Automatic Assertion Generation from Natural Language Specifications Using Subtree Analysis

Junchen Zhao
Department of Computer Science
University of California Irvine
Irvine, CA, USA
junchez3@uci.edu

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

***Abstract*—We present an approach to generate assertions from natural language specifications by performing semantic analysis of sentences in the specification document. Other techniques for automatic assertion generation use information found in the design implementation, either by performing static or dynamic analysis. Our approach generates assertions directly from the specification document, so bugs in the implementation will not be reflected in the assertions. Our approach parses each sentence and examines the resulting syntactic parse trees to locate subtrees which are associated with important phrases, such as the antecedent and consequent of an implication. Formal assertions are generated using the information inside these subtrees to fill a set of assertion templates which we present. We evaluate the effectiveness of our approach using a set of statements taken from a real specification document.**

I. Introduction

Assertion-based verification (ABV) [4] is a well used hardware verification technique and an important topic of research. An assertion is a program invariant which is evaluated during hardware simulation to perform automatic result checking. The use of assertions normally requires the manual definition of executable assertions, a process which is time-consuming and difficult. The work presented in this paper seeks to alleviate the difficulties of assertion definition by creating assertions automatically from natural language descriptions. Natural language is semantically closer to the original intent of the assertion in the mind of the human designer, so natural language descriptions are faster to formulate and are less likely to contain errors than manually-generated formal assertion descriptions. In this work we specify executable assertions using the SystemVerilog hardware verification language [15] because it is widely accepted in practice.

Several previous research efforts have attempted to generate properties and assertions using different formal languages including CTL [8], ACTL[3], SystemVerilog [6], [14], and OCL [9]. However, each of these approaches imposes limitations on the generation process in order to make the problem tractable. Several techniques rely heavily on manual interaction to convert the original natural language into a form which is easier to process [14], [9], [3]. Some techniques only process a tightly constrained English subset [8], [3]. Previous work [6] has presented an approach for this problem which depends on the use of an *attribute grammar* [11] to define the formal semantics of a subset of assertion descriptions in English. The approach is effective but it has the weakness that the attribute grammar had to be defined by hand, a tedious and time-consuming task. Research presented in [5] improves on the initial approach by using a learning technique to automatically generate the attribute grammar. In this paper we present an approach to automatically generate SystemVerilog assertions directly from natural language descriptions in English. The approach presented in this paper advances previous work by performing **subtree identification** to locate key information inside the parse tree of a sentence.

The many-to-many relationship between natural language descriptions and formal behavioral descriptions exposes the problem of **linguistic variation** which must be addressed. Linguistic variation in natural languages describes the fact that a single concept to be expressed by many acceptable natural language statements. We address linguistic variation in two ways. Subtree identification is resistant to syntactic variation because different sentence syntax may change the location of the subtree, but typically does not alter the subtree itself. So the subtree can still be located, even in a sentence with different syntax. We address morphological variation by using synonym lists and lemmatization to normalize our approach.

II. Mapping from Natural Language to SystemVerilog

Our approach maps between two domains: natural language sentences and SystemVerilog assertions. In this section we characterize the natural language sentences, the SystemVerilog assertions, and the mapping between the two domains.

A. Characterization of Sentences

We analyze sentences which express a *declarative* mood, meaning that the primary function of the sentence is to declare information. Sentences in English can express other moods such as an interrogative mood which requests information, and an imperative mood which issues commands. We assume that most sentences in a specification will express a declarative mood. We restrict our analysis to declarative sentences in one

This work was supported by the National Science Foundation (NSF) under Award NSF-CNS-1813858.

978-3-9819263-2-3/DATE19/©2019 EDAA

of two categories. We consider either *simple statements* or *conditional statements*.

A simple statement declares relationship between an object and its legal values using a single predicate. Examples of simple statements include, “signal X is HIGH”, “Y must be asserted”, and “Z must be stable”. The common grammatical structure of a simple statement is a noun phrase (NP) which contains the name of a specification object, followed by a verb phrase (VP) containing a verb and a direct object if one is present. A conditional statement is one which expresses a fact which is true only if certain conditions are met. Examples of conditional statements include, “if X is HIGH then Y is LOW”, and “A is asserted when B is HIGH”. Each conditional statement contains two clauses, a *main clause* which expresses a fact, and a *conditional clause* which expresses the conditions under which the main clause is true. For example, the sentence, “if X is HIGH then Y is LOW”, contains a main clause, “Y is LOW”, and a conditional clause, “X is HIGH”.

