Framework for Automatic Translation of Hardware Specifications Written in English to a Formal Language

Rahul Krishnamurthy

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

Michael S. Hsiao, Chair

A. Lynn Abbott

Edward A. Fox

Thomas L. Martin

Haibo Zeng

September 19, 2022

Blacksburg, Virginia

Keywords: Computational Linguistics, Dependency Grammar, Parsing, Suggestion Generation

Copyright 2022, Rahul Krishnamurthy

Framework for Automatic Translation of Hardware Specifications Written in English to a Formal Language

Rahul Krishnamurthy

(ABSTRACT)

The most time-consuming component of designing and launching hardware products to market is the verification of Integrated Circuits (IC). An effective way of verifying a design can be achieved by adding assertions to the design. Automatic translation of hardware specifications from natural language to assertions in a formal representation has the potential to improve the verification productivity of ICs. However, natural language specifications have the characteristics of being imprecise, incomplete and ambiguous. An automation framework can benefit verification engineers only if it is designed with the right balance between the ease of expression and precision of meaning allowed for in the input natural language specifications. This requirement introduces two major challenges for designing an effective translation framework. The first challenge is to allow the processing of expressive specifications with flexible word order variations and sentence structures. The second challenge is to assist users in writing unambiguous and complete specifications in the English language that can be accurately translated.

In this dissertation, we address the first challenge by modeling semantic parsing of the input sentence as a game of BINGO that can capture the combinatorial nature of natural language semantics. BINGO parsing considers the context of each word in the input sentence to ensure high precision in the creation of semantic frames.

We address the second challenge by designing a suggestion and feedback framework to assist users in writing clear and coherent specifications. Our feedback generates different ways of writing acceptable sentences when the input sentence is not understood.

We evaluated our BINGO model on 316 hardware design specifications taken from the documents of AMBA, memory controller, and UART architectures. The results showed that highly expressive specifications could be handled in our BINGO model. It also demonstrated the ease of creating rules to generate the same semantic frame for specifications with the same meaning but different word order.

We evaluated the suggestion and rewriting framework on 132 erroneous specifications taken from AMBA and memory controller architectures documents. Our system generated suggestions for all the specs. On manual inspection, we found that 87% of these suggestions were semantically closer to the intent of the input specification. Moreover, automatic contextual analysis of the rewritten form of the input specification allowed the translation of the input specification with different words and different order of words that were not defined in our grammar.

Framework for Automatic Translation of Hardware Specifications Written in English to a Formal Language

Rahul Krishnamurthy

(GENERAL AUDIENCE ABSTRACT)

The most time-consuming component of designing and launching hardware products to market is the verification of hardware circuits. An effective way of verifying a design is to add programming codes called assertions in the design. The creation of assertions can be time-consuming and error-prone due to the technical details needed to write assertions. Automatically translating assertion specifications written in English to program code can reduce design time and errors since the English language hides away the technical details required for writing assertions. However, sentences written in English language can have multiple and incomplete interpretations. It becomes difficult for machines to understand assertions written in the English language.

In this work, we automatically generate assertions from assertion descriptions written in English. We propose techniques to write rules that can accurately translate English specifications to assertions. Our rules allow a user to write specifications with flexible use of word order and word interpretations. We have tested the understanding framework on English specifications taken from four different types of hardware design architectures.

Since we cannot create rules to understand all possible ways of writing a specification, we have proposed a suggestion framework that can inform the user about the words and word structures acceptable to our translation framework. The suggestion framework was tested on specifications of AMBA and memory controller architectures.

Dedication

Dedicated to ,

Dad, S. Krishnamurthy,

Maa, Uma Krishnamurthy,

and my brother, Srikanth Krishnamurthy.

Acknowledgments

I enrolled in my Ph.D. program to pursue a Ph.D. in VLSI testing and verification. Little did I know that I would end up doing a Ph.D. in understanding language semantics. This Ph.D. has been a life-changing experience and could not have been possible without the support and advice of many people.

I want to thank my advisor, Professor Michael S. Hsiao, for introducing me to the field of natural language programming. I am also grateful to Professor Hsiao for his guidance and support that helped me in completing my Ph.D. dissertation.

I would also like to thank Dr. A. Lynn Abbott, Dr. Edward A. Fox, Dr. Thomas L. Martin, and Dr. Haibo Zeng for serving on my doctoral committee and providing valuable suggestions and comments on my Ph.D. work.

I want to thank Professor G.K.Sharma at ABV-IIITM Gwalior, under whom I learned the fundamentals of doing academic research. While working in the lab of Professor G.K.Sharma, I got an opportunity to do research and to develop research skills by engaging in fruitful discussions with his Ph.D. students, Bharat Garg and Nandakishor Yadav. I take this opportunity to thank Dr. Bharat Garg and Dr. Nandakishor Yadav for giving me helpful tips and suggestions for conducting good-quality academic research.

I could not have worked for long hours in my lab without the company of brilliant, caring, and always encouraging lab mates. I want to thank my lab mates, Dr. Yue Zhan and Aravind Krishnan Varadarajan, for making my life in Durham 367 lab so memorable.

I want to thank Shrikant Fulari for being an amazing and helpful roommate who made my time in Blacksburg filled with great memories.

Finally, I express my heartfelt gratitude to my parents and brother for their love and unwavering support. Without my parents' support, sacrifice and love, I could not have left my job and returned to school to pursue higher education.

Contents

List of Figures xiii

List of Tables xx

1 Introduction 1

1.1 Automating Assertion Generation from Specifications . . . 3

1.1.1 Semi-Automated Approaches . . . 3

1.1.2 Natural Language Based Automation . . . 4

1.1.3 Problems with Natural Language Based Translation . . . 5

1.2 Controlled Natural Language Tools for Design Verification . . . 7

1.2.1 CNL Classification . . . 7

1.2.2 Domain Specific CNL Tools for Verification . . . 9

1.2.3 Problems with Current Domain Specific CNLs . . . 14

1.3 Contributions . . . 15

1.4 Chapter Structure . . . 17

2 Basic Concepts 19

2.1 Grammar Formalism . . . 19

2.1.1 Phrase Structure Grammar . . . 19

2.1.2 Combinatory Categorial Grammar 20
2.1.3 Link Grammar 24
2.1.4 Dependency Grammar 26
2.2 Dependency Grammar-based Parsing Algorithms 29
2.2.1 Exhaustive Search-based Dependency Parser 29
2.2.2 Arc-Eager Dependency Parsing Algorithm 31
2.3 Issues in Dependency Parsing 35
2.3.1 Dependency Tree Representation of Conjunction 35
2.3.2 Dependency Parsing of Sentences with a Conjunction 36
2.4 Domain Specific Natural Language Understanding 39
2.4.1 Phoenix 39
2.4.2 Philips Automated Inquiry System 40
2.4.3 Delphi 41
2.4.4 Gemini 43
2.4.5 Understanding after Sentence Simplification 44
2.5 Authoring Tools for Controlled Natural Language (CNL) 45

3 Data Structure and Pre-Processing **48**

3.1 Node Data Structure 48
3.2 Semantic Expression 51

3.3 Tokenization 53

4 BINGO Framework **57**

4.1 Motivation 58

4.2 Framework 59

4.3 Grammar Formalism 60

4.3.1 Semantic Propagation Rules Syntax 62

4.3.2 Fill Via Node 64

4.3.3 Fill Via Slot 64

4.3.4 Fill Via SE 65

4.3.5 Fill Via SE_SE 65

4.3.6 Self Link Rules 66

4.4 2-Step BINGO Parser 66

4.4.1 Step 1 - Marking Chart Cells 69

4.4.2 Step 2 - Finding BINGO Row 70

4.5 Semantic Creation 71

4.6 Grammar Creation Examples 78

4.6.1 Contextual Grammar Rules Resolving Ambiguities 80

4.6.2 Grammar Rules Representing Conjunction Relation 86

4.7 Evaluation 98

4.7.1 RTL Specifications . . . 101
4.7.2 ACE Protocol Specifications . . . 103
4.7.3 UART and Memory Controller Specifications . . . 104
4.7.4 Detecting Incomplete Specifications . . . 106
4.7.5 Detecting Ambiguous Specifications . . . 108
4.7.6 Multiple Variations of Specifications . . . 109
4.7.7 Results and Comparison . . . 109

5 Automated Rewriting and Suggestion Generation 113

5.1 Motivation . . . 113
5.2 Parse Tree Parameters . . . 115
5.3 Methodology . . . 121
5.3.1 Selecting Incomplete Parse Trees . . . 122
5.3.2 Finding Candidate Words . . . 122
5.3.3 Ranking Mechanism . . . 123
5.3.4 Multiple Top Ranked Parse Trees . . . 128
5.3.5 Generating Suggestions . . . 133
5.4 Evaluation . . . 134

6 Conclusions and Future Work 141

6.1 Future Work . . . 143

Bibliography 144

List of Figures

1.1 Verification cost increases as the technology node shrinks [1] 2

1.2 Design productivity and design complexity [1] 2

1.3 A formal requirement written in ASSERT CNL [2] 14

2.1 Context-Free Grammar Rules to create parse tree 20

2.2 A Constituency Parse Tree of the sentence 'Value in register r1 is 1' 20

2.3 CCG derivation diagram of sentence 'value in register is 1'. 21

2.4 Link requirements for a set of words. 24

2.5 Link requirements satisfied for a sequence of words. 25

2.6 A Dependency Tree of the sentence 'Value in register r1 is 1' 28

2.7 Step-by-step description of search-based algorithm of dependency parse tree creation. A parse tree is created for the sentence 'Value in register r1 is 1.' . 30

2.8 Resultant Dependency parse tree of the sentence 'Value in register r1 is 1'. . 30

2.9 Action performed in each parsing configuration of a sentence based on Arc-eager dependency algorithm. 31

2.10 Working of arc-eager parsing algorithm on the sentence 'The value of 1 is assigned to register r2'. The resultant parse tree after each step of the algorithm is also shown. 34

2.11 Treatment of conjunction in Hudson's English Word Grammar [3] 36

2.12 Syntactic dependencies in coordinate conjunction in Melcuk's theory 36

2.13 Treatment of conjunction based on Stanford dependency manual 36

2.14 Metarule application on partial dependency parse tree due to the presence of conjunction 'and'. The parse tree is shown for the sentence 'John laughed and Mary sneezed'. [4] 37

2.15 A semantic graph generated by parser for the sentence 'What flights fly from Boston to Denver' [5]. 41

2.16 A lexicon entry of word 'fly' annotated with different types of semantic information in the Delphi system. 41

2.17 Dependency parse tree of the sentence 'I need to fly from London to Boston' [6]. 45

3.1 Illustration of words as nodes and general structure of node data structure. . 49

3.2 (a) Parse of a phrase. (b) Displays the attributes of node1. 50

3.3 Basic structure of a Semantic Expression (SE). 51

3.4 (a) SE representing phrase 'writing 1 to bit 1', (b) SE representing a phrase with a conjunction. 52

3.5 An SE containing SEs in elements list attribute of a slot. 52

3.6 (a) The first step in pre-processing converts the entire raw text of the input specification to lower case. (b) Tokenizer converts raw text to a list of words that are stored as nodes. 53

3.7 (a) Splitting lower case input text using space as a delimiter. (b) Step 2 and step 3 of the tokenizer. 54

3.8 The step 2 of tokenization is explained with an example list of words. 55

3.9 The step 3 of tokenization is explained with an example word '2'b00'. 56

4.1 Framework for parsing and translating hardware specification to Semantic Frames. 60

4.2 Structure of (a) a grammar link rule and (b) a semantic propagation rule. . . 61

4.3 Syntactic links shows syntactic requirement of words in a sentence. Semantic links represent similar linking requirement between the head and child words in a dependency relation. 61

4.4 (a) Example of Semantic propagation operation when fill_via is node. (b) Example of Semantic propagation operation when fill_via is slot. 62

4.5 (a) Example of Semantic propagation operation when fill_via is SE. (b) Example of Semantic propagation operation when fill_via is SE_SE. 63

4.6 (a) Grammar link rules for words 'before' and '5 clock cycles' to parse ' system should reach dataReady before 5 clock cycles'.(b) Translating grammar rules to BINGO chart. (c) Marking cells of links that have matched syntactic and semantic links without violation link order constraints. (d) Link replication and marking cells of replicated link. 67

4.7 (a)Initial config. of Charts based on grammar rules given in Figure 4.8. (b) Searching BINGO rows after marking cells of charts. (c) Final Linkage solution of the BINGO row shown by the Black dashed line (Linkage 1). (d) System Verilog Assertion (SVA) created from the SE of the parse tree. . . . 68

4.8 A small set of grammar rules to parse specifications 70

4.9 First four steps in the semantic creation process for connecting nodes according to BINGO solution found in parsing. 72

4.10 Nodes connected through L2 link found in the BINGO solution. 73

4.11 Steps 5 - 7 in the semantic creation process of connecting nodes and moving buffer elements to the stack. 74

4.12 (a) Semantics associated with L3 link is created, (b) New occur_frame is propagated to 'should' node after semantic creation step. 75

4.13 Steps 8 - 9 in the semantic creation process of connecting nodes and moving buffer elements to the stack. 76

4.14 (a) Semantic creation step for L4 link creation, (b) Semantic created at reach node is propagated to its head node. 77

4.15 Steps 10 - 11 in the semantic creation process of connecting nodes and moving buffer elements to the stack. 77

4.16 Semantic creation associated with L_5 doesn't lead to semantic changes, and no semantic update operation takes place. 78

4.17 (a) Semantic creation associated with the L_6 link is shown, (b) Semantic update operation for SE occur_frame takes place from node 'before' to the 'reach' node, (c) Semantic update operation is executed to propagate semantic changes from the node 'reach' to the root node 'should', (d) Final SE created at the root node 'should'. 79

4.18 Preposition phrase attachment problem with ambiguity to connect a preceding noun or verb with a preposition. 80

4.19 Declarative rules created to define correct attachments between nodes based on context. 81

4.20 Declarative rules encode semantic context to disambiguate attachment between nodes. 83

4.21 Rules demonstrating the connection of conjunction and conjunct in a sentence. 87

4.22 Output of BINGO parsing for specification and grammar rules of Figure 4.21. 87

4.23 The SE representing the meaning of Spec of Figure 4.21. 88

4.24 A parse configuration where stack top is 'dataReady' and buffer head is 'or'. 89

4.25 Partial parse tree representing parse till the node 'dataReady' of conjunction Spec given in Figure 4.21. 90

4.26 Link L_{14} is created between nodes 'or' and 'dataReady' and its semantic propagation rule is executed. 90

4.27 Semantic propagation rule of self link at node 'or' is executed. 92

4.28 A self link rule semantic propgation rule is executed at the 'dataReady' node. 94

4.29 An execution of self link rule that propagates slot Elements attribute to the Conn attribute of the destination slot. 95

4.30 Semantic changes due to the execution of self link rule is propagated till the root node. 96

4.31 Step 3 creates link connection between 'or' and 'dataWait' nodes. 97

4.32 Link L_{15} is created between nodes 'or' and 'dataWait', and its semantic propagation rule is executed. 97

4.33 Execution of semantic propagation rule of self link at 'dataWait' fills the connector_nodes slot of conj_expr SE. 98

4.34 The update in the conj_expr at 'dataWait' node is propagated to 'dataReady' node through multiple semantic update operations. 99

4.35 The occur_expr SE updated at 'dataReady' node is propagated to the root node 'should be' through multiple semantic update operations. 100

4.36 Specifications with RTL information are translated to SVA code. 101

4.37 A BINGO row does not exist for incomplete specifications and we cannot create semantic frames for these specifications. 106

4.38 Ambiguous spec generates more than one BINGO row. 108

4.39 Specs with the same meaning but different word order generates the same SEs. 109

5.1 Incomplete set of charts at the end of BINGO parsing. 116

5.2 Grammar rules for creating charts shown in Figure 5.1. 117

5.3 Incomplete parse tree information extracted from Fig 5.1 charts. 120

5.4 Parameters of incomplete Parse 1 and Parse 5 from Fig. 5.1. 120

5.5 Main blocks of suggestion framework on top of BINGO model. 120

5.6 Connecting L6+ link of 'should be' node from Parse 1. 123

5.7 Link rule used for completing the connection of L6+ link of Fig 5.6. 124

5.8 Scoring candidate words based on parse tree parameters coverage. 125

5.9 Computing a single score for a candidate words list. 126

5.10 Final set of candidate words for partial row's unconnected links and their scores. 127

5.11 Parse 1 of Figure 5.3 completed by top ranked words. Words in blue color are added by the suggestions framework. 128

5.12 Parse 2 of Figure 5.3 completed by top ranked words. 129

5.13 Parse 3 of Figure 5.3 completed by top-ranked words. 131

5.14 Parse 4 of Figure 5.3 completed by top-ranked words. 131

5.15 A flowchart of checks to generate semantically unique parse trees. 132

5.16 Semantic Theme showing only frames and slot names for parse trees. 133

5.17 Final set of suggestions generated by the framework. User-written words in Red and newly added words in Blue. 134

5.18 Signal name inference & SVA generation based on re-written spec. 134

5.19 Analysis of rewritten spec led to SVA generation of specs. 135

5.20 SVA for the re-written spec that covered all input concept_nodes. 135

5.21 Suggestion generation for the spec without standard antecedent-consequent structure. 136

5.22 Suggestion for a phrase with different words and order of words. 136

5.23 Unnecessary words removed from original spec in suggestions. 137

5.24 Suggestions generated only for phrases that could not be parsed. 138

5.25 Suggestions generated only for phrases that could not be parsed. 139

5.26 Suggestions generated for both the phrases that could not be parsed. 139

List of Tables

2.1 Link Grammar 26

4.1 Grammar size for specifications tested 98

4.2 Specification types, count, and average JavaScript processing time for Specs 109

4.3 Comparison with previous rule-based works on Hardware Assertions Synthesis. 111

Chapter 1

Introduction

Modern integrated circuits are packed with multiple embedded processors, and Intellectual Property (IP) blocks to meet the ever-increasing demand for functionality and performance. However, while achieving the high-performance target, the design must meet low area requirements, power, and other constraints simultaneously. These requirements exacerbate the hardware design complexity and increase functional verification complexity. Furthermore, due to aggressive time-to-market constraints for the design and system development, the design life cycle has been drastically shortened. As a consequence, design complexity is increasing, but the time given to design verification has relatively shrunk [1].

The overall design productivity depends on the productivity of design verification. The design verification typically takes 40% to 50% of the entire project resources and could be as high as 80 to 90% [7]. Figure 1.1 shows the cost for different parts of the design cycle. As shown in the figure, the resources devoted to the verification stage increase as the design moves towards lower technology feature sizes.

Given the difficulty of verification, the main challenge in the production of integrated circuits has focused on improving the productivity of design verification. According to Figure 1.2, compounded complexity growth per year is 58% while the compounded productivity rate is only 21%. This implies that the IC productivity is not keeping pace with the increase in circuit complexity. The major contributor to this gap is the design verification stage [1]. This gap refers to the difference in our ability to design complex ICs versus our ability to

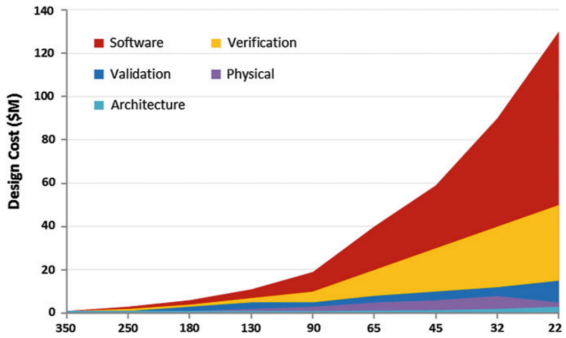


Figure 1.1: Verification cost increases as the technology node shrinks [1]

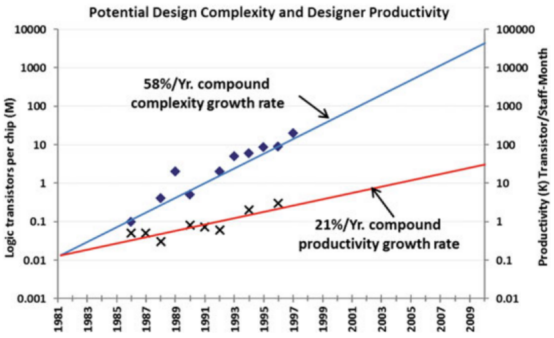


Figure 1.2: Design productivity and design complexity [1]

verify these complex designs. One way to increase verification productivity is to incorporate re-usable IPs that are standalone verified blocks. However, the introduction of IPs in design creates additional verification tests for checking the compatibility of IPs with existing design blocks [8].

Assertion-based verification (ABV) has been incorporated into the industry to mitigate this verification challenge. In ABV, assertions are introduced at various places in the design to catch bugs. The assertions can directly point to the root cause of the bugs in the design and hence reduce the time it takes to simulate, debug and verify cycles of verification stages. This is contrary to the traditional approaches where debug time is spent in tracing the bug backward from the point of failure in the circuit. Assertions provide a mechanism to describe the intent of the design unambiguously and are written in a formal language like System Verilog Assertions (SVA) or Property Specification Language (PSL). These assertions are then incorporated in the design for "pre-silicon" design verification. Also, the assertions can be synthesized into formal checkers, which helps in post-silicon debugging and run-time monitoring of hardware design. The subsequent section briefly describes the research done to generate assertions automatically.

1.1 Automating Assertion Generation from Specifications

In order to improve the verification stage productivity, much research has been conducted to generate assertions from specification documents automatically. Automatically understanding design specifications and translating them to assertions can speed up the ABV framework and improve the overall productivity of the IC design cycle by reducing the number of design errors and by early identification of incomplete and inconsistent specifications [9, 10]. In the following, this research is presented in two broad categories: (1) Semi-automated work begins by manually processing the natural language assertions to a structured language or model, and then these models are processed by the system to generate verification tests; (2) Natural language based automation research begins with natural language assertions in the document and translates them to assertions in a formal language.

1.1.1 Semi-Automated Approaches

Semi-automated approaches rely on manual interaction to translate natural language specifications into an intermediate form, which is easier to process. In [11], a semi-automated method is proposed to translate specifications written in English to System Verilog code. In this approach, specifications written in unstructured English text are manually translated to a Textual Normal Form (TNF). This TNF is automatically translated to UML class diagrams, which are manually annotated and converted to classification trees. The executable test bench for the embedded real-time systems is then automatically generated from these classification trees. Recent work on automatic formal code generation in [12, 13, 14] manually translates specifications to a formal modeling language like UML, SysML, etc. These formal

models represent specs unambiguously and can be automatically translated to hardware assertions but involve the time-consuming process of manual creation of models. Similarly, in [15], verification tests are generated from UML specifications instead of specifications written in a natural language. These formal models represent specs unambiguously and can be automatically translated to hardware assertions but involve the time-consuming process of manual creation of models.

1.1.2 Natural Language Based Automation

In contrast to manually created models, natural language based automation requires specifications written in English. In [16], a template-based approach is proposed to automatically generate assertions from natural language. In template-based work, the natural language assertions are categorized based on their syntactic and semantic structures and then mapped to a System Verilog Assertion (SVA) template. However, this approach is not fully automated. It requires the assistance of a verification engineer in the creation of these templates. In [17], partial designs have been generated from specifications by associating textual patterns with corresponding design concepts. The design concepts are understood based on the presence of textual patterns in the input sentence and are translated to design entities. In [18], translation rules are manually extracted from properties defined in specification documents to generate a formal model automatically. In addition, it also synthesized LTL formulas from natural language requirements when sufficient information about the design variables is available.

In [19], a grammar is created to parse natural language expressions, and the resulting parse tree was translated to VHDL snippets. In [20, 21], an attribute grammar is constructed, which represents vocabulary and linguistic variations of the corpus containing assertions

written in English. A parser based on this grammar enables the generation of formal assertions from natural language properties specified in the corpus. However, a new grammar has to be constructed manually for another design specification document with a different writing style. The work in [22] aims to alleviate the manual creation of the grammar by automatically learning the grammar using a training set of natural language assertions and their corresponding formal representations.

1.1.3 Problems with Natural Language Based Translation

The existing approaches described above take inputs from specification documents to automatically generate assertions. However, specifications in documents can be inconsistent, ambiguous, and even erroneous [23].

Existing approaches lack concrete solutions to handle the following inherent problems associated with the automatic understanding of natural language specifications.

- **Many-to-Many Mappings**: Existing works are based on an underlying assumption that we can unambiguously derive semantics from syntactic parse trees. For example, in [18, 24] grammatical dependencies extracted from syntactic parse tree [25] are functionally mapped to semantic relations to derive the meaning of the overall sentence. However, in reality, we can derive more than one semantic relation from a syntactic parse tree even if the sentence is syntactically unambiguous [26]. The automatic translation from natural language is actually an m-to-n relation mapping between the space of natural language specifications and the space of formal design code.

- **Context-based Understanding**: Existing approaches in [24, 27, 28, 29] lack the integration of domain specific context knowledge in their translation mechanism. A word in a specification can have different meanings in different contexts. For example,

the word "read" can have the following three meanings in three different contexts:

1. "read" as a command

   Spec: write command can follow a read only after 2 cycles.

2. "read" as a verb

   Spec: The bit is cleared when the data is read from the register.

3. "read" as a channel

   Spec: master must issue barrier transaction on read.

We need a mechanism to accurately identify the context of each word in a spec. and compositionally combine the contextual meanings of each word to create an overall meaning of the specification. However, the existing approaches in [16, 18, 24, 29] rely on syntactic parsers that were trained without contextual knowledge to create parse trees for understanding specifications. As pointed out in [30], syntactic parsing of specifications in the absence of domain-specific context knowledge fails to resolve ambiguities like preposition and coordination attachments in syntactic parsing. Parse trees with incorrect attachments between words are ineffective for any downstream natural language understanding application.

- **Ambiguous and Incompleteness**: The main source for the difficulties in validating a design is the ambiguous and incomplete nature of natural language specifications [31]. Automatic translation of ambiguous and incomplete specs may lead to incorrect translations. The output of a translation mechanism is a non-intuitive formal language, and a user may not validate its correctness unless the output is executed. Hence, a translation mechanism must detect and report ambiguous and incomplete specifications instead of translating them.

To circumvent this problem of parsing ambiguous natural language, the industry has

adopted a Controlled Natural Language (CNL) to write specifications that can be processed automatically by the machine. In the next section, we will discuss the formal definition of CNLs and analyze the existing CNLs that are created to write specifications.

1.2 Controlled Natural Language Tools for Design Verification

Controlled Natural Language (CNL) specifications are clear, coherent, and unambiguous. However, they allow only a restricted set of vocabulary and sentence structures to write specifications. In [32, 33, 34, 35, 36, 37] CNL frameworks have shown promising results in reducing time and manual efforts by automatic generation of formal models for verifying the design.

1.2.1 CNL Classification

CNLs are a subset of natural language with a restricted vocabulary and grammar. Restriction on CNL grammar and vocabulary enables the machine to efficiently and accurately process the specifications written in CNL.

A CNL with the most restricted vocabulary and grammar is close to a formal language that can be easily understood by the machine. But it becomes difficult for an engineer to express their design intent in this CNL. On the other hand, a CNL which has a large vocabulary and a wide range of syntactic variations in grammar is closer to a natural language that is difficult to process and compile by machines. We need a CNL with just enough restrictions to get the best of both natural language and formal language.

In [38], a formal classification scheme known as PENS is proposed, which quantifies this proximity of CNL to natural and formal language. This scheme defines four dimensions for a CNL. It assigns value to each of these dimensions on a scale of one to five. An extreme score in each dimension is a characteristic of either a formal or natural language. So based on the scoring on each dimension, we would know if our CNL is inclined more towards a formal language or a natural language. In the PENS scheme, these four dimensions correspond to the four properties of CNL, which are: Precision (P), Expressiveness (E), Naturalness (N), and Simplicity (S).

A brief definition of these four properties of CNL used in PENS classification is given below:

1. Precision: The Precision property refers to the degree of meaning that can be directly retrieved from the CNL text. Natural language is imprecise and has the lowest score of precision since a large amount of contextual information is required to understand the meaning of the sentence written in natural language. This imprecise nature of natural language is indicated by assigning a score of 1 to the Precision property of natural language, which is represented as P^1. The meaning of formal language is strictly based on the textual input symbols and hence gets a score of 5 on the precision property, which indicates a very high precision.

2. Simplicity: Simplicity describes the efforts needed to implement the syntax and semantics of the language in a mathematical model. To quantify the indicator of simplicity, PENS considers the count of pages needed to describe all the syntax and semantics of a language. Natural language has a score of S^1 since its grammar cannot be described in an exact and comprehensive manner. On the other hand, the formal language has a score of S^5 as its syntax and semantics can be described in an exact and comprehensive manner on a single page.

3. Naturalness: This property describes whether we can read and understand a CNL like a natural language. For example, the CNLs that employ a lot of unnatural words like brackets and special characters cannot be read like a natural language.

4. Expressiveness: Expressiveness score describes the range of propositions that can be expressed in a certain language. Based on the type of expressions that can be expressed in a language, the PENS scheme assigns an expressiveness score from 1 to 5. In the PENS scheme, propositional logic is assigned a score of 1 since it cannot express predicates with relation arguments greater than 1. Description logic has a score of 2. First-order language has a score of 3. Languages that can express second-order universal quantification over concepts and relations are assigned a score of 4. Natural language has a score of 5 since it can express anything that can be communicated between human beings.

We will describe some examples of CNL frameworks with varying scores of PENS next.

1.2.2 Domain Specific CNL Tools for Verification

It is shown in [2] that we can achieve a balance between the ease of expression and precision of meaning while writing specifications in CNL. Also, it has been shown that a machine-processable CNL can detect erroneous specifications in the early design stage, which has the advantage of **reduced time-to-market for IC and increased design productivity**. This has been the motivation for the creation of CNL tools for verifying software and hardware designs. A brief description of the latest domain-specific CNLs that assist in generating validation tests is given below:

- Template Based Natural Language Specification (TBNLS) [39] : TBNLS is a CNL to test control software of passenger vehicles. There are 15 templates in this CNL that

map to propositional logic. An example sentence of this CNL is given below:

'if Button B_4 is down P_1 occurs, then Lamp L_3 is red P_2 holds immediately, until 10 seconds T_1 elapsed.'

In the above sentence, P_1 and P_2 are propositional variables, and T_1 is a time variable.

PENS score : P^5, E^2, N^3, S^4.

1. **Precision score is P^5** . This indicates that each text has exactly one meaning associated with it. The input sentence has no lexical, syntactic, or semantic ambiguity.

2. **Expressiveness score is E^2** . This means that it can handle predicates with arguments greater than 1. For example, in the above sentence, the phrase 'Lamp L_3 is red ' is a relation with an argument list greater than 1. But it cannot handle the expressiveness of first-order logic.

3. **Naturalness score is N^3**. The naturalness property is less because the sentence uses propositional variables and time variables in the sentence. Because of these variables, we cannot read the sentence as a natural language.

4. **Simplicity score is S^4**. The CNL consists of only 15 templates describing this language's syntax and semantic rules. These syntactic and semantic rules of the language can be described in an exact and exhaustive way in less than ten pages. Hence, it has a simplicity score of 4. This means that a linguist can implement the grammar in a reasonable amount of time.

- CNL for medical device interaction [34]: Design properties and safety-critical requirements are written in Gherkin CNL, which are then translated to test cases and run-time monitors.

  The following is an example scenario for testing the number entry system for medical

devices written in Gherkin CNL:

1. Scenario: Going beyond left boundary
2. Given cursor is on leftmost position
3. When left is pressed
4. The cursor position stays the same
5. And Displayed number stays the same
6. And the user is alerted

The Gherkin CNL sentence should start with a keyword like 'Given','When','AND','BUT'.
PENS score : P^5, E^3, N^4, S^3.

1. **Naturalness score is N^4** . It has a high value since it does not use unnatural words in the input sentence.
2. **Expressiveness score is E^3** . This value is high as compared to the earlier shown TBLNS CNL. It has expressiveness equivalent to first-order logic. An increase in expressiveness increases the parsing complexity [40]. Instead of devising a sophisticated parsing algorithm, this CNL relied on regular expression patterns to understand the sentences and map them to a formal language. However, they cannot deal with ambiguity in the input sentence.
3. **Simplicity score is S^3** . The use of regular expression lowers the simplicity score of the language since Gherkin CNL doesn't use formal grammar approaches to model the syntactic and semantic rules efficiently. When we increase the language's expressiveness, we require many patterns to map the sentences to a formal language. Hence, it creates a lot of syntactic and semantic rules, which lowers the simplicity score of this language.

4. **Precision score is $\mathbf{P}^5$** . It expects the input sentence to be complete and sufficient for successful translation to formal logic. It does not rely on reasoning based on some axioms to understand the input sentence. Hence, the precision of meaning is high.

- NASA FRET [37]: FRET is a CNL tool to translate specifications to a formal language. This tool uses six semantic categories and requires input properties to be concise. This tool uses English templates to translate the input natural language specifications to CoCoSpec code [41].

  The specifications in this tool are expected to follow a strict sequential order of semantic categories. This linear order of category is shown below:

  Scope,Condition,Component,'Shall',Timing,Responses

  The word 'Shall' is a keyword used to separate semantic categories. An example requirement is 'When in roll_hold mode Autopilot shall always satisfy overshoot <=0.1'. Sequential restrictions on categories make this CNL very restricted for writing specifications in English.

PENS score : $\mathrm{P}^5, \mathrm{E}^2, \mathrm{N}^3$. We cannot comment on the exact score of Simplicity.

1. **Naturalness score is $\mathbf{N}^3$** . It cannot use English words to represent Boolean conditions. For example, it has to use '<=' in the above example sentence instead of saying 'less than or equal to'. However, there is more dominance of natural words in the sentence as compared to unnatural words. Hence, the naturalness score is three.

2. **Expressiveness score is $\mathbf{E}^2$** . It uses English templates to map sentences to formal language. Its expressiveness is defined by the allowed template structures.

3. **Precision score is P^5** . It expects the input sentence to be complete and sufficient for successful translation to formal logic. It does not rely on reasoning based on some axioms to understand the input sentence.

4. **Simplicity Score is Unknown**. The simplicity score will depend on the domain's size and the language's level of expressiveness. In this language, syntactic and semantic rules are represented by templates. We need to know either the count of templates or a formal model of the syntax of this language to decide its simplicity score.

- General Electric ASSERT tool [42]: ASSERT is a tool to capture requirements in a controlled natural language and automatically generate test cases. ASSERT is used to write requirements for a flight management application, an avionics network switch program, an aircraft engine health monitor program, and a safety controller for a power system. ASSERT detects specification errors early in the design process.

  The requirements written in this CNL follow a strict linear order of keywords and properties. The requirement consists of three lines, as shown in Figure 1.3. The first line describes the name of the requirement, which is a unique identifier. The second line is called the conclusion, which is a statement that must be true. The third is called a condition which shows a condition when this requirement can be applied. The variation of word order cannot be handled in this framework.

  PENS score : P^5, E^2, N^3. We cannot comment on the exact score of Simplicity.

  1. **Naturalness score is N^3** . The assert tool uses unnatural words to indicate which word is not a keyword in the sentence. For example, in this sentence 'Conversion(4.5,v̂alue of raw Value of channels of Px, -13.2)', they have used

v̂alue to interpret 'value' as an English word instead of a keyword. Also, they have used brackets to write arguments of the Conversion function. The above sentence cannot be read as a natural language. However, in this language, natural words are in the majority compared to unnatural words; hence, the naturalness score is three.

