UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Generating Formal Verification Properties from Natural Language Hardware Specifications

Permalink
https://escholarship.org/uc/item/11d7k48g

Author
Harris, Christopher Bryant

Publication Date
2015

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Generating Formal Verification Properties from Natural Language Hardware Specifications

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Christopher Bryant Harris

Dissertation Committee:
Professor Ian G. Harris, Chair
Professor Elaheh Bozorgzadeh
Professor Chen-Yu Phillip Sheu

2015

© 2015 Christopher Bryant Harris

DEDICATION

To my family, because no one succeeds alone.

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF TABLES **vi**

ACKNOWLEDGMENTS **vii**

CURRICULUM VITAE **viii**

ABSTRACT OF THE DISSERTATION **xii**

1 Introduction **1**
- 1.1 The Verification Problem 2
- 1.2 Dissertation Objective and Overview 4
- 1.3 Related Work 6
  - 1.3.1 Software Artifacts from Natural Language 6
  - 1.3.2 Hardware Artifacts from Natural Language 8
  - 1.3.3 Hardware Assertion Generation 9

2 A Brief Overview of Natural Language Processing **10**
- 2.1 Language and Formal Grammars 10
  - 2.1.1 Context Free Grammars 11
- 2.2 Basic NLP Analysis 12
  - 2.2.1 Part-of-Speech Tagging 12
  - 2.2.2 Syntactic Parsing 13
- 2.3 NLP Metrics and Evaluation 14

3 Template Based Assertion Translation **16**
- 3.1 Dependency Grammars 17
  - 3.1.1 Dependency Graphs 18
  - 3.1.2 Dependency Based Parsing 19
- 3.2 Information Extraction for Very Small Databases 20
- 3.3 Algorithm Description 22
  - 3.3.1 Abstraction Based Classification 23
  - 3.3.2 Similarity Based Sentence Clustering 23
  - 3.3.3 Assertion Generation 28

3.4 Experimental Results and Analysis . . . 29
3.4.1 Results . . . 29
3.4.2 Discussion . . . 34

4 Grammar Based Property Translation **36**
4.1 Semantic Parsing with Context Free Grammars . . . 37
4.2 Attribute Grammars . . . 38
4.3 Methodology . . . 41
4.3.1 System Overview . . . 41
4.3.2 Translation Engine . . . 42
4.3.3 CTL Property Equivalence . . . 46
4.4 Experimental Results and Analysis . . . 46
4.4.1 CTL Property Generation . . . 47
4.4.2 Dealing with Ambiguity . . . 51

5 Assertion Generation Using Learned Formal Grammars **53**
5.1 Grammatical Inference . . . 54
5.2 Algorithm Design . . . 55
5.2.1 Overview . . . 56
5.2.2 Primary Operators . . . 58
5.3 Experimental Results and Analysis . . . 64

6 Conclusions **66**
6.1 Criticisms . . . 66
6.2 Contributions . . . 68
6.2.1 An Algorithm for SVA Generation Using Templates . . . 68
6.2.2 A Grammar Based Translation System for Temporal Logic Properties 69
6.2.3 A Learning Algorithm to Capture the Customized Language of a Specification . . . 69
6.3 Future Work . . . 70
6.3.1 Increasing Quantity . . . 70
6.3.2 Increasing Quality . . . 71
6.3.3 From Formal Requirements to Natural Language . . . 71
6.4 Final Thoughts . . . 72

Bibliography **73**

LIST OF FIGURES

Page

1.1 Typical Simulation Testbench 2

2.1 Example Context Free Grammar 12
2.2 Part-of-Speech Tagging 13
2.3 Example Parse Tree 14

3.1 Example Dependency Relations 18
3.2 Example Typed Dependency Graph 18
3.3 Canonical Dependency Representation for Two Different Sentences 20
3.4 Rewriting Passive Voice as Active Voice 25
3.5 Two Typed Dependency Graphs 26
3.6 Sample Representative Dependency Graph 26
3.7 Translation Set of Low Abstraction NLAs 32
3.8 Cluster Representative Dependency Graph 33

4.1 Example Semantic Parse Tree 38
4.2 Attribute Evaluation Example 41
4.3 Translation System Block Diagram 42

5.1 Grammar Induction Algorithm 55

LIST OF TABLES

Page

1.1 Sample Hardware Bug Mitigation Costs 2

3.1 NLA Abstraction Level Classifier 31
3.2 SystemVerilog Assertion Templates 31

4.1 Unparseable NLAs 47
4.2 Parseable NLAs with CTL Matching Benchmark 48
4.3 Parseable NLAs with CTL not Matching Benchmark 49

5.1 Example Learning Set 57
5.2 Initial Grammar Creation 58
5.3 Merge Operator 59
5.4 Chunk Operator 60
5.5 Augment Operator 62
5.6 Selected Automatically Generated SystemVerilog Assertions 65

ACKNOWLEDGMENTS

"The strongest steel is forged by the fires of hell. It is pounded and struck repeatedly before it's plunged back into the molten fire. The fire gives it power and flexibility, and the blows give it strength. Those two things make the metal pliable and able to withstand every battle it's called upon to fight."

– Sherrilyn Kenyon, The Dark-Hunters, Vol. 1

I would like to thank my Ph.D. advisor Dr. Ian Harris for his patience and his guidance, and for teaching me the things about being a good researcher which they don't write down in books. I would also like to thank Robin Jeffers, Director of Undergraduate Student Affairs in the Henry Samueli School of Engineering, for her support and for the opportunity to contribute to the *Center for Opportunities and Diversity in Engineering* (CODE).

Finally, I would like to thank unnamed members of the UC Irvine Department of Electrical Engineering and Computer Science, and the UC Irvine Graduate Division who helped to forge me into the strongest of steel. If ever you read this, you will know who you are. Remember me... and know that I have withstood every battle, as I will withstand the battles yet to come.

CURRICULUM VITAE

Christopher Bryant Harris

EDUCATION

Doctor of Philosophy in Electrical and Computer Engineering **2015**
University of California *Irvine, CA*

Master of Science in Electrical Engineering **2001**
University of Notre Dame *Notre Dame, IN*

Bachelor of Science in Computer Engineering **1998**
University of Alabama in Huntsville *Huntsville, AL*

Bachelor of Science in Applied Mathematics **1998**
Oakwood University *Huntsville, AL*

RESEARCH EXPERIENCE

Graduate Student Researcher **2007–2015**
University of California, Irvine *Irvine, CA*

Graduate Student Researcher **2000–2001**
University of Notre Dame *Notre Dame, IN*

Staff Physicist Intern **1998–1999**
Lawrence Livermore National Laboratory *Livermore, CA*

Undergraduate Research Intern **1995**
Virginia Tech *Blacksburg, VA*

TEACHING EXPERIENCE

Adjunct Instructor **2014–2015**
Northwest Arkansas Community College *Bentonville, AR*

Teaching Assistant **2006, 2008**
University of California *Irvine, CA*

PROFESSIONAL EXPERIENCE

Emulation Engineering Intern **2010**
Intel Massachusetts, *Hudson, MA*

Verification Engineering Intern **2009**
Intel Corporation *Chandler, AZ*

Application Engineer **2003–2004**
Micro/Sys, Inc. *Montrose, CA*

Member of Technical Staff **2001–2002**
TRW, Inc. *Redondo Beach, CA*

Software Engineering Intern **1996, 1997**
BDM International *Huntsville, AL*

REFEREED CONFERENCE PUBLICATIONS

C. B. Harris and I. G. Harris, "Automatic Generation of Hardware Assertions from Natural Language using Learned Attribute Grammars," in *Under review*.

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.

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.

REFEREED WORKSHOP PUBLICATIONS

C. B. Harris and I. Harris, "Learning Grammars for Assertion Creation from Natural Language," in *Design Automation for Understanding Hardware Designs (DUHDe), 2015 Workshop on*, March 2015.

C. B. Harris and I. Harris, "Using Natural Language Documentation in the Formal Verification of Hardware Designs," in *Design Automation for Understanding Hardware Designs (DUHDe), 2014 Workshop on*, March 2014.

TECHNICAL REPORTS

C. B. Harris and I. Harris, "Using Natural Language Documentation in the Formal Verification of Hardware Designs," *TR 14-11*, Center for Embedded and Cyber-physical Systems, UC Irvine, November 2014.

SHORT PAPERS AND POSTERS

C. B. Harris and I. Harris, "Learning Grammars for Generating Hardware Assertions from Natural Language," Poster presentation, December 2014, presented at *IEEE/ACM International Symposium on Microarchitecture (MICRO) 2014 Career Workshop for Women and Minorities in Computer Architecture (CWWMCA).*

C. B. Harris and I. Harris, "Automatic Generation of Formal Verification Properties from English Language," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2014 Workshop on Supporting Diversity in Systems Research (Diversity '14)*, October 2014.

M. Soeken, **C. B. Harris**, I. Harris, N. Abdessaied, and R. Drechsler, "Automating the Translation of Assertions Using Natural Language Processing Techniques," Poster presentation, June 2014, presented at *Design Automation Conference (DAC), 2014 51st ACM / EDAC / IEEE Work-in-Progress Session.*

C. B. Harris and I. Harris, "Generating CTL Properties from Natural Language Documentation," Poster presentation, May 2014, presented at *Design Automation Conference (DAC) 2014 CRAW/CDC Discipline Specific Workshop on Diversity in Design Automation.*

C. B. Harris and I. Harris, "Using Natural Language Documentation in the Formal Verification of Hardware Designs," Poster presentation, March 2014, presented at *2014 Academic and Research Leadership Symposium (ARLS).*

PROFESSIONAL SERVICE

2014 Panelist, "Addressing Challenges in Fostering Diversity in Design Automation," May 2014, CRA-W/CDC Discipline Specific Workshop on Diversity in Design Automation, May 31-June 1, San Francisco, CA.

2009 Discussant, "Student Recommendations for Increasing Participation in Science and Engineering," April 2009, Richard Tapia Celebration of Diversity in Computing (TAPIA 2009), April 1-4, Portland, OR.

2008 Committee Member, Broader Engagement. High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, November 15-21, Austin, TX.

2008 Panel Chair, "Broader Engagement Program: Grad School - Front to Back," November 2008, High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, November 15-21, Austin, TX.

SELECTED HONORS AND AWARDS

Ford Foundation Pre-Doctoral Fellowship, Honorable Mention	**2007**
University of California Regents Fellowship	**2006**
GEM Consortium Fellowship	**1998**
University of Notre Dame Arthur J. Schmidt Presidential Fellowship	**1998**
UNCF / BDM Information Technology Scholarship	**1996**

Tau Beta Pi Engineering Honor Society, Member
Eta Kappa Nu Electrical Engineering Honor Society, Member
Alpha Chi National College Honor Scholarship Society, Member

ABSTRACT OF THE DISSERTATION

Generating Formal Verification Properties from Natural Language Hardware Specifications

By

Christopher Bryant Harris

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2015

Professor Ian G. Harris, Chair

Verification of modern digital systems can consume up to 70% of the design cycle. Verification engineers must create formal properties which reflect correct operation from design specifications or other documents. This process of creating formal correctness properties from textual descriptions is a laborious task which is difficult to automate.

In this work I investigate the creation of formal verification properties from textual descriptions written in natural language. I present two approaches that utilize natural language processing (NLP) and machine learning techniques for the automatic generation of verification properties. In the first approach a set of correctness properties expressed in natural language, called *natural language assertions* (NLAs), is divided into subsets of structurally similar sentences. A generalized formal property template for each subset is used to provide a mapping from each sentence to a well specified verification property. Experimental results show that this methodology reduces the number of formal properties which must be manually created by up to an order of magnitude.

In the second approach I create a custom attributed formal grammar which captures the English semantics of a temporal tree logic. A translation system is implemented which uses this attribute grammar to perform a semantic parsing of NLAs. Attributes for each grammatical production in the parse tree are then used to build a fully specified formal

property for each NLA. Experimental results show that valid formal properties are generated from English NLAs in over 90% of test cases.

In evaluating the translation system it was observed that translation rates for NLAs are strongly dependent on the quality of the formal grammar used. High translation rates require a finely tuned custom grammar. To facilitate the creation of such a grammar I propose a new learning algorithm which automatically generates custom attribute grammars from a small set of NLAs and formal properties. This machine generated grammar is used in the NLA translation system to create formal properties from NLAs taken from two design documents for designs in the same product family. Experimental results show that the learned attribute grammar is of sufficient quality to successfully translate up to 88% of NLAs.

Chapter 1

Introduction

As noted by the ubiquitous Moore's Law, the number of devices in digital designs has increased exponentially for the past five decades. A less discussed side-effect is that as the number of devices has increased, the complexity of digital systems has increased dramatically as well. The discipline of *Electronic Design Automation* (EDA) was created to allow the creation of ever larger and more capable digital systems in the face of this rising complexity.

However, verifying the correct operation of these increasingly complex systems is no easy task. To address this challenge the area of functional verification has arisen as a sub-discipline in EDA. Functional verification is the art and science of assuring that the operation of a digital system after implementation is consistent with that outlined in the original description of the design. This design description is often captured in the form of a written system specification.

Functional verification has become an important part of the design cycle. Verification has become so important, in fact, that it has come to dominate the design cycle. It has been consistently reported in the literature that up to 70% of the design cycle for a modern Systems on Chip (SoC) is spent on verification activities [1]. Further, the cost of verification

Table 1.1: Sample Hardware Bug Mitigation Costs

Design Phase	**Cost Estimate**	**Remedy**
Before RTL hand off	$10,000	Re-design of RTL
Before tape out	$100,000	Re-layout (including above remedy)
After engineering prototype	$1,000,000	Silicon respin (including above remedies)
After mass production	$10,000,000+	Product recall (including above remedies)

failures is high. Table 1.1 presents data from the 2004 Design Automation Conference [2] showing the cost to correct a design bug increasing exponentially with time. In short, the later in the design process a bug is detected, the more it costs to fix. In 2011, one major semiconductor manufacturer accrued costs approaching three-quarters of a billion dollars due to a verification miss and the resulting product recall [3]. Driven by these economic realities, a large amount of effort is expended in verifying the correct operation of a digital design as early as possible in the design cycle.

1.1 The Verification Problem

Why does the verification of digital systems take so long? There are several contributing factors. One contributing factor is that the verification environment is often much larger than the design itself.

