2.a = [ ] }	$P_{16} \equiv$ { S0 → ... S11 ... S0.a = [...] }
$P_6 \equiv$ { S3 → be S3.a = [==] }	$P_{17} \equiv$ { S10 → ... S2__S3 S10.a = [... $S2__S3.a$] }	$P_6 \equiv$ { S3 → be S3.a = [==] }	$P_{17} \equiv$ { S10 → ... S2__S3 S10.a = [... $S2__S3.a$] }
$P_{13} \equiv$ { S2__S3 → S2 S3 S2__S3.a = [$S2.a$ $S3.a$] }		$P_{13} \equiv$ { S2__S3 → S2 S3 S2__S3.a = [$S2.a$ $S3.a$] }	
		$P_{14} \equiv$ { S2__S3 → S2 S3 S4 S2__S3.a = [$S2.a$ $S3.a$ $S4.a$] }	
$P_{15} \equiv$ { S0 → ... S10 S4 ... S0.a = [...] }	$P_{18} \equiv$ { S11 → ... S10 S11.a = [... $S10.a$] }	$P_{15} \equiv$ { S0 → ... S10 ... S0.a = [...] }	$P_{18} \equiv$ { S11 → ... S10 S11.a = [... $S10.a$] }

mentation stage. In the augmentation stage the operator of the same name creates a new production which adds a third non-terminal symbol to the result of a previously discovered "chunk". The new, augmented, production keeps the same symbol name as the chunk from which it is derived. The augmentation of a chunk with an additional non-terminal symbol further generalizes the grammar by increasing the number of sentences in a specification which can be parsed. Attributes in an *augment* operation are handled in the same manner as a chunk operation. The attribute value is defined as the concatenation of the attribute values of its constituent symbols. Table V demonstrates the augment operator.

During each major stage in the GLAsT the algorithm, grammars in the beam are transformed using the primary operator associated with a stage in order to create candidate grammars. In the merge stage, symbols representing words which can be used interchangeably in a specification (like signal names) are combined. In the chunk stage, sequences of non-terminal symbols which appear frequently in the specification are added to the grammar. In the augmentation stage, symbols denoting phrases are augmented with an additional non-terminal symbol to generate new candidate grammars. As grammars are iteratively generalized the best of them are saved in the beam. The MDL is used as a pruning metric. As previously noted, GLAsT only moves to the stage when no additional improvement at the current stage is detected. At the conclusion of the augmentation stage, the final grammar selected as the solution is the most compact grammar which which translates the sentences in the learning set. This grammar captures the generalized structure of the sentences in the learning set and, through the use of the attribute values, maps them to SVAs patterned after those found in the learning set.

IV. EXPERIMENTAL RESULTS

A. Implementation

We implemented our learning algorithm in Python using version 2.0 of the Natural Language Toolkit [20]. A set of 88 natural language verification requirements were taken form the ARM AMBA 3 AXI Protocol Checker User Guide [21]. These 88 natural language descriptions were divided in a learning set of 17 sentences and a cross-validation set of 71 sentences. SystemVerilog Attributes were created for the 17 sentence learning set. These 17 sentences were selected to be the most representative of all sentences in the specification document in terms of both structure and word content. A short preprocessing step was also performed to resolve pronoun references (such as replacing the word "it" with the appropriate signal name). Pronoun resolution, also called anaphora resolution, has several standard approaches [22].

B. Results

The initial grammar generated from our 17 sentence learning set contained 206 words and 111 productions. Because it is highly specific, the initial grammar only parses and generates SVAs for the exact sentences in the learning set. The final grammar generated by GLAsT showed a great deal of compaction with only 16 unique symbols representing 10 distinct sentence structures.

The grammar learned by GLAsT from the 17 learning set requirements and SVAs was used in a NLP translation framework similar to [13] to generate syntactically and semantically correct SVAs from the 71 distinct sentences in the cross-validation set. These requirements were not previously seen by GLAsT but shared a similar grammatical structure to the sentences in the learning set. A subset of sentences from the cross-validation set and automatically generated SVAs are shown in Table VI[1]. SVAs created by a GLAsT generated grammar are patterned after the SVAs provided by the user in the learning set. English sentences which share a similar structure share a similar semantic meaning, and thus are mapped to the same assertion structure. High quality SVAs in the learning set result in high quality translation results. This is highlighted in the second and third natural language requirements in Table VI.