We assume that the main and conditional clauses are composed of a series of simple statements combined with the conjunctions “and” or “or”, as in the conditional statement, “if A is HIGH and B is LOW then C is greater than 2”. In this sentence, the conditional clause is the conjunction of the simple statements “A is HIGH” and “B is LOW”. In general, a main clause or conditional clause may be more complicated than the conjunction of simple statements, but we do not address such statements because they are uncommon due to their relative complexity.

System behavior is described by expressing constraints on the values of a set of *specification objects* which represent required artifacts in the final design. Our research focuses on the analysis of hardware specification and in that domain, the specification objects are registers which hold values indefinitely, and signals which hold values as long as they are driven. As an example, the statement “A is HIGH” constrains the value of either a register or a signal names “A”.

B. Characterization of Assertions

We define three SystemVerilog *templates* which capture the structure of the assertions which we generate. Each template contains a number of *slots* which are represented by variables and must be replaced by valid SystemVerilog code in order to completely specify the assertion. Figure 1 shows the three assertion templates we define.

Template 1	**?v1 ?op ?v2**
Template 2	**?op (?v1)**
Template 3	**?ante $\|->$?neg ?cons**

Fig. 1. SystemVerilog Assertion Templates

The first two templates in Figure 1 correspond to simple statements which are independent of any condition. The first template imposes a constraint on the the value of a signal/variable using a comparison operator, where ?v1 and ?v2 are two values being related, and ?op is the relation operator. Examples of SystemVerilog matching the first template include “X = 1” or “Y $>$ 5”. The second template constrains a signal/value using a unary function which is part of the SystemVerilog language, such as “$rise” or “$stable”. An example of template 2 is “$stable(X)”. The third template is a conditional relationship between an antecedent ?ante, and a consequent ?cons. The ?neg slot is a negation operator, if negation is stated in the natural language sentence. The third template includes the SystemVerilog implication symbol “$|->$”.

The three templates can be combined hierarchically in order to form more complex assertions. For example, the assertion “(X = 5) $|->$ $stable(Y)” matches template 3 and it contains an antecedent “X = 5” which matches template 1, and a consequent “$stable(Y)” which matches template 2.

C. Mapping using Subtrees

Given a SystemVerilog template with a set of slots, the task of generating an assertion is reduced to filling each slot of the template based on the natural language sentence. This requires the identification of phrases inside the natural language sentence which correspond to each slot. For example, the sentence “X must be stable.” is mapped to template 2, *?op (?v1)*, by mapping the phrase “X” to the slot ?v1, and mapping the phrase “must be stable” to the slot ?op.

We extract syntactic information about a sentence by generating a syntactic **parse tree** using a context-free grammar. The English language is not context-free [7] but the efficiency of context-free parsers have led to their acceptance for capturing English and other natural languages in practice. A context-free parser recognizes strings in the language and determines a derivation of the string from the productions of the language, if one exists. The derivation, in the form of a syntactic parse tree, represents a hierarchical mapping from between symbols based on the production rules.

We observe that each slot in a template can be associated with a **matching subtree** which is used to identify the corresponding phrase in natural language sentence. For each slot, the syntactic parse tree of the sentence is searched to find the matching subtree. The phrase which corresponds to the slot is the phrase which is underneath the matching subtree in the syntactic parse tree of the sentence.

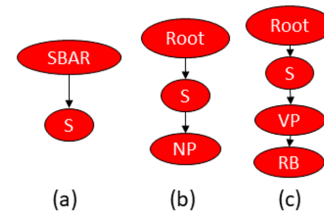

Fig. 2. Matching subtrees for slots in Template 3, (a) ?ante, (b) ?cons, (c) ?neg

The concept of a matching subtree is more easily understood by example. Figure 2 shows matching subtrees for the slots in template 3 which describes conditional assertions. Our claim is that these subtrees can be found in the syntactic parse trees

majority of conditional sentences, and that the subtrees found underneath each matching subtree describes the phrases for each slot. We show an example of the subtree identification process using the following three conditional sentences.

1) "A value of X on WLAST is not permitted when WVALID is HIGH."
2) "Asserting AWID is not allowed when TEST is LOW."
3) "When WVAID is HIGH a value of X on WLAST is not permitted."

The first two sentences are taken from the AMBA 3 AXI Protocol Checker User Guide [2] which contains a set of assertions for the AMBA 3 AXI bus protocol. The third sentence is an alternate statement of the first sentence which we created in order to demonstrate the flexibility of our approach.