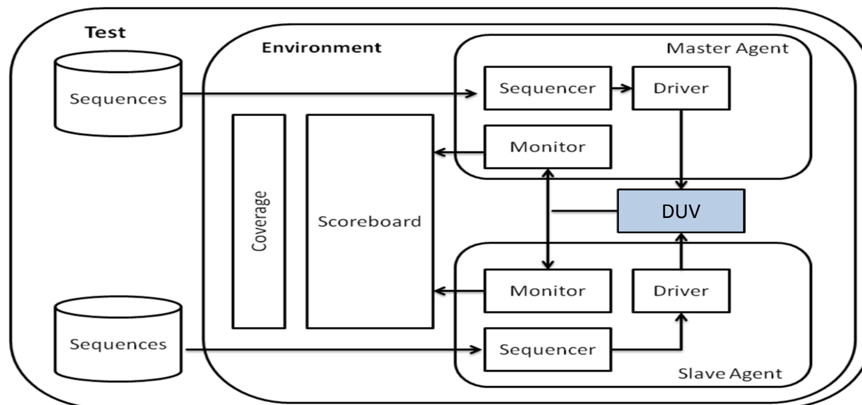


Figure 1.1: Typical Simulation Testbench

A block diagram for a typical simulation based verification testbench is shown in Fig. 1.1. The design under verification (DUV) is only a small part of the overall verification system. Because the verification infrastructure can be so much larger than the DUV, the amount of engineering effort necessary to create, debug, and maintain this infrastructure is increased proportionally. Efforts to amortize this cost across multiple designs exist in the form of reusable verification components and standardized methodologies such as the e Reuse Methodology [4], Synopsys' Verification Methodology Manual (VMM) [5], and the more recent Open Verification Methodology (OVM) [6] and Universal Verification Methodology (UVM) [7].

Another factor contributing to the verification problem is its scale. The most common approach to functional verification is simulation. However, exhaustively simulating a digital system for even a modest input size can be computationally prohibitive. As a simple example, a 32-bit comparator requires 2^{64} unique test vectors to exhaustively verify. On a one million cycle per second simulator it would take over a half million years to verify even this simple design [8]. As a result, modern verification practice generally simulates designs using only a fraction of the available test patterns. Determining which test patterns to apply (some are redundant), and how many must be successfully applied before obtaining a high level of confidence in design correctness is at the heart of modern verification practice.

An alternative to the challenge of dealing with the huge search space of simulation based verification is to apply formal verification. Formal verification uses mathematics or other techniques to formally prove or disprove a set of statements about the correctness of a design implementation for all reachable states of the design. These statements of correctness, called properties, are generally categorized as either liveness or safety properties.

Liveness properties are formal statements which state that *something good eventually happens*. Liveness properties often express temporal relationships. An example of a liveness property is the statement "If REQUEST is asserted, then GRANT must eventually be as-

serted". Because liveness properties utilize the concept of *eventually*, they cannot truly be satisfied in a finite time. Instead, they express the absence of infinite loops in a the execution of a system. As a result, in practice an approximation of the liveness property is used.

The second class of correctness properties are known as safety properties. Safety properties, also called invariants, declare that *something bad does not happen*. An example of a safety property is the statement "If READ is asserted then OUTPUT_ENABLE must also be asserted". Although the vast majority of correctness properties in use today are safety properties, the two classifications are not mutually exclusive. A single property can contain elements of both liveness and safety. The concepts of liveness and safety were first described in [9].

The most common technique used to evaluate correctness properties in formal verification is model checking. Additional techniques include automated theorem proving, Finite State Machine (FSM) based approaches such as language containment and state machine equivalence, and Assertion Based Verification (ABV). The interested reader is directed to [10] and [11] for a summary of popular techniques. Despite the diversity of formal verification techniques, all of these approaches require the generation of well defined correctness properties (a formal specification) from a high level description of the design. This high level description often takes the form of a textual description of the design details and requirements.

1.2 Dissertation Objective and Overview

It is important to take a moment to distinguish hardware verification from hardware validation. Hardware verification is strictly the determination of whether a design meets its specification. However, validation is the determination if a design is fit for a specific purpose. As is said in the industry, "Verification shows if you built the thing right; validation shows

if you built the right thing". This work is firmly in the area of functional verification. In particular, this dissertation investigates the automatic generation of formal specifications (correctness properties) for the verification of digital systems from natural English language text. The specific goals for this work are as follows:

1. To define a natural language based approach to translate English language requirements to formal specifications through the use of property templates.

2. To develop a complementary translation mechanism using a custom formal grammar to provide a mapping from natural language words and phrases to formal specification clauses, sub-clauses, and properties.

3. To combine template based and formal grammar based translation into a single translation system, capable of learning a customized formal grammar suitable for translation from a small set of natural language requirements and property templates.

In chapter 2 we cover background in the Natural Language Processing (NLP) techniques necessary to analyze textual design descriptions. In chapter 3 we present methodology for generating SystemVerilog assertions from natural language requirements using pre-determined property templates. A small set of SystemVerilog property templates, created by the user, can be used to generate a much larger set of formal specification properties for natural language requirements with similar sentence structures. In chapter 4 a formal grammar based translation method is explored. It utilizes a custom attributed grammar created to generate Computation Tree Logic (CTL) properties from text describing the verification requirements of a Peripheral Component Interconnect (PCI) local bus implementation. In chapter 5, the methodologies from the chapters 3 and 4 are combined and extended to create a new translation system. This translation system combines attribute grammar based property generation with a template based approach. In this new approach, the formal attribute grammar used in translation is induced from a small set of natural language requirements and SystemVerilog

assertion templates. Finally, chapter 6 summarizes the contributions of this research, briefly explores some future lines of inquiry, and presents some final thoughts. The remainder of this chapter will endeavor to put this work in context by presenting an overview of related work.

1.3 Related Work

Once upon a time the design of hardware and software systems were completely separate activities. Hardware engineers built a system and "then threw it over the wall" to the software developers. Programmers would create extensive software systems and "toss them over" to the hardware engineers. Then the argument would ensue about why nothing worked. As computing systems have become more complex we have begun to embrace the idea of hardware / software systems and the concept of co-design. As a consequence, ideas and techniques from the software design world have crept into hardware design and vice-versa. This trend is observed in the first area of related work, the generation of software artifacts from natural language.

1.3.1 Software Artifacts from Natural Language

It was over two decades ago that the software community first began to investigate the creation of formal structures from informal English. Early work in [12] makes the case that English nouns can serve as analogs to abstract data types with the goal of generating computer programs from textual descriptions. While Abbott was able to manually generate a formal Ada program using data types inferred in English text, he states that "a computer program that can take an informal strategy expressed in English and transform it automatically to an executable program is still a long way off".

Although generating executable code form natural language descriptions is very difficult, there is a large body of work in generating simpler software artifacts. Work in [13] uses natural language in use case documents to generate high level sequence diagrams. There is also a number of studies where nonspecific natural language descriptions are used to generate abstract, high level system models. Examples include work in [14–18] where Unified Modeling Language (UML) models or model based specifications were generated from text based summaries of system requirements .

One of the challenges in generating formal software structures from unconstrained text is the scale of the problem. Natural language is inherently ambiguous. The problem of translating English into a more formal language is only tractable if the natural language is somehow constrained. An attempt at this was made in [19] where a natural language interface was used to ask a non-expert user questions regarding desired software requirements. This attempt to constrain the input text by asking guided questions allowed software model templates to be generated from user input. A similar approach was used in the *Propel* system [20]. This system walked the user through a selection of templates in order to formalize a set of requirements. The result was a set of requirements in a English (whose language was constrained by the system) and a finite state machine (FSM) representation of each requirement.

A consistent question regarding these natural language translation systems is the usefulness of the output. Many of the systems described thus far have a variety of output formats: diagrams, charts, etc. Very few have output formats which are immediately useful in the design and verification flow. Work in [21] uses a domain specific database to generate software requirements from unstructured text. However, the requirements are presented as graphical data and relationship diagrams. Research presented in [22] uses a fuzzy rule based approach to analyze textual descriptions and generate concise sentences describing system requirements. In comparison, the authors of [20] planned to include an option to generate Computation Tree Logic (CTL) and Linear Time Logic (LTL) from their system, but unfor-

tunately it appears as if these options were never implemented. Work in [23] takes software security requirements written in natural language and expresses them in a lambda calculus. The ability to generate useful artifacts form natural language descriptions is often dependent on the specificity of the textual input. This concept will be revisited in subsequent chapters.

1.3.2 Hardware Artifacts from Natural Language

As an outgrowth of explorations into generating software from natural language, there has also been work looking at the generation of hardware from natural language descriptions. Researchers have generated partial hardware designs from natural language specifications [24, 25] by identifying a set of concepts expressed, together with a textual pattern for each concept. Any sentence which matches a textual pattern can be mapped to an intermediate graph representation, which contains information about data flow, timing, structure, and physical properties. The approach taken in [26] defines a grammar to parse natural language expressions, and generates VHDL snippets.

While the above work addresses digital design, as has been noted earlier the bottleneck in the modern design cycle lies in verification. To address the verification problem work in [27] proposes a natural language interface to query circuit simulation results. Unfortunately, the system was never fully realized. In [28] a conversion from a natural language specification to Action CTL (ACTL) is proposed. While intended to perform automatic translation the tool instead requires user input in order to resolve language ambiguity. Parallel work in [29] translates parse trees derived from natural language into an intermediate representation defined by Discourse Representation Theory [30]. Research presented in [31] features a controlled subset of English language which can be used to describe CTL properties, together with a context-free grammar to recognize the language subset. However, while the technique can recognize a CTL property described in natural language no algorithm is given to generate

CTL expressions from their proposed English language subset. While important contributions, we again observe that none of these approaches directly generate artifacts which are immediately useful in hardware verification or synthesis.

1.3.3 Hardware Assertion Generation

In recent years, assertions have emerged as an important tool to address the verification problem. They have been used in conjunction with formal verification techniques to address structures which are hard to verify using a traditional simulation based approach. As such, research aimed at automatically generating assertions has gained in prominence.

Researchers propose in [32] the generation of hardware assertions from the analysis of register transfer level (RTL) code. Assertion generation from simulation traces is proposed in [33–35]. Reference [36] presents a promising methodology that uses a failing assertion, counterexample, and a mutation model to produce alternative properties that are verified against the design and serve to make possible corrections as they provide insight into the design behavior and the failing assertion. The common characteristic of all of these approaches is that they attempt to generate assertions from an *existing implementation* of the design as opposed to a design specification. This is a subtle distinction, but the difference is important. Deriving assertions from an implementation allows you to verify the design that you *have*. However, deriving assertions from a specification allows you to verify the design that you *want*. In the subsequent chapters of this work we will explore methodologies to generate hardware assertions and other verification artifacts from natural language text describing the design that the user wants.

Chapter 2

A Brief Overview of Natural Language Processing

Natural language processing (NLP) is, on the whole, a discipline concerned with the use of computers to understand, transform, or generate text in human usable languages such as English. Processing usually takes place on a collection of textual data known as a *corpus*. Common tasks in the NLP space include parsing, part-of-speech tagging, information extraction, named entity recognition, and natural language understanding. In this section we will introduce a few of the NLP techniques which are relevant to this work.

2.1 Language and Formal Grammars

Conceptually speaking, a *language* is a set of strings, called sentences, over a finite set of symbols. These symbols are called *words* and form the *vocabulary* of the language. A *formal grammar* is the finite set of production rules which constrains the language to the specified set of strings from the set of all possible sequences of words in the vocabulary. In other

words, a formal grammar is a set of rules which tells us which strings made using words in the vocabulary are actually part of the set of sentences which make up the language.

More specifically, a formal grammar is defined as

$$G = (V_N, V_T, S, P)\,, \tag{2.1}$$

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 production rules, or simply productions (where the $*$ operator implies zero or more instances). The set of sentences which can be generated by the grammar G is the language $L(G)$.

2.1.1 Context Free Grammars

When a formal grammar is defined to have a single nonterminal symbol on the left side of each production rule it is known as a context-free grammar (CFG). These types of grammars are called context free because, given the production rule $A \rightarrow B$, the symbol B can be substituted whenever the symbol A is encountered. The context (symbols around A) does not effect the substitution. CFGs were developed in the 1950s by Chomsky [37] (and independently by Backus). Although natural language is not necessarily context free the use of CFGs is well accepted in NLP to adequately model syntax in practice.

$$V_T = \{\text{“a”}, \text{“the”}, \text{“dog”}, \text{“cat”}, \text{“chased”}, \text{“sat”}, \text{“on”}, \text{“in”}\} \tag{2.2}$$

$$V_N = \{\text{S, DET, N, V, P, NP, VP, PP}\} \tag{2.3}$$

```
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"
```

Figure 2.1: Example Context Free Grammar

Taken together, the sets defined in (2.2) and (2.3) combined with the set of productions shown in Figure 2.1 form the grammar G, an example of a simple CFG.

The productions in a CFG are examples of what is known as a *constituency relation*. Figure 2.1 shows how in productions a linguistic unit (a constituent) can be mapped to one or more other constituents. This constituency relation is reminiscent of the basic subject-predicate interpretation of sentence structure where sentences are divided in a binary manner into a noun phrase followed by a verb phrase. As a result, CFGs are sometimes referred to as phrase-structure grammars.

2.2 Basic NLP Analysis

2.2.1 Part-of-Speech Tagging

One of the fist stages of natural language processing is often to represent individual words in a sentence by distinct symbols or tokens. Part-of-Speech (POS) tags represent the part-of-speech, or grammatical category, of words in a sentence. One of the most commonly used sets of POS tags is part of the Penn Treebank Project [38, 39]. Tags for *nouns* (N), *verbs* (V), *determinants* (DET), and *adjectives* (JJ) are standard examples of POS tags.

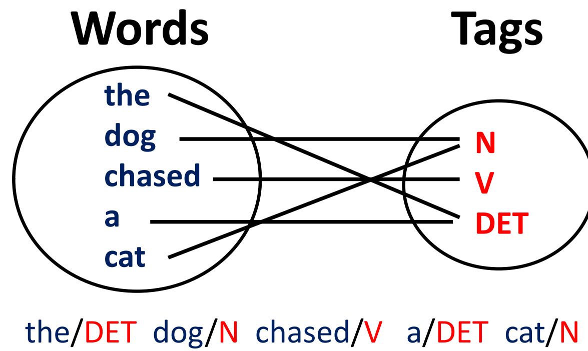


Figure 2.2: Part-of-Speech Tagging

POS tags can be used as symbols themselves, as in the previously defined grammar G, or they can simply be associated with the symbols representing words in a sentence. Consider example sentence 2.4:

$$\text{“The dog chased a cat.”} \tag{2.4}$$

Figure 2.2 shows this example sentence with part-of-speech tags specified for each word. POS tagging is especially useful in analyzing the grammatical structure of a sentence (syntactic parsing) and can reduce the ambiguity of a sentence by specifying how a word is used. However, tagging words can be useful in other types of analysis as well. For instance, words or phrases can be tagged with semantic categories in order to understand the meaning of a sentence. This is the basis of semantic parsing, which will be discussed in chapter 4.

2.2.2 Syntactic Parsing

Syntactic parsing is the grammatical analysis of the structure of a sentence according to rules defined by a formal grammar. A syntactic parsing of a sentence shows the relation of words or phrases to other sentence constituents. Results are generally displayed in a parse tree (sometimes called a constituency tree). Using the previously defined grammar G, we can now revisit sentence 2.4 and analyze its grammatical structure.