2. **Expressiveness score is $\mathbf{E}^2$** . ASSERT uses English templates to map sentences to a formal language. Its expressiveness is defined by the allowed structures of a template. It can handle predicates with more than one argument in a relation.

3. **Precision score is $\mathbf{P}^5$** . It expects the input sentence to be complete and sufficient for successful translation to formal logic. It does not rely on background knowledge to understand the input sentence.

4. **Simplicity score is not known**. For reasons similar to NASA FRET, we cannot give a specific simplicity score to this CNL.

```
Requirement LowRangeLimit1:                    // name
SYSTEM shall set lowRangeLimit of Px to 1.0    // conclusion
when engineSpeedPercentage <= 30.0 .           // condition
```

Figure 1.3: A formal requirement written in ASSERT CNL [2]

1.2.3 Problems with Current Domain Specific CNLs

In the above section, we presented four different CNL tools that are used to generate validation tests. All the CNL tools use English templates to understand the sentence. Also, none of the above tools allow for variations in the order of words that appear in a sentence. The specification in these CNLs should be written in a specific sequence of semantic and

syntactic categories. The following are the major problems that exist in domain-specific CNLs.

- **Lacks formalism that allows word order and sentence structure variations:** A user has to follow a strict sequence of categories to write the input English sentence in a CNL. They have to remember the syntax of the CNL; in the same way, they follow the strict syntax of a formal language. This hinders the user from expressing the design intent with ease.
- **Missing feedback on writing syntactically and semantically correct specifications:** A sentence meaning can be expressed through different sentence structures and order of words. However, a semantic parser can parse only a subset of words and sentence structures that are defined in the CNL grammar. Hence, a CNL framework should provide feedback to assist users in writing CNL-acceptable sentences.
- **Lacks mechanism to detect and report ambiguity and incompleteness:** As mentioned earlier, a highly expressive CNL is closer to natural language. An expressive CNL must detect ambiguities and incompleteness in specifications. We need to have a mechanism to inform the user about the ambiguity in the input sentence that cannot be translated accurately.

1.3 Contributions

The contributions of this thesis are as follows:

1. We propose a grammar-based framework called BINGO to understand specifications written in the English language. The framework comprises a grammar formalism, parser, and a module for rewriting and suggestion generation.

2. Grammar Formalism: In this framework, we have proposed a dependency grammar formalism inspired by the Link grammar [43]. The grammar is not as rigid as the existing CNL grammar and allows flexibility in word order variations and input sentence structures. The grammar creation in this work is analogous to creating BINGO charts for each word. The rows in the BINGO chart represent distinct interpretations allowed for a word. The columns in the BINGO chart describe the constraints required to complete an interpretation of a word in a specific context.

3. Contextual-based Parsing: A chart parser is proposed in the framework. The parser considers the context on both the left side and right side of a word before making a dependency arc on the word. The proposed chart parser is modeled as a game of BINGO where the BINGO chart of each word is scanned to find a horizontal BINGO row.

   We have also proposed a mechanism to detect ambiguous and incomplete specifications within the BINGO model. The proposed detection framework can identify ambiguous and missing information by analyzing the state of BINGO charts after the completion of parsing.

4. Automated Rewriting and Suggestion Generation: We propose a framework that can automatically generate suggestions by rewriting the input specification into CNL acceptable sentences with semantics closer to the user's intent. We have developed a ranking mechanism to rank a list of suggestions based on their syntactic and semantic similarity with the input sentence that was not understood.

1.4 Chapter Structure

The structure of the thesis is as follows:

Chapter 2 presents background information related to different components of our proposed framework. The chapter begins with a brief description of four existing grammar formalisms. We then introduce common dependency parsing algorithms and issues pertaining to dependency parsing. In the next section, the existing work on domain-specific natural language understanding is presented. Finally, in the end, the chapter presents a review of papers on generating suggestions and rewriting techniques for natural language sentences.

Chapter 3 introduces a node data structure that stores attributes of each word in the specification. The data structure allows compositional analysis of the meaning of each word enabling the creation of the overall meaning of the sentence. In this chapter, we present basic terminologies of the data structure that will assist in explaining the framework in subsequent chapters.

Chapter 4 presents the components of our BINGO framework. We present the structure of our grammar rules and explain different semantic propagation techniques allowed in the grammar. Next, we present the representation of rules in a chart. We then explain our chart parsing algorithm that finds linkages to connect all the specification words in a dependency tree. We present our semantic parsing technique that creates semantic frames based on the linkages found in the chart parsing step. We explain the use of grammar in providing flexibility in writing specifications with variable word order. At the same time, the rules in grammar allow disambiguation in the interpretation of words based on context. The framework evaluation is presented on four different types of hardware architectures. We further evaluated the framework by writing variations of specifications found in documents.

Chapter 5 presents our suggestions module. The chapter begins by explaining the partial parse trees that are generated after parsing specifications that lie outside the coverage of grammar. We present the parameters of the partial parse tree that assist in completing the parse tree. We then introduce our scheme to search for candidate words that can complete the parse tree. In the next section, we present our ranking scheme that selects a parse tree with intent closer to the input specification that was not understood. Finally, we present the evaluation of the suggestions module on specifications of AMBA and memory controller architectures.

Finally, Chapter 6 presents a concluding summary of the thesis and discusses directions for future work.

Chapter 2

Basic Concepts

Our translation framework comprises a grammar formalism, parser, and a suggestion module. This chapter introduces basic concepts related to different components of our translation framework.

2.1 Grammar Formalism

Specifications of sentence structures and meaning descriptions allowed in a language are formally defined in the grammar. Formalization of linguistic and semantic descriptions allows the implementation of computer programs that can automatically process natural language sentences. The following sub-sections briefly describe four types of grammar formalisms.

2.1.1 Phrase Structure Grammar

A Phrase Structure Grammar (PSG) is a type of context free grammars and is based on a set of rewrite rules that defines the set of allowed sentence structures in the language. Each rule in a PSG has a left-hand side that identifies a syntactic category and a right-hand side which defines the sub-components of this left-hand side category. For example, in Figure 2.1, the sentence is defined by the symbol S, which is composed of VP (verb phrase) and NP (noun phrase). A sentence in a PSG is defined by a series of words that can be derived

Grammar Rules for Constituents

S -> VP NP
VP -> V NP
NP -> N PP
NP -> N
PP -> N P

Figure 2.1: Context-Free Grammar Rules to create parse tree

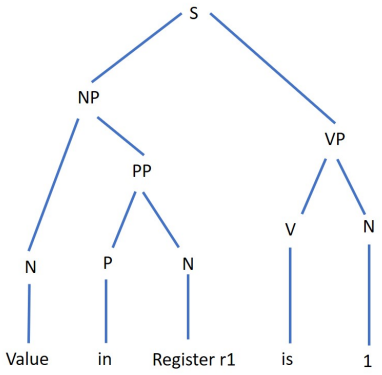


Figure 2.2: A Constituency Parse Tree of the sentence 'Value in register r1 is 1'

systematically by applying the rules starting from the rule with the symbol 'S' on the left side. Figure 2.2, shows a parse of the sentence based on the PSG shown in Figure 2.1.

2.1.2 Combinatory Categorial Grammar

In Combinatory Categorial Grammar (CCG) [44], lexical items are associated with either primitive or functional syntactic categories. Examples of some categories in CCG are:

- S: sentence, example: 'value in register is 1.'
- N: noun, example: 'register'
- PP: prepositional phrase, example: 'in register'

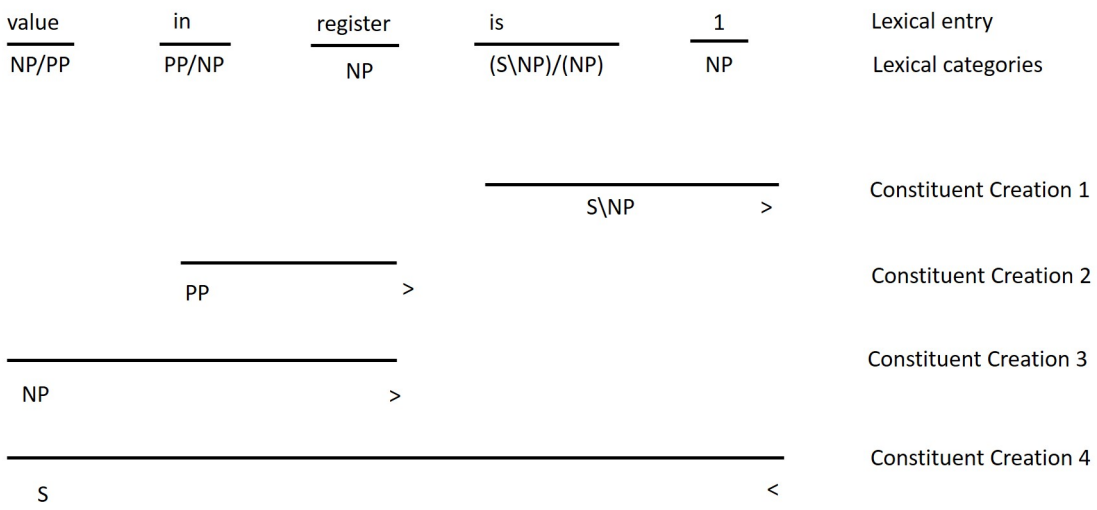


Figure 2.3: CCG derivation diagram of sentence 'value in register is 1'.

Functional Categories:

Functional categories define the process of combining constituents to form new constituents. Syntactic constituent types like determiner, verb phrase, or preposition have functional categories. Functional categories are defined as (X/Y) or $(X\backslash Y)$ where X and Y are syntactic categories. In a constituent combining process, X is the result category, and Y is the argument category.

Meaning of Functional Categories X/Y and $X\backslash Y$:

The category X/Y of a constituent implies that the constituent can be combined with another constituent of category Y on its immediate right. The resultant constituent has a category of X. In this combination of constituents to form a larger constituent, the first constituent is called 'functor', the second constituent is called 'argument', and the resultant constituent is called the 'result'.

The function category $X\backslash Y$ of a constituent (functor) shows that it can combine with a constituent with category Y (argument) on its immediate left to create a new constituent with category X (result).

For example, a preposition word 'in' is given a category of PP/NP in Figure 2.3. The category PP/NP indicates that 'in' constituent (functor) can combine with a noun phrase (argument constituent) on its right to create a prepositional phrase (result constituent) such as 'in register'.

Every constituent is associated with a list of argument slots. Each argument slot has a category and a slash direction. The constituents with the basic category have an empty list of argument slots. For example, noun category words like 'register' have no arguments defined in their category. In Figure 2.3, the category of word 'register' is NP without any arguments and slash direction.

Example CCG Derivation:

A CCG derivation of sentence 'value of register is 1' is shown in Figure 2.3. We have annotated the derivation steps with the following labels for ease of explanation.

- Lexical entry: The lexical entry shows the words in the sentence with a horizontal line underneath. The categories of these words are described beneath each word below the line.
- Lexical categories: Lexical categories of the words and their meaning are as follows:
    1. 'value': The word value has been given a category NP/PP. The category will combine with a PP argument on its immediate right.
    2. 'in' : The preposition is assigned a functional category PP/NP. The preposition will combine with a NP constituent to its right to create a PP category constituent.
    3. 'register': The 'register' has a category of NP with an empty argument list and no slash directions.

4. 'is': The verb 'is' has a category of $(S\backslash NP)/(NP)$. The first NP argument in the verb category is to create a constituent with a subject on the left side. The second NP argument creates a constituent with the object NP on the right side of the verb.

5. '1': The word '1' is assigned a NP constituent with no arguments and slash directions.

- Constituent Creation 1: The non-lexical category $(S\backslash NP)/(NP)$ combines with the NP category on its right to create a constituent with category $S\backslash NP$. As seen in the constituent creation process, the argument slot (NP) of the verb 'is' gets combined in the process and no longer appears in the resulting constituent.

  On the right side of the line, a symbol '>' or '<' is drawn to indicate the rule that creates constituents by combining child constituents. The symbol '>' indicates that we combined the functor constituent with a constituent on its right side. A '<' shows that a functor constituent was combined with an argument constituent on the left side.

- Constituent Creation 2: The category of 'in' has a NP argument with the resulting category of PP after constituent creation. The PP/NP constituent is combined with the NP constituent on the right side to create a resulting constituent of PP.

- Constituent Creation 3: The 'value' has a category of NP/PP that leads to a NP constituent after combining with a PP constituent on its right side.

- Constituent Creation 4: Finally, the constituent $S\backslash NP$ combines with the NP constituents on the left side and creates a constituent with a category S.

The derivation process ends when all the arguments are combined, and a final constituent with a category 'S' is created.

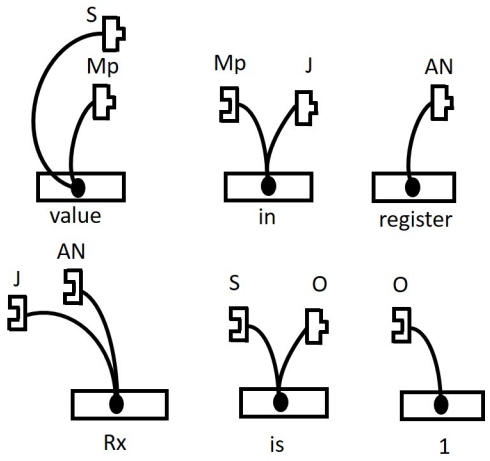


Figure 2.4: Link requirements for a set of words.

2.1.3 Link Grammar

The Link grammar [45] defines the linking requirement for a set of words. Each word in the grammar is associated with linking requirements. Figure 2.4 illustrates an example of linking requirement for words 'value', 'in', 'register', 'Rx', 'is' , and '1'. The linking requirement of each word is shown by a diagram above the word. The labeled boxes in the diagram are called links. For example, 'J' and 'AN' are links for the word 'Rx' in Figure 2.4. Satisfaction of the link rules of a word is achieved by connecting compatible links together. The compatible links have the same shape and opposite direction that allows two combined links to form a rectangle box, as shown in Figure 2.5.

Formally, the link requirements of each word are denoted by a formula as shown in Table 2.1. The link requirement formula consists of operators '&', 'or', parentheses and connector names. The direction of links in the formula is defined by '+' and '-' signs.

The 'or' between two formulas for a word's link requirement indicates that either one of the formulas has to be satisfied to complete the link requirement of the word. For example, in Table 2.1, the formula for 'Rx' is '(AN- & J-) or O-'. To satisfy the link requirement of the

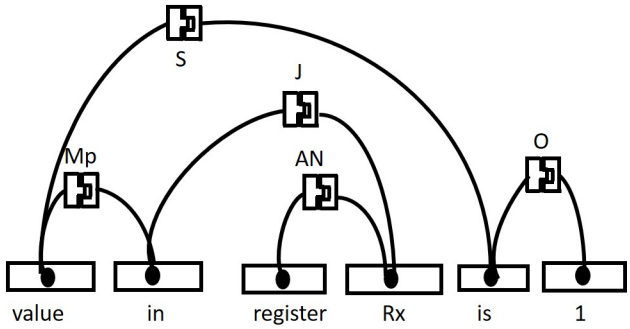


Figure 2.5: Link requirements satisfied for a sequence of words.

word 'Rx', one of the formulas connected by 'or' has to be satisfied.

The '&' operator of two formulas indicates that both formulas have to be satisfied to satisfy the overall formula.

A sentence is defined in a link grammar if there exists a way to connect links of all words in the sentence that satisfy each word's formula and the following conditions:

- Planarity: The links connecting the words in a sentence should not cross each other.

- Connectivity: All the words in the sentence should be connected together by links.

- Ordering: When a connector of the formula is traversed from left to right, then the words to which they connect proceed from near to far. For example, the formula for the word 'value' in Table 2.1 is 'M_p+ & S+'. The ordering rule implies that the word connecting to the link 'M_p' should be nearer to the word 'value' than the word that will connect to 'S+' link of the word 'value'.

- Exclusion: No two links can connect the same pair of words.

Table 2.1: Link Grammar

Words	**Formula**
value	M_p+ & $S+$
in	M_p- & $J+$
register	$AN+$
Rx	($AN-$ & $J-$) or $O-$
is	$S-$ & $O+$
1	$O-$

2.1.4 Dependency Grammar

In Dependency Grammar (DG), the syntactic structure of a sentence consists of asymmetrical binary relations between the words of a sentence. The head word in this asymmetrical relation is called the governor, and the child word is called the dependent. The dependent is usually a modifier, object, or a complement of the head word. For example, if we have a phrase consisting of words $w = w_1, w_2, ...w_i$ then the dependency between two words w_i and w_j is a directed edge represented like $w_i \rightarrow w_j$ in the graph. This edge exists only if the word w_j depends on the word w_i. In Figure 2.6, the edge between 'is' and 'value' encodes the syntactic information that 'value' is the subject of 'is'. Grammar driven dependency parse trees should adhere to the following structural constraints:

- There should be no w_i for which we have an edge $w_i \rightarrow w_i$
- For every w_k there is at most one w_j such that $w_j \rightarrow w_k$
- The dependency tree should have only one word, which is not dependent on any other word. This word is called the root of the sentence.

One of the first DG formalisms was presented in [46] , [47] by Hays and Gaifman. In the Hays and Gaifman DG formalism, dependency rules that capture all the dependents of 'x' are represented as follows:

1. x($w_1, ..., *, ..., w_k$): $w_1 ... w_k$ are dependents of x
2. x($*$): x is a leaf node
3. $*$(x): x is a sentence root node

The '$*$' represents the position of x in the linear order of words $w_1 ..... w_k$. For the sentence 'Value in register r1 is 1', we will have the following grammar rules based on this DG formalism:

1. $*$(V): The verb is the root node. In this example, the root node of the sentence will be 'is'.
2. V(N,$*$,N): The Verb 'is' is the head of Noun('value') and Noun('1'). The linear position of the verb is denoted by $*$.
3. N($*$,prep): The Noun 'value' is the head of the preposition 'in' when the linear position of the noun comes before the preposition.
4. prep($*$,N): The preposition 'in' is the head of the Noun 'register r1' when the Noun 'register r1' comes after the preposition.
5. N($*$): The Noun '1' is a leaf node.

Generative Dependency Grammar

One of the consequences of parsing natural language is that a sentence may have a large number of parse analyses [48]. We cannot efficiently evaluate every analysis and determine a correct parse for the sentence. In Context-free Grammar (CFG), the analysis of all possible parse trees is made tractable by encoding multiple parse trees in a compact representation called a shared parse forest. A shared parse forest for a CFG grammar allows a polynomial-size representation for an exponential number of parse analyses [49].

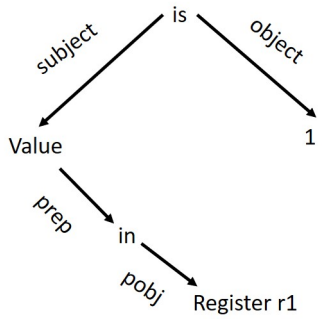


Figure 2.6: A Dependency Tree of the sentence 'Value in register r1 is 1'

In [48], a dependency grammar formalism called Generative Dependency Grammar (GDG) for encoding shared parse forest is proposed. GDG is a generative string-rewriting formalism created by extending Gaifman's [46] grammar notation with regular expressions. A GDG is a lexicalized extended context-free grammar with 4-tuple (V_N, V_T, P, S), where:

- V_N is a finite set of non-terminal symbols.
- V_T is a finite set of terminal symbols.
- P is a finite set of rules, which are ordered pairs consisting of an element of V_N and a regular expression over $V_N V_T$.
- S, a subset of V_N, contains the possible start symbols.

GDG grammar rules for representing the sentence 'Pilar saw a man with a telescope' as taken from [48], are shown below:

- The grammar, G_{Ling} is (V_N, V_T, P, S)
- $V_N = V, N, D, A, Adv$. These terminal symbols are Verb, Noun, Determiner, Adjective, and Adverb.

- V_T = Pillar, saw, man, a, telescope, with, tall, very . These are the lexical items in the grammar.
- P is a set of the following rules:

  p_1: V $\rightarrow$ N saw N P^*

  p_2: N $\rightarrow$ (Pilar | D (A) (man | telescope) P^*)

  p_3: P $\rightarrow$ with N

  p_4: A $\rightarrow$ Adv^* tall

  p_5: Adv $\rightarrow$ very

  p_6: D $\rightarrow$ a
- S = V

2.2 Dependency Grammar-based Parsing Algorithms

In our work, we have employed dependency grammar and created algorithms to parse and understand the sentences based on this grammar. Hence, in this section, we will discuss parsing algorithms based on a dependency grammar.

2.2.1 Exhaustive Search-based Dependency Parser

In search-based algorithms, we start scanning words from the beginning of the sentence and try linking words as head or dependent to all previous words. For example, Figure 2.7 illustrates this algorithm on the sentence 'value in register r1 is 1'. In this sentence, the algorithm begins by picking the word 'in'. Then we search in previous words for the dependents or head of 'in'. In this example sentence, there is only one word 'value', that precedes 'in'. Based on the grammar rules, the word 'value' is connected as the head of 'in'.

Sentence -> Value in register r1 is 1

Picked word	in	Decision
Previous word	value	Connect 'in' as dependent to 'value'
Picked word	register r1	Connect 'register r1' as dependent to 'in' , since the preposition `in' connects the subject `value' with object `register r1'
Previous word	in	
Picked word	is	Cannot Connect `is' as head of 'register r1' because `register r1' already has head word 'in'.
Previous word	register r1	
Previous word	in	
Previous word	value	Connect 'is' as head to 'value'
Picked word	1	
Previous word	is	Connect '1' as dependent of 'is'

Figure 2.7: Step-by-step description of search-based algorithm of dependency parse tree creation. A parse tree is created for the sentence 'Value in register r1 is 1.'

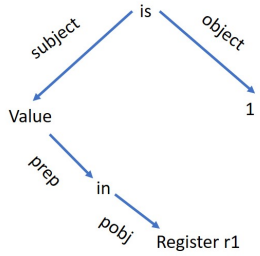


Figure 2.8: Resultant Dependency parse tree of the sentence 'Value in register r1 is 1'.

While connecting a head node and its dependents, it is important to ensure that the dependency tree constraints are satisfied. For example, in Figure 2.7, let us consider the parse configuration where we have to decide the connection between the words 'register r1' and 'is'. The word 'is' can be connected as the head of the word 'register r1'. But according to the dependency tree constraints, a word can have only one head node. Hence, we cannot connect 'is' as the head of the word 'register r1' because the word 'register r1' already has a head node 'in' connected to it.

The word 'in' is connected as the head of the word 'register r1' since it connects the subject 'value' with the object 'register r1'. However, it is not mentioned in the paper [50] how they encode this subject-object connection through a preposition in a grammar.

We continue to scan previous words until we encounter the word 'value'. The word 'value' has a dependent node 'in' but has no head node. The word 'is' can be connected as the head of 'value'. This connection does not violate any structural constraints of the parse tree. The

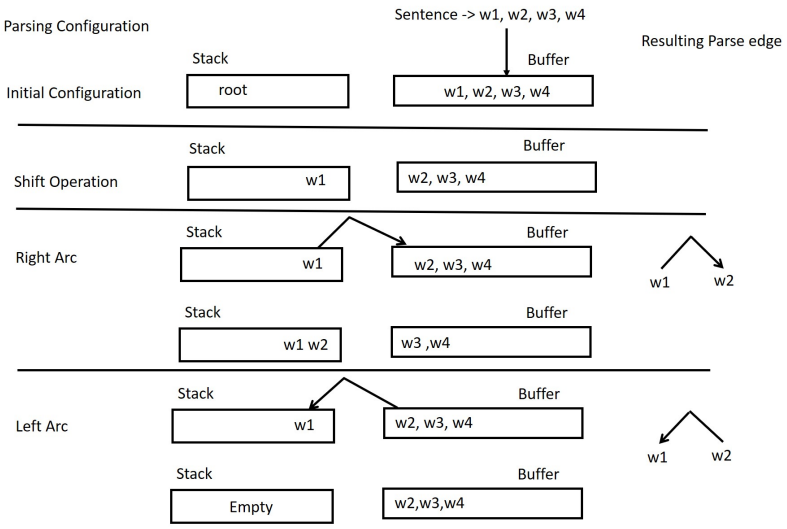


Figure 2.9: Action performed in each parsing configuration of a sentence based on Arc-eager dependency algorithm.

final dependency parse tree is shown in Figure 2.8.

2.2.2 Arc-Eager Dependency Parsing Algorithm

The steps of this algorithm [51] are similar to the shift-reduce algorithm of context-free grammar. However, the actions performed in these steps differ from the shift-reduce algorithm of CFG. The parsing configuration consists of a stack and a buffer. The parsing is initialized by appending all input words of a sentence in the buffer. A dummy root node is added to the stack in the initial parsing state.

The edge is created between the first element of the buffer (buffer head) and the top element of the stack (stack top). The following operations are performed in the parsing based on the type of words in the buffer head and stack top.

- Shift: In this operation, we move the first word in the buffer to the stack top. This action is performed when we do not find a direct dependency between the word in the

stack top and the first word in the buffer.

- Create Left arc (larc): Create an edge between the first word in the buffer and the word in the stack top. The first word in the buffer becomes the head node of the word in the stack top. This edge is allowed only if the word in the stack top does not have a head. This preserves the dependency rule that requires a maximum of one head word for each word in a sentence. Also, after creating the edge, we will remove the stack top word from the stack.

- Create Right arc (rarc): Create an edge between stack top word and buffer head word. In this edge creation, the stack top is the head of the first word in the buffer.

- Reduce: We remove the word in the stack top when the word has no more dependent words in the buffer. This action is performed only when the word in the stack top is connected to a head word.

Figure 2.9 illustrates the action performed in each of these four operations.

Figure 2.10 shows the working steps of the Arc-Eager parsing algorithm on the sentence 'The value of 1 is assigned to register r2'. Parsing begins with the creation of a buffer and a stack. All the input words are added to the buffer, and a dummy root node is added to the stack, as shown in step 1 of Figure 2.10.

The dummy word root will be connected to the main verb in the sentence. In step 2, a shift operation is performed since the root and the word 'the' have no dependency. The shift operation does not create any edges in the parse tree, and hence no parse tree was shown in this step of the algorithm.

The word 'the' is a determiner of the noun 'value', and hence 'the' is connected as the dependent of 'value'. This relation is created using a 'larc' with 'value' as the head and 'the'

as the child node. The parse tree of this edge creation is shown in step 3.

When we create a left arc (larc), the child node of the larc edge is removed from the stack after the edge creation. Step 3 showed the removal of the child node from the stack. In this step, the word 'the' gets removed from the stack after the edge creation.

In step 4, since the root and 'value' are not connected, we will shift the word 'value' into the buffer. The decision of whether two words are connected or not is fetched from the grammar.

In step 5, 'value' is made head of the word 'of' because the word 'value' is a noun and precedes a preposition 'of'. In step 6, the preposition 'of' precedes noun '1', and due to this linear order, the preposition becomes the head of the noun '1'. In a practical application of natural language understanding, we cannot rely only on the linear position of the words to make an edge. Especially for coordinate conjunctions like 'and', 'for', we need to do additional analysis to decide the semantic role of the words before making an edge.

In the reduce operation, we remove the word from the stack when it doesn't have any more connections with the words in the buffer. For example, in step 9, we remove the words 'of' and '1' from the stack because they don't have any more dependents in the buffer.

The parsing is finished when the buffer becomes empty. The final parse tree is shown in step 14 of Figure 2.10.

The above two parsing algorithms assume that all the dependencies are clearly defined in the grammar. Because natural language is ambiguous, we need to perform additional analysis to resolve ambiguity during parsing. We need a parsing algorithm that can handle the scenarios when the grammar does not have specific dependency rules for words in the stack top and buffer head. Also, the parser should report errors to the user when it cannot find dependency edges for words.

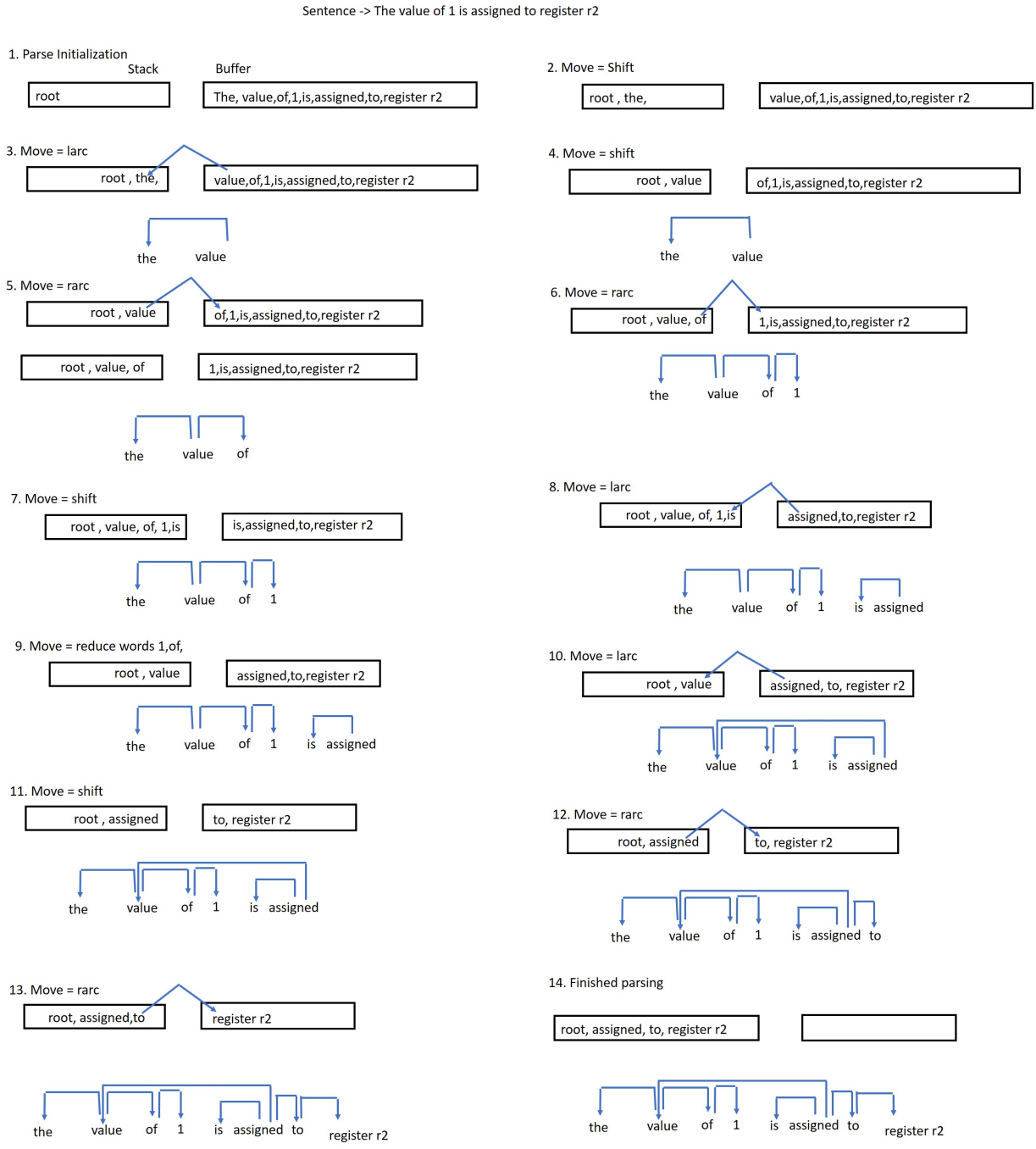


Figure 2.10: Working of arc-eager parsing algorithm on the sentence 'The value of 1 is assigned to register r2'. The resultant parse tree after each step of the algorithm is also shown.

2.3 Issues in Dependency Parsing

It is shown in [4, 52] that the parsing performance degrades as the length of the sentence increases. This reduction in the accuracy of parsing complex sentences is attributed to the presence of conjunction category words in the sentence. Dependency parsing of conjunction suffers from two main problems: (1) Representing dependency structure of conjunction relation in the sentence. (2) Identifying the conjuncts and disambiguating the role of conjunction.

2.3.1 Dependency Tree Representation of Conjunction

There is no unique way of representing a dependency relation of conjunction structures in a sentence. This is due to the fact that the general schema of dependency parsing which includes the connection of head and modifiers cannot be used to represent conjunction constructs [4]. For example, Hudson's word grammar [53] which is a dependency grammar formalism, characterizes conjuncts as word strings instead of a dependency structure connected with the rest of the sentence. In word grammar, conjuncts are internally organized as dependency structures and the root of conjunct is a word which is external to the conjunction relation.

As can be seen in Figure 2.11, the entire conjunction is considered as a word string inside curly brackets. In this figure, the root of the conjuncts is external to the conjunction relation. In Figure 2.11, both 'stools' and 'chairs' are conjuncts. These conjuncts are dependent on the word 'on', which is external to the conjunction relation. Similarly, the word 'sit' is the head of words 'now' and 'later'.

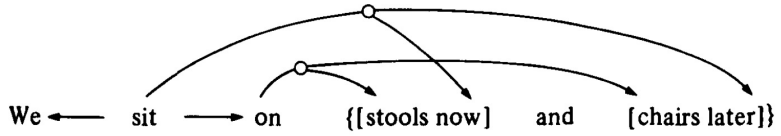


Figure 2.11: Treatment of conjunction in Hudson's English Word Grammar [3]

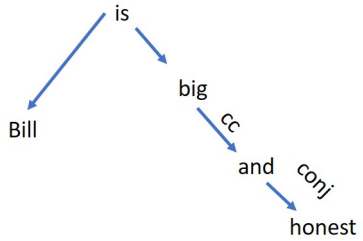


Figure 2.12: Syntactic dependencies in coordinate conjunction in Melcuk's theory

In Melcuk's analysis of conjunction, [54], one of the two conjuncts in the coordination relation is considered the head of the conjunction structure, and the conjunction element itself is the head of the other conjuncts. This is shown in Figure 2.12.

An example parse tree based on the Stanford dependency manual [55], is shown in Figure 2.13, where the conjunct 'big' is the head of the conjunction 'and' and the conjunct 'honest'.

2.3.2 Dependency Parsing of Sentences with a Conjunction

The difficulty in parsing long sentences arises because the conjunction category word can have various roles in a sentence. For example, it is shown in [56] that a comma in the English

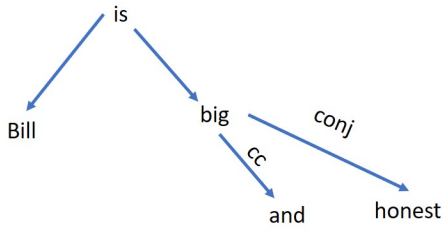


Figure 2.13: Treatment of conjunction based on Stanford dependency manual

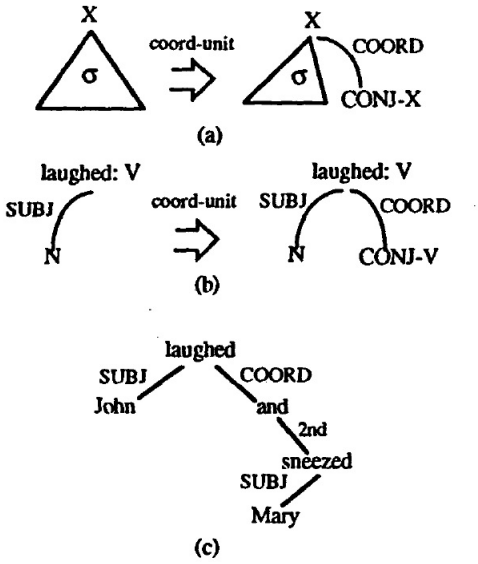


Figure 2.14: Metarule application on partial dependency parse tree due to the presence of conjunction 'and'. The parse tree is shown for the sentence 'John laughed and Mary sneezed'. [4]

language can have more than ten different roles. We may get different parse trees depending on the role played by the conjunction word in the sentence. There are two strategies adopted by the research community to deal with conjunctions.