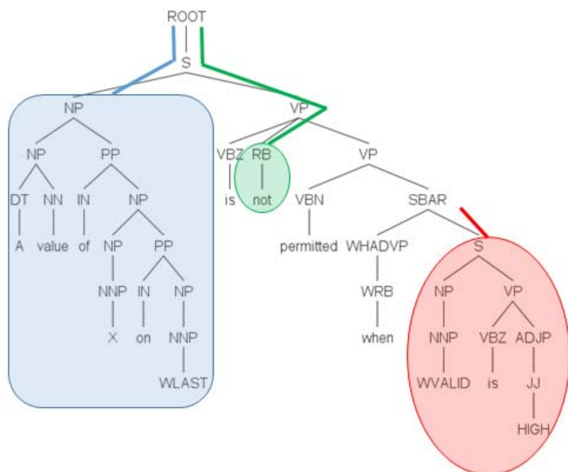

Fig. 3. Parse tree for sentence 1

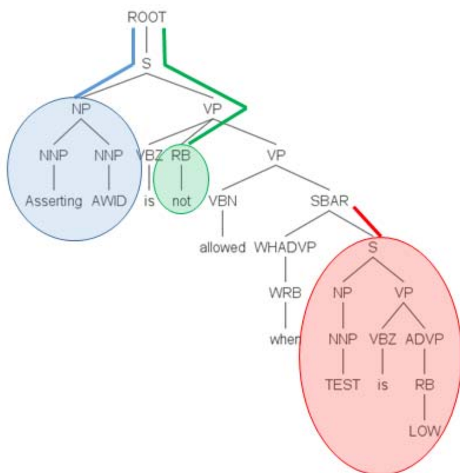

Fig. 4. Parse tree for sentence 2

Figures 3, 4, and 5 show the parse trees of sentences 1, 2, and 3, respectively. In each parse tree the matching subtrees for each slot are highlighted, as shown in Figure 2, and the

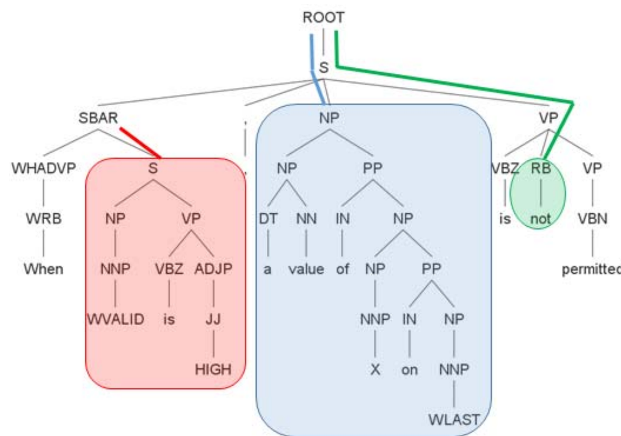

Fig. 5. Parse tree for sentence 3

subtrees beneath them. The subtree for the ?ante slot is shown in red, the ?cons slot is shown in blue, and the ?neg slot is shown in green.

The main observation to make based on Figures 3, 4, and 5 is that the same set of matching subtrees is used to identify phrases corresponding to each slot, in spite of the apparent differences in the sentence structures. Sentences 1 and 2 differ in the way in which signal assertions are referenced, "A value of X on WLAST" vs. "Asserting AWID". Sentences 1 and 3 differ in the ordering of the antecedent and the consequent: consequent first in sentence 1 and antecedent first in sentence 3. The use identification of subtrees allows the extraction of slots in the presence of significant linguistic variation.

III. Algorithm for Assertion Generation

The algorithm used to generate a SystemVerilog assertion from a natural language sentence is shown in Figures 6 and 7. The main function is the $GenerateSV()$ function shown in Figure 6. $GenerateSV()$ takes a sentence as an argument and returns either the equivalent SystemVerilog assertion, or NULL if the generation process was not successful. The loop starting at line 2 iterates through each template and attempts to fill the slots of that template by calling the $FillTemplate()$ function on line 3. If the slots of a template are filled then the SystemVerilog assertion is returned at line 5. If no template can be filled then NULL is returned at line 6.

```
1.  GenerateSV(sentence n)
2.      foreach template t
3.          sv = FillTemplate(t, n)
4.          if sv != NULL
5.              return(sv)
6.      return(NULL)
```

Fig. 6. Algorithm to Generate SystemVerilog Assertion

The $FillTemplate()$ function shown in Figure 7 takes a sentence and a template as arguments and returns either the SystemVerilog assertion produced by filling each template, or