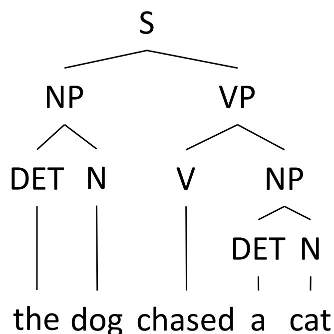


Figure 2.3: Example Parse Tree

Figure 2.3 shows the parse tree of the sentence 2.4. Each node in the tree represents a production from grammar G where the node closest to the root of the tree is the left side of the production and its child node(s) represent the right side of the production. Because of the use of POS tags as symbols in grammar G we can see in the parse tree that the sentence is divided into a noun phrase (NP) and a verb phrase (VP). The noun phrase consists of "the" and "dog" while the verb phrase is decomposed into the verb "chased" followed by the noun phrase "a cat".

2.3 NLP Metrics and Evaluation

A word must be said regarding the evaluation of the output for NLP systems. Because NLP comprises so many different distinct tasks, evaluation methods can vary widely. This work, however, focuses on generating verification artifacts from natural language data. In the verification space evaluation is often performed by comparing system output to a "gold standard". Where possible we utilize this approach as well. Where there is no "gold standard" available we borrow from the field of information retrieval and use the metrics of *accuracy*, *precision*, *recall*, and *F-measure*.

In binary classification problems accuracy gives the percentage of samples which were correctly classified. The accuracy A is defined as:

$$A = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}} \tag{2.5}$$

where **TP**, **TN**, **FP**, and **FN** denote *true positives*, *true negatives*, *false positives* , and *false negatives* respectively. The precision metric gives the percentage of positive predictions which were correctly classified and is given by equation 2.6.

$$P = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{2.6}$$

The recall metric gives the percentage of all positives which were correctly predicted, given by 2.7.

$$R = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{2.7}$$

The F-measure considers both the precision and the recall and is the harmonic mean of the two. The F-measure is given by formula 2.8.

$$F = \frac{2 \cdot P \cdot R}{P + R} \tag{2.8}$$

As we continue on to develop verification collateral from natural language data, these metrics will help us to evaluate the quality of our algorithms.

Chapter 3

Template Based Assertion Translation

In this chapter we will tackle the first objective noted in section 1.2, the translation of natural language requirements into formal properties using a template based mechanism. Although not all system requirements are expressed as text, digital system specifications contain detailed natural language descriptions of what a processor, chip, bus, or module is supposed to do. Many of these descriptions take the form of assertion-like properties. For instance, the sentence “If RESET is HIGH, ENABLE must be LOW” expresses one such invariant for correct operation. Sentences in a specification which express these types of correctness properties we call *Natural Language Assertions* (NLAs). Even though a specification contains substantial linguistic variation, sentences containing NLAs tend to exhibit grammatical similarities based on the type of property check the sentence is describing.

These NLAs can be automatically grouped into clusters of sentences of similar linguistic structure, where each NLA in the cluster can be described by an archetypical SystemVerilog Assertion (SVA). This archetypical assertion template can then be used to automatically generate a specific and correct assertion statement for each NLA in a cluster. Consider the following sentences:

$$\text{“REQUEST is only permitted to change from HIGH to LOW when ACKNOWLEDGE is HIGH”} \tag{3.1}$$

$$\text{“ACKNOWLEDGE is only permitted to change from LOW to HIGH when REQUEST is HIGH”} \tag{3.2}$$

Although the specifics differ, the NLAs in sentences 3.1 and 3.2 display a similar sentence structure and both describe an assertion check where a signal is only allowed to toggle from one specified state to another when a controlling signal is asserted.

$$\text{“}S1 \text{ is only permitted to change from } V1 \text{ to } V2 \text{ when } S2 \text{ is } V3\text{”} \tag{3.3}$$

$$\text{assert property(@(posedge clock) } (S2 \;!= V3) \;\rightarrow !((S1 == V1)\ (\#\#1\ S1 == V2))); \tag{3.4}$$

These sentences can be generalized to a sentence like that in 3.3, where $S1$ and $S2$ are the first and second signals in the sentence and $V1$, $V2$, and $V3$ are the first, second, and third signal values in the sentence respectively. The generalized SVA in 3.4 could then serve as a template for a correctness property for each of the original NLAs, shown in 3.5 and 3.6.

$$\text{assert property(@(posedge clock) } (ACK \;!= 1) \;\rightarrow !((REQ == 1)\ (\#\#1\ REQ == 0))); \tag{3.5}$$

$$\text{assert property(@(posedge clock) } (REQ \;!= 1) \;\rightarrow !((ACK == 0)\ (\#\#1\ ACK == 1))); \tag{3.6}$$

In order to facilitate the translation from English to SystemVerilog we must first develop the theory of dependency grammars, dependency based parsing, and information extraction at the individual sentence level. This is what we shall accomplish in the next two sections.

3.1 Dependency Grammars

Context free grammars utilize constituency relations to divide constituents (phrases) into smaller constituents (subphrases and individual words). Dependency grammars, on the

other hand, rely on the *dependency relation*. Dependency relations are one-to-one relations where one word is the *governor* or *head*, and the other word is the *dependent*. Recall example sentence 2.4, repeated here for convenience.

"The dog chased a cat." (3.7)

The dependency relations for this sentence are shown Figure 3.1.

det(dog, the)
nsubj(chased, dog)
det(cat, a)
dobj(chased, cat)

Figure 3.1: Example Dependency Relations

The five words in this sentence are related using three distinct dependencies. The words *dog* and *cat* each have an associated determinant, the words *the* and *a* respectively. The noun *dog* is the subject of the clause beginning with the verb *chased*, and the noun *cat* is the direct object of the clause beginning with the word *chased*.

3.1.1 Dependency Graphs

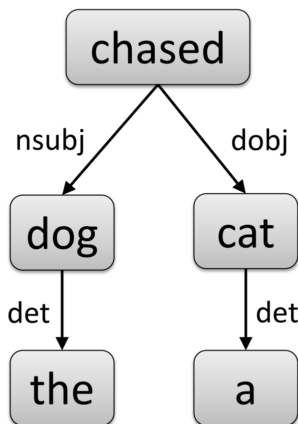


Figure 3.2: Example Typed Dependency Graph

These dependency relations can also be expressed in graphical form. Figure 3.2 shows the graphical representation of the dependencies from our example sentence. This graphical representation is called a *Typed Dependency Graph* (TDG). A TDG G can be formally defined as the 4-tuple

$$G = (V, E, r, s)\,, \tag{3.8}$$

where V is a set of vertices which represent the words in a sentence, and E is a set of directed edges which represent the dependencies between the words. Each edge $e = (g, d) \in E$ is between the governor g and the dependent d of the relation. Each edge $e \in E$ is also assigned a relation type $r(e)$ which is the type of dependency represented by the edge. Each vertex $v \in V$ is associated with a string $s(v)$ which is the word in the sentence represented by the set of vertices.

3.1.2 Dependency Based Parsing

A dependency based language parser takes a string in a specific language and generates the appropriate set of dependency relations between words as represented by a typed dependency graph. In order to represent the grammatical structure of NLAs for this investigation we use the Stanford typed dependency representation [40] which is generated automatically by the Stanford Natural Language Parser [41]. [1]

One important reason for using the Stanford dependency representation is its canonicity in the presence of a large degree of linguistic variation. The techniques used to generate the dependency representations are robust and can identify dependencies even when the grammatical structures of the sentence are reordered. In other words, there can exist two

[1] As of Version 3.5.2 (April, 2015), the Stanford parser switched its default output to use the Universal Dependency representation available at http://universaldependencies.github.io/docs/. Stanford Dependency representation is still supported via an optional switch.

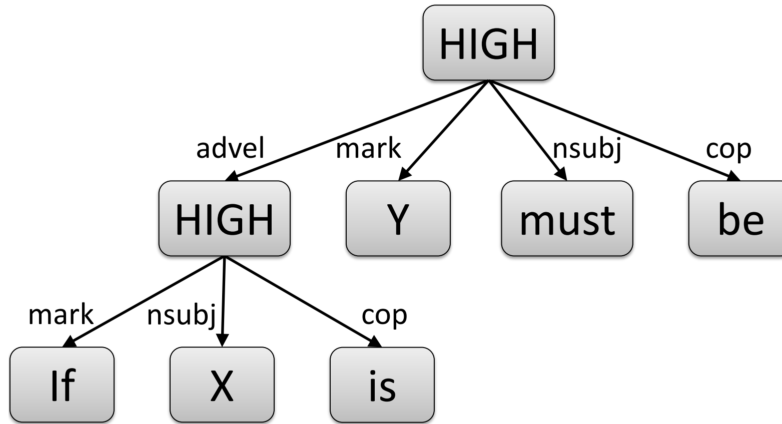


Figure 3.3: Canonical Dependency Representation for Two Different Sentences

sentences $S_1 \neq S_2$ such that the TDGs of the sentences are equivalent. An example can be seen in the representations of the following two sentences.

$$S_1 = \text{“If X is HIGH, Y must be HIGH”} \tag{3.9}$$

$$S_2 = \text{“Y must be HIGH if X is HIGH.”} \tag{3.10}$$

These two sentences have the same semantic meaning but the ordering of the phrases in sentence 3.9 is different than the ordering in 3.10. However, the dependency representations of these two sentences is the same, independent of the ordering of their constituents, as illustrated in Figure 3.3. The ability of the Stanford dependency representation to capture grammatical relations independent of ordering is extremely useful in processing a wide range of writing styles used by possibly different authors.

3.2 Information Extraction for Very Small Databases

In this chapter we also make unique use of database extraction techniques. In short, we treat each individual NLA as its own database. The idea is that for a given sentence a database is extracted that contains a variety of linguistic information about the sentence. We make

use of *triplestore* databases which are represented as sets of 3-tuples, where each 3-tuple represents the manner in which two entities relate to each other. Information extraction is then performed by applying a query to the database which captures the type of information that should be extracted. The specified information can then be extracted from the query result. Given the three sentences

"X must be HIGH", (3.11)

"Y must be LOW", and (3.12)

"X is not equal to Y", (3.13)

the requisite task is to extract signal names and their required values from sentences 3.11 and 3.12. Sentence 3.13 does not match and so is not considered. Dependency based parsing supported by the Stanford Parser is used to construct the database.

The database for sentence 3.11 contains the following triples:

```
<HIGH-4> aux   <must-2>
<HIGH-4> cop   <be-3>
<HIGH-4> nsubj <X-1>
<X-1>    word  "X"
<must-2> word  "must"
<be-3>   word  "be"
<HIGH-4> word  "HIGH"
...
```

Notice that for each word in the sentence a word item exists as an entity in the triplestore. This is required since sentences may contain a word more than once. The first three triples represent the typed dependencies. The latter triples represent word literals for each word.

While this data extraction task can also be accomplished with regular expressions, such an implementation is not as robust as our proposed solution. This is due to the canonicity of typed dependency representation as illustrated in the previous section. We chose the

SPARQL Protocol and RDF Query Language (SPARQL, [42]) in order to query the generated databases. SPARQL queries consist of triples as they are found in the database but allow the use of variables and additional constraints with respect to those variables.

In order to extract the signal/value pairs as shown in sentences 3.11 and 3.12 the following query is created and evaluated on each database that is generated for each sentence:

```
SELECT ?signal ?value WHERE {
  ?w4 aux ?w2. ?w4 cop ?w3. ?w4 nsubj ?w1.
  ?w2 word "must". ?w3 word "be".
  ?w1 word ?signal. ?w4 word ?value. }
```

Six variables are used in this query where only two of them are *global* (`?signal` and `?value`) and appear in the result set which is obtained after evaluating the query. The other three variables are *locally* used and represent the corresponding word items in this case. In the case of a matching query we can examine the results set to determine the name of the name of a signal and the value associated with it. This database can be used to extract this information from any sentence with a matching grammatical structure.

3.3 Algorithm Description

The proposed algorithm has three phases, which will be described in the following subsections.

3.3.1 Abstraction Based Classification

Natural language descriptions of requirements can be at a high level of abstraction or a low level of abstraction. Only descriptions at a low level of abstraction are suitable for direct translation. As such, a method is required to identify these low abstraction level requirements.

A good heuristic to determine the abstraction level of natural language assertions heavily depends on the writing style of the specification. As a result, a general heuristic cannot be provided. Instead, we propose a classification method that only requires one SPARQL query as input by the designer. After a brief inspection of some assertions in the specification the designer can determine common characteristics of low level assertions, e.g. the use of some particular words or a special formatting. The observations are described in terms of a SPARQL query which is applied to each assertion. An assertion is classified low level, if and only if the query matches the assertion. In addition, we only consider assertions which can be expressed in a single sentence.

3.3.2 Similarity Based Sentence Clustering

The clustering step aims at partitioning a set of NLAs based on grammatical similarity (as represented by the TDG) while generating a minimal number of clusters. We discovered that sometimes sentences are similar when read by an informed expert but were misclassified due to a lack of support for discipline specific semantics in our toolset. Therefore we developed preprocessing and postprocessing steps to aid in reducing classification errors. The preprocessing step occurs before a TDG is generated from a sentence, while postprocessing is applied to a resulting TDG.

3.3.2.1 Sentence Preprocessing

The preprocessing step aims at modifying words that might be misinterpreted by the NLP parser. In the specifications we have considered, often binary literals, e.g. `3'b101`, or signal logic levels, e.g. `LOW`, are directly been used. The NLP parser regards them as normal words. As a result, the literal is split into two words separated by an apostrophe and the signal logic level is sometimes detected as an adjective. In order to avoid misinterpretation we applied preprocessing rules that remove apostrophes in binary literals and insert double quotes around signal logic levels in order to enforce correct POS tagging.

3.3.2.2 Sentence Postprocessing

After a TDG is generated for each sentence representing a NLA, a postproccessing step based on graph transformation is applied. This postprocess step allows for removing semantically irrelevant words and transforming structures that are semantically equivalent. In our implementation we apply a couple of different postprocessing rules.

One rule removes prepositions as well as auxiliary verbs such as the word *must*. For instance, sentences 3.14 and 3.15 are treated in an equivalent manner.