The first strategy handles conjunctions by doing extra-grammatical analysis. Such an approach is followed in [57, 58, 59]. Proponents of this strategy believe that the conjunction is difficult to model using the same formal tools that are used to describe the conjunction-free part of the language. According to [58], analysis of conjunction should be treated as a parsing problem. For example, in [57], multiple parsing of the sentence is performed to deal with linguistic phenomena arising from conjunction. In [57], parsing steps were interleaved with special processing steps for conjunction elements. The decision to process conjunction is postponed till sufficient information about conjunction is parsed.

The second strategy to deal with coordinate sentences is to add specific syntactic rules in the

grammar that can detect different patterns of conjunction structure. For example, the work in [4] employs metarules on the existing dependency rules to modify dependency trees that have conjunction category words. This application of metarules to the dependency parse tree is shown in Figure 2.14. This figure shows the parse tree for the sentence 'John laughed and Mary sneezed'. In Figure 2.14 (a), metarule 'coord-unit' is applied on the parse tree since it contains conjunction 'and'. Figure 2.14(a) is a metarule that shows the processing of conjunction when one conjunct is the head of the parse tree. According to this rule, we will extract a word with a category similar to the first conjunct. This word is extracted from the right dependents of the first conjunct. The extracted word and the conjunction are then connected to the head conjunct node.

This application of the rule is described by an example in Figure 2.14(b) and (c). The head of the parse tree is a conjunct 'laughed'. Since this conjunct is of verb category, we will extract the verb category word and the conjunction word from the right dependents of the word 'laughed'. Then we will connect these extracted words with the word 'laughed'. This connection is shown in Figure 2.14(c).

The work in [4] extracts conjuncts from the conjunction relation without explaining how these conjuncts were identified in the first place. It has been shown in the research that finding conjuncts in a sentence is not a trivial task [60, 61].

In our work, we use domain-specific knowledge along with syntactic and semantic analysis to understand the conjunction relation and then translate this relation into a semantic expression.

2.4 Domain Specific Natural Language Understanding

In this work, we have built a grammar-based framework to understand English specifications for a specific domain. In this section, the existing frameworks for domain-specific natural language understanding are presented. These frameworks take spoken language queries as their input. An example query is: 'How many American Airline flights leave Denver on June tenth'.

2.4.1 Phoenix

Phoenix is a natural language understanding system used in CMU's spoken language understanding system [62]. Phoenix uses semantic frames to parse and understand the natural language utterances.

The frames contain slots, which are semantic entities. For example, 'depart loc' and 'arrive loc' are slots in a frame. This system parses the semantic fragments instead of a complete sentence. These semantic fragments are identified in the sentence using word patterns. Phoenix uses Recursive Transition Networks to define the word patterns corresponding to each slot in the frame.

The parser operates by searching for word patterns in the input text. A word pattern may be assigned slots that exist in multiple frames. This gives rise to multiple interpretations of a matched word pattern. All possible interpretations for a matched word pattern are pursued in parallel.

The final output of the parser will have more than one filled frame for a given input sentence. The parser applies a scoring mechanism and picks the best scoring frame as the final interpretation for the sentence. The highest scored frame is the one that has the maximum

number of words of the input sentence.

The Phoenix system was evaluated on 1001 utterances of the ATIS dataset and reported an accuracy of 93.5% on transcript inputs.

2.4.2 Philips Automated Inquiry System

Philips created a system to understand queries regarding train timetable information [63]. The system is built based on the idea that the meaning of a database query is expressed in standalone phrases. For example, the phrase 'to <stationname>' refers to the destination railway station. These phrases may occur in any order in the sentence. The concept phrases may be separated by words that have no contribution to the meaning of a database query. For example, words like 'Good morning' or 'I want to go' do not contribute to the semantics of a database query.

The understanding module locates only these conceptual words and connects them in a conceptual graph. This is similar to the understanding algorithm of Phoenix, where the entire sentence is not parsed, but only the semantic words are selected and assigned to slots of a frame.

This system uses semantic context-free grammar which describes the connection between conceptual words. The processing begins by selecting the concept words in a sentence and connecting them in a particular order according to the semantic grammar. This connection of semantic words in the input sentence creates a conceptual graph. This conceptual graph is then mapped to a formal query language.

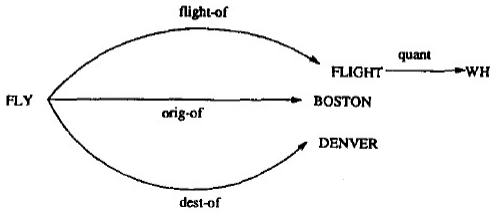


Figure 2.15: A semantic graph generated by parser for the sentence 'What flights fly from Boston to Denver' [5].

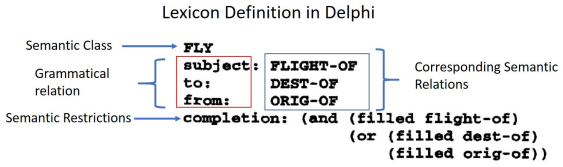


Figure 2.16: A lexicon entry of word 'fly' annotated with different types of semantic information in the Delphi system.

2.4.3 Delphi

Delphi is a natural language processing component of the BBN spoken language system [5]. To parse the input natural language queries of air travel booking, Delphi uses a linguistic grammar written in Definite Clause Grammar formalism. Even though the grammar is linguistically motivated, the output of the parser is a semantic graph instead of a parse tree with syntactic edges. Apart from the syntactic grammar rules, Delphi has rules that convert a syntactic edge between words to a semantic relation.

A semantic graph with semantic relations between nodes is shown in Figure 2.15. To create such a graph, explicit semantic information was encoded in the lexical definition. An annotated lexical entry for the word 'fly' with different types of semantic information is shown in Figure 2.16. In this figure, we have shown three types of semantic information, namely, Semantic class, Semantic relations corresponding to grammatical relations, and Semantic restrictions.

The word 'FLY' in Figure 2.16 shows the semantic class for the word 'fly'. The entries 'subject', 'to', and 'from' in the lexicon entry represents the type of syntactic edges that can

be connected with the word 'fly'. The entries 'FLIGHT-OF', 'DEST-OF', and 'ORIG-OF' represent the relation edges for the corresponding syntactic edges for the word 'fly'. For example, in Figure 2.15, 'DELTA' is the subject of the word 'FLIGHT' and is connected by a semantic relation 'airline-of'. These relation edges are similar to the slots in a frame of the semantic class 'FLY'.

Semantic restrictions in Figure 2.16 show the type of slots that are required to be filled. As shown in this figure, the semantic restrictions require that the slot 'Flight-OF' must be filled, and either the slot 'DEST-OF' or 'ORIG-OF' has to be filled. This implies that we cannot use the phrase 'the flight flies', since this phrase does not have the words corresponding to the 'Flight-of' slot.

A crucial component of Delphi's chart parser is to understand sentences that are either ungrammatical or lie outside the coverage of the grammar. A parse of a partially understood sentence in Delphi's chart parser will have a set of fragment constituents in the chart. Each of these fragments will have an associated semantic graph. The semantic linker module of Delphi connects these semantic graphs into a single connected graph by adding links in different sub-graphs. In [5], an example of the semantic linker is shown on the following query:

Show me the meals on the morning flight, on American at 12:30

A semantic linker connects 'American' from the semantic graph of the second phrase to the 'flight' node of the semantic graph of the first phrase. Also, the word 'morning' is replaced with '12:30' using a variant of the semantic linker.

Delphi was evaluated on 1000 utterances taken from the ATIS dataset. It reported that 14.7% of queries were either answered incorrectly or not answered at all.

2.4.4 Gemini

Gemini is a tool that translates natural language queries for booking airline tickets to a formal query language. The grammar and lexicon of Gemini [64] is a variant of the formalism used in the Core Language Engine (CLE) [65]. CLE is a device for mapping natural language sentences to a logical representation. The categories in Gemini grammar are associated with feature constraints. An example definition of a noun phrase category in Gemini is as follows:
np:[wh = ynq, case = (nom $\vee$ acc), pers_num = ($3rd \wedge sg$)]
The above definition shows a noun phrase and its wh-feature constraint. This 'np' category is assigned to a noun phrase that has wh-words like 'which book', 'who'. The noun phrase wh-feature is assigned a 'ynq' value. According to CLE, the value 'ynq' refers to a yes/no question. The case feature of this 'np' category can be either 'accusative' (acc) or 'nominative' (nom). The person number feature for the np category should be singular and in the third person form. This 'np' category and its feature constraint is an example of a building block for creating a constituent grammar in Gemini.

Apart from syntactic rules, this system also has semantic restrictions applied to linguistic categories. For example, the object of the transitive verb 'depart' can only be of type 'city' and 'airport'. This implies that we can say the phrases like 'flight departing Boston' but we cannot say the phrase 'flight departing after two days'. Also, similar restrictions are applied to adjectives. For example, the word 'further' can take arguments of type distances only, like the phrase 'thirty miles further'.

Gemini uses a chart parser where all possible edges accepted by the grammar are created. The parser does subsumption checking during chart parsing to prevent adding redundant edges. The subsumption filter checks if there is an edge in the chart spanning the same nodes as the new incoming edge. If an edge exists, then the check does not add the new

edge in the chart. The parser also performs semantic constraint checking while creating a syntactic edge. An edge is added to the chart only if it satisfies both syntactic and semantic constraints.

Gemini has a separate utterance grammar to understand incompatible utterances in a single sentence. These incompatible utterances cannot be joined in a single parse tree and represent sentence structures that are not covered by the constituent grammar. The analysis using the utterance grammar is carried out after parsing the sentence.

2.4.5 Understanding after Sentence Simplification

In frame semantic parsing, understanding a sentence consists of two tasks, namely, intent determination and slot filling. The complex syntactic structure of the sentence makes it difficult to identify the correct intent (frames) and slots for the sentence. For example, it is shown in [66] that the failure in intent determination occurs in a sentence structure that has a prepositional phrase in a noun phrase. In this sentence type, the prepositional phrase suggests a different intent than the actual one. For example, in the utterance 'Capacity of the flight from Boston to Orlando', the noun phrase 'Capacity' is the actual intent of the sentence. But the understanding frameworks usually assign 'flight' as the intent of the sentence.

Similarly, there are errors in slot filling due to long-distance dependencies in a sentence. For example, in the sentence 'Find flights to New York arriving no later than next Saturday' a 6-gram context is required to understand that Saturday is the arrival date.

In [6], the error in finding the intent and slots is reduced by simplifying the sentence. In [6], the dependency parse tree of the input sentence is used to remove auxiliary predicates. The simplified sentence contains only the main sentence predicate and its dependents. For exam-

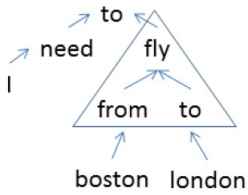


Figure 2.17: Dependency parse tree of the sentence 'I need to fly from London to Boston' [6].

ple, let us consider the sentence 'I need to fly from London to Boston' and its dependency parse tree, as shown in Figure 2.17. In this parse tree, the word 'fly' is the highest non-auxiliary predicate. The dependents of the word 'fly' is 'from' and 'to'. The entire sentence is simplified to 'fly from to'. It is evident from this example that the simplified sentence need not be grammatically correct. The simplified sentence is passed to a classifier, which predicts the intent and slots of the sentence. The overall objective of simplification is to remove the phrases that complicate the classification task. For example, in the sentence 'how much does it cost to fly from London to Boston', the phrase 'how much does it' is removed as a result of simplification. The resultant simplified sentence consists of the predicate 'cost' and its dependents connected with 'from' and 'to'

2.5 Authoring Tools for Controlled Natural Language (CNL)

A user can express the same semantics in a natural language through different words and different order of words. However, a semantic parser can parse only a subset of words and sentence structures that are defined in the CNL grammar. Hence, users have to learn CNL-specific templates or grammar rules to write CNL-acceptable sentences.

A common approach to alleviate user efforts in learning CNL rules is to employ an auto-completion feature in the CNL editor. Auto-completion feature can predict the next word based on the word's syntactic compatibility with the already written words of the sentence. The predictive text editor in [67] performs auto-completion using a lookahead analysis based on Prolog DCG grammar rules and parser. The predictive editor in [68] is implemented using a statistical n-gram language model. However, merely predicting the next syntactically accurate word may not be enough to generate a CNL-acceptable sentence. A syntactically accurate sentence may still be semantically incorrect and may not create the intended output.

The CNL editor proposed in [69] provides a combination of a predictive editor and a menu-based graphical user interface that allows the user to edit the underlying intermediate representation of the CNL. However, as shown in [70], users can be overwhelmed by many options in a drop-down menu.

Work in [71] has proposed an authoring tool that displays parsed and un-parsed segments of specification in different colors. According to this work, the coloring of segments is a less intrusive approach than predictive editors and visualization of intermediate representations. Moreover, the work states that the coloring of segments is well suited for users who are in haste and not keen on checking next word predictions, and couldn't understand the syntactic visualization or intermediate representation of a sentence.

In [72, 73], context vectors obtained from the parsing of an erroneous sentence are used to generate suggestions.

In [74], semantic parsing of sentences with unknown words and sentence structures is achieved by rewriting the sentence into an acceptable form while preserving the input sentence semantics. A sentence is rewritten using rules and templates that are extracted from a paraphrase database. However, each rewritten sentence is parsed to pick a logical form that is seman-

tically closer to the input sentence. Semantically parsing all possible rewritten sentences may consume a lot of time and may not be a viable option for an input sentence with many rewritten forms in a real-time system.

In [75], rewrite patterns and relaxed grammar rules are created to find and rewrite the erroneous phrase in a sentence. The rewrite rules are created in addition to the existing grammar rules needed to parse the sentence. The manual process of creating rewrite patterns in addition to hand-crafted grammar rules may not scale well to cover a large spectrum of sentences that cannot be parsed.

Chapter 3

Data Structure and Pre-Processing

In this chapter, we introduce a node data structure to store and process the words of an input sentence. We present a data structure for the meaning representation of a sentence used in the framework. We also present our pre-processing module that generates words from raw text.

3.1 Node Data Structure

In the framework, the words of an input specification are represented as nodes of a tree. For example, as shown in Figure 3.1, the words of the specification 'Writing a 1 to bit 1 clears the receiver FIFO' are stored as a sequence of nodes in our framework.

A node data structure is a JSON object. A node is created for each word of the specification. The node data structure stores word attributes and provides ease in the processing of the word's meaning based on its connections with other words of the sentence.

As shown in Figure 3.1, a node of an input specification has the following four attributes:

1. word: Stores the user written raw word as a string.

2. processed_word: Stores the processed word corresponding to the raw text stored in the 'word' attribute.

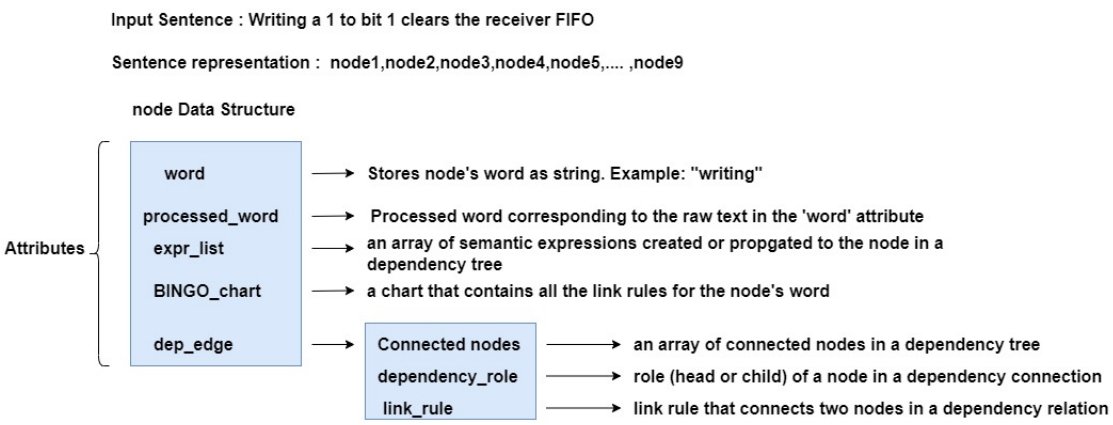


Figure 3.1: Illustration of words as nodes and general structure of node data structure.

3. expr_list: An expr_list keeps an array of semantic expressions that are either created at the node or propagated to the node in a dependency tree.

4. BINGO_chart: BINGO_chart contains a chart corresponding to the node's grammar rules.

5. dep_edge: dep_edge attribute stores the dependency edge information for the node in a dependency parse tree.

In the rest of the thesis, the words are considered nodes without explicitly specifying them as nodes. For example, Figure 3.2 (a) illustrates a dependency parse tree where edges of the parse tree connect nodes of a sentence. The nodes are not explicitly written at the ends of the connecting edges but are represented by the word that is stored in the node's 'word' attribute.

In the framework, words can also be combined and processed as n-grams. In Figure 3.2, the word 'bit 1' is analyzed as one word and represented by a single node.

Figure 3.2 (b) presents attributes of Writing node as it exist in the dependency parse. In the figure, Writing node contains the word 'Writing' in its 'word' attribute. The expr_list

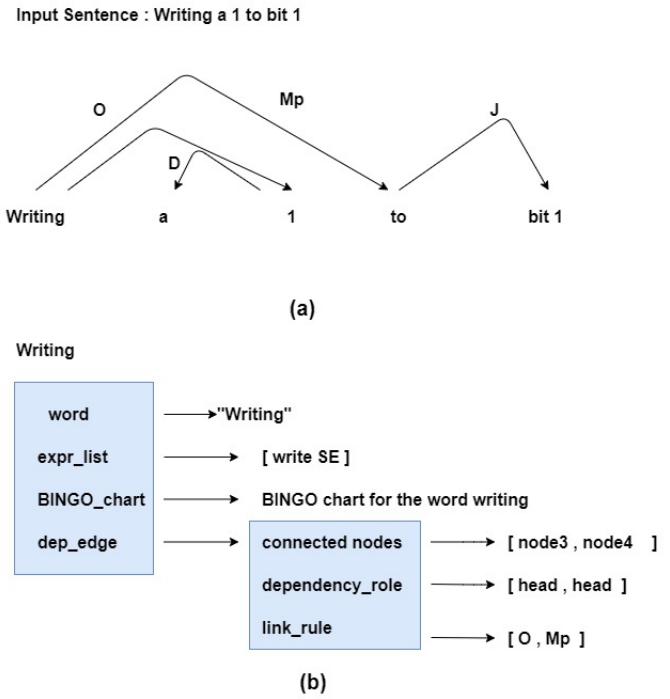


Figure 3.2: (a) Parse of a phrase. (b) Displays the attributes of node1.

of Writing node contains 'write SE' that represents the meaning of the parse tree rooted at the Writing node.

The grammar rules for the word 'Writing' is stored in the attribute 'BINGO_chart' of Writing node. The dependency edge information describes that the node is connected to two nodes, node3 and node4. The role of the Writing node in these dependency edges is 'head' as specified in the 'dependency_role' key. The links rule name for the respective edges is shown by the key 'link_rule' in the dep_edge attribute.

The subsequent sub-sections present the structure of a Semantic Expression (SE) that stores the semantics of words and the overall sentence.

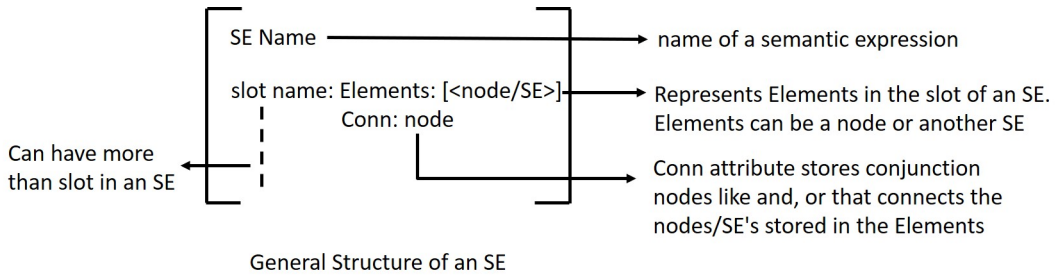


Figure 3.3: Basic structure of a Semantic Expression (SE).

3.2 Semantic Expression

A Semantic Expression (SE) is similar to a semantic frame [76] that represents semantics created by analyzing nodes in the framework. Figure 3.3 displays a general structure of SE in the framework. Example SEs with values are shown in Figure 3.4. Figure 3.4 illustrates a write SE corresponding to the meaning of spec 'writing 1 to bit 1'.

An SE structure has the following two parts:

1. SE name: An SE has a name denoted by 'SE Name' at the top of the SE that signifies the broader meaning of a semantic expression. For example, the name of the SE in Figure 3.4 (a) is 'write SE', which signifies that we are talking about write operation in a hardware verification domain.

2. slot name: An SE consists of a number of slots that explicitly point to a specific meaning within the broader domain of 'SE name'. For example, in Figure 3.4, slot names in 'write SE' are 'write_what_value' and 'write_to_bit'. These slot names indicate that we are talking about writing a specific value to a specific bit in a register.

In the framework, there are two attributes in a slot to store the information. These attributes are as follows:

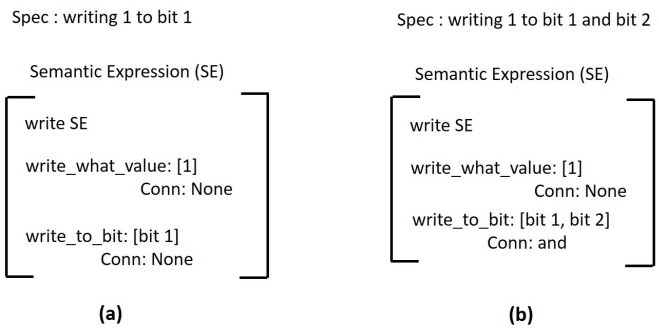


Figure 3.4: (a) SE representing phrase 'writing 1 to bit 1', (b) SE representing a phrase with a conjunction.

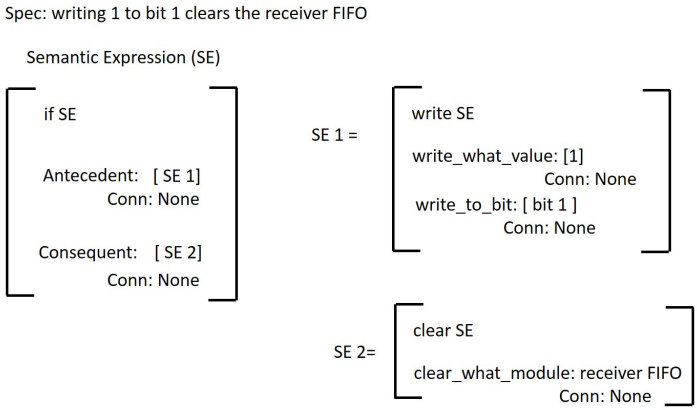


Figure 3.5: An SE containing SEs in elements list attribute of a slot.

- Elements: The values in a slot are stored in a list. Elements is an array to store slot values that are either a node or another SE. For example, in Figure 3.4 (a), the slot 'write_what_value' has an elements list that contains only one node. In Figure 3.4 (b), the 'write_to_bit' slot has two nodes in the elements list.

- Conn attribute: The Conn attribute is used to store a conjunction node that connects multiple nodes or SEs stored in the elements list of a slot. For example, slot 'write_to_bit' has two nodes bit 1 and bit 2, in the elements list. The Conn attribute of the slot has an 'and' node. The 'and' node indicates that the nodes in the elements list are in conjunction relation where the conjunct is 'and'.

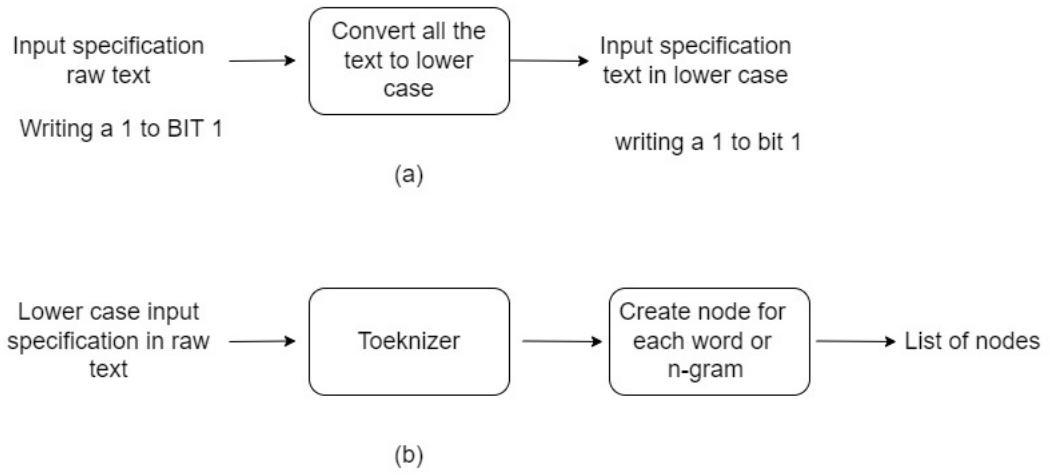


Figure 3.6: (a) The first step in pre-processing converts the entire raw text of the input specification to lower case. (b) Tokenizer converts raw text to a list of words that are stored as nodes.

The semantics of the spec 'writing 1 to bit 1 clears the receiver FIFO' is illustrated by the 'if SE' in Figure 3.5. The 'if SE' contains slots Antecedent and Consequent that contain SEs instead of nodes. The content of 'if SE' implies that if a 1 is written in bit 1 then receiver FIFO is cleared.

3.3 Tokenization

In our system, the input specification is first pre-processed before being sent to the BINGO framework. The pre-processing begins by first converting the entire raw text to lowercase. For example, in Figure 3.6 (a), all the text in the input specification 'Writing a 1 to BIT 1' is converted to lowercase text 'writing a 1 to bit 1'.

As shown in Figure 3.6 (b), the specification in lowercase is then passed to a tokenization module. The tokenizer reads the raw text of the input specification and generates words or multi-word expressions. Each word generated by the tokenizer is stored in a node data

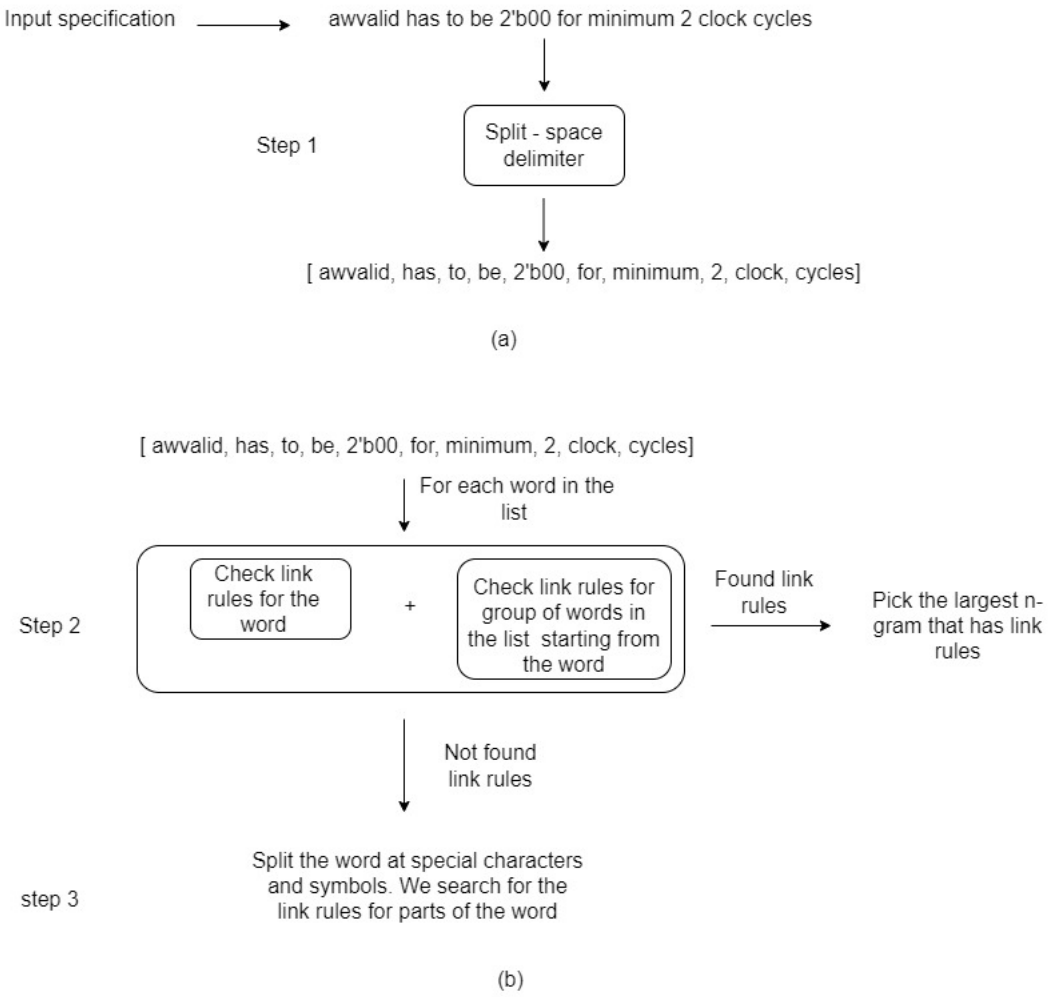


Figure 3.7: (a) Splitting lower case input text using space as a delimiter. (b) Step 2 and step 3 of the tokenizer.

structure. The final output generated by the system is a list of nodes corresponding to the words or multi-word expressions generated by the tokenizer.

We have illustrated our tokenization steps in Figure 3.7. The block diagram of Figure 3.7 is explained in the following steps:

1. In step 1, the input specification is split into a list of words using space as a delimiter, as shown in Figure 3.7 (a).

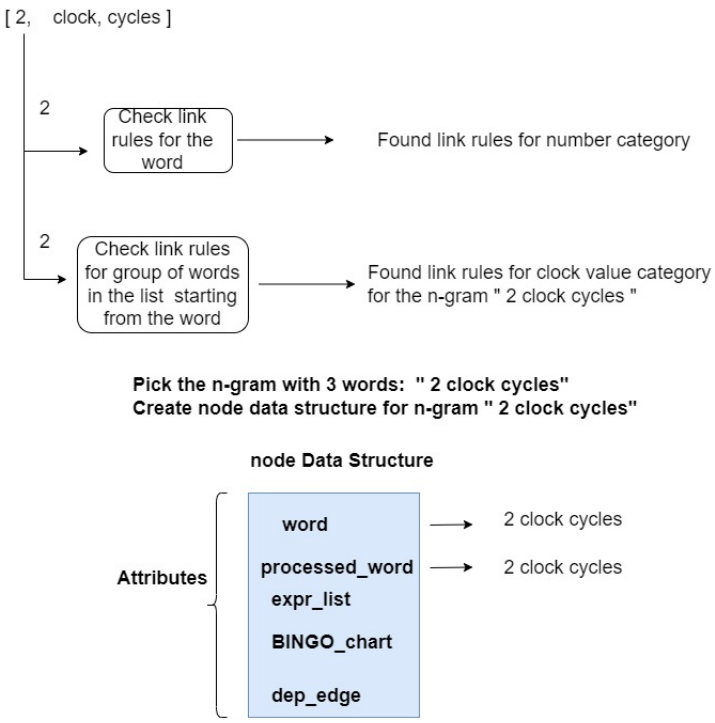


Figure 3.8: The step 2 of tokenization is explained with an example list of words.

2. In step 2, we pick a word from the list created in step 1 and search for the word's link rules in our grammar. If we find the link rules, then we save this information.

   In step 2, we also search for a multi-word expression that can be created using the picked word. The objective is to pick the largest multi-word expression that has link rules in our grammar.

   When we find link rules for a multi-word expression starting from the picked word, then we create a node data structure for the entire multi-word expression. If we found the link rules only for the picked word, then we create a node data structure for that word. The step 2 is explained using the example list $[2, clock, cycles]$ in Figure 3.8.

   In Figure 3.8, the module begins by checking the link rules for the first word in the list. The system identifies '2' as a number and selects the link rules of the 'number' category for the word '2'.

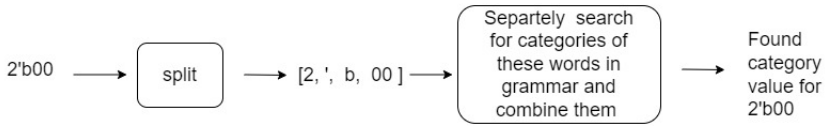


Figure 3.9: The step 3 of tokenization is explained with an example word '2'b00'.

The system further searches for n-gram starting from '2' in the list that can have link rules in our grammar.

The system finds n-gram '2 clock cycles' and identifies it as a clock value category n-gram. The system picks the link rules of the 'clock value' category for the n-gram.

Since the largest n-gram found in the system is '2 clock cycles', the system merges three words in the input list to '2 clock cycles' and creates a node for the n-gram, as shown in Figure 3.8. The node in Figure 3.8 shows values of only two attributes, 'word' and 'processed_word'. The values of the remaining attributes of the node is filled during the node's processing in the BINGO framework.

3. Step 3 is executed when we don't find the link rules for the word or multi-word expression. The word is considered an out-of vocabulary word, and we break the word into parts and continue to search for link rules for the parts of the word.

   The word is broken at special characters and symbols. The system searches for the link rules of the smaller parts of the word. If the link rules are found then, the corresponding node is created for the word. If the link rule is not found, then the word is considered unknown to our grammar. A node for an unknown word is still created with no link rules in the 'BINGO_chart' attribute of the node data structure. The unknown word and its node is processed in the suggestion generation module.

   Figure 3.9 shows the analysis of step 3 for the word '2'b00'. The word is split at character symbols, and the words in the list are separately searched in the vocabulary. After the analysis, a 'value' category and its link rules were assigned to the word '2'b00'.

Chapter 4

BINGO Framework

In this chapter, we present a dependency grammar-based framework to understand hardware specifications written in English. The formal output generated by a semantic parser of a translation framework can be non-intuitive, and the user may not be able to validate its correctness unless the output is executed. However, natural language specifications are generally written at the early design stages when an executable prototype is not yet available. As a result, the semantic parser output cannot be immediately executed and verified. It becomes the responsibility of the parser to generate only correct translations of the natural language specifications. It is evident in [77, 78], machine learning-based semantic parsing approaches require thousands of input-output examples to achieve high accuracy. Unavailability of large numbers of examples resulted in many rule-based translation works like [37, 79, 80]. These rule-based approaches achieve high accuracy in understanding specifications by imposing strict restrictions on the order of words in the input specifications.

The novelty of this work is in developing a framework that allows an automatic understanding of input sentences with variations in word order and input sentence structures. Our framework comprises of two distinct components. The first component is a declarative specification of rules analogous to creating BINGO charts for each word. The second component is a chart parser that takes the BINGO chart of each word as input and performs two steps similar to the game of BINGO: The first step marks cells in the chart of each word. The second step selects a single horizontal BINGO row that passes through the rows of charts of

all the words and covers only marked cells. The declarative rules component is inspired by the syntactic link rules of the link grammar [43]. In our grammar, we have added semantic links to the existing syntactic links of the link grammar. The semantic links serve the following purposes: (1) they represent the semantic context with which a word can be used in the specification (2) they define semantic propagation operations that should be executed when a link between two words is created.