NULL if one or more slots cannot be filled. The main loop of the function, starting on line 3, iterates through each slot and attempts to fill the slot by calling the $FindMatch()$ function on line 4. If no matching phrase is found for a slot the NULL is returned at line 6. If the phrase matches a slot refers to only terminals (signals, constants, SystemVerilog operators) then the SystemVerilog is directly generated from each terminal to produce the assertion, on line 8. If the matching phrase contains at least one non-terminal sub-phrase, then the function $GenerateSV()$ is called recursively with the phrase as an argument to generate its SystemVerilog.

```
1.  FillTemplate(template t, sentence n)
2.      filledT = NULL
3.      foreach slot s in t
4.          matchText = FindMatch(s, t, n)
5.          if matchText == NULL
6.              return(NULL)
7.          if matchText is Terminal
8.              matchSV = Terminal(matchText)
9.          else
10.             matchSV = GenerateSV(matchText)
11.         Substitute(matchSV, filledT)
12.     return(filledT)
```

Fig. 7. Algorithm to Fill Slots in a Template

The $FindMatch()$ function identifies the phrase in a sentence which matches a slot in a template. This is performed by identifying the matching subtrees for the slot, as described in Section II-C. We use the Stanford Parser [13] to generate syntactic parse trees for each sentence, and we use the Tregex tool [12] to locate subtrees within a parse tree.

IV. Experimental Results

To evaluate our approach, we have used our system to generate SystemVerilog assertions from natural language property statements created for the AMBA AXI 3 bus protocol [1] developed by ARM Inc. Our benchmark set consists of a set natural language assertion statements taken from the AMBA Protocol Checker [2] which presents a set of assertions which can be used to validate an AMBA implementation. Our current approach comprehends assertion statements which directly constrain signals and registers defined in the protocol. For this reason, we evaluate our system with the subset of 81 assertion statements taken from [2] which directly refer to system signals and registers. Our code is implemented in Java and all results were executed using a 1.6 GHz Intel i5 processor with 8Gb RAM. We use the Stanford Parser [10] to generate parse trees and we use the Tregex tool [12] to locate subtrees within parse trees.

Out of the set of 81 natural language statements, our system correctly generated SystemVerilog assertions for 71, resulting in 87.6% correctness. The total CPU time required to process all 81 assertions is 0.57 seconds.

V. Conclusions

We present an approach to generate formal logic assertions directly from English text found in hardware specifications. The approach allows designers to describe behavioral constraints using the language which they are most comfortable with, natural language. By raising the abstraction level of the description, this approach saves designer time in generating assertions, and designer time in debugging errors in complex assertions.

We have evaluated the approach with text from a real hardware specification and found that our technique is effective for a range of English used in specifications. The approach also has some limitations in terms of the English which it can process, such as references to abstract objects. We have identified several of these limitations through testing and we intend to address them in future work.

References

[1] ARM Ltd. *AMBA AXI Protocol Specification, v1.0*, 2003.

[2] ARM Ltd. *AMBA 3 AXI Protocol Checker User Guide*, 2009.

[3] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisting requirement formalization by means of natural language translation. *Formal Methods in System Design*, 4(3), 1994.

[4] Harry Foster. *Applied Assertion-Based Verification:An Industry Perspective*. Now Foundations and Trends, 2009.

[5] C. B. Harris and Ian G. Harris. Glast: Learning formal grammars to translate natural language specifications into hardware assertions. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016.

[6] Ian G. Harris. Capturing assertions from natural language descriptions. In *Workshop on Natural Language Analysis in Software Engineering (NaturaLiSE)*, May 2013.

[7] James Higginbotham. *English is Not a Context-Free Language*, pages 335–348. Springer Netherlands, 1987.

[8] Alexander Holt and Ewan 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*, 1999.

[9] O. Keszocze, M. Soeken, E. Kuksa, and R. Drechsler. Lips: An IDE for model driven engineering based on natural language processing. In *Natural Language Analysis in Software Engineering (NaturaLiSE), 2013 1st International Workshop on*, May 2013.

[10] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, 2003.

[11] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[12] R. Levy and G. Andrew. Tregex and tsurgeon: tools for querying and manipulating tree data structures. In *International Conference on Language Resources and Evaluation*, 2006.

[13] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[14] Wolfgang Mueller, Alexander Bol, Alexander Krupp, and Ola Lundkvist. Generation of executable testbenches from natural language requirement specifications for embedded real-time systems. In Mike Hinchey, Bernd Kleinjohann, Lisa Kleinjohann, Peter A. Lindsay, Franz J. Rammig, Jon Timmis, and Marilyn Wolf, editors, *Distributed, Parallel and Biologically Inspired Systems*, volume 329 of *IFIP Advances in Information and Communication Technology*. Springer Berlin Heidelberg, 2010.

[15] Chris Spear and Gref Tumbush. *SystemVerilog for Verification*. Springer, 2012.