"The signal remains LOW" (3.14)

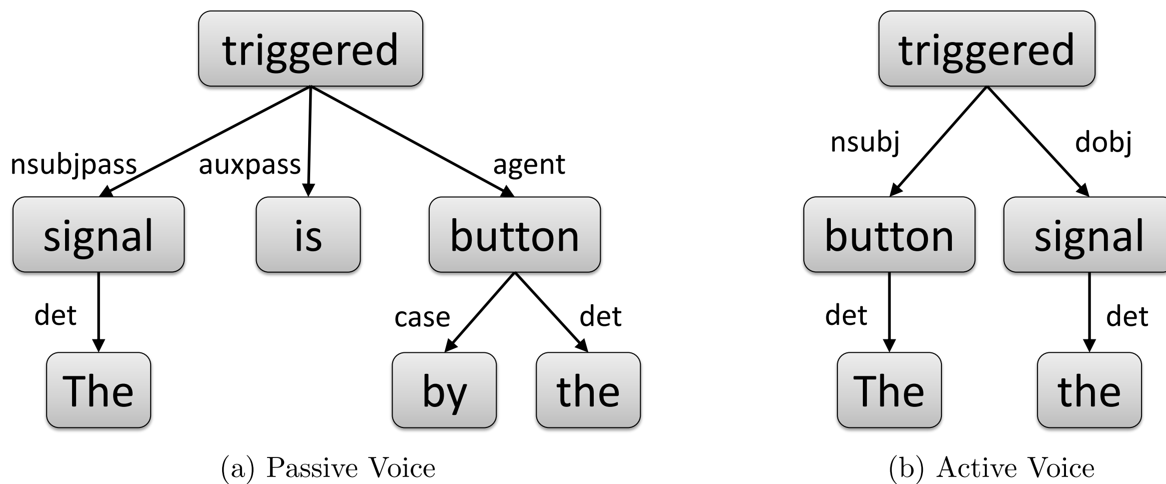
"The signal must remain LOW" (3.15)

Performing this step at the graph level is advantageous compared to sentence level because the word's dependencies can be taken into account. As an example, an auxiliary verb or preposition is only removed when no other words depend on it which is equivalent to being a sink vertex in the TDG.

A more complex postprocessing rule rewrites sentences in passive voice into active voice. Compare the sentence in 3.16 which is written in passive voice to the active voice representation in 3.17.

$$\text{“The signal is triggered by the button”} \tag{3.16}$$

$$\text{“The button triggered the signal”} \tag{3.17}$$



(a) Passive Voice (b) Active Voice

Figure 3.4: Rewriting Passive Voice as Active Voice

The TDGs for these two sentences are illustrated in Figure 3.4. After the preposition and auxiliary verb in Figure 3.4a are removed, the edges *nsubj* and *dobj* of the graph representing the active voice must be correlated with the edges *agent* and *nsubjpass* of the graph representing the passive voice, respectively. These postprocessing graph transformations allow a greater correlation between similar NLAs in subsequent steps and thus allow a better partitioning to be achieved.

3.3.2.3 Representative Dependency Graphs

In order to determine similar sentences we make use of a *Representative Dependency Graph* (RDG), which is a generalized description of a set of TDGs. This generalization is performed by al-

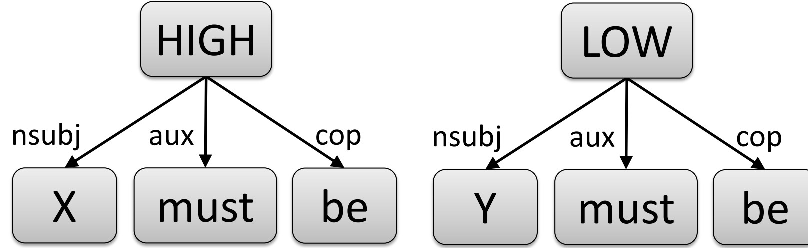


Figure 3.5: Two Typed Dependency Graphs

lowing a subset of vertices to represent variables which can represent any word. An RDG is a TDG $G = (V, E, r, s)$ whose set of vertices V is partitioned into two sets, W, which represents words in a sentence, and A, which are variables and may represent any string.

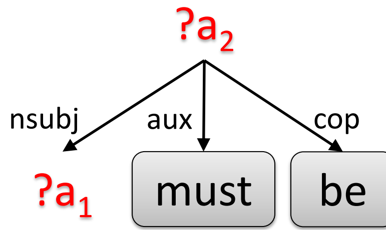


Figure 3.6: Sample Representative Dependency Graph

Figure 3.5 shows the TDGs of the sentences 3.11 and 3.12. Figure 3.6 shows an RDG which describes the graphs of both sentences. The RDG contains two variables $?a_1, ?a_2 \in A$ which capture the semantic values for the signal name in the sentence and signal value, respectively.

Simply put, two sentences S_1 and S_2 are *similar* if they have the same TDG when disregarding the words associated with the vertices. If we designate the function $tdg(s)$ to return the TDG for a sentence s then given the TDGs for two sentences S_1 and S_2

$G_1 = (V_1, E_1, r_1, s_1) = tdg(S_1)$

and

$G_2 = (V_2, E_2, r_2, s_2) = tdg(S_2)$,

we can determine that

$G_1 \simeq G_2$.

In other words, there exists a graph isomorphism $f : V_1 \to V_2$ where additionally the dependency relations need to be preserved, i.e., for each $e_1 = (u, v) \in E_1$ the property $r(e_1) = r(e_2)$ holds, where $e_2 = (f(u), f(v)) \in E_2$.

3.3.2.4 Sentence Partitioning and RDG Generation

For the partitioning and construction of the RDG we make use of a data structure

$$\texttt{CLUSTER}(S_1, \ldots, S_n, w_1, \ldots, w_m)$$

that represents one subset of the partition and stores a set of similar sentences $S_1, \ldots, S_n$ and a set of words $w_1, \ldots, w_m$ that are common in all sentences. A word w is common if for each pair of similar sentences S_i, S_j, their TDGs $G_i = (V_i, E_i, r_i, s_i), G_j = (V_j, E_j, r_j, s_j)$ and the induced isomorphism f as defined above, there exist one $v \in V_i$ such that $w = s_i(v) = s_j(f(v))$.

The low level assertions are partitioned in a sequential manner by checking for each processed assertion S whether there already exists a cluster such that a graph isomorphism can be determined. If this is the case, the sentence is added to the cluster and the common words are adjusted by intersection with the words from S. Otherwise a new cluster is created where S is the only sentence and all words from S are common.

Once the partitioning is completed and all clusters have been obtained, a representative dependency graph is constructed for each cluster $\texttt{CLUSTER}(S_1, \ldots, S_n, w_1, \ldots, w_m)$ by means of a SPARQL query defined as

```
SELECT ?a_1, ..., ?a_ℓ WHERE {
    ?src(e) r(e) ?dest(e).    // for each e ∈ E.
    ?v word s(v).             // if ∃ w ∈ W : w = s(v)
    ?v word ?a_k. }           // if ∄ w ∈ W : w = s(v)
```

where (V, E, r, s) is equal to the TDG after postprocessing and $W = \{w_1, \ldots, w_m\}$. That is, first the TDG structure is reassembled, secondly all common words are inserted, and finally all variable words are associated with the remaining vertices. Applying this SPARQL query to a triple store obtained from a sentence of the cluster directly returns the non-common variable words in the sentence.

3.3.3 Assertion Generation

Assertion generation occurs in two stages. First, an assertion template is created for each cluster. This template contains variables in the positions where assertion specific information such as signal names, logic levels, or numerical constants would normally appear. If we recall the RDG from Figure 3.6 we can see that a generalized sentence can be intuitively constructed from the information in the graph. This generalized sentence is representative of all sentences in a cluster. Using this generalized sentence a designer or verification engineer can manually design an appropriate SystemVerilog Assertion template.

It is important to note that in a normal verification process this mapping of English to a SystemVerilog Assertion would occur dozens if not hundreds of times for a large design. In our process it is only necessary once per cluster. The automation inherent in our process affords the designer or verification engineer the opportunity to apply their expertise where it will be the most valuable, crafting fewer, higher quality assertions.

In the second stage of assertion generation, the assertion template is populated for each NLA in the cluster. Variables are read from the typed dependency graph of each NLA. These

variables are combined with simple cluster specific mapping functions in order to generate the unspecified values for the assertion template. These mapping functions translate the English language symbols for signal names, logic values, or other verification parameters to their SystemVerilog equivalents. This is often realized as a direct or very simple mapping. This second stage results in a fully specified SystemVerilog assertion for each NLA in a cluster.

3.4 Experimental Results and Analysis

We have implemented the proposed algorithm in Java using the Stanford NLP library for natural language processing tasks and the JENA API for the triple store based information extraction.

We applied the algorithm to the *AMBA 3 AXI Protocol Checker* [43] user guide that consists of 145 natural language assertions for the *AMBA AXI 3 Protocol* [44]. We will now describe the implementation decisions and evaluate the main results of the experiment.

3.4.1 Results

To illustrate the experimental evaluation we make use of the four example sentences from the AMBA specification shown in 3.18 - 3.21.

"AWID must remain stable when AWVALID is asserted and AWREADY is LOW." (3.18)

"A write transaction with burst type WRAP has an aligned address." (3.19)

"AWVALID is LOW for the first cycle after ARESETn goes HIGH." (3.20)

"BRESP remains stable when BVALID is asserted and BREADY is LOW." (3.21)

3.4.1.1 Abstraction Level Classification

We have prepared the data for the experimental evaluation by first classifying all assertions manually. These expected values were then compared to the result of the classifier from which we computed the accuracy, precision, recall, and F-measure as defined in 2.5, 2.6, 2.7, and 2.8 respectively.

We discovered that most of the low level assertions contain one of the signal names listed in a summary table in the specification. Further we discovered that many local parameters are constrained. In the set of four NLAs above, sentence 3.19 is a high level assertion, whereas 3.18, 3.20, and 3.21 are low level. We have extended the triple store generation algorithm including a field in the database denoting whether a word is a signal name. For this purpose, the predicate *isSignalName* has been used which associates each word item with a boolean value. The SPARQL query provided to the classifier algorithm then checks whether a signal name is present or whether the word "parameter" occurs. The query reads:

```
SELECT ?signal ?someword WHERE {
  { ?signal isSignalName "true". } UNION
  { ?someword word "parameter". } }
```

From 145 assertions 100 have been classified as having a low level of abstraction and are candidates for translation. Numbers for each metric are listed in Table 3.1.

3.4.1.2 Partitioning based on Sentence Similarity

For the partitioning we have implemented all pre- and postprocessing rules as described in Section 3.3.2. This led to a partitioning of 11 clusters for the 100 assertions that were

Table 3.1: NLA Abstraction Level Classifier

Metric	Value
Accuracy	93.01%
Precision	95.00%
Recall	93.13%
F-measure	94.06%

been defined as low level in the previous step, i.e. each cluster contains approximately 9 sentences on average. Of the four example sentences presented in 3.18 - 3.21, sentences 3.18 and 3.21 belong to the same cluster. This becomes evident after the auxiliary verbs have been removed in the postprocessing step. Sentence 3.20 belongs to a different cluster. In order to understand the effect of the classification of the previous step to the clustering we have also computed the partition based on all 145 assertions. This lead to 50 clusters with approximately 3 sentences per cluster on average.

3.4.1.3 Assertion Generation

After partitioning the 100 NLAs into 11 distinct clusters a set of 11 corresponding SystemVerilog Assertion templates were generated. These assertion templates are presented in Table 3.2 with the variables highlighted in red. For each cluster mapping functions were also generated. Two types of mapping functions were used. The "signal name" mapping function simply passes the input value to the output. The other mapping function used was

Table 3.2: SystemVerilog Assertion Templates

<table>
<tr><th>Cluster</th><th>SVA Template</th></tr>
<tr><td>1</td><td>assert property(@(posedge clock)<signal1> >= <value>);</td></tr>
<tr><td>2</td><td>assert property(@(posedge clock) (<signal2> == <value2>) |-> (<signal1> != <value1>));</td></tr>
<tr><td>3</td><td>assert property(@(posedge clock) (<signal1> == <value1>) |-> (##1 $stable(<signal1>) [*1:$] ##1 (<signal2> == <value1>)));</td></tr>
<tr><td>4</td><td>assert property(@(posedge clock) RESET != 1 |-> (<signal1> != <value>));</td></tr>
<tr><td>5</td><td>assert property(@(posedge clock) (<signal2> == <value2>) |-> (<signal1> != <value1>));</td></tr>
<tr><td>6</td><td>assert property(@(posedge clock) <signal2> == <value2> |-> ((<signal1> == <value1>) ##1(<signal1> == <value1>)));</td></tr>
<tr><td>7</td><td>assert property(@(posedge clock) ((<signal1> == <value1>) && (<signal2> == <value2>)) |-> (<signal3> == <value3>));</td></tr>
<tr><td>8</td><td>assert property(@(posedge clock) ((<signal2> == <value1>) && (<signal3> == <value2>)) |-> $stable(<signal1>));</td></tr>
<tr><td>9</td><td>assert property(@(posedge clock) (<signal1> == <vlaue1>) |-> ((<signal2> == <value2>) |-> (<signal3> == <value3>)));</td></tr>
<tr><td>10</td><td>assert property(@(posedge clock) (<signal2> == <value1>) |-> ##[1:<parameter1>](<signal1> == <value1>));</td></tr>
<tr><td>11</td><td>assert property(@(posedge clock) (<signal2> != <value3>) |-> !((<signal1> == <value1>) && (##1 <signal1> == <value2>)));</td></tr>
</table>

the "logic level" mapping function. This function maps an abstract logic level such as *HIGH* or *LOW* to a logical 1 or 0 respectively. While these mapping functions might at first seem superfluous, they allow our technique much greater flexibility. By utilizing these mapping functions our method can not only handle *HIGH*, and *LOW* but also words such as *asserted*, *deasserted* and their synonyms.

3.4.1.4 Assertion Translation Walkthrough

We will now walk through an end-to-end example of our algorithm as applied to a portion of a real data set. Recall the four natural language assertions from [43] given in 3.18 - 3.21. Upon application of step 1 of the algorithm the sentences are separated into high and low abstraction sentences. The low abstraction sentences 3.18, 3.20, and 3.21, which are shown in Figure 3.7, are supplied to the next stage of the algorithm while the high abstraction sentence 3.19 is discarded from the translation set.

"AWID must remain stable when AWVALID is asserted and AWREADY is LOW."
"AWVALID is LOW for the first cycle after ARESETn goes HIGH."
"BRESP remains stable when BVALID is asserted and BREADY is LOW."

Figure 3.7: Translation Set of Low Abstraction NLAs

In step 2 the sentences are partitioned into clusters and an RDG for each cluster is generated. The three remaining sentences in our example are partitioned into two clusters based on sentence similarity as determined by their TDGs. Sentences 3.18 and 3.21 are in one cluster while sentence 3.20 is in a second cluster. The RDG for the cluster with sentences 3.18 and 3.21 is shown in Figure 3.8.

In step 3 we will look at the translation of sentences 3.18 and 3.21 while remembering that the steps outlined will be performed for each cluster in turn.