We build semantic frames for a hardware specification compositionally using the link parse tree as the syntactic structure. The creation and propagation of the semantic frames from the individual nodes to the root node of the parse tree are governed by the semantic propagation rules defined in the semantic links.

4.1 Motivation

Dependency parse trees are useful in extracting semantic relations between entities and have close correspondence to the semantic representation of the sentence [50, 81]. Driven by these advantages, dependency parsing has been used as the core component in the recent works [16, 18, 29, 82, 83, 84] to automatically understand input specifications written in natural language. However, due to the lack of domain-specific data to train the dependency parser, an off-the-shelf dependency parser trained on general natural language is employed in these applications. It has been shown in [85] that the accuracy of a syntactic parser may reduce when applied on text outside its training corpus. Work in [18, 29, 84] attempts to improve the accuracy of the syntactic parser by introducing a pre-processing step to recognize and parse the domain-specific phrases. However, no concrete solution was proposed to resolve ambiguities like preposition and coordination attachments in syntactic parsing. Parse trees with incorrect attachments between words are ineffective for any downstream

natural language understanding application. As pointed out in [30], the main reason for the inaccuracy in syntactic parsing of specifications is the absence of domain-specific context knowledge and its integration with the parser.

In this thesis, we propose a grammar-based understanding framework that considers context on both the left side and right side of a word before making a dependency arc on the word. Our dependency grammar is inspired by the link grammar formalism [43] that allows encoding of syntactic context by using binary associative operators '&' and '*or*'. The existing link grammar parser is a complicated algorithm that is similar to finding an optimal triangulation of a convex polygon using dynamic programming [43]. Also, the link grammar has no provision to define semantics for link rules. We propose a simple parser for the link grammar by modeling link parsing as a game of BINGO. Moreover, we define semantics for each link connection of a word and then combine the semantics of each link connection in a transition-based dependency parsing framework. The final output of the parser is a semantic frame that corresponds to the meaning of the input specification written in the English language. The final semantic frame can be translated to a System Verilog Assertion (SVA) if all the Register Transfer Level (RTL) information is available in the semantic frame.

4.2 Framework

Figure 4.1 shows the flow of the major modules of our framework. We first create a BINGO chart for each word in a given specification based on the underlying grammar. The BINGO parser marks cells in the BINGO chart of each word according to the word's syntactic and semantic links. After completing the marking process, we search for a BINGO row that is a horizontal row spanning the marked rows of all the charts. The BINGO row represents a solution to connect all the words in the sentence after taking into account each word's

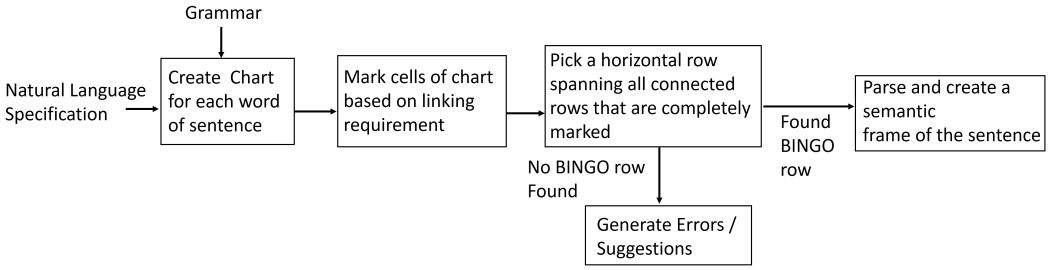


Figure 4.1: Framework for parsing and translating hardware specification to Semantic Frames.

context. In order to create a semantic frame, we execute semantic propagation rules for each connection in the BINGO row in a transition-based dependency parsing framework.

In subsequent sections, we present the grammar formalism of the BINGO framework. Then, we present the parsing methodology of the framework, give examples of grammar creation, and discuss the evaluation of the work.

4.3 Grammar Formalism

A link rule in our proposed grammar consists of a syntactic link and a semantic link. Figure 4.2 (a) shows a general structure of a link rule in the grammar. The syntactic link is taken from the link grammar [43]. The meaning of syntactic link rules can be found in [45].

Similar to the notion of syntactic links that represent the linking requirement of a word with its left and right words, we present semantic links that represent the linking requirement for the semantic composition of nodes in a dependency relation. Our semantic links consist of a semantic propagation rule and an indicator of the node's dependency role in the dependency relation.

In the link grammar, two matching syntactic link connectors have the same name and dif-

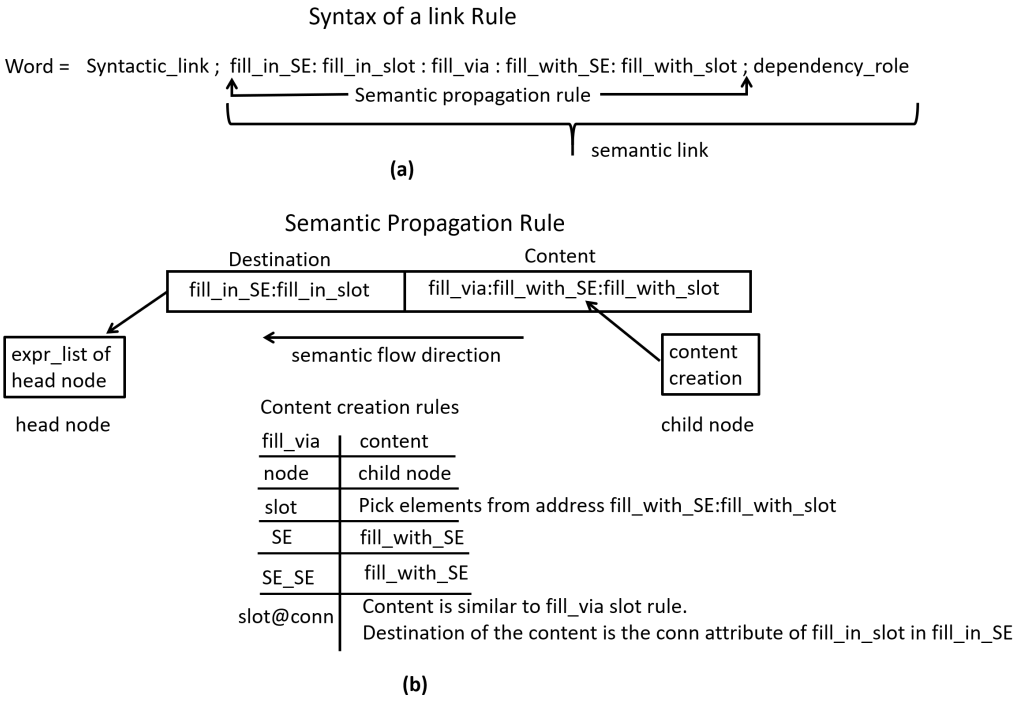


Figure 4.2: Structure of (a) a grammar link rule and (b) a semantic propagation rule.

ferent polarity of + and - directions. Similarly, in our grammar two matching semantic links have the same semantic propagation rule but have an opposite polarity of head and child dependency roles. The semantic composition between two nodes is possible only if they have matching semantic links.

In our framework, we connect two nodes in a dependency relation only if they satisfy the criteria of both syntactic and semantic linking as illustrated in Figure 4.3. The syntactic

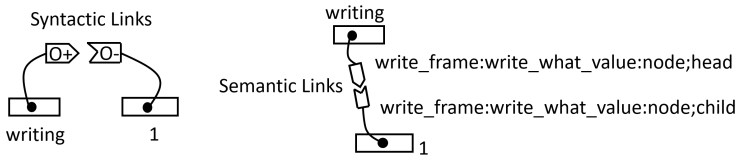


Figure 4.3: Syntactic links shows syntactic requirement of words in a sentence. Semantic links represent similar linking requirement between the head and child words in a dependency relation.

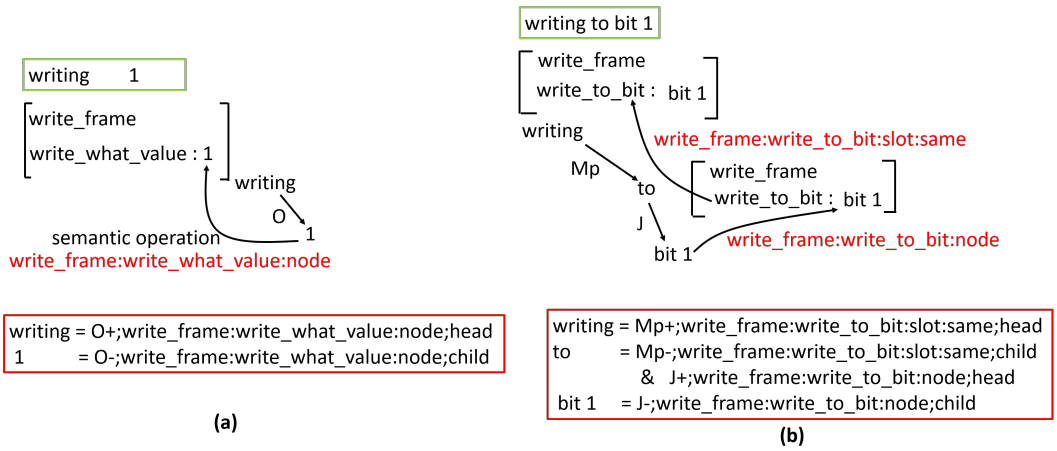


Figure 4.4: (a) Example of Semantic propagation operation when fill_via is node. (b) Example of Semantic propagation operation when fill_via is slot.

link in this figure demonstrates that the node 'writing' expects an object (O+) on its right side, and the (O-) at node '1' indicates that it can be connected as an object of a node on its left side. In this figure, semantic composition is possible between the nodes 'writing' and '1' because they have the same semantic propagation rule write_frame:write_what_value:node where the node 'writing' is the head, and '1' is the child node.

4.3.1 Semantic Propagation Rules Syntax

A semantic propagation rule in a semantic link consist of five parameters separated by ":" as shown in Figure 4.2 (a). A semantic propagation rule is needed to perform two tasks. The first task is to define the semantic content that will be propagated from the child node in a dependency relation. The second task is to define the destination SE and slot of the head node where the content will be transferred. Figure 4.2 (b) illustrates the function of the semantic propagation rule parameters. The first task is accomplished by the following three parameters of the rule: fill_via, fill_with_SE, and fill_with_slot. These parameters create semantic content according to the content creation rules shown in Figure 4.2 (b). When the

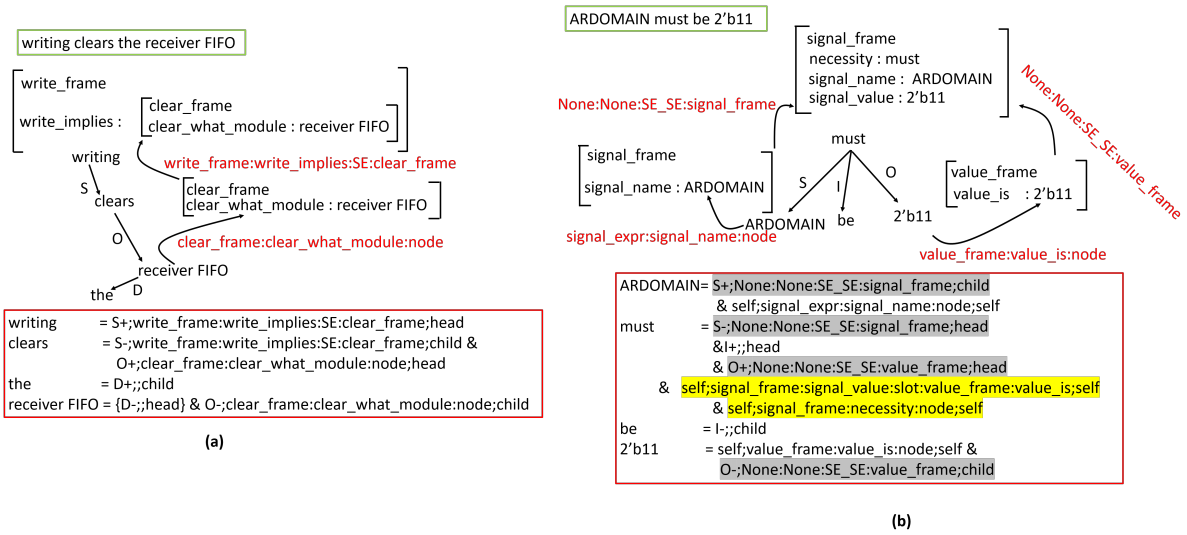


Figure 4.5: (a) Example of Semantic propagation operation when fill_via is SE. (b) Example of Semantic propagation operation when fill_via is SE_SE.

parameter fill_via is node, then the content to be propagated is the child node. Elements of a particular slot (fill_with_slot) from an SE (fill_with_SE) of the child node's expr_list is propagated when the fill_via value is slot. In the case of an SE or SE_SE, we propagate an SE from the expr_list of the child node defined by the fill_with_SE parameter of the rule.

The destination of the semantic content is determined by the fill_in_SE and fill_in_slot parameters of the semantic propagation rules. These parameters refer to the SE and slot of the head node that are stored in the expr_list of the head node.

In the fill_via slot propagation rule, the content in the source slot's elements array and conn parameter is propagated to the corresponding elements array and conn parameter of the destination slot in head node's SE. To transfer the content of the elements array from the source slot to the conn parameter of the destination slot, we created a fill_via slot@conn rule. The rule is needed to transfer conjunction nodes like 'and' , 'or' from source slot's elements array to the conn parameter of the destination slot.

We illustrate the working of different semantic propagation operations in Figure 4.4 and

Figure 4.5. In these figures, we have represented the example phrase in the top green box and the corresponding grammar rules in the red box at the bottom of the figure.

4.3.2 Fill Via Node

The fill_via node semantic propagation operation represents a scenario where a child node of the dependency edge directly fills the slot of an SE located in the head node's expr_list. The application of this rule between nodes 'writing' and '1' is illustrated in Figure 4.4 (a). As shown in the grammar rules, the words in dependency relation satisfy each other's syntactic link and semantic link requirements. The semantic fill_via node operation is carried out in dependency parsing by placing the node '1' in the slot write_what_value of the head node's SE write_frame.

4.3.3 Fill Via Slot

A child node of a dependency edge cannot always be a direct argument of the head node's SE. A child node can also act as a bridge propagating semantic information between its connected nodes. We can express a child node as a bridge in our grammar by using fill_via slot and fill_via SE semantic propagation operations. Figure 4.4 (b) illustrates a fill_via slot operation where the node 'to' acts as a bridge. In this figure, the node 'to' plays the role of both head and child node in the conjunctive rule (Mp-;write_frame:write_to_bit:slot:same;child & J+;write_frame:write_to_bit:node;head). The node 'to' receives the content of the slot write_to_bit in the write_frame from the child node 'bit 1' using a fill_via node semantic operation. This content is then transferred to the node 'writing' using a fill_via slot semantic operation through the edge Mp. The word 'same' in the rule is used for the ease of writing semantic propagation rules that have the same source and destination parame-

ters. For example, in the fill_via slot rule of the word 'to', 'same' indicates that parameters fill_with_SE and fill_with_slot have values equal to the values of the parameters fill_in_SE and fill_in_slot.

4.3.4 Fill Via SE

Figure 4.5 (a) illustrates a fill_via SE operation where the node 'clears' creates a clear_frame and passes it to a slot in its head node's SE. In this figure, the SE at the node 'clears' receive 'receiver FIFO' in its slot using a fill_via node operation. The entire SE created at the node 'clears' is sent to a slot in the writing node's write_frame using a fill_via SE semantic operation. It can be seen in this figure that every syntactic edge is not associated with a semantic propagation operation. In the red box at the bottom of the figure, the rule for syntactic edge 'D' has an empty semantic propagation rule that indicates the absence of a semantic propagation operation when a syntactic edge is created.

4.3.5 Fill Via SE_SE

Figure 4.5 (b) shows that the grammar can also express the composition operation of two SEs. This figure illustrates grammar rules and semantic operations for the phrase 'ARDOMIAN must be 2'b11'. The node 'must' receives SE's from its subject ARDOMAIN and object 2'b11 using a fill_via SE_SE operation as highlighted in the grey color in the red box. A SE_SE semantic operation transfers the entire SE of the child node to its head node's expr_list. In this rule, the SE is not transferred to a slot of an SE in the head node's expr_list. This is defined in the SE_SE rule by specifying 'None' in both the fill_in_SE and fill_in_slot parameters of the rule. Since the SEs have already reached the 'must' node's expr_list, we combine them using self link rules highlighted in yellow color in the red box in the Figure

4.5 (b).

4.3.6 Self Link Rules

We created self link rules in the grammar that allows to manipulate SEs that exist at the node's expr_list. The self link rule doesn't represent a connection between nodes and has a dummy syntactic link component (self) that is not matched with the syntactic link of any other node. The word 'self' in the dependency role of the self link rule indicates that the final semantic information created by the self link rule remains at the node and is not propagated to its head node.

4.4 2-Step BINGO Parser

In parsing, we search for all possible set of links that can connect all the nodes of the input specification in a dependency parse tree and satisfy the syntactic-semantic link requirements of all the nodes. This set of links is called linkages in [45]. Figure 4.7(c) shows the final linkages found at the end of parsing for the input specification 'system should reach dataReady before 5 clock cycles'.

The working of the parser can be visualized as a game of BINGO. The grammar rules of each node can be arranged in a chart where the rows in the chart represent all possible syntactic-semantic links for a specific interpretation of the node. Each cell in a row of a chart contains a link from the node's grammar rule. The total number of cells in a row represents the total number of links required by the node to complete an interpretation. For example, in Figure 4.6 (a), the grammar for the node 'before' allows two different combinations of syntactic-semantic connections. This set of rules for 'before' translates to two rows of the

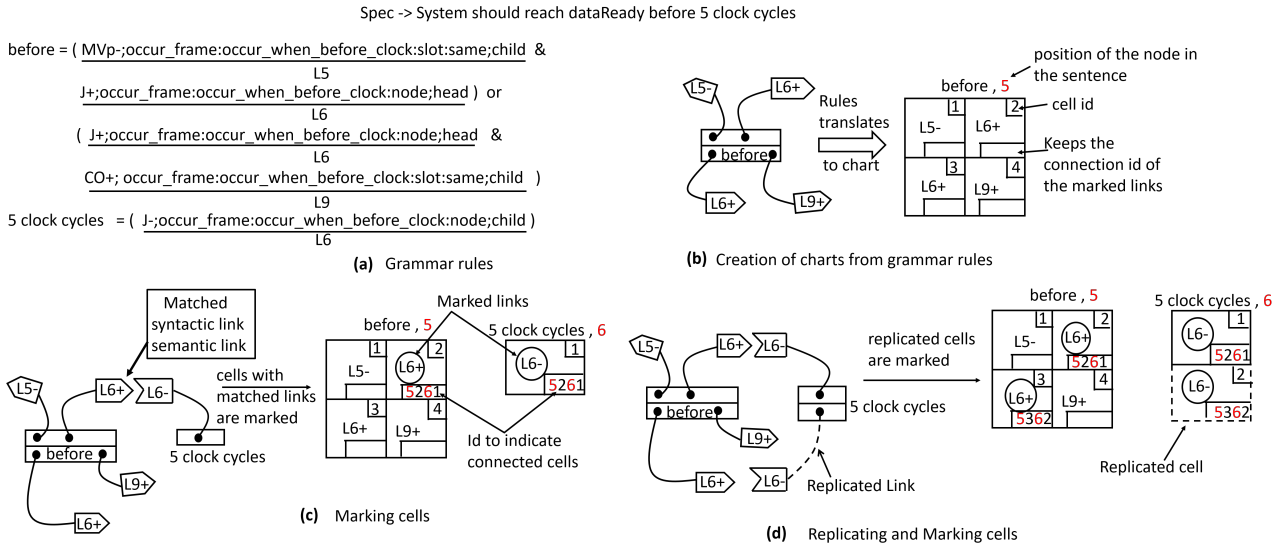
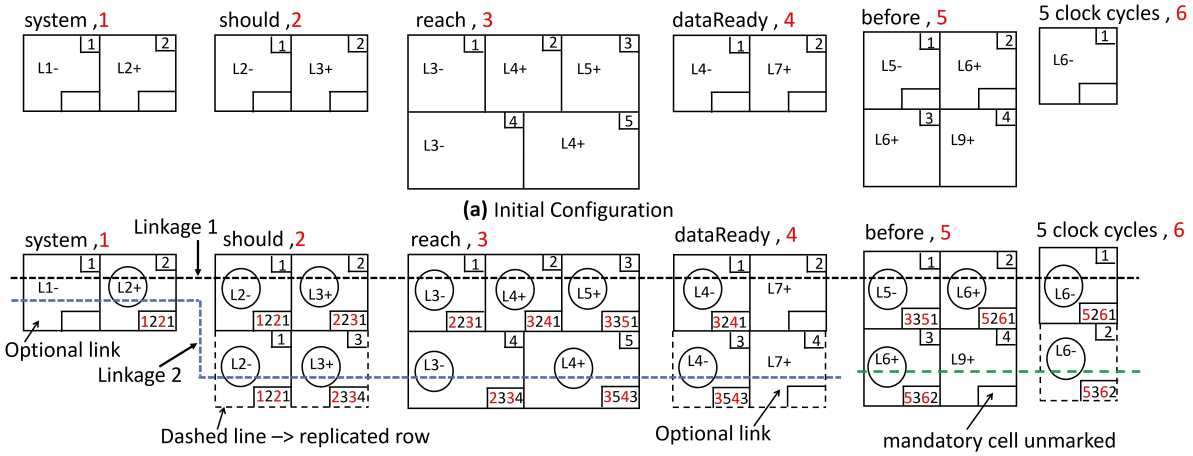


Figure 4.6: (a) Grammar link rules for words 'before' and '5 clock cycles' to parse ' system should reach dataReady before 5 clock cycles'.(b) Translating grammar rules to BINGO chart. (c) Marking cells of links that have matched syntactic and semantic links without violation link order constraints. (d) Link replication and marking cells of replicated link.

chart as shown in Figure 4.6 (b). In the chart, each row represents a unique combination of conjunct links that a node can have. For example, the chart for the node 'before' implies that the node 'before' should have either L_5 and L_6 or L_6 and L_9 links connected to it in a dependency parse tree.

Charts of all the nodes are given as input to the parser. The parser performs the following two tasks. First, the parser marks cells of all the charts based on the links inside the cells. Secondly, the parser searches for a set of BINGO rows that are horizontal rows spanning charts of all the nodes and cover only marked cells. A BINGO row represents a linkage that has satisfied all the syntactic-semantic constraints to build the parse tree.



nodes ->	system	should	reach	dataReady	before	5 clock cycles	Status
Linkage 1	L2	L2 L3	L3 L4 L5	L4	L5 L6	L6	Complete Coverage
Linkage 2	L2	L2 L3	L3 L4	L4			Incomplete Coverage

(c) Output of 2 Step BINGO parsing

Final SVA : assert property (@(posedge clk) ##[0:5] (dataReady)) ;

(d)

Figure 4.7: (a)Initial config. of Charts based on grammar rules given in Figure 4.8. (b) Searching BINGO rows after marking cells of charts. (c) Final Linkage solution of the BINGO row shown by the Black dashed line (Linkage 1). (d) System Verilog Assertion (SVA) created from the SE of the parse tree.

4.4.1 Step 1 - Marking Chart Cells

Our algorithm marks the cells of the charts as follows: We pick a node and match the cells of its chart with the chart cells of all the previous nodes of the sentence. We mark and connect a pair of cells if the link rules inside these cells match each other's syntactic and semantic links without violating syntactic link order constraints of the link grammar.

In Figure 4.6, we have shown grammar rules and marking in the charts of two nodes 'before' and '5 clock cycle'. The rules in this figure are labeled as L_i (where i is an index to the rule). Matched link rules have the same L_i and opposite polarity of syntactic and semantic links. For example, in Figure 4.6, the L_6 link rule in the first row of 'before' node chart matches the L_6 link rule of '5 clock cycles' chart. These matched L_6 link rules do not violate the syntactic link order constraints and can be connected.

As illustrated in Figure 4.6 (c), we record the connection between these links by circling them in the cell. These links can be part of the final linkage when all the mandatory links in their rows are connected. For example, the L_6 link in the first row of the 'before' chart can be part of a final linkage if we find a matching link for the L_5 link of the first row.

We continue to match the remaining links between the two charts. The L_6 link rule in the second row of the 'before' chart can also be matched with the L_6 link rule of the '5 clock cycles' chart. This connection can result in a new linkage that will include L_6 and L_9 links. Since the L_6 of the '5 clock cycles' chart is already marked, we will create a new row by replicating the L_6 cell of the '5 clock cycle' chart as shown in Figure 4.6 (d). The replicated L_6 cell is then connected with the L_6 link rule of the second row of the 'before' chart. Replication of marked cells creates a new set of unique linkages.

As illustrated in Figure 4.6, the cells of connected links are recognized by assigning a unique cell connection Id at the bottom right corner of these cells. A cell connection Id is derived

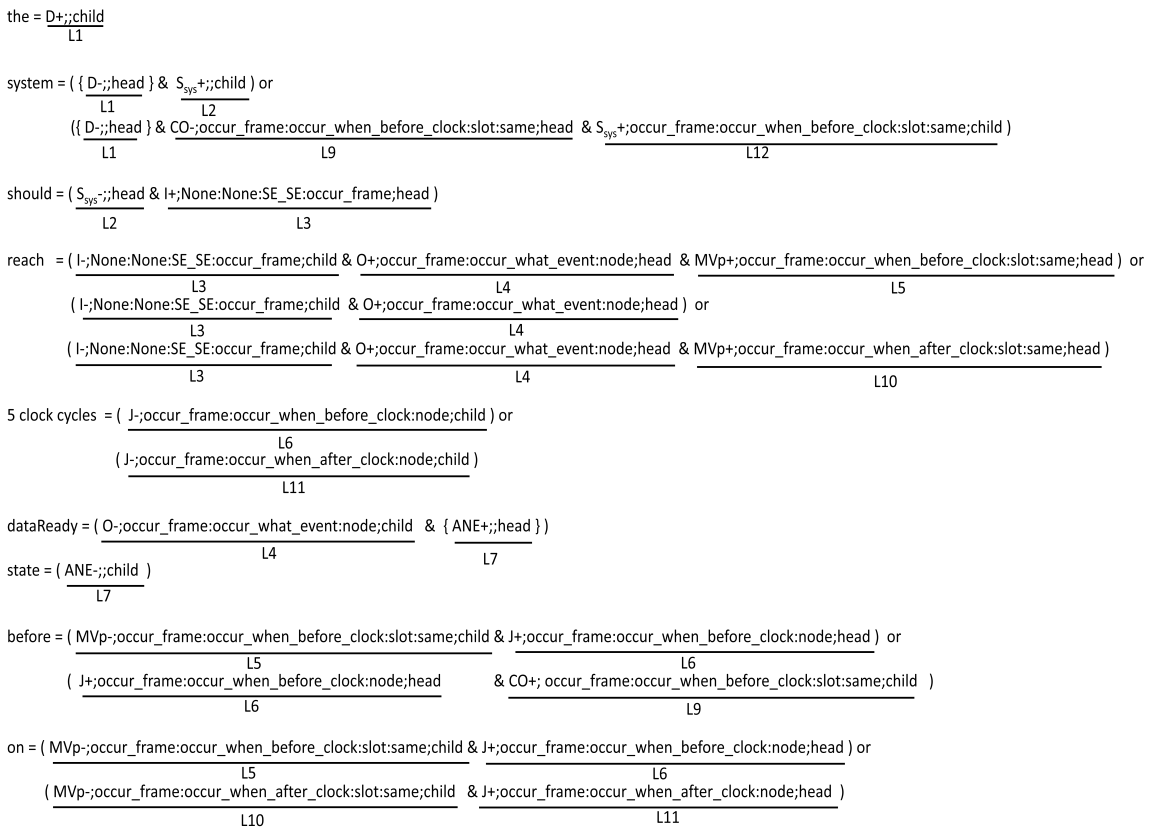


Figure 4.8: A small set of grammar rules to parse specifications

from the node and cells Ids of the marked links. A node Id (marked in red) represents the node position in the sentence, and a cell Id (marked in black) is shown in each chart cell. For example, in '5 clock cycles' chart, the cell connection Id of the L_6 link in the first row is 5261 (node 5 cell 2 and node 6 cell 1) and is different from the L_6 link of the second row. A unique cell connection Id acts as a pointer to the connected cell and assists in traversing the connected cells while selecting the BINGO rows.

4.4.2 Step 2 - Finding BINGO Row

After marking all possible cells in the chart, we scan the charts to find BINGO rows that pass through the connected cells and contain chart rows that have all their mandatory cells marked. Figure 4.7 illustrates the output after each step of the parsing for the specification

'system should reach dataReady before 5 clock cycles'. In Figure 4.7 (a), the initial charts are shown with a subset of grammar rules of Figure 4.8.

Figure 4.7 (b) shows lines that span rows containing marked cells. The green line cannot be a BINGO row since it covers an unmarked mandatory cell L_9+. The Blue dashed line (Linkage 2) in the figure is a BINGO row with incomplete coverage since it does not cover all the nodes of the sentence. A complete coverage BINGO row is found by the Black dashed line that contains all the connected cells with mandatory cells marked and passes through the chart of all the nodes. The output of parsing is shown in Figure 4.7 (c), where the final linkage is represented by the links covered by the BINGO row of the Black dashed line (Linkage 1).

4.5 Semantic Creation

A BINGO row provides linkages after taking into account the context of each node in the sentence. When a link is created between nodes, then the semantic propagation rule associated with the link is executed. The execution of semantic rules creates and updates SEs in the expr_list of nodes. The final SE representing the meaning of the entire specification is created and propagated to the root node of the parse tree. A detailed explanation of semantic creation that resulted in the SVA illustrated in Figure 4.7 (d) is presented next.

The semantic creation process begins by creating an empty stack and loading all input specification nodes in the buffer, as shown in step 1 of Figure 4.9. The following operations are performed during the transfer of nodes from the buffer to the stack.

- link_creation: In a link_creation step, we make all the links that the front node of a buffer can make with nodes in the stack. When all the link connections between the

Step no.	Stack	Buffer	Action Performed
1.	[]	[system,should,reach,dataReady,before,5 clock cycles]	
2.	[system]	[should,reach,dataReady,before,5 clock cycles]	Action: Shift Semantic_creation: No semantic action performed
3.	[system]	[should,reach,dataReady,before,5 clock cycles]	Action: link_creation L2 link created between node 'should' and 'system' Semantic_creation: No semantic is associated with L2 link
4.	[system,should]	[reach,dataReady,before,5 clock cycles]	Action: shift Semantic_creation: No semantic action performed

Figure 4.9: First four steps in the semantic creation process for connecting nodes according to BINGO solution found in parsing.

front node of buffer and stack nodes are made, then we decide to either shift or reduce a node. The link_creation operation initiates the following two operations when there exists a semantic propagation rule with the created link:

- semantic creation: In the semantic creation operation, the semantic propagation rule associated with links created between nodes is executed. The execution creates an SE in the expr_list of the head node defined in the semantic rule.
- semantic update: Once an SE is modified in a node due to the execution of a semantic rule, then the changes created in the SE of a node are propagated to its head node. Changes in a node's SE are propagated till the root node of the parse configuration.

- shift: In a shift operation, a node at the front of the buffer is pushed on the stack when the front node in a buffer cannot make any more edges with the nodes on the stack.
- reduce: In a reduce operation, we pop the node on the stack top when the node has made all its edges found in the BINGO solution.

We now present the step-by-step process of the creation of an SE for the specification "system should reach dataReady before 5 clock cycles".

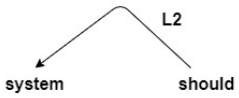


Figure 4.10: Nodes connected through L2 link found in the BINGO solution.

1. In Figure 4.9, step 1 is the initialization step where the nodes are loaded in the buffer, and the stack is empty. The semantic creation process continues until all the buffer nodes are moved to the stack.

2. In step 2, the front node of the buffer 'system' is pushed on the stack using shift operation. The shift operation was used since the node 'system' could not be connected with any nodes in the stack but could make edges with other nodes in the buffer.

3. In step 3, the front node of the buffer 'should' has a link L_2 with 'system' node as found in the BINGO solution. A link_creation operation is executed connecting the two nodes as shown in Figure 4.10.

   The L_2 link in the node 'should' is '$S_sys-;;head$'. The rule does not create semantics, as indicated by the absence of a semantic propagation rule in '$S_sys-;;head$'. The link_creation operation does not initiate semantic creation and semantic update operations due to the absence of a semantic propagation rule in the L_2 link.

4. In step 4, the node 'should' is shifted in the stack since it has made all its connections with stack nodes. The node 'should' is not reduced since it can make link connections with other nodes in the buffer.

5. In step 5 shown in Figure 4.11, a link L_3 is created between nodes 'should' and 'reach'. The link L_3 has semantic propagation rule 'None:None:SE_SE:occur_frame' that leads to semantic creation and semantic update operations as shown in Figure 4.12.

   semantic creation: The occur_frame on node 'reach' is propagated to the node 'should'

Step no.	Stack	Buffer	Action Performed
5.	[system,should]	[reach,dataReady,before ,5 clock cycles]	Action: link_creation L3 link created Semantic_creation: occur_frame of node reach transferred to node should through SE_SE operation
6.	[system,should,reach]	[dataReady,before,5 clock cycles]	Action: shift Semantic_creation: No semantic action performed
7.	[system,should,reach ,dataReady]	[before,5 clock cycles]	Action: link_creation L4 link created Semantic_creation: Node 'dataReady' is placed in the slot 'occur_what_event' in 'occur_frame' of node 'reach'. Semantic_update: Semantic operation associated with L3 edge is executed again to update the occur_frame that exist in the node 'should'.

Figure 4.11: Steps 5 - 7 in the semantic creation process of connecting nodes and moving buffer elements to the stack.

using SE_SE semantic propagation rule.

semantic update: The occur_frame received at the node 'should' has to be propagated to its head node. Since in the partial parse configuration, the 'should' node has no head node, the occur_frame remains at the 'should' node.

6. In step 6, the node 'reach' is pushed to the stack using a shift operation.

7. In step 7, link L_4 is created between the 'reach' node and the 'dataReady' node. The node 'dataReady' is then shifted onto the stack.

   semantic creation: The semantic propagation rule of L_4 is 'occur_frame:occur_what_event:node'. The semantic creation operation for L_4 is illustrated in Figure 4.14 (a). As shown in the figure, the 'dataReady' node is placed in the slot 'occur_what_event' of the 'occur_frame'. The semantics are created in the expr_list of the 'reach' node.

   semantic update: The semantic update operation is illustrated in Figure 4.14(b). Since the SE occur_frame of node 'reach' is modified by L_4 link creation, the update in semantics will be propagated to the head node of 'reach' node. The node 'should' is

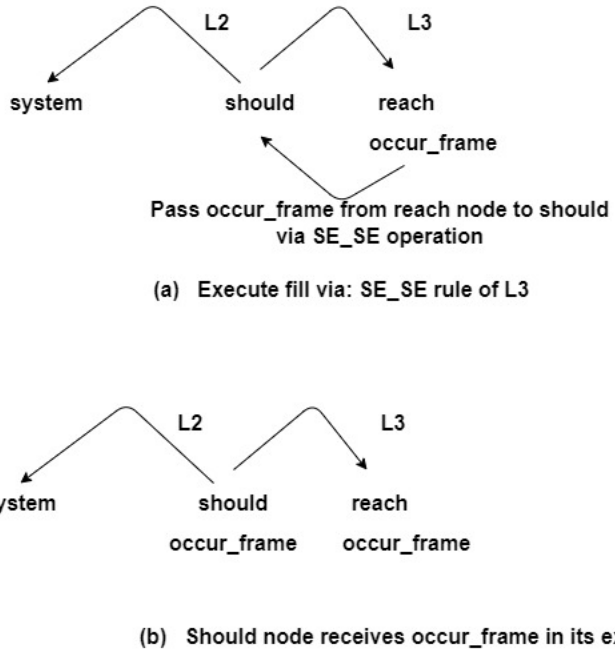


Figure 4.12: (a) Semantics associated with L3 link is created, (b) New occur_frame is propagated to 'should' node after semantic creation step.

connected to the 'reach' node through the L_3 link. The semantic operation associated with the L_3 link is executed again to update the occur_frame in the node 'should'. In Figure 4.14(b), the node 'should' is provided new occur_frame that has been written with 'dataReady' node in the slot 'occur_what_event'.