[1] A full set of English requirements and autogenerated SVA translations is available at https://www.dropbox.com/sh/rt9phnpyh6iifpv/AAAJ2r063nz-nthHs4qMylpIa?dl=0

TABLE VI
SELECTED AUTOMATICALLY GENERATED SYSTEMVERILOG ASSERTIONS

<table>
<tr><th>Natural Language Requirements</th><th>SystemVerilog Assertion</th></tr>
<tr><td>BVALID is LOW for the first cycle after ARESETn goes HIGH</td><td>{assert property (@(posedge clk) $rose(ARESETn) |→ (BVALID == 0));}</td></tr>
<tr><td>A value of X on WUSER is not permitted when WVALID is HIGH</td><td>{assert property (@(posedge clk) (WVALID == 1) |→ (WUSER != X));}</td></tr>
<tr><td>A value of X on CACTIVE is not permitted when not in reset</td><td>{assert property (@(posedge clk) RESET == 0 |→ (CACTIVE != X));}</td></tr>
<tr><td>RLAST remains stable when RVALID is asserted and RREADY is LOW</td><td>{assert property (@(posedge clk) (RVALID == 1 && RREADY == 0)
|→ $stable (RLAST));}</td></tr>
<tr><td>Parameter WDEPTH must be greater than or equal to 1</td><td>{assert property (@(posedge clk) WDEPTH >= 1);}</td></tr>
</table>

We also evaluated the grammar produced by the GLAsT algorithm on sentences from a different specification. A sample of 48 natural language descriptions of assertions were taken from the AMBA 4 AXI Protocol User Guide [23]. The AMBA AXI 4 Protocol Guide was chosen because although it is a separate document from the one used by GLAsT to generate the translation grammar, it shares a similar writing style. The sentences from the AMBA 4 specification were also preprocessed to remove pronoun references and new signal names were added to the input set. Our GLAsT generated grammar was able to successfully translate these 48 unseen assertion descriptions into fully formed SVAs.

C. Conclusions and Future Work

We have constructed a new *grammatical inference* algorithm capable of learning a formal attribute grammar which can be used to successfully generate SystemVerilog attributes from previously unseen English verification descriptions across multiple documents with similar writing styles. Although the grammar is automatically generated, thus removing the necessity for an NLP expert when utilizing a grammar based translation system, the quality of the learned grammar is still strongly dependent on the initial learning set. The extent to which the sentences chosen for the learning set capture the linguistic variation used in the specification impact the number sentences which can be translated to SVAs. Future work will investigate heuristics to choose learning sets that give good coverage of a target family specification documents.

REFERENCES

[1] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, March 2010, pp. 626–629.

[2] A. Hekmatpour and A. Salehi, "Block-based schema-driven assertion generation for functional verification," in *Test Symposium, 2005. Proceedings. 14th Asian*, Dec 2005, pp. 34–39.

[3] E. El Mandouh and A. Wassal, "Automatic generation of hardware design properties from simulation traces," in *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, May 2012, pp. 2317–2320.

[4] D. Deeptimahanti and M. Babar, "An automated tool for generating uml models from natural language requirements," in *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, Nov 2009, pp. 680–682.

[5] R. Drechsler, I. Harris, and R. Wille, "Generating formal system models from natural language descriptions," in *High Level Design Validation and Test Workshop (HLDVT), 2012 IEEE International*, 2012, pp. 164–165.

[6] R. Sharma, S. Gulia, and K. Biswas, "Automated generation of activity and sequence diagrams from natural language requirements," in *Evaluation of Novel Approaches to Software Engineering (ENASE), 2014 International Conference on*, April 2014, pp. 1–9.

[7] R. Verma and M. Beg, "Generation of test cases from software requirements using natural language processing," in *Emerging Trends in Engineering and Technology (ICETET), 2013 6th International Conference on*, Dec 2013, pp. 140–147.

[8] E. Sarmiento, J. Sampaio do Prado Leite, and E. Almentero, "C amp;l: Generating model based test cases from natural language requirements descriptions," in *Requirements Engineering and Testing (RET), 2014 IEEE 1st International Workshop on*, Aug 2014, pp. 32–38.

[9] 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.

[10] M. Soeken, C. B. Harris, I. G. Harris, N. Abdessaied, and R. Drechsler, "Automating the translation of assertions using natural language processing techniques," in *Specification and Design Languages (FDL), 2014 Forum on*, October 2014.

[11] 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.

[12] I. G. Harris, "Extracting design information from natural language specifications," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1256–1257.

[13] C. B. Harris and I. G. Harris, "Generating formal hardware verification properties from natural language documentation," in *Semantic Computing (ICSC), 2015 IEEE International Conference on*, February 2015.

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

[15] K.-S. Fu and T. L. Booth, "Grammatical inference: Introduction and survey - part i," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. SMC-5, no. 1, pp. 95–111, Jan 1975.

[16] 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.

[17] G. Petasis, G. Paliouras, V. Karkaletsis, C. Halatsis, and C. D. Spyropoulos, "E-GRIDS: Computationally Efficient Grammatical Inference from Positive Examples," *GRAMMARS*, vol. 7, pp. 69–110, 2004.

[18] R. Bisani, "Beam search," in *Encyclopedia of Artificial Intelligence*, 2nd ed., S. C. Shapiro, Ed. New York, NY, USA: John Wiley & Sons, Inc., 1992.

[19] J. Rissanen, *Stochastic Complexity in Statistical Inquiry Theory*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1989.

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

[21] *AMBA 3 AXI Protocol Checker User Guide*, r0p1 ed., ARM, Jun. 2009.

[22] R. Mitkov and W. W. Sb, "Anaphora resolution: The state of the art," Tech. Rep., 1999.

[23] *AMBA 4 AXI4, AXI4-Lite, and AXI4-Stream Protocol Assertions User Guide*, r0p1 ed., ARM, Jul. 2012.