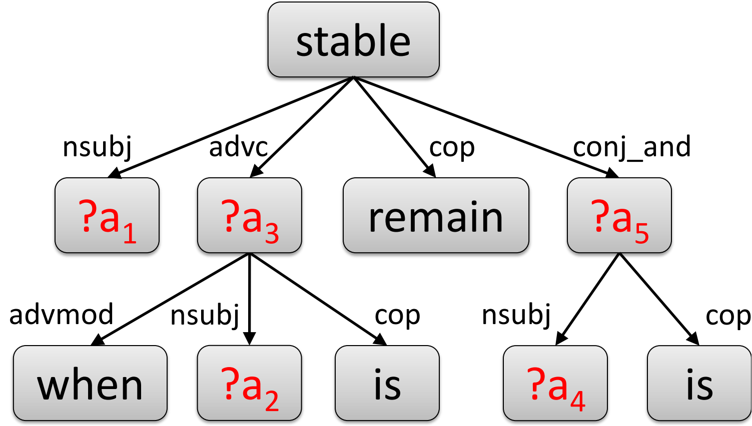


Figure 3.8: Cluster Representative Dependency Graph

We have now ascertained that sentences 3.18 and 3.21 are in the same cluster and share an RDG, which was generated in the previous step. We now manually construct the representative sentence

$$\text{“}a_1 \text{ remain stable when } a_2 \text{ is } a_3 \text{ and } a_4 \text{ is } a_5\text{”} \tag{3.22}$$

from the RDG for our cluster of interest. We also construct the assertion template

assert property(@(posedge clock) ((⟨signal2⟩ == ⟨value1⟩) && (⟨signal3⟩ == ⟨value2⟩)) → $stable(⟨signal1⟩));

which corresponds to entry 8 of Table 3.2 but is repeated here for convenience. SystemVerilog Assertions for every NLA in the cluster will be generated using this single template. Although the cluster of interest in the example only contains two NLAs, recall that our input data set yielded an average of 9 NLAs per cluster. In the final stage of the algorithm we read the variables for sentences 3.18 and 3.21 from their individual TDGs. The variables are applied to the mapping functions and combined with the assertion template resulting in the two fully realized SVAs shown in 3.23 and 3.24.

assert property(@(posedge clock)((AWVALID == 1)&&(AWREADY == 0)) → $stable(AWID)); (3.23)

assert property(@(posedge clock)((BVALID == 1)&&(BREADY == 0)) → $stable(BRESP)); (3.24)

3.4.2 Discussion

Although we have shown the ability to generate 100 SystemVerilog Assertions from requirements expressed in natural language while manually writing assertions for only a tenth of the input set, it is instructive to consider possible areas of vulnerability. We evaluate our results in consideration of three such areas below.

What happens if an assertion is classified wrong? There are two cases in which an assertion can be wrongly classified. If the NLA is classified as a high abstraction level sentence although it is low level, a manual reclassification is required. If a NLA is classified as a low abstraction level sentence although it is high level, it will likely end up in a single sentence cluster. Writing a translation rule for a cluster with only one sentence is equal to translating a high level assertion since no common words can be extracted. Hence, a wrongly classified assertion cannot cause any harm.

What if too many assertions are classified high level? In the worst case all assertions are classified as high level which corresponds to a conventional verification flow in which all NLAs are manually translated into formal representations. Although the proposed approach can therefore never be worse than the conventional one, a bad classification result is nevertheless unsatisfactory. In order to obtain a better result, the SPARQL query for classification can be enhanced.

What if the number of sentences per cluster is too small? If in the worst case each cluster contains of only one sentence, the proposed translation flow again performs equally compared to a conventional one. In order to obtain larger clusters one can inspect the assertions and provide additional tuning of the pre- and postprocessing rules during the partitioning stage.

However, one must be cautious to verify that sentences in the same cluster still have the same semantic meaning and can be translated using common translation templates.

Chapter 4

Grammar Based Property Translation

In this chapter we will develop, implement, and evaluate a method to generate formal verification properties from natural language which does not utilize templates as in chapter 3, but is fully realized through the use of a formal grammar. This corresponds to the second objective outlined in section 1.2.

This chapter presents a custom attribute grammar and a methodology for automatically extracting a set of *Computation Tree Logic* (CTL) properties from inline HDL code comments written in English. These properties can then be directly used, without any additional processing, in the formal verification of a design via model checking. Although the experiment in this chapter utilizes inline HDL code comments, this methodology is equally applicable to other natural language design documents such as system specifications. Before presenting our translation methodology we will first explore semantic parsing using the context free grammars first described in section 2.1 and extend these grammars by developing the theory of attribute grammars.

4.1 Semantic Parsing with Context Free Grammars

Recall the formal definition of a Context Free Grammar (CFG) from section 2.1, repeated here for convenience.

$$G = (V_N, V_T, S, P)\,, \tag{4.1}$$

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.

In syntactic parsing a sentence consisting of terminal symbols is converted to a parse tree consisting of both terminal and nonterminal symbols. This parse tree displays the constituents and syntactic structure of a sentence. 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.

$$G_{SEM} = (V_{TSEM}, V_{NSEM}, S, P_{SEM}) \tag{4.2}$$

$$V_{TSEM} = \{\text{“asserted”}, \text{“deasserted”}, \text{“be”}, \text{“should”}, \text{“reset”}\} \tag{4.3}$$

$$V_{NSEM} = \{\mathrm{S}, \mathrm{SETSIG}, \mathrm{SIGNAME}, \mathrm{VAL}\} \tag{4.4}$$

$$P_{SEM} = \{P_0, P_1, P_2, P_3\} \tag{4.5}$$

$$P_0 \equiv \mathrm{S} \rightarrow \mathrm{SETSIG} \tag{4.6}$$

$$P_1 \equiv \mathrm{SETSIG} \rightarrow \mathrm{SIGNAME}\ \text{“should”}\ \ \text{“be”}\ \mathrm{VAL} \tag{4.7}$$

$$P_2 \equiv \mathrm{SIGNAME} \rightarrow \text{“reset”} \tag{4.8}$$

$$P_3 \equiv \mathrm{VAL} \rightarrow \text{“asserted”} \mid \text{“deasserted”} \tag{4.9}$$

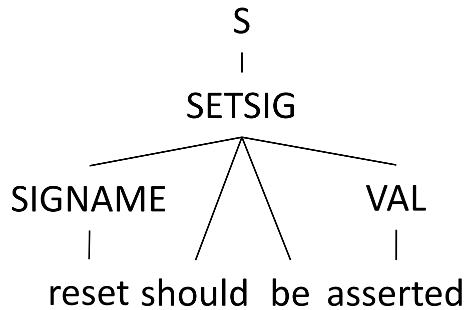


Figure 4.1: Example Semantic Parse Tree

Consider the semantic grammar G_{SEM} defined in (4.2)-(4.9). The nonterminal symbols in G_{SEM} no longer represent syntactic categories such as *noun* or *verb* but semantic categories such as *signal name* (SIGNAME) or *signal level* (VAL). Using the semantic grammar G_{SEM} we can now perform a semantic parse of the example sentence

$$\text{“Reset should be asserted”} \tag{4.10}$$

resulting in the semantic parse tree shown in Figure 4.1.

4.2 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 [45]. 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 relation $p = X_i \rightarrow (X_{j=1 \ldots n})^*$ has one attribution rule with which it is associated. The rule is applied every time that relation 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} from (4.2) to form a semantic attribute grammar with productions P_0, P_1, P_2, and P_3 from (4.6)-(4.9) associated with rules R_0, R_1, R_2, and R_3 (4.11)-(4.14) respectively.

$$R_0 \equiv \text{S.v} = \text{SETSIG.v} + \text{``;''} \tag{4.11}$$

$$R_1 \equiv \text{SETSIG.v} = \text{SIGNAME.v} + \text{``=''} + \text{VAL.v} \tag{4.12}$$

$$R_2 \equiv \text{SIGNAME.v} = \text{``module1.rst_''} \tag{4.13}$$

$$R_3 \equiv \text{VAL.v} = \text{``0''} \tag{4.14}$$

This new attribute grammar will be called G_{ATT} and is formally defined in (4.15)-(4.22). Note that all productions in P_{ATT} consist of a single production *relation* and a single *attribute*.

The attribute is named *value* and is denoted with a lowercase v.

$$G_{ATT} = (V_{TATT}, V_{NATT}, S, P_{ATT}) \tag{4.15}$$

$$V_{TATT} = \{\text{“asserted”}, \text{“deasserted”}, \text{“be”}, \text{“should”}, \text{“reset”}\} \tag{4.16}$$

$$V_{NATT} = \{\mathrm{S}, \mathrm{SETSIG}, \mathrm{SIGNAME}, \mathrm{VAL}\} \tag{4.17}$$

$$P_{ATT} = \{P_0, P_1, P_2, P_3\} \tag{4.18}$$

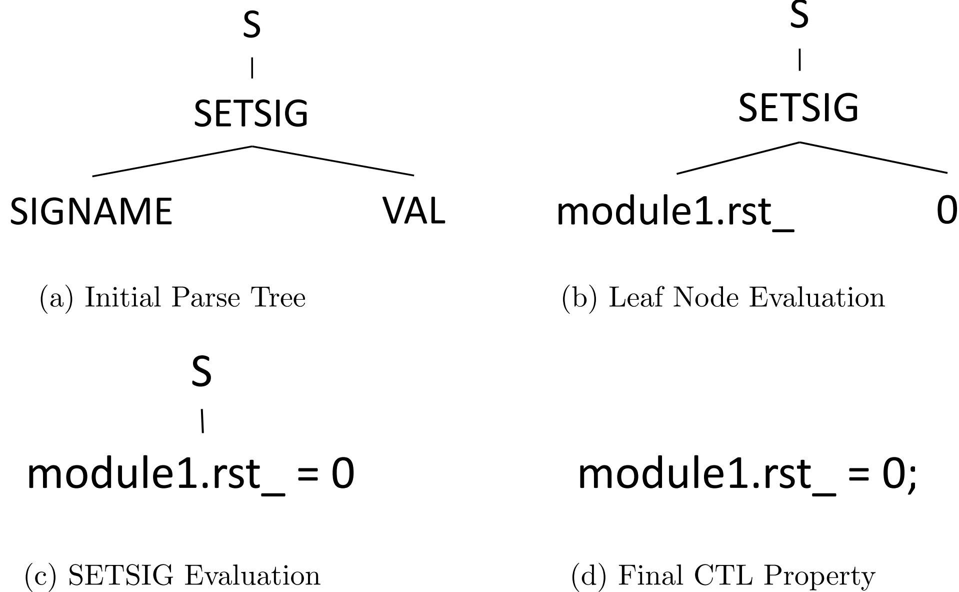
$$P_0 \equiv \left\{ \begin{array}{l} \mathrm{S} \rightarrow \mathrm{SETSIG} \\ \mathrm{S.v} = [\mathrm{SETSIG.v} + \text{“;”}] \end{array} \right\} \tag{4.19}$$

$$P_1 \equiv \left\{ \begin{array}{l} \mathrm{SETSIG} \rightarrow \mathrm{SIGNAME}\ \text{“should”}\ \text{“be”}\ \mathrm{VAL} \\ \mathrm{SETSIG.v} = [\mathrm{SIGNAME.v} + \text{“=”} + \mathrm{VAL.v}] \end{array} \right\} \tag{4.20}$$

$$P_2 \equiv \left\{ \begin{array}{l} \mathrm{SIGNAME} \rightarrow \text{“reset”} \\ \mathrm{SIGNAME.v} = [\text{“module1.rst_”}] \end{array} \right\} \tag{4.21}$$

$$P_3 \equiv \left\{ \begin{array}{l} \mathrm{VAL} \rightarrow \text{“asserted”} \mid \text{“deasserted”} \\ \mathrm{VAL.v} = [\text{“0”} \mid \text{“1”}] \end{array} \right\} \tag{4.22}$$

We are now prepared to revisit our example sentence and translate the NLA in 4.10 into a formal CTL property though the application of production attributes in G_{ATT} to the parse tree in Figure 4.1. 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 (4.21) and (4.22) respectively. This step is shown in Figure 4.2b. The values of these leaf nodes are then used to evaluate the attribute value of their parent node $SETSIG$. In Figure 4.2c we replace the symbol $SETSIG$ with its attribute value $SETSIG.v$ evaluated according to (4.20). Finally, in Figure 4.2d we evaluate the final symbol S which is defined in (4.19) to be the value of the $SETSIG$ attribute terminated by a semicolon. The use of attribute



(a) Initial Parse Tree

(b) Leaf Node Evaluation

(c) SETSIG Evaluation

(d) Final CTL Property

Figure 4.2: Attribute Evaluation Example

grammars in this manner allows a NLA represented by a semantic parse tree, with the appropriate choice of attributes, to be translated to a different formalization.

4.3 Methodology

We will now provide an overview of our end to end NLA translation methodology.

4.3.1 System Overview

Figure 4.3 depicts a high level block diagram of the translation system. In this diagram English language sentences are harvested from inline HDL 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

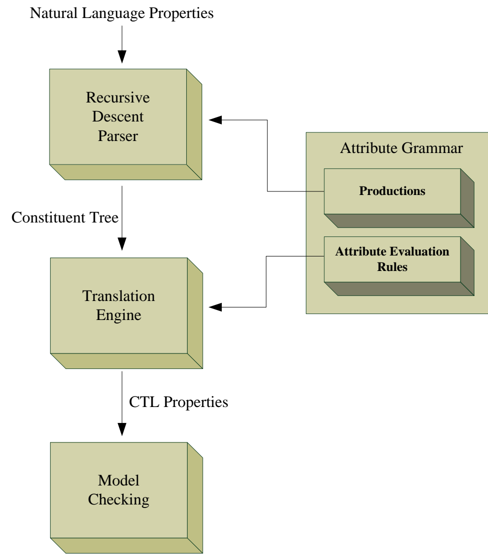


Figure 4.3: Translation System Block Diagram

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 [46].

4.3.2 Translation Engine

The system utilizes a stock recursive descent parser distributed as part of the Natural Language Toolkit (NLTK) [47]. NLTK is a general NLP environment written in Python which supports sentence parsing, text tokenization and classification, and other syntactic analysis [47]. However, the heart of our system is the translation engine which utilizes a custom attribute grammar. Our custom grammar is a semantic attribute grammar which models the designer intent of textual comments in addition to their syntactic structure.