8. In step 8 shown in Figure 4.13, the 'dataReady' node has made all its connections found in the BINGO solution. The 'dataReady' is removed from the stack with a reduce operation.

9. In step 9, link L_5 is created between nodes 'reach' and 'before'. The node 'before' is then shifted to the stack.

   semantic creation: The semantic propagation rule of L_5 is '$occur_frame : occur_when_before_clock : slot : same$'. The semantic operation transfers the content of the source slot occur_when_before_clock of occur_frame to

Step no.	Stack	Buffer	Action Performed
8.	[system, should,reach]	[before,5 clock cycles]	Action: reduce The stack top has made all its edges found in the BINGO solution. We will pop the stack top 'dataReady' node.
9.	[system, should,reach ,before]	[5 clock cycles]	Action: link_creation link_creation: L5 link is created Semantic_creation: Execute the fill_via slot semantic operation associated with L5 edge. The content in the source address (slot: occur_when_before_clock SE: occur_frame) of the node 'before' is transferred to the destination address (slot: occur_when_before_clock SE: occur_frame) of node 'reach' Semantic Update: Since semantics of an SE of node 'reach' is modified by L5 edge creation, the update in semantics will be propagated to the head of reach node. Head of node 'reach' is 'should' through L3 edge. Semantic operation associated with L3 edge is executed again to update the occur_frame that exist in the node 'should'.

Figure 4.13: Steps 8 - 9 in the semantic creation process of connecting nodes and moving buffer elements to the stack.

the destination slot occur_when_before_clock of occur_frame in the 'reach' node. The semantic creation operation for L5 is illustrated in Figure 4.16.

semantic update: The source slot in 'before' node does not have any content. The transfer of the source slot to the destination slot does not cause any change in the semantics of the destination node 'reach'. Hence, a semantic update is not initiated.

10. In step 10 in Figure 4.15, link L_6 is created between nodes 'before' and '5 clock cycles'. The node '5 clock cycles' is then shifted to the stack.

    semantic creation: The semantic propagation rule of L_6 is '$occur_frame : occur_when_before_clock : node$'. The semantic creation operation for L_6 is illustrated in Figure 4.17. As shown in the figure, the '5 clock cycles' node is placed in the slot 'occur_when_before_clock' of the 'occur_frame'. The semantics are stored in the expr_list of the 'before' node.

    semantic update: The semantic update operation is illustrated in Figure 4.14(b) and

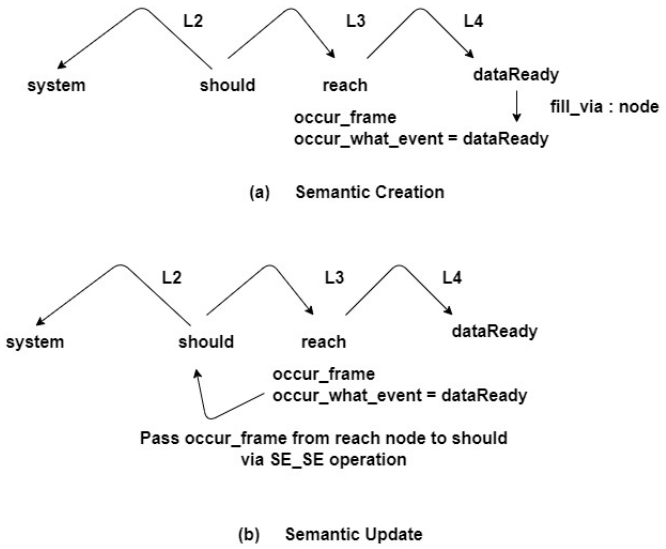


Figure 4.14: (a) Semantic creation step for L4 link creation, (b) Semantic created at reach node is propagated to its head node.

Step no.	Stack	Buffer	Action Performed
10.	[system, should, reach, before, 5 clock cycles]	[ ]	Action: rarc link_creation: L6 edge created Semantic_creation: Execute the fill_via node semantic operation associated with L6 edge. The node '5 clock cycles' is transferred to the destination address (slot: occur_when_before_clock SE: occur_frame) of the node 'before' Semantic_update: Since semantics of an SE of node 'before' is modified by L6 edge creation, the update in semantics will be propagated to the head of 'before' node. Head of 'before' is node `reach' connected through L5 edge. Semantic operation associated with L5 edge is re-executed.  Semantic_update: Since semantics of an SE of node `reach' is modified by the re-execution of L5 rule. Semantic operation associated between node 'reach' and its head 'should' will be executed again. Head of node 'reach' is 'should' through L3 edge. Semantic operation associated with L3 edge is executed again to update the occur_frame that exist in the node 'should'.
11.	'should' node is picked as the root node		Semantic Parsing is completed. The final SE in the expr_list of node 'should' represents the semantics of the spec.

Figure 4.15: Steps 10 - 11 in the semantic creation process of connecting nodes and moving buffer elements to the stack.

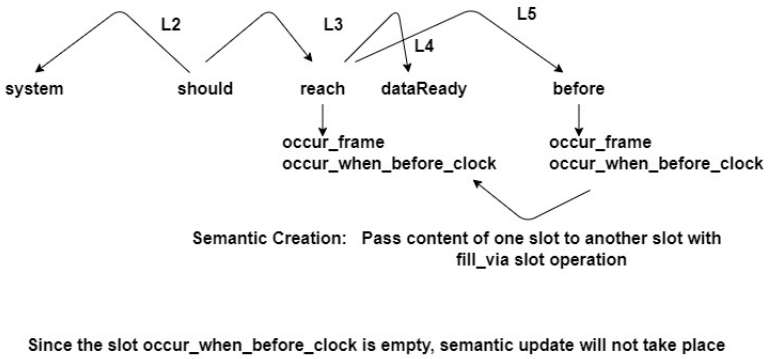


Figure 4.16: Semantic creation associated with L_5 doesn't lead to semantic changes, and no semantic update operation takes place.

4.14(c). The changes in semantics due to L6 link creation are propagated to the root node of the parse tree. The semantics of 'occur_frame' at node 'before' is updated in L_6 link semantic_creation. The update in semantics is propagated to its head node 'reach' as shown in Figure 4.14(b). In Figure 4.14(c), the changes in the 'reach' node semantics is propagated to its head node 'should'.

11. In step 11, the buffer becomes empty. All the nodes are connected in a parse tree according to the BINGO solution. The entire parse tree is rooted at the node 'should'. The SE in the expr_list of the 'should' node represents the meaning of the input spec.

4.6 Grammar Creation Examples

In this section, we will show the creation of declarative grammar rules with contextual information to resolve ambiguities. We also explain the creation of grammar rules for defining conjunction relations in specifications.

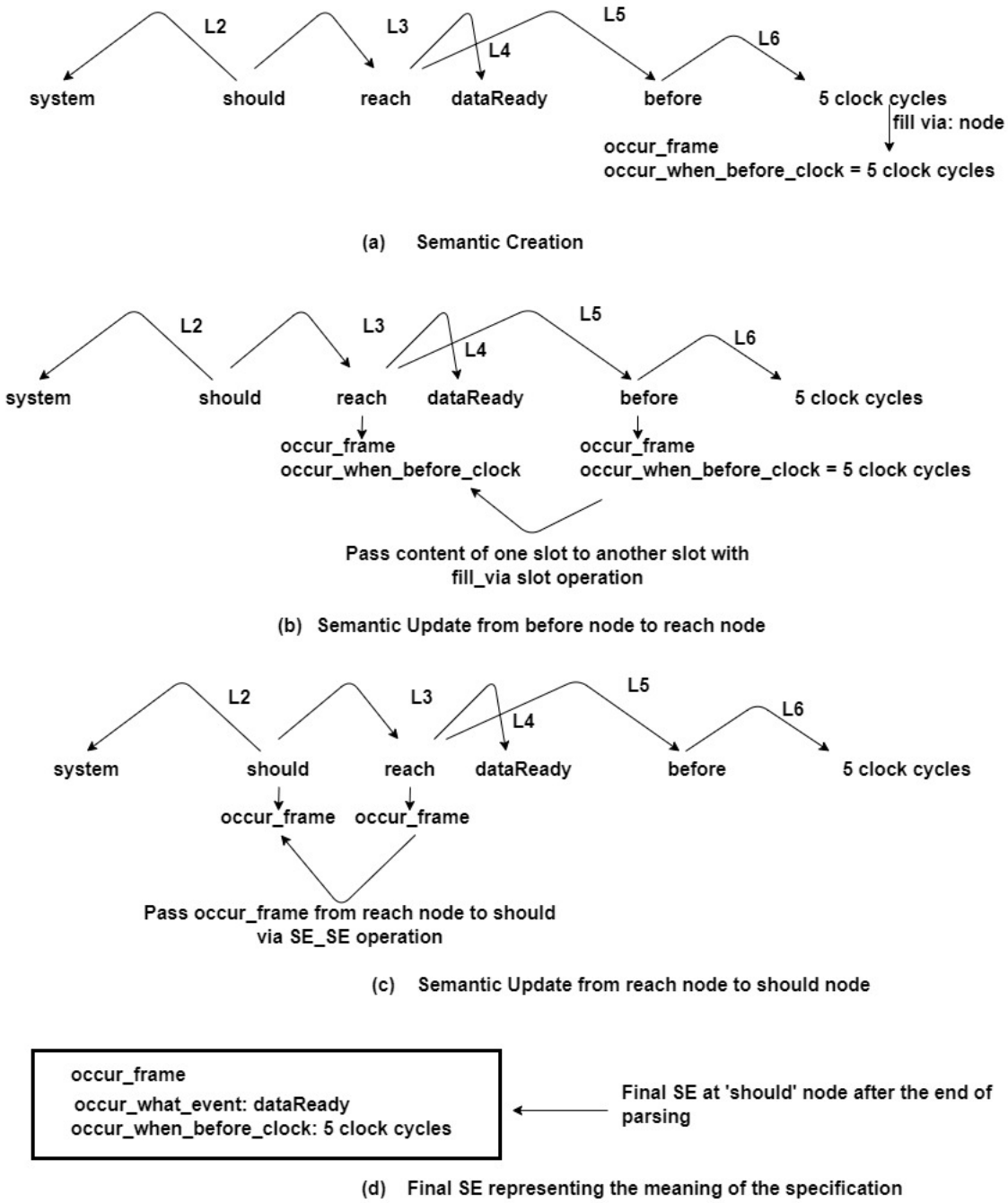


Figure 4.17: (a) Semantic creation associated with the L_6 link is shown, (b) Semantic update operation for SE occur_frame takes place from node 'before' to the 'reach' node, (c) Semantic update operation is executed to propagate semantic changes from the node 'reach' to the root node 'should', (d) Final SE created at the root node 'should'.

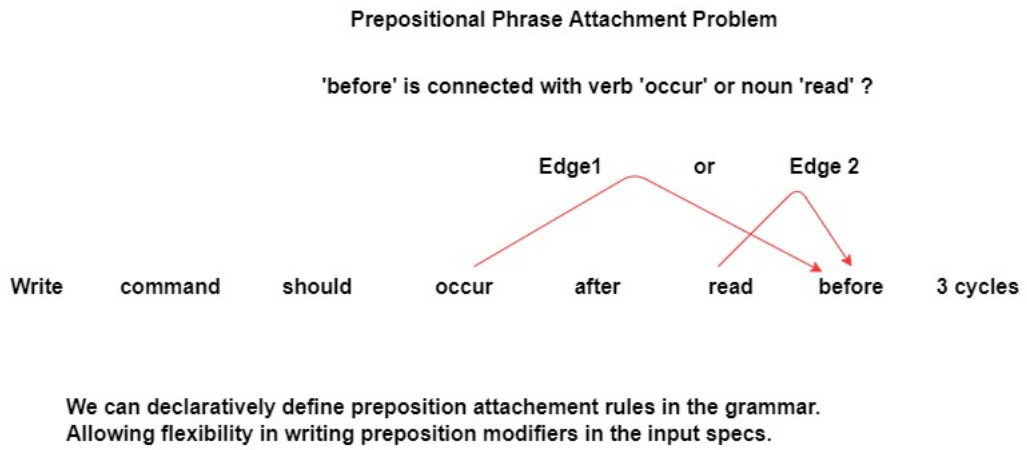


Figure 4.18: Preposition phrase attachment problem with ambiguity to connect a preceding noun or verb with a preposition.

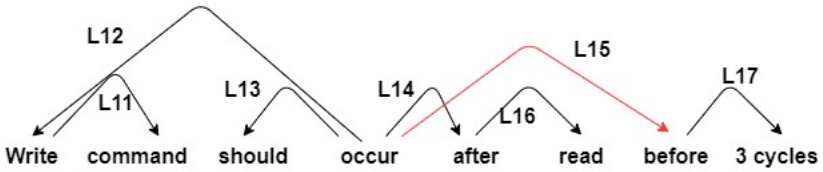
4.6.1 Contextual Grammar Rules Resolving Ambiguities

Figure 4.18 demonstrates an example of ambiguity associated with prepositional phrase attachment. In prepositional phrase attachment, there exists more than one possible connection for a preposition in a sentence. In Figure 4.18, the preposition 'before' can be a modifier of the verb 'occur' through dependency edge 'Edge1'. The node 'before' can also be a modifier of a noun 'read' through edge 'Edge 2', as shown in Figure 4.18. Moreover, in this sentence, there also exists ambiguity in the interpretation of the node 'read'. The semantic role of node 'read' has to be understood as 'read command' and not as a 'read verb'.

In our grammar, we create declarative rules to disambiguate attachments between nodes based on context. Moreover, we can also disambiguate the interpretation of nodes used in the sentence using grammar rules.

Labeling (L_{ij}) of Grammar Rules: In order to explain the grammar rules with ease, we have again labeled the rules as L_{ij}. We have labeled the rules as L_{ij} to indicate the matching links in the grammar. As shown in Figure 4.2, a link rule has two components: syntactic link and semantic link. The link rules that match syntactic and semantic links and do not

Input Sentence : Write command should occur after read before 3 cycles



(a)

```
Write = AN+;;head & S+;occur_expr:ocur_what_command:node;child
          L11                 L12

command = AN-;;child
            L11

should  = Pv+;;child
            L13

occur   = Pv-;;head & S-;occur_expr:ocur_what_command:node;head &
            L13                         L12

          MV+;occur_expr:occur_when_after_command:slot:same;head &
                                L14

         MV+;occur_expr:occur_when_before_clock:slot:same;head
                                L15

after =  MV-;occur_expr:occur_when_after_command:slot:same;child &
                                L14

       J+;occur_expr:occur_when_after_command:node;head
                                L16

read = J-;occur_expr:occur_when_after_command:node;child    or
                                L16

       MV+;read_expr:read_when_before_clock:slot:same;head
                               L18


before =   MV-;occur_expr:occur_when_before_clock:slot:same;child &
                                L15

          J+;occur_expr:occur_when_before_clock:node;head
                                L17

3 cycles = J-;occur_expr:occur_when_before_clock:node;child
                                L17
```

(b)

Figure 4.19: Declarative rules created to define correct attachments between nodes based on context.

violate the link order constraint have the same L_{ij} label.

For example, the following constraints are satisfied for the links labeled L_{11}:

- Syntactic Link: The link L_{11} of the node 'Write' and 'command' match their syntactic link 'AN' with opposite polarity of '+' and '-'.

- Semantic Link: The semantic link in L_{11} is ';child' in node 'command'. The semantic link in L_{11} for the node 'Write' is ';head'. The link L_{11} of the node 'Write' and 'command' match their semantic link with opposite dependency polarity of 'head' and 'child'.

  The L_{11} labeled link has no semantic propagation rule as indicated by a single semicolon followed by the dependency role like ';child' for the node command. The absence of a semantic propagation rule indicates that the node connections do not propagate semantics between them.

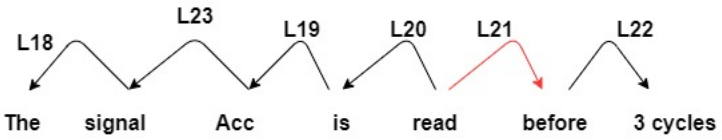
- link order: The order of links in 'Write' is $L_{11} < L_{12}$. The order implies that in the sentence, the node that makes L_{11} labeled link with 'Write' should be closer to the node 'Write' than the node that makes L_{12} labeled link with 'Write'.

  This requirement is satisfied for links L_{11} of nodes 'Write' and 'command' in the specification shown in Figure 4.19 (a). The link order is satisfied since, in this specification, the node 'command' that makes L_{11} labeled link with 'Write' is closer to 'Write' than the node 'should' that makes L_{12} labeled link with 'Write'.

Figure 4.19(b) demonstrates grammar rules to unambiguously parse the specification shown in Figure 4.19(a).

In order to disambiguate prepositional phrase attachments, the grammar allows us to explicitly define the semantic role of words that can be connected to prepositions in different

Input Sentence : The signal Acc is read before 3 cycles



(a)

```
The = D+;;child
          L18
signal = D-;;head & AN+;;child
            L18          L23

Acc = AN-;;head & S+;read_expr:read_what_signal:node;child
          L23                    L19

is = S-;read_expr:read_what_signal:node;head & Pv+;read_expr:read_what_signal:slot:same;child
               L19                                                  L20

read = Pv-;read_expr:read_what_signal:slot:same;head
                 L20
        & MV+;read_expr:read_when_before_clock:slot:same;head
                 L21

before = MV-;read_expr:read_when_before_clock:slot:same;child &
                 L21

          J+;read_expr:read_when_before_clock:node;head
                 L22

3 cycles = J-;read_expr:read_when_before_clock:node;child
                  L22
```

(b)

Figure 4.20: Declarative rules encode semantic context to disambiguate attachment between nodes.

contexts. These semantic roles of connecting words are defined in the link rules of prepositional words.

For example, the link rule L_{15} of node 'before' in Figure 4.19 (b) is:

$$MV-; occur_expr : occur_when_before_clock : slot : same; child$$

According to the semantic link of the above rule, the node 'before' is propagating the content in the slot 'occur_when_before_clock' of SE 'occur_expr' to a head node. It implies that the head node of the preposition has to be connected with some event, and the preposition 'before' is providing semantics that specify the duration within which this event takes place.

In order to connect to the correct head node, we need to explicitly specify the semantic role of the head node in its link rules. The link rules of node 'occur' in Figure 4.19 (b) are:

$$\begin{gathered}(Pv-; ; head \ \& \ S-; occur_expr : ocur_what_command : node; head \\ \& \\ MV+; occur_expr : occur_when_after_command : slot : same; head \\ \& \\ MV+; occur_expr : occur_when_before_clock : slot : same; head \ )\end{gathered}$$

According to the above link rules, the node 'occur' has 'occur_expr' in its expr_list. The 'occur_expr' has a filled slot 'occur_what_command', 'occur_when_after_command' and 'occur_when_before_clock'. It implies that the node 'occur' represents the occurrence of a command before some clock cycles and after the occurrence of another command.

The semantic role of 'occur' matches the semantic role of the head node defined in the link rule of node 'before'.

Finally, the link rules of 'read' that can be another probable head of the preposition 'before' are:

$$(J-; occur_expr : occur_when_after_command : node; child$$

$$or$$

$$MV+; read_expr : read_when_before_clock : slot : same; head)$$

The first link rule has a semantic propagation rule '$occur_expr : occur_when_after_command : node$'. The rule states that the node 'read' fills the slot 'occur_when_after_command' in SE 'occur_expr' through fill via node operation. A 'read' node in the slot 'occur_when_after_command' implies that the node 'read' in Figure 4.19(a), is used in the context of an event that defines the start time of some other event.

The first link rule of 'read' is in a disjunct (or) relation with the rule '$MV+; read_expr : read_when_before_clock : slot : same; head$'. The disjunct rule states that the 'read' can also be a read operation event that expects to be over before some clock cycles. The 'read' node with this link rule matches the semantic role of the head node defined in the link rules of 'before'.

However, the read in the specification is used in the context that is defined by the link rule '$J-; occur_expr : occur_when_after_command : node; child$'.

The parser will not assign 'read' node a link rule that has a context of a read operation for the specification shown in Figure 4.19.

The semantic role with which 'read' is used in the specification does not match the semantic role of the head node in the link rule of 'before'.

Hence, the above example shows that we can explicitly define contextual grammar rules to overcome ambiguity in connecting nodes in a specification. The node 'before' will be

attached to the node 'occur' as defined in the grammar rules.

In Figure 4.20, the node 'before' is used in a context that can be connected with the 'read' node. In Figure 4.20 (b), the link rule of node 'before' is as follows:

$$MV-; read_expr : read_when_before_clock : slot : same; child$$

The rule states that the node 'before' is propagating the content in the slot 'read_when_before_clock' of SE 'read_expr' to a head node in the sentence. It implies that the head node of the preposition can be a read operation, and the preposition 'before' provides semantics that specifies the duration within which this read operation takes place.

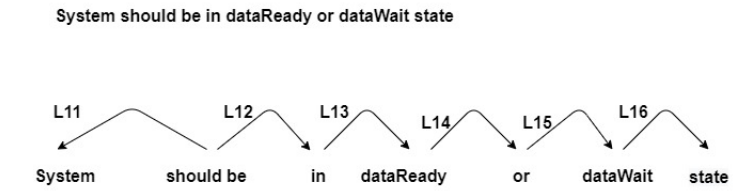
The semantic role of 'read' is explicitly defined as a read operation in the grammar rules, as shown in Figure 4.20(b).

The semantic role of 'read' matches the semantic role of the head node defined in the link rules of 'before' in the context of the specification shown in Figure 4.20.

4.6.2 Grammar Rules Representing Conjunction Relation

A specification with conjunction relation and the grammar to parse this specification is illustrated in Figure 4.21. In Figure 4.21 (b), '<state_category>' represents a category of words that denote system state in a design. For example, 'dataReady' and 'dataWait' are design states and belong to category '<state_category>'. The link rules for '<state_category>' is applied to all the words of this category.

The specification in Figure 4.21 (a) has conjunction 'or' between states 'dataReady' and 'dataWait'. The linkage solution generated by the parser for this specification is shown in



(a)

```
System = S+;;child
              L11

should be = S-;;head & MV+;occur_expr:occur_what_event:slot:same;head
                  L11                  L12

in = MV-;occur_expr:occur_what_event:slot:same;child &
                      L12

     J+;occur_expr:occur_what_event:slot:same;head
                      L13


<state_category> = ( J-;occur_expr:occur_what_event:slot:same;child  &
                                        L13

      evconjoin+;None:None:SE_SE:conj_expr;head &  self;conj_expr:connector_nodes:node;self &
                                    L14

self;occur_expr:occur_what_event:slot:conj_expr:connector_nodes;self  &

self;occur_expr:occur_what_event:slot@conn:conj_expr:conjunction;self

 ) or

( evconjoin-;None:None:SE_SE:conj_expr;child & {ANE+;;head} &
                        L15                                    L16

self;conj_expr:connector_nodes:node;self  )


or  = evconjoin-;None:None:SE_SE:conj_expr;child &
                        L14

       evconjoin+;None:None:SE_SE:conj_expr;head & self;conj_expr:conjunction:node;self
                        L15

state = ANE-;;child
              L16
```

(b)

Figure 4.21: Rules demonstrating the connection of conjunction and conjunct in a sentence.

nodes →	System	Should be	in	dataReady	or	dataWait	state
Linkage →	L11	L11 L12	L12 L13	L13 L14	L14 L15	L15 L16	L16

Output of BINGO parsing for spec: System should be in dataReady or dataWait state

Figure 4.22: Output of BINGO parsing for specification and grammar rules of Figure 4.21.

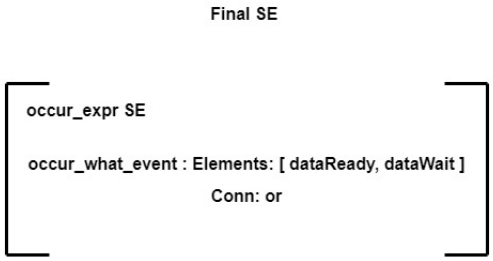


Figure 4.23: The SE representing the meaning of Spec of Figure 4.21.

Figure 4.22.

conj_expr SE: In order to create semantics for specifications with a conjunction, we introduce a conj_expr SE with two slots: 'connector_nodes' and 'conjunction'. The slot 'connector_nodes' stores conjunct nodes that are in a conjunction relation. The slot 'conjunction' stores the conjunction nodes like 'and', 'or'.

The conj_expr is an intermediate SE that assists in placing the conjuncts and conjunction nodes in respective slot attributes of the final SE. As illustrated earlier, in Chapter 3, a slot has the following two attributes:

1. Elements: Elements is a list that contains nodes or SEs.

2. Conn: The Conn attribute contains the conjunction node that connects the nodes or SEs in the Elements list in a conjunction relation.

For example, the SE representing the meaning of the specification 'System should be in dataReady or dataWait state' is illustrated in Figure 4.23. In Figure 4.23, the Elements attribute of the slot 'occur_what_event' in the SE contains the 'dataReady' and 'dataWait' conjunct nodes. The Conn attribute of the slot 'occur_what_event' stores the node 'or' that specifies the conjunction relation between the nodes stored in the Elements list.

Semantic Creation Process for Conjunction Spec: We now present the process for

Step no.	Stack	Buffer	Action Performed
1.	[system,should be,in, dataReady]	[or,dataWait,state]	
2.	[system,should be,in, dataReady,or]	[dataWait,state]	Action: link_creation L14 link created There are three self link rules that are in conjunct with L14 link rule of node <state_category>. There is one self link in conjunct with L14 link in the rule of node 'or'. All these self links are executed and subsequent semantic creation and semantic update operations will take place.

Figure 4.24: A parse configuration where stack top is 'dataReady' and buffer head is 'or'.

creating the SE shown in Figure 4.23 according to the grammar rules given in Figure 4.21 (b). In our explanation, we will skip the common semantic creation steps illustrated in section 4.5. We have highlighted the steps that signify the role of self links in grammar and the propagation of semantics through conj_expr. The connecting links in the following steps are taken from the BINGO solution shown in Figure 4.22.

1. We start with the initial configuration of the parse shown in Step 1 of Figure 4.24. In this configuration, the stack top is 'dataReady', and the buffer front node is 'or'. The parse tree corresponding to this configuration is shown in Figure 4.25. The parse tree shows the semantic creation between nodes 'in' and 'dataReady' based on the L_{13} connecting link.

2. In step 2 shown in Figure 4.24, link L_{14} is created between nodes 'dataReady' and 'or'.

   semantic_creation: The semantic propagation rule of L_{14} is '$None : None : SE_SE : conj_expr$'. The semantic operation transfers the SE conj_expr from the expr_list of node 'or' to the node 'dataReady', as illustrated in Figure 4.26. Initially, the content of conj_expr is empty, and hence no semantic update operation takes place.

   The node 'dataReady' is assigned link rules mentioned for '<state_category>' in the

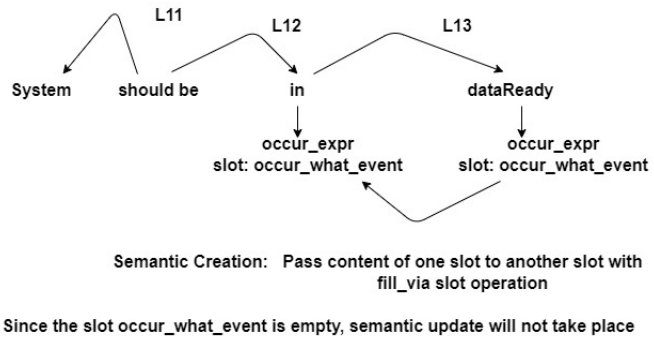


Figure 4.25: Partial parse tree representing parse till the node 'dataReady' of conjunction Spec given in Figure 4.21.

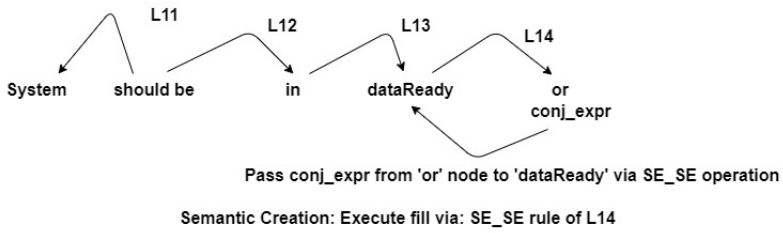


Figure 4.26: Link L_{14} is created between nodes 'or' and 'dataReady' and its semantic propagation rule is executed.

grammar shown in Figure 4.21 (b). The link rules assigned to 'dataReady' in the BINGO solution are as follows:

$$(J-; occur_expr : occur_what_event : slot : same; child$$
$$\&$$
$$evconjoin+; None : None : SE_SE : conj_expr; head)$$

The above rules are labeled L_{13} & L_{14} in the grammar of Figure 4.21 (b). As shown in the grammar, these link rules are in conjunct with the following three self link rules:

$$self; conj_expr : connector_nodes : node; self$$
$$\&$$
$$self; occur_expr : occur_what_event : slot : conj_expr : connector_nodes; self$$
$$\&$$
$$self; occur_expr : occur_what_event : slot@conn : conj_expr : conjunction; self$$

Similarly, the link rule L_{14} of node 'or' is in conjunction with the following self link rule:

$$self; conj_expr : conjunction : node; self$$

The self link rules in conjunction with other non-self link rules of a node imply that we have to execute the semantic propagation rule of self links whenever the other non-self link rules are executed.

Purpose of self link rules: We have created self link rules to manipulate SEs that exist at the node's expr_list. When semantic information is received or propagated

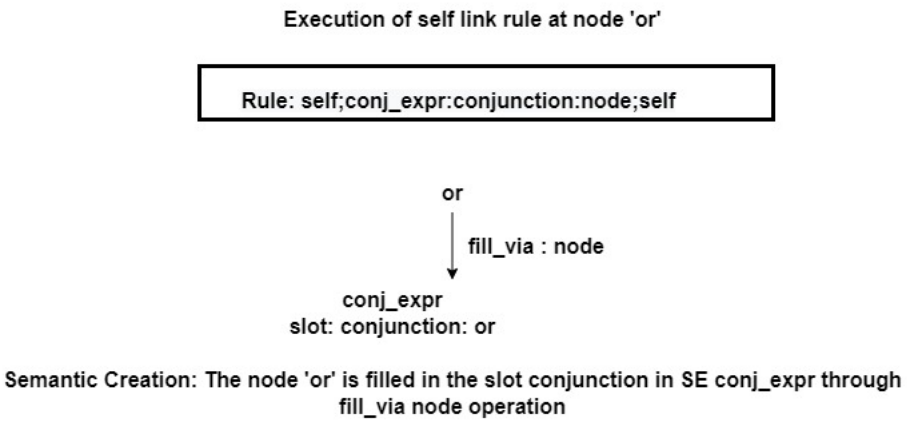


Figure 4.27: Semantic propagation rule of self link at node 'or' is executed.

from a node then the self links are executed to perform further processing on the semantic information on that node.

It implies that when 'dataReady' receives information through L_{14} link, all three self links of 'dataReady' node must be executed to update the SE on node 'dataReady'. Similarly, the self link rule of node 'or' will be executed when the semantic information is sent through its L_{14} link.

Execution of self link rule of 'or' node: Execution of the self link of 'or' is displayed in Figure 4.27. According to the rule, the slot 'conjunction' of SE 'conj_expr' is filled by the node 'or' through a fill_via 'node' operation.

Propagation of SE modified by self link rule: A 'self' specified in the dependency role component of a self link rule indicates that the SE modified by the rule remains at the node and is not propagated to a head node by the self link rule. The modified SE is propagated to a head node through other non-self link rules of the node that are in conjunction (&) with self link rules. For example, after the execution of self link rule at 'or' node, the modified 'conj_expr' is propagated to the head node through the L_{14} link as illustrated in Figure 4.26.

Execution of self link rule of 'dataReady' node: The 'dataReady' node has three self link rules to capture the conjunction relation in conj_expr and to propagate the conj_expr content to SEs in a head node. The execution of these three self link rules is explained as follows:

- Execution of semantic propagation rule of '$self; conj_expr : connector_nodes : node; self$' is shown in Figure 4.28. According to the self link rule, we pass the 'dataReady' node to the slot 'connector_nodes' in the conj_expr SE at the 'dataReady' node. **The rule has captured one of the conjuncts in the connector_nodes slot of conj_expr SE.**
- Execution of semantic propagation rule of '$self; occur_expr : occur_what_event : slot@conn : conj_expr : conjunction; self$' is shown in Figure 4.29. In this rule, semantic propagation takes place through fill_via 'slot@conn'. The content in the Elements attribute of the slot 'conjunction' is propagated to the 'Conn' attribute of the destination slot 'occur_what_event', as illustrated in Figure 4.29. **This rule has propagated the captured conjunction ('or') from conj_expr to occur_expr slot of 'occur_what_event' SE.**
- Execution of semantic propagation rule of '$self; occur_expr : occur_what_event : slot : conj_expr : connector_nodes; self$' is shown in Figure 4.30. This self link rule propagates the conjunct stored in connector_nodes slot to occur_what_event slot of occur_expr SE, as illustrated in Figure 4.30 (a). In this semantic propagation of slot content, both the destination and source SEs are at the same 'dataReady' node.

  Since an occur_expr at node 'dataReady' is updated, the update in the SE will be propagated to the root node of the partial parse tree.

  The updated occur_what_event slot at the node 'dataReady' is propagated to

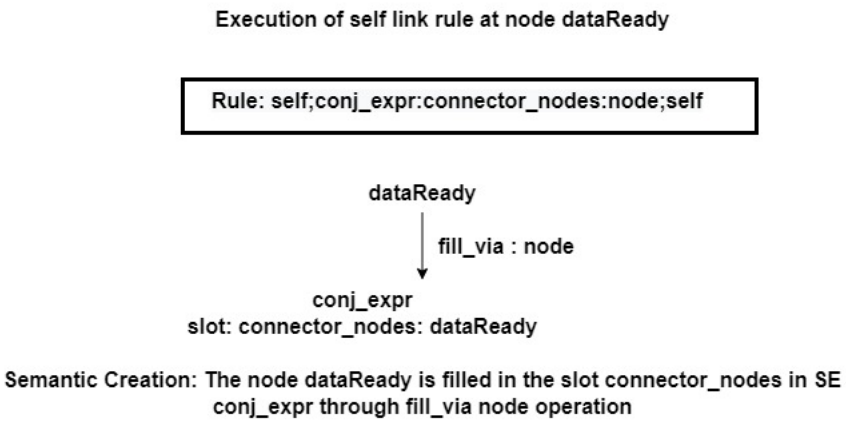


Figure 4.28: A self link rule semantic propgation rule is executed at the 'dataReady' node.

node 'in' through L_{13} link. A semantic change in this slot triggers semantic update operation through the L_{12} link, as shown in Figure 4.30.

The execution of all the self link rules has propagated the conjunction 'or' and conjunct node 'dataReady' to the respective attributes of the slot 'occur_what_event' of occur_expr SE at the root node 'should be'.