A new grammar is commonly developed through the analysis of a large collection of discipline specific text samples known as a *corpus*. We utilized a subset of NL comments from the Verilog implementation of a PCI Local Bus from the Texas-97 Verification Benchmark suite [48] with additional information from the PCI Local Bus specification [49]. The use of specification information in the generation of the grammar allows the ability to correctly

translate design abstractions such as *bus transaction*, *memory read*, *initialize*, or *assert*. Our grammar is well tailored to the design which we are modeling. However, 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 the inline HDL comments used in this study 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 with similar meanings and combined to form a set of symbols $X_i \in X$. 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 relation $S \rightarrow PP\ CTLS$ the value of the attribute $S.v$ is defined 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 relation $S \rightarrow PP\ CTLS$ would successfully parse the NL sentence "In the clock cycle that REQ_ is low, GNT must be high". Example relations from the top level productions shown in 4.23 and 4.24 demonstrate that a $CTLS$ symbol can consist of an implication symbol $IMPLI$ or a $DELAYED_IMPLI$ symbol. The attribute value for the $CTLS$ symbol uses the CTL operator AG which acts on the attribute value of

the node which is the first child of $CTLS$ in the parse tree (where child node order is specified to be left to right).

$$P_{TL1} \equiv \left\{ \begin{array}{l} \text{S} \rightarrow \text{PP CTLS} \mid \text{NP VP} \\ \text{S.v} = [\text{LAST_CHILD.v}] \end{array} \right\} \tag{4.23}$$

$$P_{TL2} \equiv \left\{ \begin{array}{l} \text{CTLS} \rightarrow \text{IMPLI} \mid \text{DELAYED_IMPLI} \\ \text{CTLS.v} = [\text{AG(FIRST_CHILD.v)}] \end{array} \right\} \tag{4.24}$$

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. The productions in 4.25 and 4.26 show examples from this category.

$$P_{I1} \equiv \left\{ \begin{array}{l} \text{IMPLI} \rightarrow \text{``if'' CTL_TRANS ``then'' CTL_SET_NEXT} \\ \text{IMPLI.v} = [(\text{FIRST_CHILD.v}) \rightarrow (\text{SECOND_CHILD.v})] \end{array} \right\} \tag{4.25}$$

$$P_{I2} \equiv \left\{ \begin{array}{l} \text{DELAYED_IMPLI} \rightarrow \text{CTL_TRANS CTL_SET} \mid \text{SBAR CTL_UNTIL} \\ \text{DELAYED_IMPLI.v} = [(\text{FIRST_CHILD.v}) \rightarrow \text{AX(THIRD_CHILD.v)}] \end{array} \right\} \tag{4.26}$$

3. *Wait-Until:* Productions 4.27, 4.28, and 4.29 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.

$$P_{WU1} \equiv \begin{Bmatrix} \text{CTL_UNTIL} \rightarrow \text{WAIT_ACTION “until” WAIT_COND} \\ \text{CTL_UNTIL.v} = [\text{A(WAIT_ACTION.v U WAIT_COND.v)}] \end{Bmatrix} \tag{4.27}$$

$$P_{WU2} \equiv \begin{Bmatrix} \text{WAIT_ACTION} \rightarrow \text{CTL_SET} \mid \text{DIST2} \\ \text{WAIT_ACTION.v} = [\text{CHILD.v}] \end{Bmatrix} \tag{4.28}$$

$$P_{WU3} \equiv \begin{Bmatrix} \text{WAIT_COND} \rightarrow \text{CTL_SET} \mid \text{CTL_ANP} \\ \text{WAIT_COND.v} = [\text{CHILD.v}] \end{Bmatrix} \tag{4.29}$$

4. *Signal Value:* This category contains the symbols which denote a logical one or zero.

5. *Signal Assignment and Edge Transitions:* The productions in 4.30, 4.31, and 4.32 cover signal assignment as well as rising or falling edges.

$$P_{SA1} \equiv \begin{Bmatrix} \text{CTL_SET} \rightarrow \text{CTL_NP “has to be” CTL_VAL} \mid \text{CTL_NP MD “be” CTL_VAL} \\ \text{CTL_SET.v} = [(\text{FIRST_CHILD.v} = \text{LAST_CHILD.v})] \end{Bmatrix} \tag{4.30}$$

$$P_{SA2} \equiv \begin{Bmatrix} \text{SET_FUTURE} \rightarrow \text{CTL_NP “must be” CTL_VAL} \\ \text{SET_FUTURE} \rightarrow \text{CTL_NP “should” CTL_RB “be” CTL_VAL} \\ \text{SET_FUTURE.v} = [\text{AF}(\text{CTL_NP.v} = \text{CTL_VAL.v})] \end{Bmatrix} \tag{4.31}$$

$$P_{SA3} \equiv \begin{Bmatrix} \text{CTL_TRANS} \rightarrow \text{CTL_NP “has been” CTL_VAL} \mid \text{CTL_NP “is” CTL_VAL} \\ \text{CTL_TRANS.v} = [(!(\text{CTL_NP.v} = \text{CTL_VAL.v}) * \text{AX}(\text{CTL_NP.v} = \text{CTL_VAL.v}))] \end{Bmatrix} \tag{4.32}$$

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.

4.3.3 CTL Property Equivalence

The ready availability of the Texas-97 Benchmark suite [48] informed the choice of CTL as the target formalism for this study. In addition to CTL verification properties the Texas-97 suite [48] 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.

4.4 Experimental Results and Analysis

The natural language based formal verification system outlined above was implemented in Python using Natural Language Toolkit 2.0 [47]. The Texas-97 Verification Benchmark

suite [48], 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?

4.4.1 CTL Property Generation

A limited pre-processing step was performed on the natural language comments. This pre-processing 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 [48] a total of 22 CTL properties were identified which contained inline descriptions in English. Two of these HDL 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

Table 4.1: Unparseable NLAs

TextID	NLA Code Comment
PCI12	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	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).

Table 4.2: Parseable NLAs with CTL Matching Benchmark

TextID	**NLA Code Comment** Automatically Generated CTL Property
PCI0	Once FRAME_ has been asserted, the CBE_ lines should be driven until the end of the transaction. $AG(((!(m1.rFRAME_ = 0) * AX(m1.rFRAME_ = 0))) \to AX(A((m1.OE_CBE_ = 1) \cup ((m1.rIRDY_ = 0) * (t1.rTRDY_ = 0) + (m1.rIRDY_ = 0) * (t1.STOP_ = 0)))));$
PCI1	In the clock cycle that FRAME_ is deasserted, IRDY_ has to be asserted. $AG(((!(m1.rFRAME_ = 1) * AX(m1.rFRAME_ = 1))) \to AX((m1.rIRDY_ = 0)));$
PCI3	Whenever Trigger is set, a transaction must be initiated by the master. $AG(((!(m1.Trigger = 1) * AX(m1.Trigger = 1))) \to (AF(m1.rFRAME_ = 0)));$
PCI4	Once FRAME_ has been asserted, it should eventually be deasserted. $AG(((!(m1.rFRAME_ = 0) * AX(m1.rFRAME_ = 0))) \to (AF(m1.rFRAME_ = 1)));$
PCI7	Once TRDY_ has been asserted, TRDY_ and STOP_ must remain unchanged until the end of the current data phase. $AG(((!(t1.rTRDY_ = 0) * AX(t1.rTRDY_ = 0))) \to AX(A((((t1.rTRDY_ = 0) * AX(t1.rTRDY_ = 0)) * ((t1.STOP_ = 0) * AX(t1.STOP_ = 0))) \cup (((t1.rTRDY_ = 0) * (m1.rIRDY_ = 0)) + ((m1.rIRDY_ = 0) * (t1.STOP_ = 0))))));$
PCI8	Once STOP_ has been asserted, STOP_ and DEVSEL_ must remain unchanged until the end of the transaction. $AG(((!(t1.STOP_{=}0) * AX(t1.STOP_{=}0))) \to AX(A((((t1.STOP_{=}0) * AX(t1.STOP_{=}0)) * ((t1.DEVSEL_{=}0)*AX(t1.DEVSEL_{=}0)))U((m1.rIRDY_{=}0)*(t1.rTRDY_{=}0)+(m1.rIRDY_{=}0)*(t1.STOP_{=}0)))));$
PCI9	Once STOP_ has been asserted, TRDY_ and STOP_ must remain unchanged till the end of the transaction. $AG(((!(t1.STOP_ = 0)*AX(t1.STOP_ = 0))) \to AX(A((((t1.rTRDY_ = 0)*AX(t1.rTRDY_ = 0)) * ((t1.STOP_ = 0) * AX(t1.STOP_ = 0))) \cup ((m1.rIRDY_ = 0) * (t1.rTRDY_ = 0) + (m1.rIRDY_ = 0) * (t1.STOP_ = 0)))));$
PCI10	If REQ_ is asserted, GNT_ is eventually asserted. $AG(((!(m1.REQ_ = 0) * AX(m1.REQ_ = 0))) \to (AF(m1.GNT_ = 0)));$
PCI11	Observed that if devsel is removed then FRAME_ gets deasserted. $AG(((!(t1.DEVSEL_ = 1) * AX(t1.DEVSEL_ = 1))) \to (AX(m1.rFRAME_ = 1)));$

currently be processed by our system. These unparseable properties are shown in Table 4.1. While semantic analysis across sentence boundaries can be facilitated though the use of Discourse Representation Structures [30] 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.

Table 4.3: Parseable NLAs with CTL not Matching Benchmark

TextID	**NLA Code Comment** Automatically Generated CTL Property
PCI2	If DEVSEL_ is asserted, ultimately TRDY_ should be asserted. $AG(((!(t1.DEVSEL_ = 0) * AX(t1.DEVSEL_ = 0))) \rightarrow (AF(t1.rTRDY_ = 0)));$
PCI5	In a transaction if STOP_ and IRDY_ are asserted, FRAME_ must be deasserted. $AG(((m1.rFRAME_ = 0) * ((t1.STOP_ = 0) * (m1.rIRDY_ = 0))) \rightarrow (AF(m1.rFRAME_ = 1)));$
PCI6	This is achieved by ensuring that TRDY_ is asserted at least one clock cycle after FRAME_ is asserted. $AG((!(m1.rFRAME_ = 0) * AX(m1.rFRAME_ = 0)) \rightarrow AF((!(t1.rTRDY_ = 0) * AX(t1.rTRDY_ = 0))));$
PCI14	Whether any transaction completes successfully? $EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 0 * m1.TransStatus[1] = 0 * m1.TransStatus[0] = 0);$
PCI15	Whether any transaction completes with Retry? $EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 0 * m1.TransStatus[1] = 0 * m1.TransStatus[0] = 1);$
PCI16	Whether any transaction completes as a disconnect? $EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 0 * m1.TransStatus[1] = 1 * m1.TransStatus[0] = 1));$
PCI17	Whether any transaction completes as target Abort? $EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 0 * m1.TransStatus[1] = 1 * m1.TransStatus[0] = 0);$
PCI18	Whether any transaction sees the master to be busy? $EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 1 * m1.TransStatus[1] = 1 * m1.TransStatus[0] = 0);$
PCI19	Whether any transaction completes with master preempted? $EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 1 * m1.TransStatus[1] = 0 * m1.TransStatus[0] = 1);$
PCI20	Whether transaction status becomes Incomplete? $EF(m1.TransStatus[3] = 1 * m1.TransStatus[2] = 0 * m1.TransStatus[1] = 0 * m1.TransStatus[0] = 0);$
PCI21	Whether any transaction completes with Device Time Out? $EF(m1.TransStatus[3] = 0 * m1.TransStatus[2] = 1 * m1.TransStatus[1] = 0 * m1.TransStatus[0] = 0);$

The results of the experiment are presented in Tables 4.2- 4.1. Of our 22 natural language code 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). Over half of these correctly formed CTL properties,

however, did not match the CTL in the benchmark. These properties are shown in Table 4.3. Upon detailed analysis it was discovered that the automatically generated CTL statements in all but three of the properties in Table 4.3 are logically equivalent to the benchmark CTL. Properties PCI2, PCI5, and PCI6 were found to not be logically equivalent to the benchmark CTL. We will briefly analyze the automatically generated CTL for property PCI5, which was found to not be logically equivalent to the benchmark.

$$\text{In a transaction if STOP and IRDY are asserted, FRAME must be deasserted.} \tag{4.33}$$

$$AG((t1.STOP_ = 0) * (m1.rIRDY_ = 0) \rightarrow AX(m1.rFRAME_ = 1) \tag{4.34}$$

$$AG(((m1.rFRAME_ = 0) * ((t1.STOP_ = 0) * (m1.rIRDY_ = 0))) \rightarrow (AF(m1.rFRAME_ = 1))) \tag{4.35}$$

The NLA labeled PCI5 is shown in 4.33. The benchmark CTL property in 4.34 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 4.35. 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.

This observation highlights one of the benefits of our approach. Because it is easier for an engineer to express herself in a natural language such as English, it is more likely that a natural language requirement is fully defined. If that natural language description can be faithfully machine translated into a formal representation, as demonstrated in our approach, information is not lost and a verification miss is less likely to occur.

4.4.2 Dealing with Ambiguity

There is, however, one additional difference between the equations 4.34 and 4.35. 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. This situation results from the ambiguity of the phrase “must be” in the natural language.

Looking again at the natural language comment, the phrase “FRAME_ must be deasserted” shows no obvious temporal reference. However, it was discovered during the analysis of the natural language text used to 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 relation for the symbol with a semantic mapping to the CTL structure AF. In short, the phrase “must be” is understood to mean “in the future” in the limited context of our discipline specific corpus. The text of this corpus “trains” our grammar, incorporating this association 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. This type of analysis is similar to the weighted production approach utilized in *Probabilistic Context Free Grammars* (PCFGs). While there is no way to determine if the ambiguity was resolved correctly and we understand that this methodology does not address all types of ambiguities, we believe that it is a reasonable simplification for this common class of ambiguities. It does, however, indicate that 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. In addition, the "training" of the grammar using our corpus relies on human intuition. This moderately labor intensive process slightly mitigates the benefit realized from automatic verification property generation. However, context free grammars can be generated from sample text using a process known as *grammatical inference* [50]. We will further explore automatic grammar generation through this process in the next chapter.

Chapter 5

Assertion Generation Using Learned Formal Grammars

In previous chapters we have explored the automatic generation of formal verificaiton properties from natural language assertions (NLAs) using property templates and attributed context free grammars (CFGs). Results from the template based approach in chapter 3 showed that by writing 11 assertion templates, 100 NLAs could be translated in SystemVerilog. This is approximately a 10x increase in productivity for a verification engineer. However, this approach was limited in that NLAs must share a similar grammatical structure in order to best take advantage of the property templates. In chapter 4 an attribute grammar based translation approach was explored. This approach demonstrated a translation rate of over 90% for NLAs with a range of grammatical structures, but the approach relies on custom attribute grammars tailored to a single design or design family. The effort involved in designing such custom attribute grammars is prohibitive. In this chapter we address the third objective of this work. We combine elements of the template based approach in chapter 3 with the attribute grammar based approach in chapter 4 to create a grammar based translation approach in which a high quality custom attribute grammar is automatically generated from

a small sample set of NLAs and property templates. We develop a new learning algorithm to *infer* a suitable attribute grammar from a small sample set.

5.1 Grammatical Inference

Grammatical inference is also known in the literature as language learning, grammar learning, or grammar induction. It is the process of, given some information about an unknown language, inferring a set of grammatical rules to form a grammar G which can generate or parse sentences in the language. This information usually takes the form of a set of sample sentences from the language to be learned [51].

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 knowledge 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

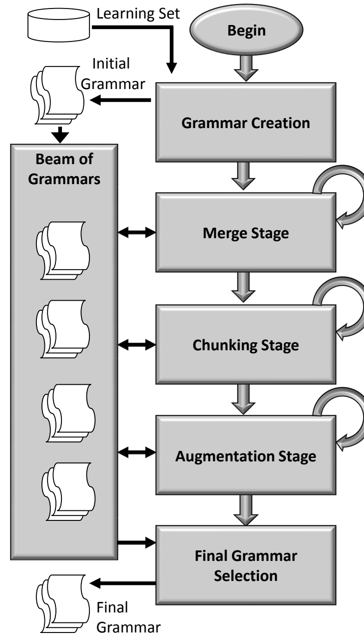


Figure 5.1: Grammar Induction Algorithm

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 [50]. In this chapter we also develop an unsupervised algorithm using positive examples. This algorithm will be detailed in the next section.

5.2 Algorithm Design

Our algorithm extends the E-GRIDS algorithm presented in [52] to support our special form of attributed grammars. E-GRIDS performs grammatical induction on CFGs, but does not support the type of attribute grammar necessary for assertion generation via NLA translation.

5.2.1 Overview

An overview of our algorithm architecture is shown in Figure 5.1. The user must supply a learning set consisting of natural language sentences and corresponding SystemVerilog Assertions (SVAs), as well as a list of signal names used in the design. In the initial *Grammar Creation* 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 initial mappings provided by the user. For example, the symbol $SIGNAME$ in equation (4.8) would have the value *module1.rst* mapped to its attribute value $SIGNAME.v$. This is because the signal name *module1.rst* is the semantic value, or meaning, of the word *reset* which is the right hand side of the production relation in (4.8).

After each unique word in the learning set of example sentences has been assigned a token (and attribute if appropriate), a top level relation for each sentence is created by replacing each word in the sentence with its previously defined token. Finally, each top level relation is assigned as attribute to form a top level production. 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 tokenized words 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 relations. 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 relation. A small learning set is shown in Table 5.1 and the initial grammar which it generates is presented in Table 5.2. Note that in attribute $S0.a$ of production P_1 the references $S1\text{-}1.a$ and $S1\text{-}2.a$ refer to the first and second instances of the $S1$ symbol on the right hand side of relation $S0$.

Table 5.1: Example Learning Set

	Learning Set
NLA 1	Reset should be high
SVA 1	assert property(@(posedge clk) module1.rst == 1;
NLA 2	AWBURST remains stable when AWVALID is asserted
SVA 2	assert property (@(posedge clk) (AWVALID == 1) |→ $stable(AWBURST_));

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 [53]. The number of search states retained at any given time is fixed and is stored in a data structure called a "beam". In our implementation, the beam can hold between 3 and 5 candidate grammars at a time. The heuristic used for pruning a grammar is a metric called the *minimum description length* (MDL). MDL, first proposed by Rissanen [54], 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 [52].

Our algorithm explores the search space of potential grammars by proceeding sequentially in three main stages. At each algorithm stage grammars in the beam are modified using one of three primary operators: *merge*, *chunk*, or *augment*. Each primary operator is used only during the algorithm stage corresponding to its name and accepts a pair of non-terminal symbols from a candidate grammar. During the corresponding stage, each operator is repeatedly applied to every candidate grammar in the beam. Each application of a primary operator is performed with a different pair of non-terminal symbols and generates a new candidate grammar. When all grammars in the beam have been modified using a primary operator, any candidate grammar which is found to have an MDL smaller that of a grammar

Table 5.2: Initial Grammar Creation

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

in the beam replaces the grammar in the beam 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 iteration, further exploring the search space by permuting and evaluating the grammars in the beam. Only when no new grammars enter the beam during an iteration does the algorithm advance 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. This process is illustrated in Figure 5.1. We will now detail the function of each of the three primary operators.

5.2.2 Primary Operators

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 has several key functions:

Table 5.3: Merge Operator

Before Merge			
$P_0 \equiv$	S0 → S1 S2 S3 S4 S0.a = [assert property(@(posedge clk) $S1.a$ $S3.a$ $S4.a$;]		
$P_5 \equiv$	S2 → should S2.a = [ ]	$P_7 \equiv$	S4 → high S4.a = ["1"]
$P_6 \equiv$	S3 → be S3.a = [==]	$P_{11} \equiv$	S8 → is S8.a = [==]
After Merging Symbols S2 with S4 and S3 with S8			
$P_0 \equiv$	S0 → S1 S2_S4 S3_S8 S2_S4 S0.a = [assert property(@(posedge clk) $S1.a$ $S3_S8.a$ $S2_S4\text{-}2.a$;]		
$P_5 \equiv$	S2_S4 → should S2_S4.a = [ ]	$P_7 \equiv$	S2_S4 → high S2_S4.a = ["1"]
$P_6 \equiv$	S3_S8 → be S3_S8.a = [==]	$P_{11} \equiv$	S3_S8 → is S3_S8.a = [==]

1. It accepts two non-terminal symbols S1 and S2, merging them into the new non-terminal symbol S3.
2. It replaces all occurrences of S1 or S2 with the new non-terminal symbol S3.
3. It removes any production rules with duplicate relations created by symbol replacements.

The merge operator preserves attribute assignments associated in a production. However, attribute values that appear in the definition(s) of higher level productions may need to be updated. The merge operator has the effect of generalizing an attribute grammar.

The effect of a merge operation on two pairs of symbols from the initial grammar is shown in Table 5.3. The top half of Table 5.3 shows a five production subset of our previously defined initial grammar. The bottom half of Table 5.3 show the changes in the grammar as a result of the merge operations highlighed in red. The operator takes the symbols $S2$ and

Table 5.4: Chunk Operator

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

$S4$ and merges them to create the new symbol $S2_S4$. The new symbol $S2_S4$ then replaces symbols $S2$ and $S4$ everywhere that they appear (namely in productions P_0, P_5, and P_7). Note that this replacement occurs in both the relation as well as the attribute portion of the production. In all other respects, the attribute portion of the productions are preserved. The same procedure is shown merging the symbols $S3$ and $S8$ into the symbol $S3_S8$.

The second primary operator, the *chunk* operator, could have also appropriately been named the concatenation operator. It uses two existing non-terminal symbols to create a new production. The two existing non-terminals appear sequentially on the right hand side of the relation for the new production. The new production which is created by the chunk operation is assigned an attribute value which is the concatenation of the attribute values of its constituent symbols. The operator then searches all existing productions and replaces each sequence of the two existing non-terminals (the "chunk") in a relation with the symbol for the newly created relation. This operator typically reduces the generality of a grammar by requiring that two symbols appear sequentially in a sentence.

The effect of the chunk operator on a subset of productions from the initial grammar is shown in Table 5.4. The top half of the table again shows a subset of the initial grammar. The bottom half of Table 5.4 shows the effect of chunking the $S2$ symbol with the $S3$ symbol (with new elements again highlighted in red). First, the new production P_{13} is created. The relation in P_{13} defines the new symbol $S2__S3$, which is simply the symbol $S2$ followed by $S3$. The attribute of P_{13} is set to the concatenated values of the attributes $S2.a$ and $S3.a$. The only effect on production P_0 is that the "chunk" $S2S3$ on the right hand side of the relation is replaced with the new symbol $S2__S3$. Similarly, the attribute value of P_0 is updated. The productions P_5 and P_6 are unchanged.

The third operator for our consideration is the *augmentation* operator. In the third and final stage of our algorithm, the augmentation operator creates a new production based on the result of a *chunk* operation from the previous stage. In this stage the relation created by a chunk operation is augmented by adding a third non-terminal symbol. The relation of the new, augmented, production keeps the same left hand side as the "chunked" production 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 which can be parsed. Attributes in an augment operation are handled in a similar manner to a chunk operation in that the attribute value for the new production is defined as the concatenation of the attribute values of its constituent symbols.

Table 5.5 demonstrates the behavior of the augment operator. The top half of the table shows the subset of a grammar in which a *chunk* operation has already occurred between symbols $S2$ and $S3$. The "chunked" relation is shown in the production P_{13}. The bottom half of the figure shows the effect of augmenting the symbol $S2__S3$ of production P_{13} with the $S4$ symbol. First, the new production P_{14} is created where the symbol $S4$ and attribute value $S4.a$ are simply added to the relation and attribute value, respectively, of production P_{13}. The effects on other productions in the grammar is more extensive. In production P_0

Table 5.5: Augment Operator

Before Augmentation			
$P_0 \equiv$	$\left\{ \begin{array}{l} \text{S0} \rightarrow \text{S1 S2__S3 S4} \\ \text{S0.a} = [\text{assert property(@(posedge clk)}\ S1.a\ S2__S3.a\ S4.a;] \end{array} \right\}$		
$P_5 \equiv$	$\left\{ \begin{array}{l} \text{S2} \rightarrow \text{should} \\ \text{S2.a} = [\ ] \end{array} \right\}$	$P_{15} \equiv$	$\left\{ \begin{array}{l} \text{S0} \rightarrow \text{... S10 S4 ...} \\ \text{S0.a} = [...] \end{array} \right\}$
$P_6 \equiv$	$\left\{ \begin{array}{l} \text{S3} \rightarrow \text{be} \\ \text{S3.a} = [\ ==\ ] \end{array} \right\}$	$P_{16} \equiv$	$\left\{ \begin{array}{l} \text{S0} \rightarrow \text{... S11 S4 ...} \\ \text{S0.a} = [...] \end{array} \right\}$
$P_{13} \equiv$	$\left\{ \begin{array}{l} \text{S2__S3} \rightarrow \text{S2 S3} \\ \text{S2__S3.a} = [S2.a\ S3.a] \end{array} \right\}$	$P_{17} \equiv$	$\left\{ \begin{array}{l} \text{S10} \rightarrow \text{... S2__S3} \\ \text{S10.a} = [...\ S2__S3.a] \end{array} \right\}$
		$P_{18} \equiv$	$\left\{ \begin{array}{l} \text{S11} \rightarrow \text{... S10} \\ \text{S11.a} = [...\ S10.a] \end{array} \right\}$
After Augmenting Symbol S2__S3 with Symbol S4			
$P_0 \equiv$	$\left\{ \begin{array}{l} \text{S0} \rightarrow \text{S1 S2__S3} \\ \text{S0.a} = [\text{assert property(@(posedge clk)}\ S1.a\ S2__S3.a;] \end{array} \right\}$		
$P_5 \equiv$	$\left\{ \begin{array}{l} \text{S2} \rightarrow \text{should} \\ \text{S2.a} = [\ ] \end{array} \right\}$	$P_{15} \equiv$	$\left\{ \begin{array}{l} \text{S0} \rightarrow \text{... S10 ...} \\ \text{S0.a} = [...] \end{array} \right\}$
$P_6 \equiv$	$\left\{ \begin{array}{l} \text{S3} \rightarrow \text{be} \\ \text{S3.a} = [\ ==\ ] \end{array} \right\}$	$P_{16} \equiv$	$\left\{ \begin{array}{l} \text{S0} \rightarrow \text{... S11 ...} \\ \text{S0.a} = [...] \end{array} \right\}$
$P_{13} \equiv$	$\left\{ \begin{array}{l} \text{S2__S3} \rightarrow \text{S2 S3} \\ \text{S2__S3.a} = [S2.a\ S3.a] \end{array} \right\}$	$P_{17} \equiv$	$\left\{ \begin{array}{l} \text{S10} \rightarrow \text{... S2__S3} \\ \text{S10.a} = [...\ S2__S3.a] \end{array} \right\}$
$P_{14} \equiv$	$\left\{ \begin{array}{l} \text{S2__S3} \rightarrow \text{S2 S3 S4} \\ \text{S2__S3.a} = [S2.a\ S3.a\ S4.a] \end{array} \right\}$	$P_{18} \equiv$	$\left\{ \begin{array}{l} \text{S11} \rightarrow \text{... S10} \\ \text{S11.a} = [...\ S10.a] \end{array} \right\}$

both the relation and the attribute are updated to use the values of production P_{14} instead of P_{13}.

The impact of the new production on productions P_{15} and P_{16} is more subtle. The relation in production P_{15} contains the symbol $S10$ followed by $S4$. If we were to replace the $S10$ symbol in P_{15} using the relation in P_{17} we would get the sequence in equation 5.1. The subsequent replacement of symbol $S2__S3$ using the relation in P_{13} would yield the sequence

in 5.2.

$$\ldots\ldots S2__S3\ S4\ldots \tag{5.1}$$

$$\ldots S2\ S3\ S4\ldots \tag{5.2}$$

This sequence, we can see is equivalent to the right hand side of the relation defined in our new production P_{14}. What this means is that the sequence in 5.1 can be replaced by the $S2__S3$ symbol as defined by production P_{14} (the $S4$ symbol is effectively subsumed into the $S2__S3$ symbol). This subsuming of $S4$ allows the sequence in the relation of production P_{15} to go from $S10$ $S4$ to simply $S10$. Similarly, the $S4$ symbol in production P_{16} is subsumed resulting in the sequence $S11$ $S4$ transforming into simply $S11$.

For a walkthrough of the entire algorithm, we revisit Figure 5.1. We begin by creating an initial grammar from the learning set. The resulting grammar is very specific but will parse the learning set exactly. The initial grammar is placed in the beam and the algorithm moves to the *merge* stage. In the merge stage the first grammar in the beam (the initial grammar) is considered and the *merge* operator is executed on each pair of non-terminal symbols in the grammar. Each execution of the operator generates a new candidate grammar. The MDL is calculated for each candidate grammar (smaller is better). If a candidate grammar has a smaller MDL than a grammar in the beam, then the candidate grammar replaces the beam grammar with the lowest MDL. When all pairs of non-terminal symbols in the first grammar have been merged and candidate grammars evaluated, the second grammar in the beam is considered. After all grammars in the beam have been considered then the stage iteration is complete. If a new grammar was placed in the beam during the iteration then the stage repeats. If no new grammar entered the beam then the algorithm moves to the the next stage. The second and third stages continue in a similar manner, only using the *chunk* and *augmentation* operators respectively. During each stage the grammars in the beam are

further generalized. After the third and final stage the grammar with the lowest MDL in the beam is selected as the final grammar.

5.3 Experimental Results and Analysis

We implemented our learning algorithm in Python using version 2.0 of the Natural Language Toolkit [47]. A set of 98 natural language verification requirements were taken form the ARM AMBA 3 AXI Protocol Checker User Guide [43]. These 98 natural language descriptions were divided in a learning set of 17 sentences and a cross-validation set of 81 sentences. SystemVerilog Attributes were created for the 17 sentences learning set. The learning set was selected to be representative of as many sentences in the specification document as possible in terms of both structure and word content. In addition, all sentences were pre-processed to perform pronoun resolution, replacing pronouns with the related noun (generally a signal name).