3. In step 3 shown in Figure 4.31, link L_{15} is created between nodes 'or' and 'dataWait'. The link will allow the propagation of conj_expr SE from the 'dataWait' node to the 'or' node through semantic propagation rule $None : None : SE_SE : conj_expr$, as illustrated in Figure 4.32.

   The node 'dataWait' is assigned link rules mentioned for '<state_category>' in grammar of Figure 4.21 (b). According to the context with which 'dataWait' is used in the specification, the BINGO solution found the following link connections for the 'dataWait' node:

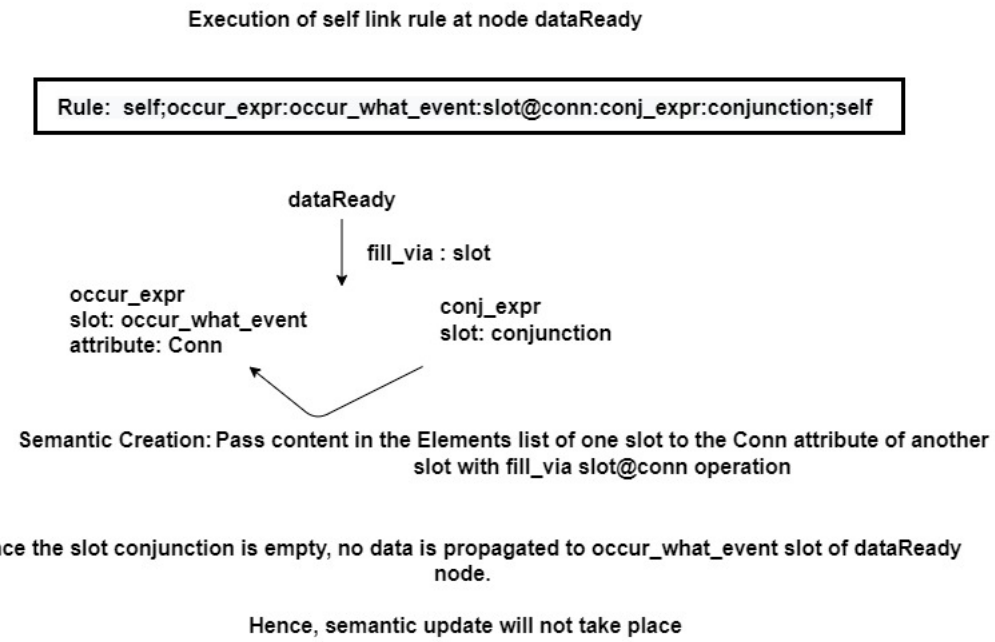


Figure 4.29: An execution of self link rule that propagates slot Elements attribute to the Conn attribute of the destination slot.

$$(evconjoin-; None : None : SE_SE : conj_expr; child$$
$$\&$$
$$\{ANE+; ; head\})$$

The above rules are labeled L_{15} & L_{16} in the grammar of Figure 4.21 (b). The link rule L_{16} in curly brackets indicates an optional link for the node 'dataWait'. The notion of optional links is adopted from the link grammar [45]. As shown in the grammar, these link rules are in conjunct with the following self link rule:

$$self; conj_expr : connector_nodes : node; self$$

The self link rule will be executed when any one of the conjunct links is executed. Execution of self link updates the conj_expr at 'dataWait' node as illustrated in Figure

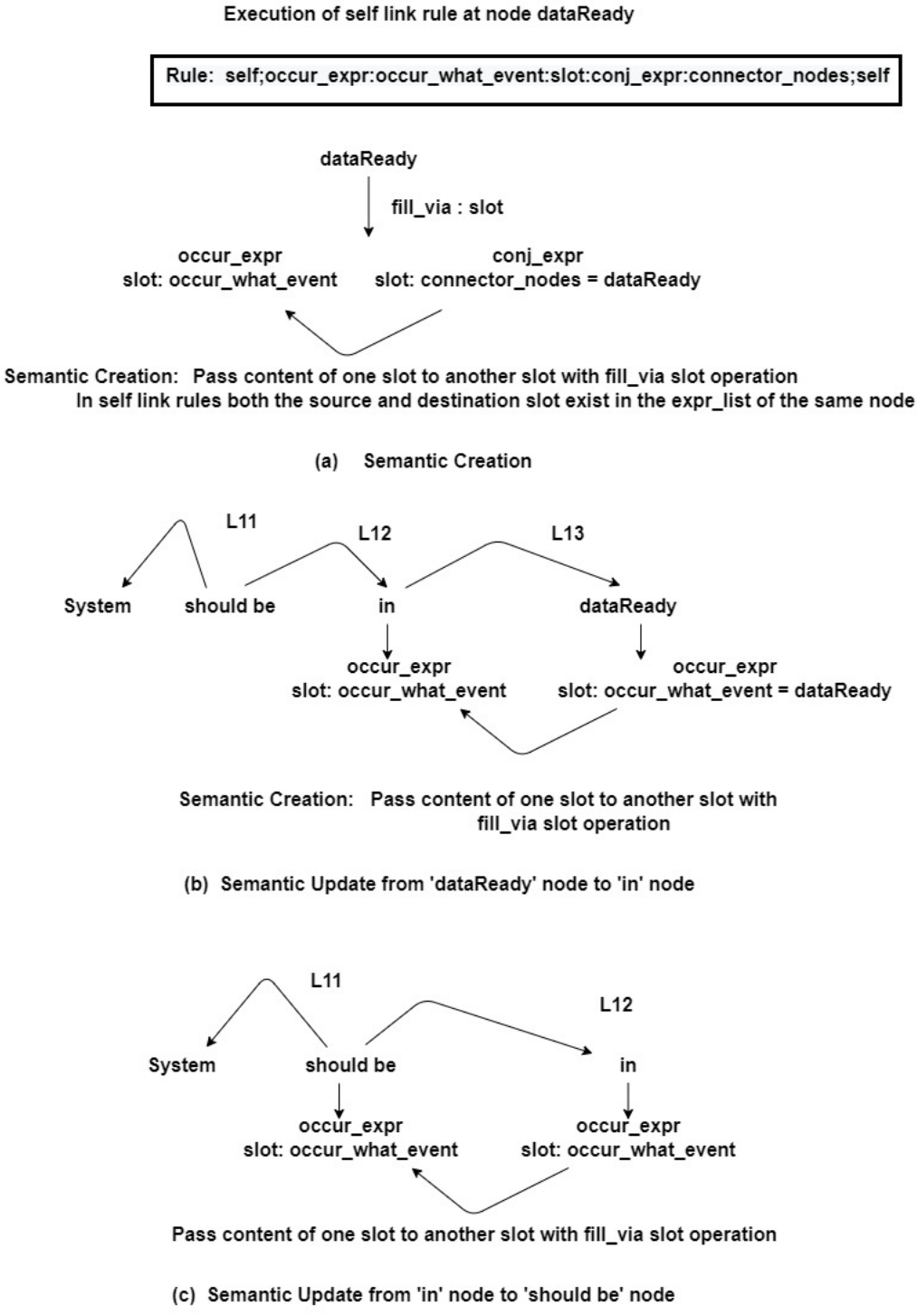


Figure 4.30: Semantic changes due to the execution of self link rule is propagated till the root node.

Step no.	Stack	Buffer	Action Performed
3.	[system,should be,in, dataReady,or,dataWait]	[state]	Action: link_creation L15 link created between nodes `or' and `dataWait' All the self link rules in conjunct with L15 links are executed. Execution will lead to semantic creation and semantic update operations till the root node 'should be'

Figure 4.31: Step 3 creates link connection between 'or' and 'dataWait' nodes.

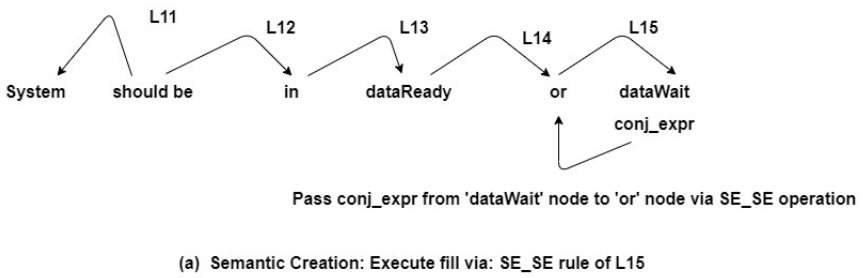


Figure 4.32: Link L_{15} is created between nodes 'or' and 'dataWait', and its semantic propagation rule is executed.

4.33. The changes in conj_expr trigger semantic_update operation through the L_{15} link and L_{14} links. The semantic update results in a conj_expr that has two nodes in the 'connector_nodes' slot at the node 'dataReady', as illustrated in Figure 4.34. The self links in the rules of the 'dataReady' node further process the conj_expr SE to update the occur_expr SE at the 'dataReady' node. The resulting occur_expr at 'dataReady' node is shown in Figure 4.34 (b).

The link L_{13} propagates the updated occur_expr SE from 'dataReady' node to 'in' node. Updated SE propagation continues until the root node 'should be'. The resultant occur_expr SE at 'should be' node represents the final meaning of the overall specification, as illustrated in Figure 4.35.

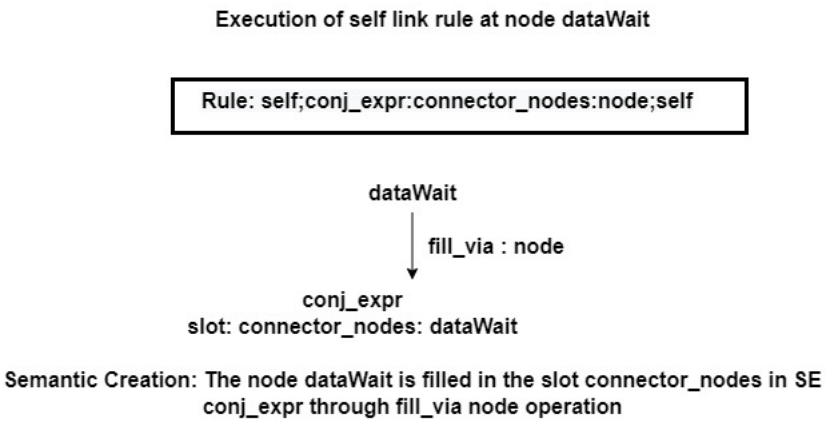


Figure 4.33: Execution of semantic propagation rule of self link at 'dataWait' fills the connector_nodes slot of conj_expr SE.

4.7 Evaluation

The BINGO framework is written in JavaScript and is executed in the Node.js platform. All experiments were run on a machine with 1.8 GHz Intel Core i7-8550u processor and 16GB RAM. We evaluated the framework by creating the grammar with syntactic and semantic link rules that can parse specifications found in documents. As shown in Table 4.1, our grammar consisted of a vocabulary of 440 words. We created 279 charts and 1079 rows to represent syntactic and semantic connections of the words in our grammar. We evaluated our framework on both Register-Transfer-Level (RTL) and high-level specifications written in English. High-level specifications are architecture-dependent and require the knowledge of architecture-specific concepts like the attributes of transactions and responses. In the subsequent sub-sections, we present the nature of English specifications that were used in the evaluation.

Table 4.1: Grammar size for specifications tested

Total Words	Total Chart	Total rows
440	279	1079

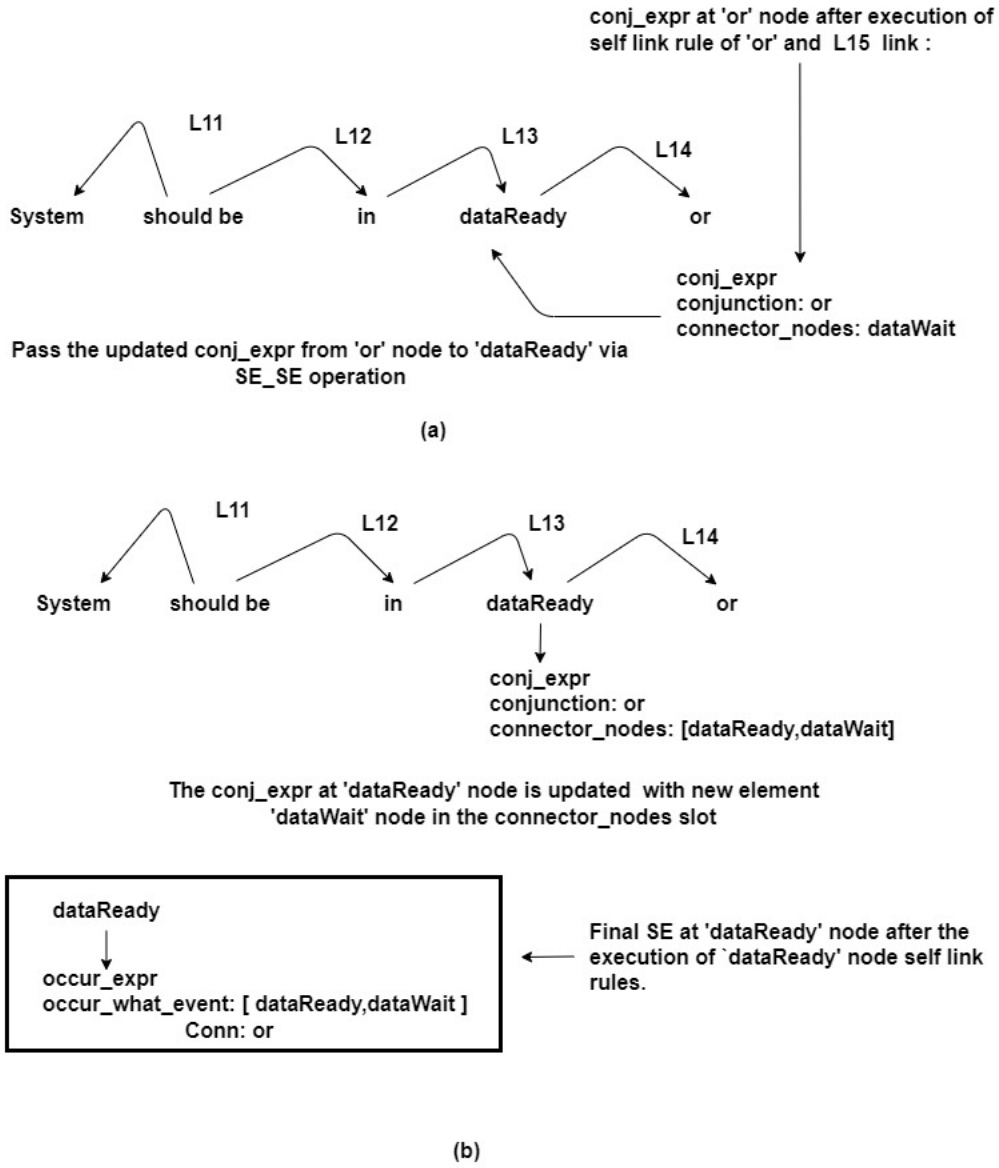


Figure 4.34: The update in the conj_expr at 'dataWait' node is propagated to 'dataReady' node through multiple semantic update operations.

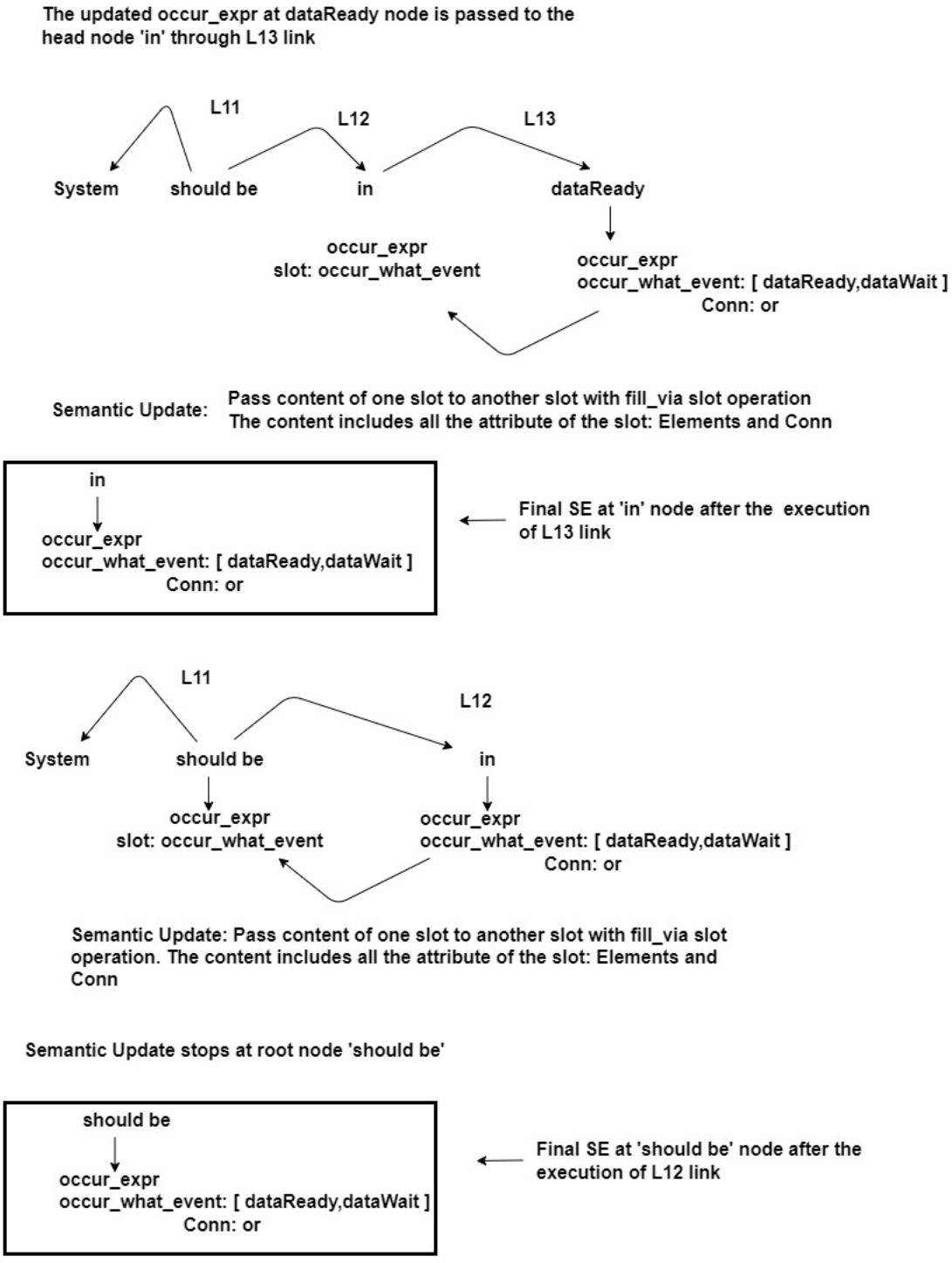


Figure 4.35: The occur_expr SE updated at 'dataReady' node is propagated to the root node 'should be' through multiple semantic update operations.

SPECS	TRANSLATIONS
AWID must remain stable when AWVALID is asserted and AWREADY is low.	assert property (@(posedge clk) (AWVALID == 1) & (AWREADY == 0) |-> $stable(AWID));
A value of X on AWADDR is not permitted when AWVALID is high	assert property (@(posedge clk) (AWVALID == 1) |-> AWADDR != X);
When BVALID is asserted then it must remain asserted until BREADY is high.	assert property (@(posedge clk) (BVALID == 1) |-> $stable(BVALID)[*1:$] (BREADY == 1));
ACREADY should be asserted within MAXWAITS cycles of ACVALID being asserted	assert property (@posedge clk) (ACVALID == 1) |-> ##[1:MAXWAITS] ACREADY);

Figure 4.36: Specifications with RTL information are translated to SVA code.

4.7.1 RTL Specifications

We evaluated the framework on the RTL specification of AMBA AXI3 [86], and the AMBA ACE protocol checker user guide [87]. The documents consisted of 123 RTL specifications in English. The syntactic structures of these specifications contained conjunctions and conditional words like if, when, etc. For example, the following specs contain conjunctions and conditional words:

1. When ACVALID is asserted, then it must remain asserted until ACREADY is asserted.
2. AWID must remain stable when AWVALID is asserted and AWREADY is low.
3. When AWVALID is high and AWCACHE[1] is low then AWCACHE[3:2] are also low.
4. CSYSREQ is only permitted to change from HIGH to LOW when CSYSACK is high.

There are also specifications in these documents where prepositional words are also used to denote antecedent and consequent relations in the specification. For example,

1. CDREADY should be asserted within MAXWAITS cycles of CDVALID being asserted.
2. BVALID is low for the first cycle after ARESETn goes high.

However, these RTL specifications had **little syntactic and semantic variations**. For example, the specification ‘A value of X on AWLEN is not permitted when AWVALID is

HIGH' is syntactically similar to the specification 'A value of X on AWSIZE is not permitted when AWVALID is HIGH'. The only difference between these specs is the use of different signal names, AWLEN and AWSIZE. That prompted us to write linguistic variations of these specifications and then evaluate all these variations in our tool. We created variations by using synonyms and changing the order of words in the existing specifications.

The following are four variations of the specification 'A value of X on RDATA valid byte lanes is not permitted when RVALID is high.'

1. If RVALID is high then value on Rdata valid byte lanes cannot be X.
2. Rdata valid byte lanes cannot be assigned X if signal RVALID value is high.
3. If value of signal RVALID is high then a value of X on signal Rdata valid byte lanes is not permitted.
4. signal Rdata should not have a value of X if signal RVALID value is high.

The system generated the same output assertion for all the above specifications: assert property(@(posedge clock) (RVALID == 1) | − > (RDATA != X));

It was necessary to test these variations to ensure that the tools allow flexibility in writing specifications in the English language. The tool successfully translated all the assertions to SVA code.

Previous work like [22, 27, 28] also evaluated their translation framework on specifications given in [86]. The work in [22], alleviates the manual creation of the grammar by automatically learning the grammar using a training set of natural language assertions and their corresponding formal representations. However, the overall sentences in the test set lacked variations in sentence structure. For example, a sentence in the test-set is 'A value of X

on AWID is not permitted when AWVALID is HIGH' and the other is 'A value of X on AWADDR is not permitted when AWVALID is HIGH'. Both these sentences are equivalent except for the signals' name, 'AWID' in the first sentence and 'AWADDR' in the second sentence. Similarly, the chatbot framework of [28], and the subtree-analysis based framework of [27] were only evaluated on the specifications of [86] that lacked variations in sentence structures.

4.7.2 ACE Protocol Specifications

In order to evaluate the framework for different types of sentence structures and wide-ranging semantics, the proposed approach was tested on the protocol rules of ACE and ACE-Lite AMBA architecture that are given in [87]. The protocol contains specifications for transactions and their attributes. Specifications cover semantics ranging from requirements of cache-line size for transactions, sharing of channels among modules, signal requirements for different responses between slaves and masters modules, data-width of channels, etc.

We created grammar rules to generate semantic frames for high-level abstraction specs of the ACE protocol checker document [87]. We successfully parsed a total of 113 high-level specs. A few example specifications that were successfully translated to semantic frames by our framework are:

1. WriteUnique,WriteLineUnique,WriteBack,WriteClean and Evict transactions are required to have AWBURST != 1 .

2. The total number of transaction bytes must be less than the cache line size.

3. If shareable access is inner domain or outer domain then AXlock must be 0.

4. A slave must not give an Isshared(RESP[3] = 'b1) response to a readsnoop, readunique,

cleanunique, cleaninvalid, makeinvalid or makeunique transaction.

5. A slave must not give an Isshared and PassDirty response to a ReadNotSharedDirty.
6. If transaction data transfer is in progress then RRESP must be constant.
7. The EXOKAY response is only permitted for a writenosnoop transaction.
8. Cache line size must be defined as 16, 32, 64, 128, 256, 1024, or 2048 bytes.
9. For a data bus width or snoop data width of 32 bits, the minimum cache line size is 16 bytes and the maximum cache line size is 64 bytes.
10. At the end of simulation, all snoop transactions have completed.
11. A master must issue barrier transactions on both the read and write channels.
12. When ACVALID is asserted, ACSNOOP must be one of the legal value.
13. CR payload must remain stable while CRVALID is asserted and CRREADY is de-asserted.
14. A DVM response must use CRRESP[4:0] = 5'b000x0.

4.7.3 UART and Memory Controller Specifications

We further evaluated the tool on different functionalities of RTL assertions. We manually wrote the spec of UART [88] and memory controller [89] in English and then used our tool to generate the corresponding SVA or semantic frames. In total, we wrote 80 specifications: 40 Assertions of UART and 40 assertions of the memory controller. A few of these memory controller specifications are listed below.

1. If an active command is issued then the controller cannot issue another active command within trc (6 clock cycles).

2. If an active command is issued then the next active command should be issued within 12000 clock cycles.

3. If the memory controller has issued an active command then read or write command cannot be issued within tRCD cycles.

4. When a read command is in progress, the addr[10] is set to 0.

5. If data transfer size is 64 bits, each read or write operations takes 2 cycles.

6. In read cycle time, the output enable and chip select signals are asserted.

7. If an active command is issued to bank 0 then an active command cannot be issued to other banks 1,2 and 3 within tRRD.

8. Chip select signal and address signal should be stable for a minimum 15 clock cycles before data is valid.

A few of UART specifications are listed below:

1. If register fcr[7:6] is set to 01 and amount of data in register receiver buffer is less than 4 bytes then value of register iir can never be 010.

2. If register ier[1] is assigned 0 then register iir[3:1] should never be 001.

3. On reset the value of register ier should be 00h.

4. If we set register ier[2] to 1 and register lsr[1] has a value of 1 then value of register iir[3:1] should be 011.

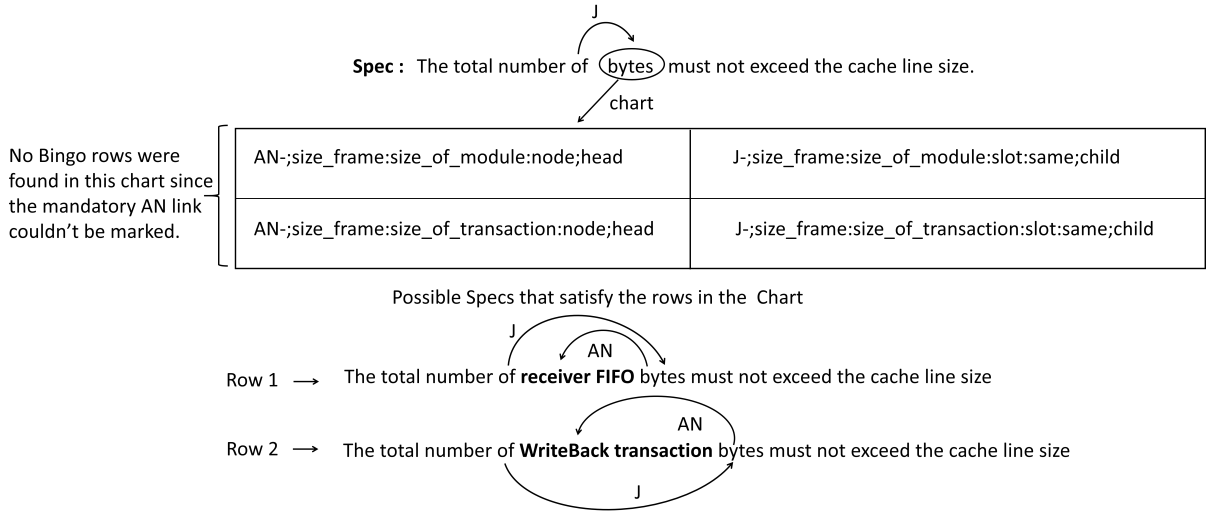


Figure 4.37: A BINGO row does not exist for incomplete specifications and we cannot create semantic frames for these specifications.

5. If register lcr[6] is set to 1 then signal STX_PAD_O should be 0.

6. The registers can be accessed when the 7th (DLAB) bit of the Line Control Register is set to 1.

7. The divisor latches can be accessed by setting the 7th bit of LCR to 1.

Correct translations were generated for all of the 80 specifications.

4.7.4 Detecting Incomplete Specifications

Our framework could not infer data that was not explicitly present in the specification. These incomplete specifications were detected when no BINGO rows were generated. Figure 4.37 illustrates an example of an incomplete specification 'The total number of bytes must not exceed the cache line size'. The specification does not explicitly specify a module name or a transaction whose bytes are being referred. As shown in this figure, the bytes could

belong to a receiver FIFO module of UART or can be a part of the WriteBack transaction of the AMBA protocol.

In Figure 4.37, the chart represents the syntactic-semantic connections for the node 'bytes'. In this chart, the node 'bytes' require an AN- syntactic link to fill either a size_of_module or size_of_transaction slot in the size_frame. In the example specification, the AN- linkage requirement for the node 'bytes' could not be satisfied, and no BINGO row was found for the spec. Consequently, no SE could be produced from this incomplete spec. We had to rewrite the incomplete specifications with the complete information needed for accurate understanding. The rewritten specs were accurately parsed and translated to semantic frames.

Few incomplete specifications and their rewritten forms are illustrated below:

1. **Incomplete Spec 1**: A normal transaction that shares its ID with an outstanding barrier or DVM message must not be issued.
   Re-written Spec 1: If a transaction Id of a normal transaction is equal to **the transaction Id** of an outstanding barrier or DVM message then the normal transaction should not be issued.

2. **Incomplete Spec 2**: The total number of bytes must not exceed the cache line size.
   Re-written Spec 2: The total number of **transaction** bytes must not exceed the cache line size.

3. **Incomplete Spec 3**: The ID of a write barrier must not be the same as any outstanding non-barrier transaction.
   Re-written Spec 3: The ID of a write barrier must not be same to the **id of** any outstanding non-barrier transaction.

4. **Incomplete Spec 4**: The TLB invalidate message indicated by ARADDR is not supported.

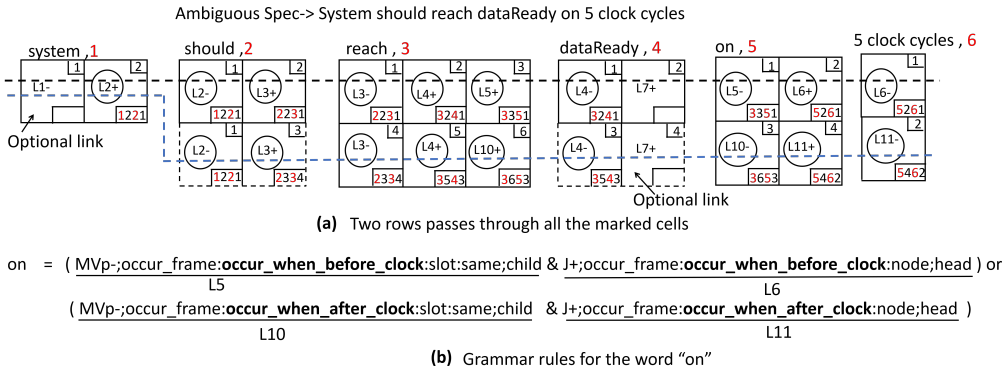


Figure 4.38: Ambiguous spec generates more than one BINGO row.

Re-written Spec 4: An error should be generated when TLB Invalidate message occurs and ARADDR is not valid.

4.7.5 Detecting Ambiguous Specifications

In our framework, ambiguous sentences can be detected when more than one BINGO row with complete coverage is generated in the parsing stage. Figure 4.38 shows an example of an ambiguous specification 'System should reach dataReady on 5 clock cycles.' The specification is ambiguous since it is unclear if the dataReady state should reach before or after 5 clock cycles. The node 'on' can connect with node '5 clock cycles' using two combinations of link rules as illustrated in the Figure 4.38 (a). The connection with link L_6 passes the node 5 clock cycles to the slot occur_when_before_clock, and the connection with link L_{11} propagates the 5 clock cycles to the slot occur_when_after_clock. The connections result in two different interpretations that cover all the nodes in the spec.

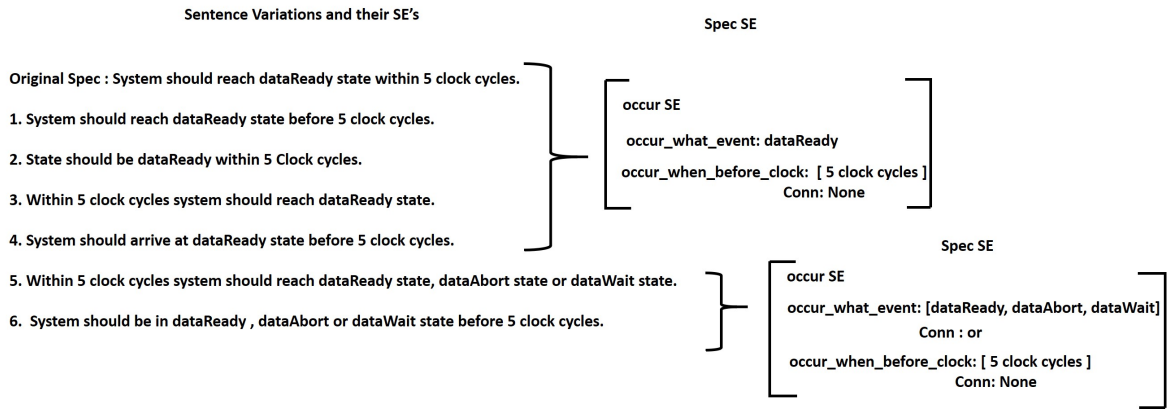


Figure 4.39: Specs with the same meaning but different word order generates the same SEs.

4.7.6 Multiple Variations of Specifications

We also tested sentence structure variations of specifications and generated the same SEs for specs with the same intent. Figure 4.39 illustrates some variations of the specification 'System should reach dataReady state within 5 clock cycles' that were translated to the same SE. The first four specifications have the same intent and are translated to the same SE as illustrated in Figure 4.39. We added conjunction with two variations of the specification and generated the same SEs, as shown in sentences 5 and 6 in Figure 4.39.

4.7.7 Results and Comparison

Table 4.2: Specification types, count, and average JavaScript processing time for Specs

Spec type	Count of Spec	Avg. time
RTL Spec	123	245 ms
High Level Spec	113	526 ms
Memory controller Spec	40	101 ms
UART Spec	40	178 ms

We report the framework's performance in Table 4.2 in terms of the type of specifications and number of specifications parsed, and the average time taken to parse each specification.

In Table 4.2, RTL Spec type under the Spec type column refers to low abstraction level specifications that have explicitly defined signals and registers in specifications. We successfully created the grammar for 123 specifications of RTL Spec type and generated the corresponding SVA code. A small set of these specifications with the translated SVA is shown in Figure 4.36.

In the second row of Table 4.2, the High level Spec type represents specifications of a higher abstraction level that can only be translated to Frames due to the lack of low-level design variable names in these specs. These are the specs taken from the AMBA ACE protocol document [87].

In Table 4.2, the "Avg. time" column represents the average time taken to parse and create semantic frames for each spec. For example, it took 30.2 seconds to create semantic frames for all 123 RTL Spec, which gives an average time of 245 ms to parse each RTL spec. Specifications with many conjuncts took the maximum amount of time to parse. For example, the specification taken from [87]: "A slave must not give an Isshared (RESP[3] = 'b1) response to a readsnoop, readunique, cleanunique, cleaninvalid, makeinvalid or makeunique transaction" takes 2.5 seconds to parse.