The initial grammar generated from our learning set was found to contain 206 words, 66 of which were unique. The grammar contained a total of 111 productions. After completing the learning algorithm, the final grammar exhibited a great deal of compaction with only 16 unique symbols representing 10 top level sentences over 115 productions.

The resulting learned grammar was used in the grammar based translation system from chapter 4 to generate SVAs from the 81 sentences in the cross-validation set. Out of these 81 NLAs, 71 sentences were parsed and translated into syntactically correct SystemVerilog, a translation rate of 88%. A subset of sentences from the cross-validation set and the automatically generated SVAs are shown in Table 5.6.

We also used our learned grammar to translate natural language requirements from a second specification document. This second analysis was used to examine if our learned grammar

Table 5.6: Selected Automatically Generated SystemVerilog Assertions

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

was general enough to translate NLAs from a document in the same *family* as our initial specification. We denote a *document family* to be a set of documents which share a similar writing style. A sample of 68 NLAs was taken from the AMBA 4 AXI Protocol User Guide [55]. Of these 68 sentences, unseen by our learning algorithm, SVAs were successfully generated for over 70% (48 out of 68). These results indicate that the generalization which takes place during grammar learning results in an attribute grammar which can be used across multiple related specifications. Based on this analysis, we anticipate that the use of a learning set which captures more of the linguistic variation in a document family will yield even higher translation rates on documents other than the one used to train the grammar.

Chapter 6

Conclusions

In this section we will discuss this dissertation in its entirety. We will first discuss criticisms to our approach before summarizing the contributions of this work. We will then briefly discuss some possible future branches of inquiry before

6.1 Criticisms

The use of Natural Language Processing has a substantial history in the area of web search and consumer devices. However, the use of NLP in engineering design is a new and largely unexplored area. The hardware community has voiced some criticism of the suitability of NLP techniques for addressing the verification problem. These criticisms are best summarized by arguments presented in [56].

1. It is difficult to model unstated assumptions and common sense.

2. Not all system requirements are captured in text.

3. With the increasing importance of formal verification, formal requirements are necessary.

4. It is difficult to express the complexity of modern systems unambiguously.

We will address the first and the last of these arguments together.

It is difficult to model unstated assumptions and common sense.
It is difficult to express the complexity of modern systems unambiguously.

We rebut these two criticisms by noting that it is the exact purpose of a specification to disambiguate complex systems. A specification which does not do so is generally a poorly written specification and is not fit for its purpose. The use of "unstated assumptions" is poor engineering practice. Many are the engineers who used undocumented"common sense" in their designs only to incur huge costs at a later date when the design was revisited for either maintenance, subsystem reuse, or the addition of new functionality. Further, in digital design it is useful to discover poorly constructed specifications early in the design cycle. Intervention at the beginning stages of the design cycle allows bugs to be mitigated at lower cost (see Table 1.1).

A more valid critical argument is the observation that not all system requirements are captured in textual form. Specifications often contain diagrams, charts, tables, and graphs which capture system requirements in addition to text. For example, it is not uncommon in software design to represent an entire system as a series of UML diagrams.

We view a solution to this problem as an extension of our existing approach. Specifications with requirements in other formats can be preprocessed such that textual requirements are generated from the nontextual data. For instance, image processing techniques such as edge detection techniques can be applied to timing diagrams to support natural language requirement extraction. A set of preprocessing modules, one for each requirement format

(graphical, tabular, etc), can be applied to a specification in order to increase the set of natural language requirements available for translation.

The final criticism, the importance of formal verification and the necessity of formal requirements, is best addressed in a summary of our contributions.

6.2 Contributions

The contribution of this work is broadly stated to be a set of methodologies and approaches to automatically generate formal verification properties from hardware requirements captured in natural language. Specifically, we have contributed the following:

6.2.1 An Algorithm for SVA Generation Using Templates

In chapter 3 we presented an algorithm for generating SystemVerilog Assertions from specification sentences called natural language assertions. Instead of the normal process of manually translating each sentence into an appropriate SVA, our approach categorizes assertions based on their abstraction level and then partitions low level abstraction level sentences into clusters based on structural similarity. All assertions in a partition can then be translated using the same SVA template, significantly decreasing verification effort. Experimental results showed a tenfold increase in efficiency over a fully manual process.

6.2.2 A Grammar Based Translation System for Temporal Logic Properties

In chapter 4 we defined and implemented a methodology to automatically generate Computation Tree Logic properties from natural language text descriptions. We created a custom attribute grammar which captured the semantics of our target document. This attribute grammar allowed us to perform a semantic parsing of the textual CTL property descriptions. The productions of the attribute grammar were then used to substitute attribute values which captured semantic meaning for the symbols in the semantic parse tree, resulting in fully formed CTL properties. We characterized our system using a verification benchmark suite, and verified that our automatically generated CTL properties successfully verified the associated model implementation. Finally, we compared the quality of the CTL generated by our method to CTL included in the benchmark suite and found that our automatically generated CTL met or exceeded the quality of the CTL found in the benchmark.

6.2.3 A Learning Algorithm to Capture the Customized Language of a Specification

Our third primary contribution was detailed in chapter 5. In this chapter we developed a learning algorithm which automatically generated a customized attribute grammar for use in our grammar based property translation system. We extended the E-GRIDS grammar induction algorithm to support attribute grammars and incorporated property templates for use as attributes of sentence level productions in the grammar. This allowed the increased linguistic variation exhibited by a grammar based translation approach to be combined with the leveraging of human designer intelligence through the creation of a small number of high quality SystemVerilog Assertion templates. Our results showed that almost 90% of natural language assertions were successfully translation to SVAs using our automatically learned

grammar. In addition, the generated grammar was able to successfully produce SVAs for over 70% of natural language assertions from the specification for an entirely different design in the same product family.

6.3 Future Work

Because NLP techniques have yet to be assiduously applied to engineering design, this area is ripe for exploration. Future work can focus on increasing both the quality and the quantity of translated correctness properties.

6.3.1 Increasing Quantity

We have previously discussed in this chapter the use of image processing to extract additional natural language requirements from figures and graphs. In addition to this approach, the number of natural language requirements available for translation can be increased by decomposing text containing higher level abstractions into sets of lower abstraction requirements. An example of this might be understood by examining the phrase *read transaction*. In a document where read transactions are specified, there exists an explanation of the low level steps necessary for such a transaction to occur. All of our techniques in this work operate on requirements captured in individual sentences. Semantic analysis across sentence boundaries can facilitate the translation of these higher level abstractions and other more complex requirements. In addition, syntactic analysis across sentence boundaries can allow automation of some steps which were manually completed in this work. Pronoun resolution being a primary example. The increased linguistic variation enabled by pronoun resolution will increase the number of low abstraction level sentences available for translation.

6.3.2 Increasing Quality

Increasing the quality of automatically generated requirements means generating more appropriate properties from the natural language. This can be assisted through better metrics. The core aim of this work is closely relabeled the the NLP task of machine translation. However, while machine translation is generally from one natural language to another natural language this work translates natural language into formal representations such as high level computer languages and mathematical representations. Metrics such as the BLEU score [57] which measure the quality of natural language translations are not suitable for the natural to formal language translations we explore in this work. Development of more appropriate metrics will allow a more nuanced tuning of our grammar generation algorithm and prevent such inefficiencies as overfitting and underfitting of grammar models.

6.3.3 From Formal Requirements to Natural Language

This work focuses on functional verification, the determination if a system is built according to specification. However, information flows both up and down the design stack during system design. The validation process determines if an implementation meets the purpose for which it is intended. For this task, a natural language description of a system is useful. The generation of natural language descriptions from formal software requirements are useful in the validation process as discussed in [58] so that the customer and engineer agree on the system parameters. This is the reverse of the task addressed in this work and thus a possible avenue of future inquiry.

6.4 Final Thoughts

The work in this dissertation has shown the promise of applying NLP techniques to the generation of hardware requirements for the functional verification of digital systems. This is a young area of research. Results indicate that not all requirements are successfully translated info formal properties using our nascent techniques. Even so, we submit that sufficiently high percentages of natural language requirements **are** translated such that our techniques can act as a “force multiplier” for verification engineers to increase their productivity and directly impact the fastest growing bottleneck in the modern digital ASIC design cycle.

Bibliography

[1] A. Raynaud, "The new gate count: What is verification's real cost?" *Electronic Design*, October 2003. [Online]. Available: http://electronicdesign.com/products/new-gate-count-what-verifications-real-cost

[2] R. Damiano, B. Bentley, K. Baty, K. Normoyle, M. Ishii, and E. Yogev, "Verification: What works and what doesn't," in *Proceedings of the 41st Annual Design Automation Conference*, ser. DAC '04. New York, NY, USA: ACM, 2004, pp. 274–274, moderator-Bacchini, Francine.

[3] B. Crothers, "Intel's sandy bridge chipset flaw: The fallout," January 2011, [Accessed: 11-June-2015]. [Online]. Available: http://www.cnet.com/news/intels-sandy-bridge-chipset-flaw-the-fallout/

[4] S. Iman and S. Joshi, "e reuse methodology," in *The e Hardware Verification Language*. Springer US, 2004, pp. 267–278.

[5] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale, *Verification Methodology Manual for SystemVerilog*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[6] M. Glasser, *Open Verification Methodology Cookbook*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[7] *Universal Verification Methodology (UVM) 1.2 Class Reference*, Accellera Systems Initiative, June 2014, [Accessed: 11-June-2015]. [Online]. Available: http://accellera.org/images/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf

[8] R. Stolzman, "Understanding assertion-based verification," *EE Times*, August 2002. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1275847

[9] L. Lamport, "Proving the correctness of multiprocess programs," *Software Engineering, IEEE Transactions on*, vol. SE-3, no. 2, pp. 125–143, March 1977.

[10] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 4, no. 2, pp. 123–193, Apr. 1999.

[11] R. Drechsler, *Advanced Formal Verification*. Norwell, MA, USA: Kluwer Academic Publishers, 2004.

[12] R. J. Abbott, “Program design by informal english descriptions,” *Commun. ACM*, vol. 26, no. 11, pp. 882–894, Nov. 1983.

[13] J. S. Thakur and A. Gupta, “Automatic generation of sequence diagram from use case specification,” in *Proceedings of the 7th India Software Engineering Conference*, ser. ISEC ’14. New York, NY, USA: ACM, 2014, pp. 20:1–20:6.

[14] T. Yue, L. Briand, and Y. Labiche, “A systematic review of transformation approaches between user requirements and analysis models,” *Requirements Engineering*, vol. 16, no. 2, pp. 75–99, 2011.

[15] S. Nanduri and S. Rugaber, “Requirements validation via automated natural language parsing,” *J. Manage. Inf. Syst.*, vol. 12, no. 3, pp. 9–19, Dec. 1995.

[16] E.-V. Chioaşcă, “Using machine learning to enhance automated requirements model transformation,” in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE ’12, 2012, pp. 1487–1490.

[17] G. S. A. Mala and G. V. Uma, “Automatic construction of object oriented design models [uml diagrams] from natural language requirements specification,” in *Proceedings of the 9th Pacific Rim International Conference on Artificial Intelligence*, ser. PRICAI’06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 1155–1159.

[18] E. G. Amoroso, “Creating formal specifications from requirements documents,” *SIGSOFT Softw. Eng. Notes*, vol. 20, no. 1, pp. 67–70, Jan. 1995.

[19] D. Chin, K. Takea, and I. Miyamoto, “Using natural language and stereotypical knowledge for acquisition of software models,” in *Tools for Artificial Intelligence, 1989. Architectures, Languages and Algorithms, IEEE International Workshop on*, Oct 1989, pp. 290–295.

[20] R. L. Cobleigh, G. S. Avrunin, and L. A. Clarke, “User guidance for creating precise and accessible property specifications,” in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT ’06/FSE-14. ACM, 2006, pp. 208–218.

[21] V. Ambriola and V. Gervasi, “An environment for cooperative construction of natural-language requirement bases,” in *Software Engineering Environments, Eighth Conference on*, Apr 1997, pp. 124–130.

[22] ——, “Processing natural language requirements,” in *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference*, Nov 1997, pp. 36–45.

[23] C. O’Halloran, “On requirements and security in a ccis,” in *Computer Security Foundations Workshop V, 1992. Proceedings.*, Jun 1992, pp. 121–134.

[24] 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.

[25] J. J. Granacki, A. C. Parker, and Y. Arena, "Understanding system specifications written in natural language," in *Proceedings of the 10th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'87. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, pp. 688–691.

[26] W. R. Cyre, J. Armstrong, M. Manek-Honcharik, and A. J. Honcharik, "Generating vhdl models from natural language descriptions," in *Proceedings of the Conference on European Design Automation*, ser. EURO-DAC '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 474–479.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] P.-H. Chang and L.-C. Wang, "Automatic assertion extraction via sequential data mining of simulation traces," in *Design Automation Conference, 2010. Proceedings of the ASP-DAC 2010. Asia and South Pacific*, 2010, pp. 607–612.

[35] 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.

[36] B. Keng, S. Safarpour, and A. Veneris, "Automated debugging of SystemVerilog assertions," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, 2011, pp. 323–328.

[37] N. Chomsky, "Three models for the description of language," *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 113–124, 1956.

[38] M. Marcus, G. Kim, M. A. Marcinkiewicz, R. MacIntyre, A. Bies, M. Ferguson, K. Katz, and B. Schasberger, "The penn treebank: Annotating predicate argument structure," in *Proceedings of the Workshop on Human Language Technology*, ser. HLT '94. Stroudsburg, PA, USA: Association for Computational Linguistics, 1994, pp. 114–119.

[39] B. Santorini, "Part-Of-Speech tagging guidelines for the Penn Treebank project (3rd revision, 2nd printing)," Department of Linguistics, University of Pennsylvania, Philadelphia, PA, USA, Tech. Rep., 1990.

[40] M.-C. de Marneffe and C. D. Manning, "The stanford typed dependencies representation," in *CrossParser*, 2008, pp. 1–8.

[41] M.-C. de Marneffe, B. MacCartney, and C. D. Manning, "Generating typed dependency parses from phrase structure parses," in *LREC*, 2006, pp. 449–454.

[42] *SPARQL 1.1 Query Language*, W3C, Mar. 2013.

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

[44] *AMBA AXI and ACE Protocol Specification*, ARM, Oct. 2011.

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

[46] 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.

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

[48] 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/

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

[50] 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.

[51] 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.

[52] 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.

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

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

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

[56] K. Ryan, "The role of natural language in requirements engineering," in *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, Jan 1993, pp. 240–242.

[57] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. Stroudsburg, PA, USA: Association for Computational Linguistics, 2002, pp. 311–318.

[58] A. Salek, P. Sorenson, J. Tremblay, and J. Punshon, "The review system: from formal specifications to natural language," in *Requirements Engineering, 1994., Proceedings of the First International Conference on*, Apr 1994, pp. 220–229.