We have reported the comparison of our work with earlier rule-based works [16, 20, 27] in Table 4.3. The earlier works in rule-based translation have only reported translation of low-level specifications taken from AMBA AXI 3 bus protocol [86]. These specifications are marked as AXI3 in Table 4.3 under the column 'Count of spec Translated'. The earlier works can be categorized as follows:

Attribute-Grammar based Approach:

An attribute grammar-based translation of specifications is proposed in [20]. As reported in this column, this work translated 52 specs of AXI3 to SVA. However, the approach could not

Table 4.3: Comparison with previous rule-based works on Hardware Assertions Synthesis.

Paper (Year)	Spec type tested	Multi-clock spec translated	Arch. Tested	Detected Ambiguity	Detected Incompleteness	Count of Spec Translated
[20] (2013)	Low Level Lacked Variations	No	AXI3	No	No	52 (AXI3)
[16] (2014)	Low Level Lacked Variations	-	AXI3	No	No	100 (AXI 3)
[27] (2019)	Low Level Lacked Variations	-	AXI3	No	No	71 (AXI3)
This work	Low Level with Variation, High Level Specs with variations	Yes	Memory Controller, UART, AXI3, ACE	Yes	Yes	123 (AXI3,ACE) 40 (Mem) 40 (UART) 113 (ACE)

parse sequential constraints that contain multiple clock cycles. For example, the spec "Recommended that wready is asserted within MAXWAITS cycles of WVALID being asserted" could not be parsed using attribute grammar in [20].

Clustering and Template based Approach:

In [16], a template-based approach was employed to translate 100 AXI3 specs to SVA. The approach is semi-automatic since it needed a manual process to create translation rules after automatically partitioning specifications into a group of similar clusters. The approach was only evaluated on AMBA 3 AXI architecture as reported under the column 'Arch. tested' in Table 4.3.

Sub-tree Pattern Matching based Approach:

The work in [27] searches for a sub-tree pattern in the input specifications parse tree. The matched sub-tree pattern is associated with antecedent-consequent phrases that are mapped to a template for generating formal assertions in SVA. The work tested the approach only on AXI3 and translated 71 specs to SVA.

BINGO Framework:

Our work translated a total of 123 low-level specs from AXI3, and ACE [87] documents. Moreover, we tested the approach on high level specs with various functionality of hardware design like memory controller, UART and AMBA ACE protocol.

None of the earlier works are evaluated on specifications with flexible syntactic and semantic structures. Moreover, earlier rule-based approaches have not proposed techniques to detect incompleteness and ambiguity in the input specification.

Chapter 5

Automated Rewriting and Suggestion Generation

In this chapter, an automated sentence rewriting and suggestion generation framework is proposed. The framework is required to process English specifications that have words and sentence structures not defined in our grammar. The novelty of this work is in the selection of a minimal set of rewritten sentences that are semantically closer to the input sentence. The framework generates sentences as suggestions when an input sentence is not understood. The suggestions are generated based on a ranking mechanism that ranks a suggestion highest that is similar to the semantic and syntactic structure of the input sentence that was not understood. The framework was evaluated by creating grammar for specifications of one document and then evaluating suggestions framework on specifications of a different document. The suggestions framework was also evaluated on different variations of specifications taken from documents.

5.1 Motivation

A user can express the same semantics in a natural language through different words and different order of words. However, a semantic parser can parse only a subset of words and sentence structures that are defined in the Controlled Natural Language (CNL) grammar.

Hence, users have to learn CNL specific templates or grammar rules to write CNL acceptable sentences. A common approach to alleviate user efforts in learning CNL rules is to employ an auto-completion feature in the CNL text editor. The existing auto-completion approaches [67, 68] predict the next word in the sentence based on the word's syntactic compatibility with already written words. However, merely predicting the next syntactically accurate word may not be enough to generate a CNL acceptable sentence. A syntactically accurate sentence may still be semantically incorrect and may not create the intended output. In [74], semantic parsing of sentences with unknown words and sentence structures is achieved by rewriting the sentence into an acceptable form while preserving the input sentence semantics. However, each rewritten sentence is parsed to pick a logical form that is semantically closer to the input sentence. Semantically parsing all possible rewritten sentences may consume a lot of time and may not be a viable option for an input sentence with many rewritten forms in a real-time system. In [75], rewrite patterns and relaxed grammar rules are created to find and rewrite the erroneous phrase in a sentence. The rewrite rules are created in addition to the existing grammar rules needed to parse the sentence. The manual process of creating rewrite patterns in addition to hand-crafted grammar rules may not scale well to cover a large spectrum of sentences that cannot be parsed.

In this chapter, the focus is on rewriting the input sentence that couldn't be parsed into a CNL acceptable sentence. This work leverages the existing grammar of the BINGO model to rewrite a sentence instead of creating additional rewriting patterns or rules. Moreover, the rewriting process ensures the generation of syntactically and semantically accurate sentences without the need of parsing all possible rewritten forms of the input sentence. The rewriting approach utilizes the knowledge of syntactic-semantic dependency relations between words defined in our grammar and operates on incomplete parse trees produced by the BINGO model.

5.2 Parse Tree Parameters

This section defines the parameters associated with partial parse trees of the BINGO framework. These partial parse tree terminologies will be useful in explaining the proposed rewriting and suggestion generation framework.

BINGO model: As explained in earlier chapters, in the BINGO model [90], dependency grammar creation is analogous to creating BINGO charts for each word. In this model, a chart parser scans charts of all words to find a completed BINGO row. A completed BINGO row represents the meaning of an input specification. The proposed suggestion analysis is initiated when either (1) multiple completed BINGO rows are created or (2) no single completed BINGO rows are found at the end of parsing.

The words of the input spec in Figure 5.1 are represented as nodes of a parse tree in the BINGO framework. The nodes are further labelled by their position (in red color) in the input sentence.

Figure 5.2 illustrates grammar rules for nodes used to create charts in Figure 5.1. The links are labelled as L_i (where i is an index to the link) for the ease of explaining them in the node charts.

Complete Parse or BINGO row: We understand a node unambiguously when only one BINGO row is found. A node with an unambiguous interpretation is said to be fully connected in a parse tree. When all the nodes of the input spec are fully connected, then a complete parse tree containing all the nodes can be found. A complete parse tree is represented by a single BINGO row that passes through all the charts of all the nodes and contains only marked cells.

In Figure 5.1, the chart of some nodes have incomplete marking in their rows and can never

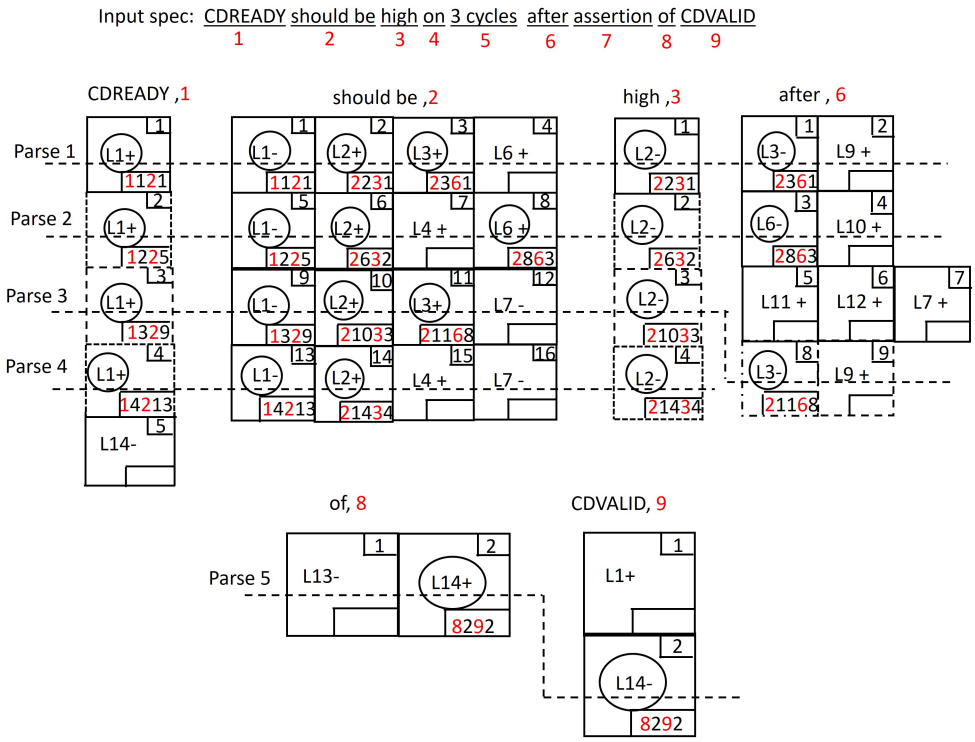


Figure 5.1: Incomplete set of charts at the end of BINGO parsing.

```
<signal> =  S+;signal_expr:signal_node:node;child  or   J-;change_expr:change_what_signal:node;child
                                  L1                                    L14
should be =  S-; signal_expr:signal_node:node;head  &   O+; signal_expr:signal_value_is:node;head &
                                  L1                                    L2
              ( MV+;occur_expr:occur_when_after_clock:slot:same;head or MV+;occur_expr:occur_when_before_clock:slot:same;head   ) &
                                  L3                                                  L4
               self;occur_expr:occur_what:SE:signal_expr;self & ( MV+;if_expr:ante:slot:same;head  or MV-;if_expr:ante:slot:same;head   ) &
                                  L5                                    L6                                  L7
               self;if_expr:conse:SE:occur_expr;self
                                  L8
<value> = O-; signal_expr:signal_value_is:node;child
                                  L2
after = ( MV-;occur_expr:occur_when_after_clock:slot:same;child & J+;occur_expr:occur_when_after_clock:node;head ) or
                                  L3                                                  L9
        ( MV-;if_expr:ante:slot:same;child &  J+;if_expr:ante:slot:same;head ) or ( J+;if_expr:ante:slot:same;head & {prep_comma+;;head} &
                                  L6                    L10                   L11                         L12
         MV+;if_expr:ante:slot:same;child )
                                  L7
of   =   Mf-;change_expr:change_what_signal:slot:same;child &  J+;change_expr:change_what_signal:node;head
                                  L13                                   L14
before = MV-;occur_expr:occur_when_before_clock:slot:same;child & J+;occur_expr:occur_when_before_clock:node;head
                                  L4                                    L15
<clock_cycle> = J-;occur_expr:occur_when_before_clock:node;child or J-;occur_expr:occur_when_after_clock:node;child
                                  L15                                   L9
asserted = Pa-;assert_expr:assert_what_signal:slot:same;head & J-;if_expr:ante:slot:same;child & self;if_expr:ante:SE:assert_expr;self
                                  L16                                   L11                         L17
is         = S-;assert_expr:assert_what_signal:node:same;head  & Pa+;assert_expr:assert_what_signal:slot:same;child
                                  L18                                   L16
<signal> = S+;assert_expr:assert_what_signal:node:same;child
                                  L18
```

Figure 5.2: Grammar rules for creating charts shown in Figure 5.1.

be fully connected in a parse tree. For example, the chart of node 'should be' does not have a single row that is completely marked. Moreover, the nodes 'on','assertions' and '3 cycles' have no marking in their charts, and their charts are not shown in this figure. As a result, we can never find a complete parse tree for all the nodes of the input spec shown in Figure 5.1.

Incomplete parse trees: We can extract all the incomplete parse trees by searching for a line that passes through all the connected marked cells. The incomplete parse trees extracted from the charts of Figure 5.1 are shown in Figure 5.3. In our framework, every Parse i (where i is an index for an incomplete parse tree) is stored as a list of elements. Each element of a Parse i contains the node, the row Id (Rid) of the node that is a part of the parse tree, and the status of the node's row Id. The status of row Id is 'full' when all the cells in the row

are marked otherwise the status is set to 'partial'.

Semantic role: The semantic role of a node in a parse tree is represented by the row of the node's chart that is a part of the parse tree. For example, in Figure 5.3, the semantic role of node 'CDREADY' in Parse 1 is represented by row Id 1 of the 'CDREADY' chart. The row Id 1 of 'CDREADY' chart contains only one link 'S+;signal_expr:signal_node:node;child' that defines the semantic role of the node 'CDREADY'. The link indicates that the 'CDREADY' node has a semantic role of a signal that can be placed in a slot 'signal_node' in the semantic frame 'signal_expr'.

concept node: A concept node is a node that contributes to the semantics of the domain. For example, 'CDREADY' is a concept called signals. The node 'assertion' refers to the concept of asserting a signal value. The concept nodes in the input spec of Figure 5.1 are 'CDREADY','high','3 cycles','assertion' and 'CDVALID'.

non_concept node: A non_concept node is a node that has a syntactic purpose of connecting nodes in a parse tree. The non_concept nodes in the input spec in Figure 5.1 are 'should be', 'on', 'after' and 'of'.

Every parse tree is associated with the following four parameters:

missing_input_concept_node_in_parse (MIC): MIC i for a Parse i is a list of concept nodes of the input spec that are not present in the Parse i tree. For example, in Figure 5.4, MIC 1 contains concept nodes '3 cycles', 'assertion' and 'CDVALID' since these are not part of Parse 1 tree. MIC 5 for Parse 5 contain nodes 'CDREADY', 'high', '3 cycles' and 'assertion'.

missing_input_concept_node_sem_role_in_parse (MICS): MICS i for a Parse i is a list of semantic roles for missing concept nodes of Parse i. These semantic roles for concept nodes exist in some other parse tree except for Parse i. In Figure 5.4, MICS 1 for

Parse 1 represents semantic role for each node in the MIC 1 list. The semantic roles for '3 cycles' and 'assertion' in MICS 1 are both denoted by 'None' since these nodes are not part of any parse tree. On the other hand, the third element $\{CDVALID, [Rid : 2]\}$ in MICS 1 indicates that the CDVALID node link rule corresponding to row Id 2 is missing in Parse 1 but exist in some other parse tree. MICS 5 for Parse 5 contains $\{CDREADY, [Rid : 1]\}$ for the missing CDREADY concept node. The row Id 1 of CDREADY corresponds to the L_1+ link that exists in Parse 1, Parse 2, Parse 3 and Parse 4 but is missing in Parse 5.

missing_input_non_concept_node_in_parse (MIN): MIN i for a Parse i is a list of non_concept nodes of the input spec that are not present in the Parse i tree. In Figure 5.4, MIN 1 list contains the nodes 'on' and 'of' since these non_concept nodes are not part of Parse 1 tree. Similarly, MIN 5 contains nodes 'should be', 'on' and 'after'.

missing_input_non_concept_node_sem_role_in_parse (MINS): MINS i for a Parse i is a list of semantic roles for missing non_concept nodes of Parse i. These semantic roles for non_concept nodes exist in some other parse tree except for Parse i.

For example, in Figure 5.4, the element $\{of, [Rid : 1]\}$ in MINS 1 indicates that Parse 1 does not contain link rules of row Id 1 from node 'of' chart. MINS 5 contains all the row Ids of 'should be' and 'after' that correspond to their unique link rules that are not part of Parse 5 tree but exist in other parse trees. For example, in element $\{\text{should be}, [Rid : 1, Rid : 2, Rid : 3, Rid : 4]\}$, all the rows of 'should be' corresponds to different link rules that are not part of MINS 5 but exist in the Parse 1, Parse 2, Parse 3 and Parse 4 trees. Similarly, MINS 5 contains the element $\{after, [Rid : 1, Rid : 2]\}$ that has unique rows of 'after' chart that exist in other parse trees except for Parse 5.

Parse 1: [{ node: CDREADY, Rid:1,status: full }, { node: should be, Rid:1,status: partial },
{ node: high, Rid:1,status: full }, { node: after, Rid:1,status: partial }]

Parse 2: [{ node: CDREADY, Rid:2,status: full }, { node: should be, Rid:2,status: partial },
{ node: high, Rid:2,status: full }, { node: after, Rid:2,status: partial }]

Parse 3: [{ node: CDREADY, Rid:3,status: full }, { node: should be, Rid:3,status: partial },
{ node: high, Rid:3,status: full }, { node: after, Rid:4,status: partial }]

Parse 4: [{ node: CDREADY, Rid:4,status: full }, { node: should be, Rid:4,status: partial },
{ node: high, Rid:4,status: full }]

Parse 5: [{ node: of, Rid:1,status: partial }, { node: CDVALID, Rid:2,status: partial }]

Figure 5.3: Incomplete parse tree information extracted from Fig 5.1 charts.

Parse 1: [{ node: CDREADY, Rid:1,status: full }, { node: should be, Rid:1,status: partial },
{ node: high, Rid:1,status: full }, { node: after, Rid:1,status: partial }]

Parse 5: [{ node: of, Rid:1,status: partial }, { node: CDVALID, Rid:2,status: partial }]

MIC 1 : [3 cycles, assertion, CDVALID]	MIC 5 : [CDREADY, high, 3 cycles, assertion]
MICS 1: [None, None, {CDVALID,[Rid:2] }]	MICS 5 : [{CDREADY,[Rid:1]}, {high,[Rid:1]}, None, None]
MIN 1 : [on, of]	MIN 5 : [should be, on, after]
MINS 1 : [None, {Of,[Rid:1] }]	MINS 5: [{should be, [Rid:1, Rid:2, Rid:3, Rid:4]}, None, {after, [Rid:1,Rid:2]}]

Figure 5.4: Parameters of incomplete Parse 1 and Parse 5 from Fig. 5.1.

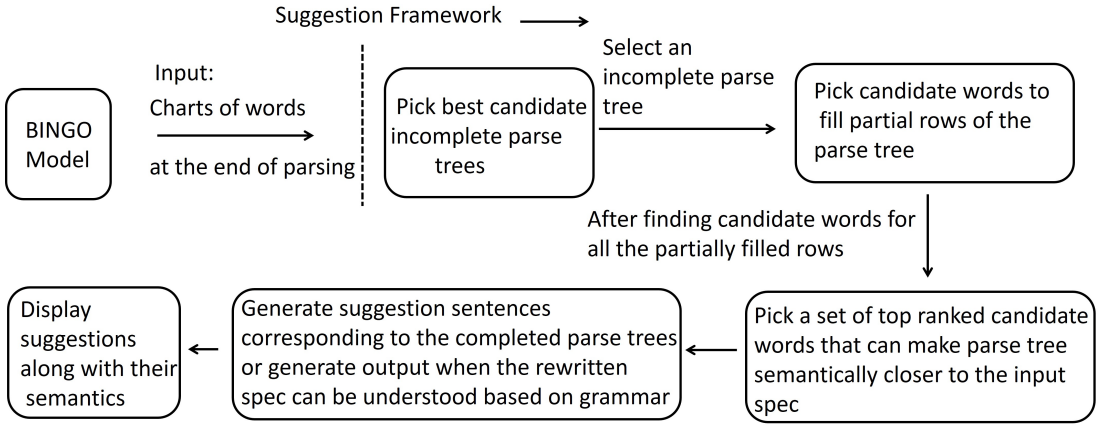


Figure 5.5: Main blocks of suggestion framework on top of BINGO model.

5.3 Methodology

Our suggestion framework takes incomplete parse charts as input from the BINGO model to generate feedback for users.

Figure 5.5 represents the major modules of our suggestion framework. Inputs to our suggestion framework are the charts of nodes processed through the BINGO model but could not generate a single complete BINGO row. We extract all incomplete parse trees from the charts as illustrated in Figure 5.1. The analysis begins by selecting a set of incomplete parse trees that are the best candidates to be completed. The incomplete parse trees contain some rows that have status set as partial, as shown in Figure 5.3. The partially filled rows have some mandatory cells that are left unmarked. The unmarked cells in the rows correspond to the syntactic-semantic links that could not be connected during parsing.

The next step in the suggestion framework is to pick candidate words from the vocabulary that can fulfill the linking requirement of unmarked cells in the partially filled rows. A ranking mechanism is developed to rank candidate words where the top-ranked candidate words can create suggestions with intent closer to the input specifications semantics. After finding candidate words that can complete a parse tree, we pick a set of candidate words that can generate suggestions with the highest semantic similarity score. Finally, the top-ranked candidate words are connected in the parse tree. The completed parse tree is analyzed to determine if it can be accepted as a re-written form of input specification or if we should process it as a suggestion. The complete list of suggestions and their semantics are displayed to the user.

In the following subsections, we explain the suggestion framework blocks in detail through an example spec: “CDREADY should be high on 3 cycles after assertion of CDVALID”. The spec could not be parsed and generated incomplete charts as shown in Figure 5.1.

5.3.1 Selecting Incomplete Parse Trees

As illustrated in Figure 5.1, we can extract five incomplete parse trees from the charts. Our objective is to generate suggestions that are semantically close to the user written specification.

An obvious approach to achieving the target is to pick parse trees with a maximum number of connected nodes of the input spec. The maximally connected parse tree will have maximum words used by the user and can be completed with a structure that is closer to the semantics and syntax of the user written spec. However, a maximally connected tree may still have fewer concept nodes than a parse tree with fewer nodes. In order to ensure the meaning of the suggestion is closer to the user's intent, we pick the parse tree with maximum connected concept nodes.

In Figure 5.1, Parse 1, Parse 2, Parse 3 and Parse 4 contain maximum concept nodes: 'CDREADY' and 'high'. These are the best candidate parse trees and will be processed for generating suggestions.

5.3.2 Finding Candidate Words

After selecting the incomplete parse trees, we begin the search for candidate words that can be connected in the parse tree to create a meaningful sentence according to our grammar.

A parse of a spec is incomplete due to the existence of partially filled rows, as illustrated in Figure 5.3.

We attempt to complete these partially filled rows by adding words from the vocabulary that can fulfill the linking requirement of the unmarked links. The process of searching candidate words for the incomplete row of 'should be' chart in Parse 1 is illustrated in Figure 5.6. The

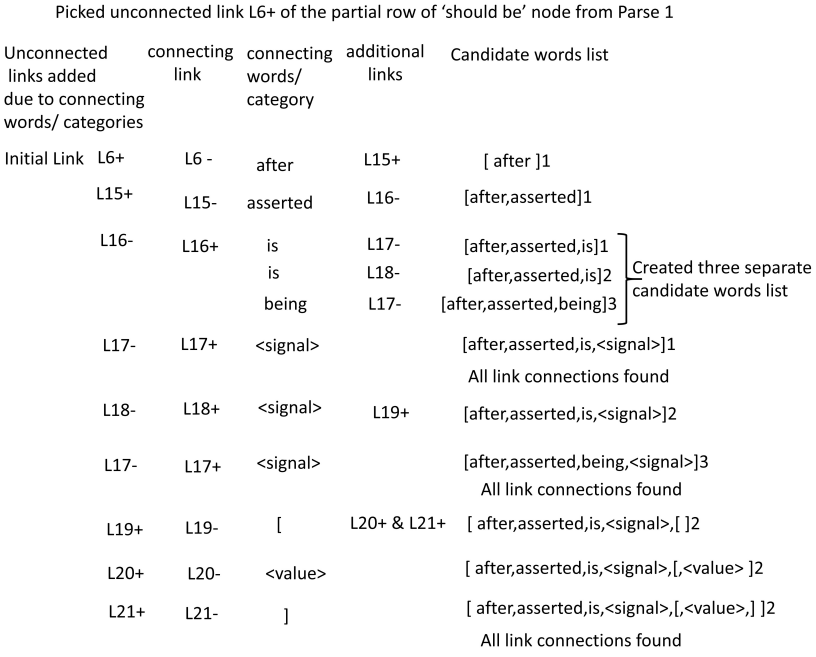


Figure 5.6: Connecting L6+ link of 'should be' node from Parse 1.

complete set of labeled link rules for the candidate words/categories is shown in Figure 5.7.

The 'should be' row in Parse 1 has an unconnected link L_6+. The framework searches for a matching L_6- link in the grammar and finds it in the link rules for the word 'after'. The system picks the word 'after' as a candidate word that can satisfy the L_6+ link requirement. However, as shown in Figure 5.7, the word 'after' has L_6- link in conjunction with $L_{15}+$ link. As a result, the addition of L_6- link in Parse 1 will also add the $L_{15}+$ link as an unmarked link in the parse tree. We have to mark the new $L_{15}+$ link to complete the Parse 1 tree.

5.3.3 Ranking Mechanism

Many lists of candidate words can be generated through the process illustrated in Figure 5.6. The ranking mechanism assists in selecting candidate words list that can generate

```
after =    MV-;if_expr:ante:slot:same;child & J+;if_expr:ante:slot:same;head
                        L6                                    L15
asserted = Pa-;assert_expr:assert_what_signal:slot:same;head &
                        L16
              J-;if_expr:ante:slot:same;child & self;if_expr:ante:SE:assert_expr;self
                        L15
is = ( S-;assert_expr:assert_what_signal:node;head &
                        L17
      Pa+;assert_expr:assert_what_signal:slot:same;child   ) or
                        L16
      ( S-;None:None:SE_SE:signal_expr;head &
                        L18
       Pa+;assert_expr:assert_what_signal:slot:same;child )
                        L16
being = S-;assert_expr:assert_what_signal:node;head &
                        L17
           Pa+;assert_expr:assert_what_signal:slot:same;child
                          L16

<signal> = ( S+;assert_expr:assert_what_signal:node;child )  or
                        L17
           ( bitsqob+;signal_expr:signal_bit_position:slot:same;head &
                        L19
             S+;None:None:SE_SE:signal_expr;child )
                          L18

 [   = bitsqob-;signal_expr:signal_bit_position:slot:same;child &
                          L19
           bitsqob_value+;signal_expr:signal_bit_position:node;head & csqb+;;head
                          L20                                                  L21

  ] = csqb-;;child
         L21

<value> = bitsqob_value-;signal_expr:signal_bit_position:node;child
                                L20
```

Figure 5.7: Link rule used for completing the connection of L6+ link of Fig 5.6.

```
function compute_sugg_score(candidate_nodes_n_links,param)
    /* pscore is an object to store parse parameter scores */
    pscore = { MICS = 0 , MIC = 0 ,sim_cat = 0,  MINS = 0 , MIN = 0}

    for(each candidate node ni in candidate nodes list  ), do
         if(ni.word and ni.links covers semantic role in
           param.MICS list), then
                pscore.MICS = pscore.MICS + 1

         else if(ni.word covers words  in param.MIC list), then
                pscore.MIC = pscore.MIC + 1

         else if(ni.word category similar to category of words in
                param.MIC list), then
                pscore.sim_cat = pscore.sim_cat + 1

         else if(ni.word and ni.links covers semantic role in
                param.MINS list), then
                pscore.MINS = pscore.MINS + 1

         else if(ni.word covers words  in param.MIN list), then
                pscore.MIN = pscore.MIN + 1

     end for loop

   return pscore  /* pscore has parameter values for candidate words */

end function
```

Figure 5.8: Scoring candidate words based on parse tree parameters coverage.

```
/*   param is an object that contain the parameter lists of the parse tree*/
/*    candidate_nodes_n_links is an object that contain candidate words
         and their links picked from vocabulary  */

pscore = compute_sugg_score(candidate_nodes_n_links,param)

combined_score = (5*pscore.MICS + 4* pscore.MIC + 3*pscore.sim_cat + 2*pscore.MINS + pscore.MIN)
                                  Total new words being added in suggestion

if(combined_score == 0) , then
{
 combined_score = -1 * Total new words being added in suggestion
}
```

Figure 5.9: Computing a single score for a candidate words list.

suggestions with semantics closer to the intent of user written specification. The ranking mechanism assigns a score to different lists of candidate words based on a simple counting scheme. Our counting scheme counts the number of covered elements of parameter lists by the newly added words and links.

As discussed earlier, every parse tree in our framework is associated with four parameters: MIC, MICS, MIN and MINS. The parameters list stores the words (MIC and MIN list) and the semantic role of words (MICS and MINS list) of the input spec that were not covered by the nodes in the parse tree. We employ a counting scheme as shown in Figure 5.8 to assign five different scores to the candidate words list based on their coverage of the information in the parameters list.

For example in Figure 5.8, pscore.MICS refers to the count of missing semantic role of words that are covered by the candidate words and their links. If a candidate word covers a semantic role given in the MICS list, then the pscore.MICS score is incremented by one. Higher coverage of user written words or semantic role of words in the suggestion would take the suggestion semantics closer to the user intent.

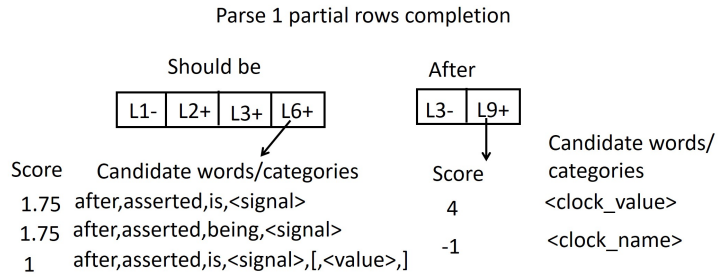


Figure 5.10: Final set of candidate words for partial row's unconnected links and their scores.

We combine these five different scores and generate a single score for a suggestion using a weighted equation shown in Figure 5.9. In combined_score computation, we have given the highest weight of five to the score corresponding to the semantic role coverage. Weight for all other scores is decremented by one according to their importance in capturing the semantics of the suggestion. For example, concept words related score pscore.MICS, pscore.MIC and pscore.sim_cat are given higher weights than the non_concept words coverage scores.

Because we want to generate concise suggestions, we have used the count of new words taken from the vocabulary as the denominator in the combined_score metric for the suggestion.

The combined_score value will be zero when a candidate words list fails to cover any elements of the parameters list. In this scenario, we assign a score to the suggestion solely based on the count of new words added to the partial parse. Moreover, we assign these suggestions a negative value to indicate the absence of any similarity between the newly added words and the user-written words of the input spec. The suggested words list with higher score are ranked higher.

Figure 5.10 illustrates the final set of candidate words required to complete the partial rows in Parse 1. As shown in the figure, the L_9 link of the word 'after' requires a connection with either a word of category $< clock_value >$ or a word with a category of $< clock_name >$. The input spec does not contain any word that has $< clock_name >$ category, and hence

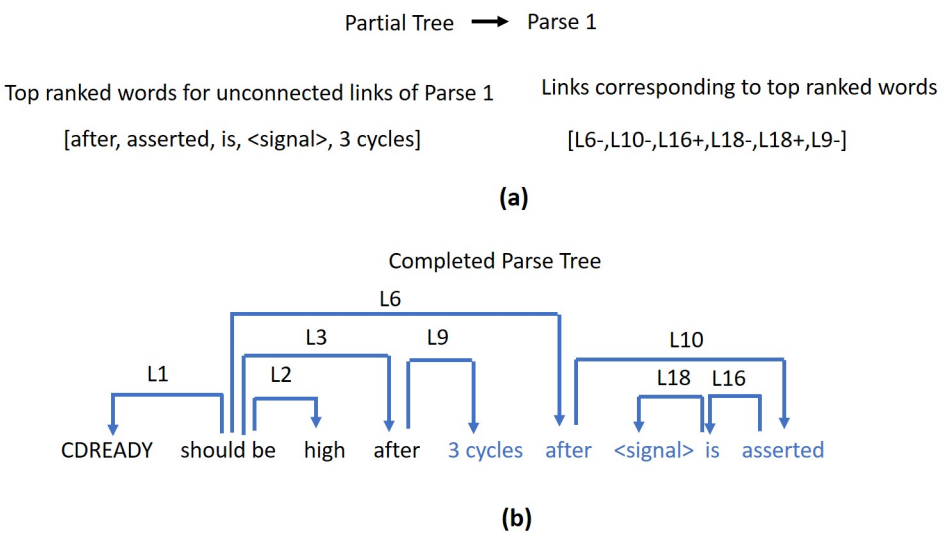


Figure 5.11: Parse 1 of Figure 5.3 completed by top ranked words. Words in blue color are added by the suggestions framework.

the suggestion for using $< clock_name >$ is assigned a negative score by the system. The framework prefers the use of $< clock_value >$ due to the presence of *clock_value* category word '3 cycles' in the input spec that is not covered by the Parse 1 tree.

5.3.4 Multiple Top Ranked Parse Trees

At the end of ranking mechanism, we have the top-ranked candidate words and their links that can satisfy the unmarked links of partial rows of incomplete parse trees. For example, Figure 5.11 (a) illustrates the highest ranked candidate words that can complete all the partial rows of Parse 1 tree of Figure 5.3. Figure 5.11 (a) also shows the links of the candidate words that can be attached to their corresponding matched links in Parse 1 tree. For example, the unconnected link L_6- of node "after" is connected to L_6+ link of the node "should be" to give rise to the edge L_6 in the parse tree shown in Figure 5.11 (b). The grammar rules with complete description of these links are shown in Figure 5.2.

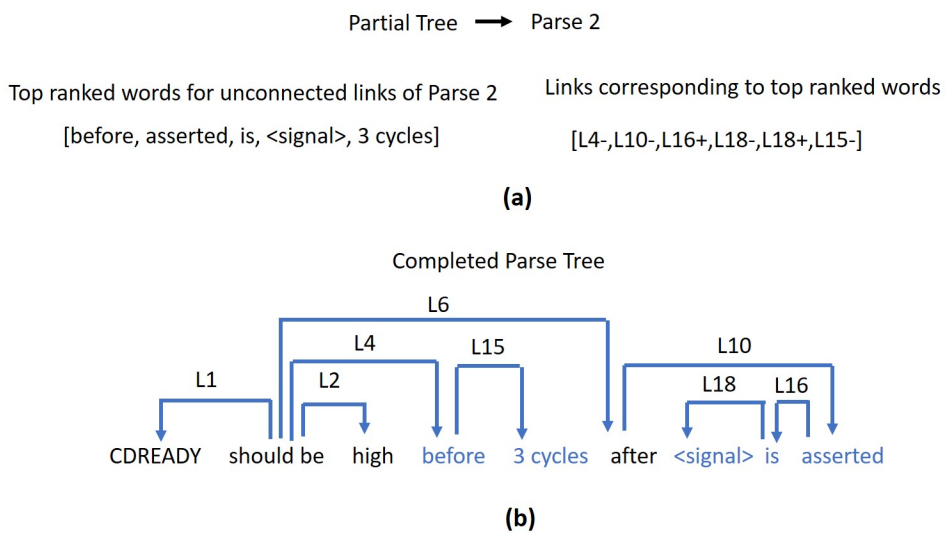


Figure 5.12: Parse 2 of Figure 5.3 completed by top ranked words.

Similarly, in Figures 5.12, 5.13, and 5.14, the partial parse trees of Figure 5.3 are completed with the top ranked words and links found through ranking mechanism.

The multiple top ranked parse trees are further pruned to select parse trees that can be used to generate a minimal set of suggestions. Figure 5.15 illustrates a flow chart to compare two parse trees and selects parse trees that cover unique set of concept nodes of the input sentence, and can produce semantically unique suggestions. The comparison of parse trees is illustrated by analysing the completed Parse 2 (Figure 5.12) and Parse 4 (Figure 5.14) trees through the steps described in the flow chart.

Flowchart steps illustrated through following example:

1. **Inputs**: The inputs to flow chart for comparison are Parse 2 and Parse 4. The resulting sentence for the completed Parse 2 tree is: "CDREADY should be high before 3 cycles after <signal> is asserted "

   Parse 4 corresponds to a sentence : "after <signal> is asserted, CDREADY should be

high before 3 cycles"

2. **Check 1. Concept nodes coverage**: We pick a parse tree with maximum number of concept nodes coverage.

   Concept nodes covered in Parse 2 are: "CDREADY, high, 3 cycles, < *signal* >, asserted"

   Concept nodes covered in Parse 4 are "CDREADY, high, 3 cycles, < *signal* >, asserted"

   These parse trees cover same set of concept nodes.

3. **Check 2. Difference in number of words**: When two parse trees have the same coverage of concept words then we would like to remove a parse tree that has more words. We have set a minimum difference limit called delta_min to allow parse trees that have lengths within a difference of few words. The delta_min is set to 3 words in our implementation. In our current example, length of Parse 2 and Parse 4 has a difference of one word. This short distance in parse trees length cannot be used to discard a parse tree based on its length.

4. **Check 3. Semantic theme check**: When the length of parse trees is almost equal, then we use their semantic theme to discard one of the parse trees. A semantic theme indicates the frames and slots that are created by semantic parsing. Parse 2 and Parse 4 have a similar semantic theme as shown in Figure 5.16. A similar semantic theme implies that these parse trees convey the same semantics. We can pick any of these parse trees to explain to the user the way a sentence can be written for specific semantics.

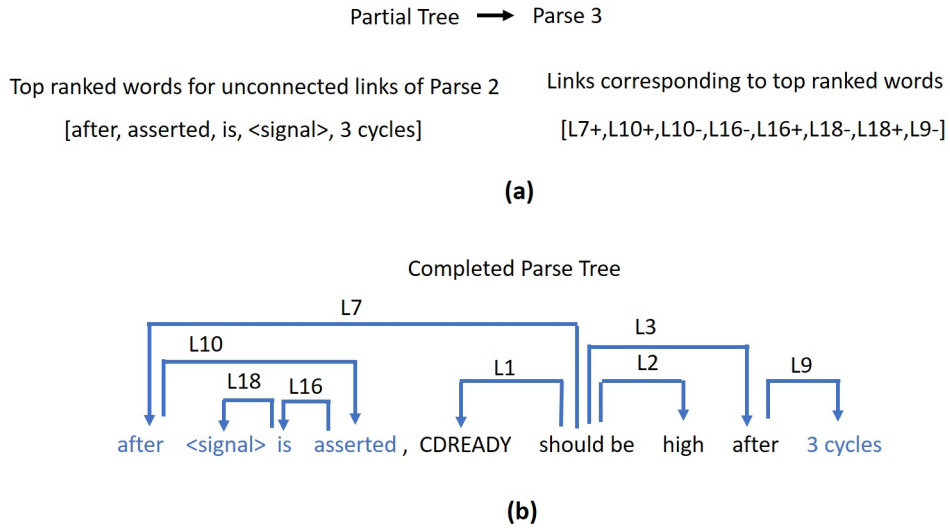


Figure 5.13: Parse 3 of Figure 5.3 completed by top-ranked words.

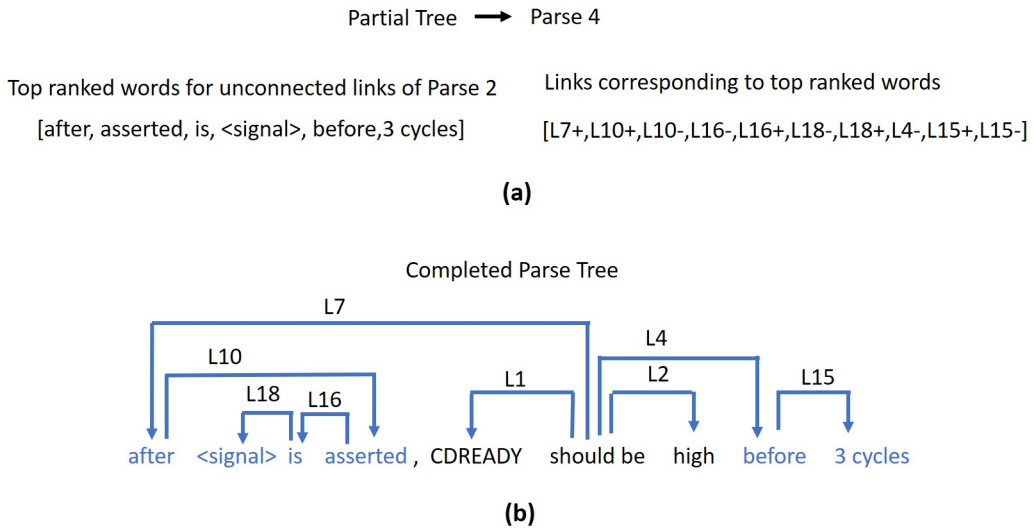


Figure 5.14: Parse 4 of Figure 5.3 completed by top-ranked words.

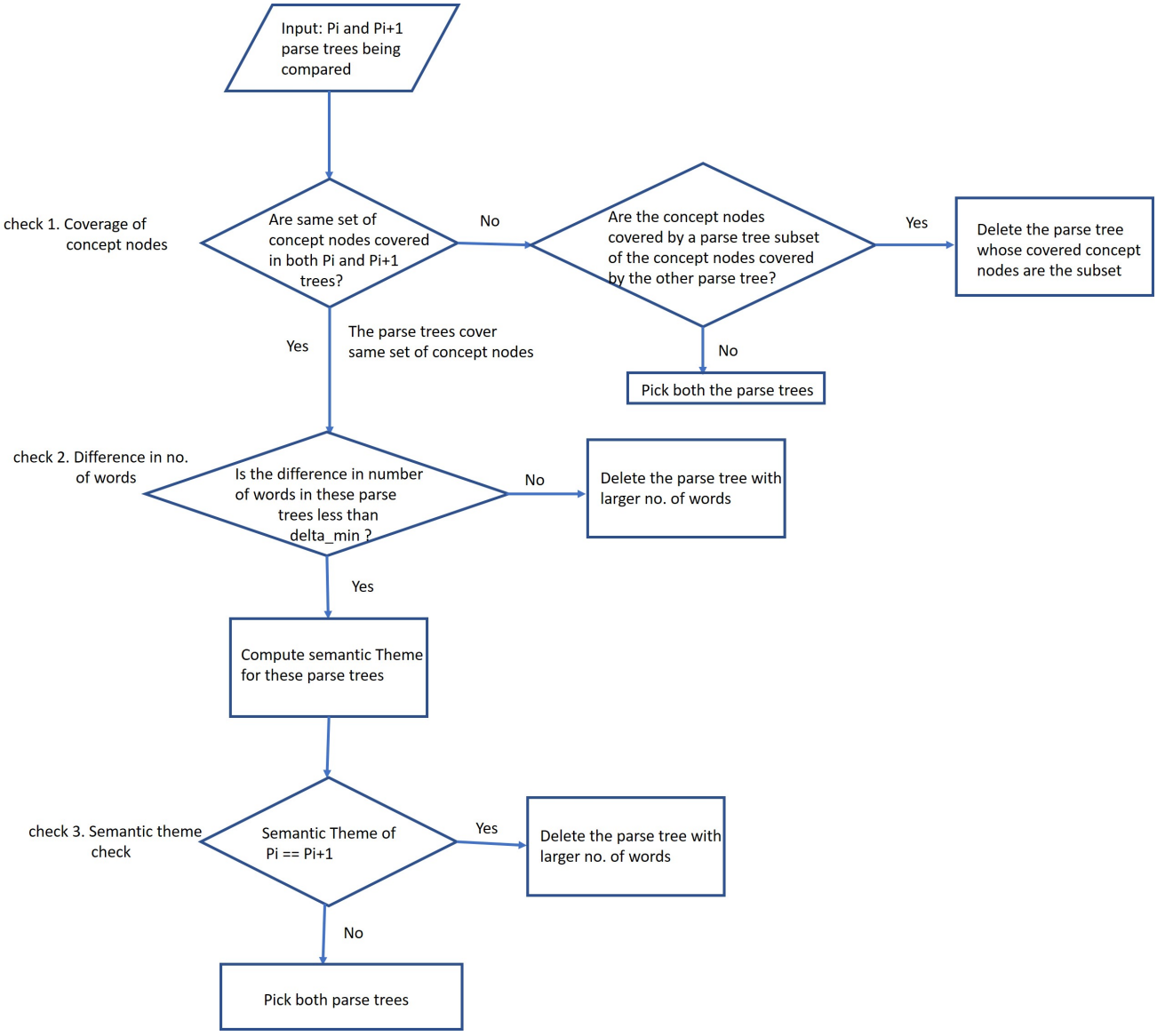


Figure 5.15: A flowchart of checks to generate semantically unique parse trees.

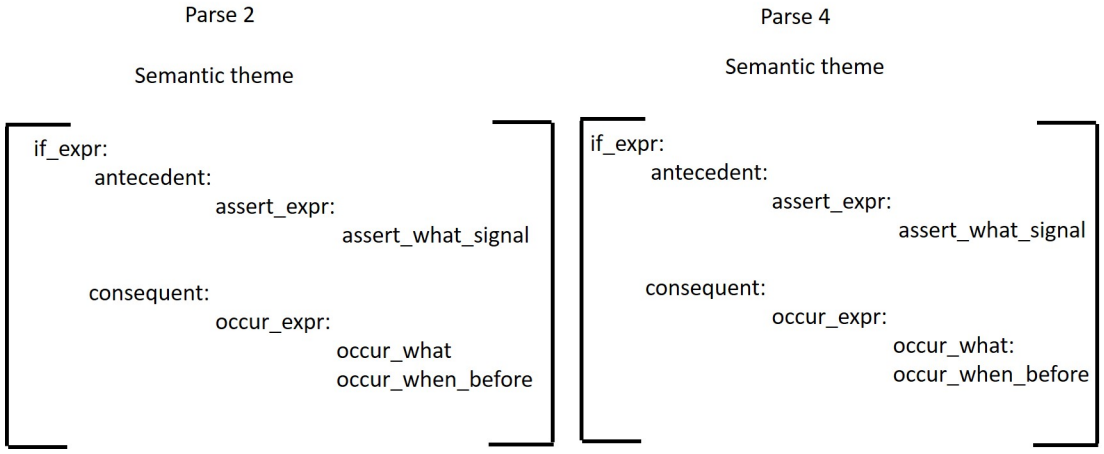


Figure 5.16: Semantic Theme showing only frames and slot names for parse trees.

5.3.5 Generating Suggestions

The top-ranked candidate words and their links are connected in the parse tree to generate suggestions. The final set of suggestions after processing all the incomplete parse trees Parse 1, Parse 2, Parse 3 and Parse 4 is shown in Figure 5.17. The red color words in the suggestions are taken from the user written spec. The blue color words are the new words added to the spec from the vocabulary according to the grammar rules.

The framework generates four suggestions for the input specification but chose only two suggestions for display. The final list of suggestions is selected based on the semantics generated by the suggestions. The suggestions that can generate unique semantics in a few words are selected in the final output. As shown in Figure 5.17, the semantics of the generated suggestions are different, and hence they were selected to be displayed.

When the rewritten specification generated by the suggestion mechanism is acceptable by the grammar and covers all the concept nodes, then through contextual analysis of each word in the input spec and the rewritten spec, we can infer design variable names and translate the rewritten spec to SVA, as shown in Figure 5.18.

Input Specification: CDREADY should be high on 3 cycles after assertion of CDVALID

The following suggestions are created by extending partially understood phrases of the input sentence:

Suggestion 1: CDREADY should be high after 3 cycles after CDVALID is asserted.

Suggestion 2: After CDVALID is asserted, CDREADY should be high before 3 cycles.

Figure 5.17: Final set of suggestions generated by the framework. User-written words in Red and newly added words in Blue.

Unknown design variables

Input spec: TID remains stable when TVALID is asserted and TREADY is Low.

Re-written Spec: <signal> remains stable when <signal> is asserted and <signal> is Low.

SVA : assert property(@(posedge clock) ((TVALID == 1) && (TREADY == 0)) |-> $stable(TID));

Figure 5.18: Signal name inference & SVA generation based on re-written spec.

5.4 Evaluation

The suggestions framework was implemented in JavaScript and executed on Node.js platform. Experiments were run on a machine with 1.8 GHz Intel Core i7-8550u processor and 16GB RAM. A major goal of this research is to generate different suggestions that are semantically closer to the specifications written in unconstrained natural language. We also wanted the system to have the ability to use the rewritten form of input spec to infer variable names.

We created grammar rules and vocabulary based on specifications taken from the AMBA ACE [87] document. We manually wrote specifications in English based on the semantics of assertions verifying memory controller architecture given in [89].

We then created restricted grammar rules and vocabulary for these memory controller specifications. We never stored the names of design variables in our vocabulary and instead wanted the system to infer them automatically based on the context.

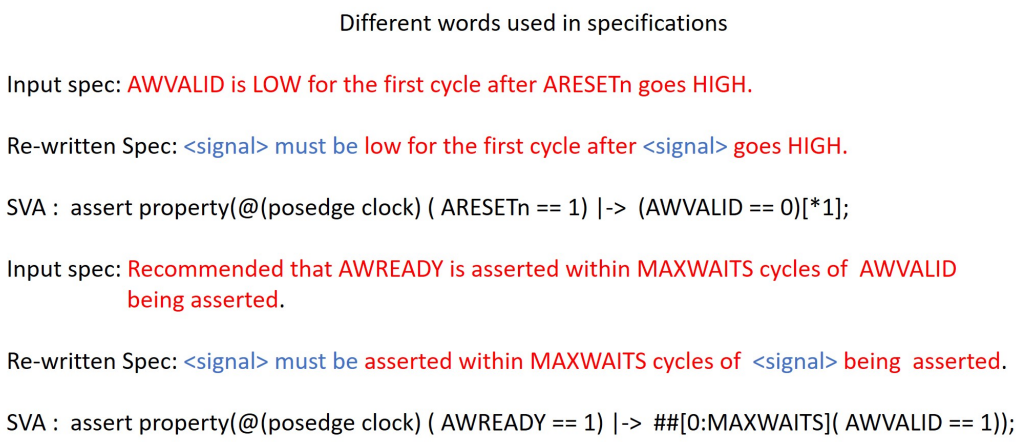


Figure 5.19: Analysis of rewritten spec led to SVA generation of specs.

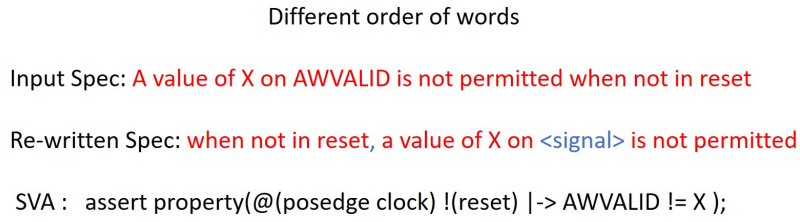


Figure 5.20: SVA for the re-written spec that covered all input concept_nodes.

We tested the framework by using input specifications from AMBA 3 AXI Protocol Checker [86], and AMBA AXI4 Stream Protocol Assertions [91] documents. These specification documents have semantics similar to the specifications of [87] but have variations in sentence structures. We took natural language specifications from [89] that have sentence structures and vocabulary not covered by our grammar.

All the specifications in our test set had different sentence structures and some words that were not defined in our grammar. However, 71% of these test specifications had concept_nodes with similar semantics as existed in our grammar. As a result, the test set contained some specifications that can be understood based on their rewritten form and some specs needed more clarity in their meaning.

There were a total of 94 specifications in [86] and [91] that had different sentence structures

Input spec: active to active command cannot come within trc

Suggestion 1: active command cannot come after < command > within trc

Following is the explanation of the categories used in above suggestions.

" < command > " : command is a category to represent specific commands used in a design.
Example words in this category are: write, read etc.

Figure 5.21: Suggestion generation for the spec without standard antecedent-consequent structure.

Input spec: Whenever a read command is issued, the data is available after the cas latency.

Suggestion 1: read data is available with latency of < sp_clk_value > after read command is issued

Following is the explanation of the categories used in above suggestions.

" < sp_clk_value > " : sp_clk_value is a category representing specific clock values. Example words in this category are: 5 clocks, 3 cycles, 5 clock cycles ...

Figure 5.22: Suggestion for a phrase with different words and order of words.

and words, but their semantics were similar to the specifications in [87]. We used these 94 specifications to test our framework.

Inferring Unknown Design Variable Names: The system was not aware of design variable names in the input specifications. The suggestions framework rewrote the specification with unknown design variables and added a '$< signal >$' category in these specifications where it expected a signal name, as shown in Figure 5.18. Instead of generating suggestions, the system was able to unambiguously pick words from the input spec that can replace '$< signal >$' and generated SVA in the output. We inferred correct design variable names in all 94 specifications.

Rewritten Specs parsed without suggestions: The test specifications contained non_concept words in a sentence structure that were different from the words used in the specifications of our training set. However, the grammar had categories defining the usage

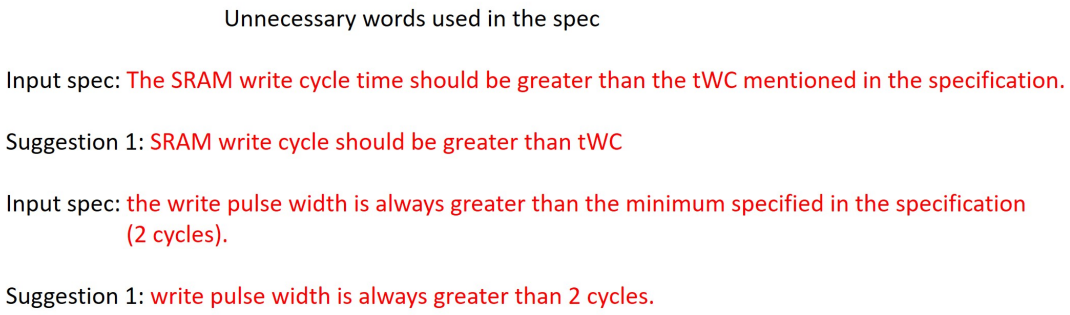


Figure 5.23: Unnecessary words removed from original spec in suggestions.

of these non-concept words. An example is illustrated in Figure 5.19. The system rewrote the input spec with non_concept words of our grammar as shown in 'Re-written Spec'. The rewritten spec captured all the concept_nodes of the input spec, and the semantic role of input concept nodes did not change in the rewritten spec. Moreover, the system was also aware of the category of non_concept nodes of the input specification that could not be parsed. The rewritten specification was considered semantically equivalent to the input based on contextual analysis, and an SVA code was generated instead of a suggestion.

Similarly, in Figure 5.20, the system generated a rewritten specification that contained all the input concept and non_concept words with the same semantic roles. An SVA was generated for this specification with a different order of words.

Results Analysis: All 94 specifications had design variables unknown to our system. Of these specifications, 41% had different order of phrases as shown in Figure 5.20 and 59% of specifications had different words as shown in Figure 5.19. By manual validation, we verified that all the suggestions were correct and semantically close to the intent of the input suggestion.

Memory Controller Test set: In contrast to above test set, we used 38 specifications of memory controller that had more unknown words and different concept words that were not defined in our grammar. For example, as shown in Figure 5.21, the specification does

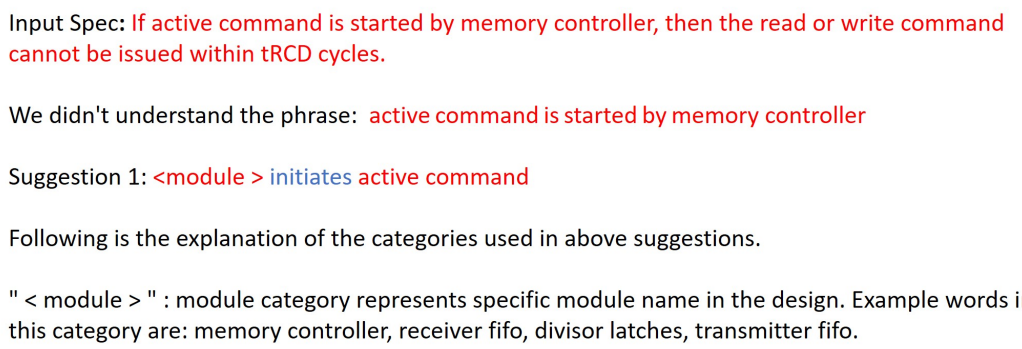


Figure 5.24: Suggestions generated only for phrases that could not be parsed.

not have explicit antecedent and consequent phrases. The system was able to generate suggestions explaining the required conditional structure, as shown in Figure 5.21. The system generated suggestions where red words indicated the words or categories similar to the user written input specification. The blue words in the suggestion are the words that were added in the partial parse tree to make the suggestion intent closer to the intent of the input specification.

Since the suggestion has categories marked in brackets like '< >', an explanation of these categories are presented in the suggestion, as illustrated in Figure 5.21.

Presence of unknown words allowed only suggestions: When there were unknown words in the input specification, then the system could not perform contextual analysis to check if the rewritten phrase could be similar to the input phrase. The system didn't perform contextual analysis since the effect of the unknown word on the overall meaning of specification cannot be determined.

Figure 5.22 shows suggestions when the input specification had an unknown word 'Whenever'.

The system was able to remove unnecessary words from specifications to generate suggestions that were semantically closer to the input specification, as shown in Figure 5.23.

The system generated suggestions only for a phrase, when the system could not parse only

Input spec: If a 1 is written to register fcr[2] then register debug2[7:3] should have a value of 0.

We didn't understand the phrase: 1 is written to register fcr [2]

Suggestion 1: value of < value > is assigned to register fcr [2]

Suggestion 2: value of < value > is assigned to register fcr [2 : < value >]

Following is the explanation of the categories used in above suggestions.

" < value > " : value category represent numbers that can be used as values in a design. Example words in this category are: 1,2,3... etc.

Figure 5.25: Suggestions generated only for phrases that could not be parsed.

Input Spec: If register ier[1] gets a value of 0 then value of 001 should not be on register iir[3:1]

We didn't understand the phrase: register ier [1] gets value of 0
The following suggestions are created by extending partially understood phrases of the input sentence:

Suggestion 1: register ier [1] should have value of < value >

Suggestion 2: register ier [1 : < value >] should have value of < value >

We didn't understand the phrase: value of 001 should not be on register iir [3 : 1]

Suggestion 1: value of register iir [3 : 1] should not be < value >

Following is the explanation of the categories used in above suggestions.

" < value > " : value category represent numbers that can be used as values in a design. Example words in this category are: 1,2,3... etc.

Figure 5.26: Suggestions generated for both the phrases that could not be parsed.

that phrase in a specification. For example, in Figure 5.24, the system could not parse only the phrase 'active command is started by memory controller'. The suggestion analysis generated suggestions for only one phrase as illustrated in Figure 5.24.

Similarly, in Figure 5.25, only one phrase was not understood. The system generated two suggestions for the phrase '1 is written to register fcr[2]'.

In Figure 5.26, suggestions were generated for more than one phrase.

Results Analysis: Of all the 38 specifications, 55% had unknown words and 45% had different sentence structures. Our approach generated suggestions for all 38 specifications. Manual inspection of suggestions has showed that 60% of the suggestions were semantically close to user's intent like the suggestions in Figure 5.23. In the other 40%, the system was able to explain the sentence structural requirements to the user.

The system translated 71% of the entire test specifications to SVA without requiring any changes in the input spec. The framework generated suggestions for the remaining 29% specs that required more clarity from the user. Overall the system generated either SVA of the rewritten spec or semantically close suggestions for 87% of the entire test specifications.

Chapter 6

Conclusions and Future Work

In this thesis, we tackled the problem of creating a translation framework that can accurately understand specifications written in the English language. Our application of translation was the hardware verification domain, where a user writes hardware design specifications and expects the output in a formal language like System Verilog Assertions. Formal language in the hardware verification is non-intuitive. Even the verification engineers cannot confirm the accuracy of the formal assertions without the knowledge of the underlying hardware architecture. Machine learning-based understanding frameworks require many training examples for developing a high-accuracy translation framework. However, the output generated by machine learning approaches may lack explainability, and a verification engineer may have to validate the accuracy of the automatically generated assertions before using them in the hardware design.

Our work is driven by an objective to provide a balance between the expressiveness of specifications and the precision of meaning to users in writing specifications. Expressiveness will benefit users in writing their design intent with ease, and the precision of meaning will ensure that the written specification is unambiguous and complete. In this thesis, we adopted a rule-based translation mechanism that can achieve high accuracy in translation.

Our conclusions are as follows:

1. It is well known that dependency grammar allows flexibility in word order while creat-

ing rules for parsing sentences. Also, dependency parse trees are closer to the semantic representation of the sentence. However, dependency parsing cannot resolve ambiguous attachments solely based on syntactic rules.

2. In Chapter 4, we developed a BINGO formalism that coupled syntactic relation between words and semantic interpretation of words.

3. Our formalism was inspired by the Link grammar and allowed declarative specification of semantic context rules for each word and category in the grammar.

4. Contextual grammar rules provided flexibility to users in writing expressive specifications with variable word order. Semantic contextual rules allowed the creation of grammar that can resolve ambiguous attachments like prepositional modifiers and conjunction attachments. Moreover, the grammar allowed us to include multiple interpretations of words based on the context.

5. In Chapter 4, we also introduced our semantic parser with systematic steps that allowed the creation of semantic frames based on the linkages found in the BINGO parsing. The creation and analysis of semantic frames for the specification were made possible by the node data structure that was created for each word of the specification. The node data structure allowed the ability to store and process the semantic interpretation of each word in the word's node data structure.

6. The BINGO chart created for the overall specification allowed the detection of ambiguous and incomplete specifications.

7. Even though our framework allows flexible syntactic and semantic structures for parsing specifications, the rule-based translation mechanism restricts the vocabulary and grammar rules in writing specifications. We developed a suggestion module in Chapter 5 to alleviate user efforts in learning our grammar rules.

6.1 Future Work

Our suggestion module coverage is limited by the grammar rules. We cannot generate suggestions with intent closer to the user's requirement if most of the words in the input specification are not available in our grammar. We can use Large Language Models (LLM) like GPT3 and OPT-175B to overcome the limitations of a small set of rules in our grammar.

The LLMs can be trained to rewrite input specifications that are not covered by our grammar into a specification with the same intent that lies within the coverage of our grammar rules. When the grammar accepts a rewritten specification, then we can create semantic frames for that specification. Our suggestion module will be initiated when both the input and rewritten specifications lie outside the grammar coverage.

An LLM in this configuration can enhance the understanding coverage of our grammar-based framework.

Bibliography

[1] Ashok B Mehta. *SystemVerilog Assertions and Functional Coverage*. Springer, 2020.

[2] Abha Moitra, Kit Siu, Andrew W Crapo, Michael Durling, Meng Li, Panagiotis Manolios, Michael Meiners, and Craig McMillan. Automating requirements analysis and test case generation. *Requirements Engineering*, 24(3):341–364, 2019.

[3] Richard Hudson. Coordination and grammatical relations. *Journal of Linguistics*, 24(2):303–342, 1988.

[4] Vincenzo Lombardo and Leonardo Lesmo. Unit coordination and gapping in dependency theory. In *Proceedings of the Workshop on Processing of Dependency-Based Grammars*, pages 11–20, 1998.

[5] Madeleine Bates, Robert Bobrow, Robert Ingria, and David Stallard. The Delphi natural language understanding system. In *Proceedings of the fourth conference on Applied natural language processing*, pages 132–137. Association for Computational Linguistics, 1994.

[6] Gokhan Tur, Dilek Hakkani-Tür, Larry Heck, and Sarangarajan Parthasarathy. Sentence simplification for spoken language understanding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5628–5631. IEEE, 2011.

[7] Harry D Foster. Trends in functional verification: A 2014 industry study. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[8] Pietro Fezzardi, Fabrizio Ferrandi, and Christian Pilato. Enabling automated bug detection for IP-based designs using high-level synthesis. *IEEE Design & Test*, 35(5):54–62, 2018.

[9] Brian Keng, Sean Safarpour, and Andreas Veneris. Automated debugging of SystemVerilog assertions. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[10] Andreas Veneris, Brian Keng, and Sean Safarpour. From RTL to Silicon: The case for automated debug. In *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*, pages 306–310. IEEE, 2011.

[11] Wolfgang Mueller, Alexander Bol, Alexander Krupp, and Ola Lundkvist. Generation of executable testbenches from natural language requirement specifications for embedded real-time systems. In *Distributed, Parallel and Biologically Inspired Systems*, pages 78–89. Springer, 2010.

[12] Tomas G Moreira, Marco A Wehrmeister, Carlos E Pereira, Jean-Francois Petin, and Eric Levrat. Automatic code generation for embedded systems: From UML specifications to VHDL code. In *2010 8th IEEE International Conference on Industrial Informatics*, pages 1085–1090. IEEE, 2010.

[13] Emad Ebeid, Franco Fummi, and Davide Quaglia. HDL code generation from UML/MARTE sequence diagrams for verification and synthesis. *Design automation for embedded systems*, 19(3):277–299, 2015.

[14] Marcela Leite and Marco Aurélio Wehrmeister. System-level design based on UML/-MARTE for FPGA-based embedded real-time systems. *Design Automation for Embedded Systems*, 20(2):127–153, 2016.

[15] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *International Conference on the Unified Modeling Language*, pages 416–429. Springer, 1999.

[16] Mathias Soeken, Christopher B Harris, Nabila Abdessaied, Ian G Harris, and Rolf Drechsler. Automating the translation of assertions using natural language processing techniques. In *Proceedings of the 2014 Forum on Specification and Design Languages (FDL)*, volume 978, pages 1–8. IEEE, 2014.

[17] JJ Granacki and Alice C Parker. Phran-span: a natural language interface for system specifications. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 416–422, 1987.

[18] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. ARSENAL: automatic requirements specification extraction from natural language. In *NASA Formal Methods Symposium*, pages 41–46. Springer, 2016.

[19] Walling R Cyre, Jim Armstrong, M Manek-Honcharik, and Alexander J Honcharik. Generating VHDL models from natural language descriptions. In *Proceedings of the conference on European design automation*, pages 474–479, 1994.

[20] Ian G Harris. Capturing assertions from natural language descriptions. In *2013 1st International Workshop on Natural Language Analysis in Software Engineering (NaturaLiSE)*, pages 17–24. IEEE, 2013.

[21] Christopher B Harris and Ian G Harris. Generating formal hardware verification properties from natural language documentation. In *Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing (IEEE ICSC 2015)*, pages 49–56. IEEE, 2015.

[22] Christopher B Harris and Ian G Harris. GLAST: Learning formal grammars to translate natural language specifications into hardware assertions. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 966–971. IEEE, 2016.

[23] Sandip Ray, Ian G Harris, Goerschwin Fey, and Mathias Soeken. Multilevel design understanding: from specification to logic. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6. IEEE, 2016.

[24] Rongjie Yan, Chih-Hong Cheng, and Yesheng Chai. Formal consistency checking over specifications in natural languages. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1677–1682. IEEE, 2015.

[25] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D Manning. Generating typed dependency parses from phrase structure parses. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, 2006.

[26] Ralph Debusmann, Denys Duchier, Alexander Koller, Marco Kuhlmann, Gert Smolka, and Stefan Thater. A relational syntax-semantics interface based on dependency grammar. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 176–182, 2004.

[27] Junchen Zhao and Ian G Harris. Automatic assertion generation from natural language specifications using subtree analysis. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 598–601. IEEE, 2019.

[28] Oliver Keszocze and Ian G Harris. Chatbot-based assertion generation from natural language specifications. In *2019 Forum for Specification and Design Languages (FDL)*, pages 1–6. IEEE, 2019.

[29] Shiyu Zhang, Juan Zhai, Lei Bu, Mingsong Chen, Linzhang Wang, and Xuandong Li.

Automated generation of LTL specifications for smart home IOT using natural language. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 622–625. IEEE, 2020.

[30] Imran Sarwar Bajwa, Mark Lee, and Behzad Bordbar. Resolving syntactic ambiguities in natural language specification of constraints. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 178–187. Springer, 2012.

[31] Daniel M Berry and Erik Kamsties. Ambiguity in requirements specification. In *Perspectives on software requirements*, pages 7–44. Springer, 2004.

[32] Flávia A Barros, Laís Neves, Érica Hori, and Dante Torres. The ucsCNL: A controlled natural language for use case specifications. In *Proceedings of the 2011 International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 250–253, 2011.

[33] Gustavo Carvalho, Flávia Barros, Ana Carvalho, Ana Cavalcanti, Alexandre Mota, and Augusto Sampaio. NAT2TEST tool: From natural language requirements to test cases based on CSP. In *SEFM 2015 Collocated Workshops*, pages 283–290. Springer, 2015.

[34] Abigail Cauchi, Christian Colombo, Adrian Francalanza, Mark Micallef, and Gordon Pace. Using gherkin to extract tests and monitors for safer medical device interaction design. In *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 275–280, 2016.

[35] Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.

[36] MW Esser and Peter Struss. Obtaining models for test generation from natural-

language-like functional specifications. In *International workshop on principles of diagnosis*, pages 75–82, 2007.

[37] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. Generation of formal requirements from structured natural language. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 19–35. Springer, 2020.

[38] Tobias Kuhn. A survey and classification of controlled natural languages. *Computational Linguistics*, 40(1):121–170, 2014.

[39] MW Esser and Peter Struss. Obtaining models for test generation from natural-language-like functional specifications. In *International workshop on principles of diagnosis*, pages 75–82, 2007.

[40] Teruko Mitamura and Eric Nyberg. Controlled English for knowledge-based MT: Experience with the KANT system. *Proceedings of TMI-95*, 145(146-147):216, 1995.

[41] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. Cocospec: A mode-aware contract language for reactive systems. In *International Conference on Software Engineering and Formal Methods*, pages 347–366. Springer, 2016.

[42] Kit Siu, Abha Moitra, Michael Durling, Andy Crapo, Meng Li, Han Yu, Heber Herencia-Zapana, Mauricio Castillo-Effen, Shiraj Sen, Craig McMillan, et al. Flight critical software and systems development using ASSERT™. In *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, pages 1–10. IEEE, 2017.

[43] Daniel D Sleator and David Temperley. Parsing English with a Link Grammar. In *Proceedings of the Third International Workshop on Parsing Technologies*, pages 277–292, 1993.

[44] Mark Steedman and Jason Baldridge. Combinatory categorial grammar. *Non-Transformational Syntax: Formal and Explicit Models of Grammar. Wiley-Blackwell*, pages 181–224, 2011.

[45] Daniel Sleator and Davy Temperley. Parsing English with a Link Grammar. Carnegie Mellon University Computer Science technical report cmu-cs-91-196. 1991. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/link/pub/www/papers/ps/tr91-196.pdf.

[46] Haim Gaifman. Dependency systems and phrase-structure systems. *Information and control*, 8(3):304–337, 1965.

[47] David G Hays. Dependency theory: A formalism and some observations. *Language*, 40(4):511–525, 1964.

[48] Alexis Nasr and Owen Rambow. A simple string-rewriting formalism for dependency grammar. In *Proceedings of the Workshop on Recent Advances in Dependency Grammar*, pages 17–24, 2004.

[49] Bernard Lang. Towards a uniform formal framework for parsing. In *Current issues in parsing technology*, pages 153–171. Springer, 1991.

[50] Michael A. Covington. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102, 2001.

[51] Joakim Nivre. An efficient algorithm for projective dependency parsing. In *Proceedings of the eighth international conference on parsing technologies*, pages 149–160, 2003.

[52] Li Shiuan Peh and C Ting. A divide-and-conquer strategy for parsing. In *Proceedings of the International Workshop on Parsing Technology (IWPT)*, 1996.

[53] Richard A Hudson. *English word grammar*. B. Blackwell, 1991.

[54] Igor Aleksandrovic Mel'cuk et al. *Dependency syntax: theory and practice*. SUNY press, 1988.

[55] Marie-Catherine De Marneffe and Christopher D Manning. Stanford typed dependencies manual. Technical report, Stanford University, 2008.

[56] Geoffrey Nunberg. *The linguistics of punctuation*. Number 18. Center for the Study of Language (CSLI), 1990.

[57] Valmi Dufour-Lussier, Bruno Guillaume, and Guy Perrier. Parsing coordination extra-grammatically. In *Language and Technology Conference*, pages 55–66. Springer, 2011.

[58] Crit Cremers. Coordination as a parsing problem. In *Proceedings of the 6th Twente Workshop on Language Technology*, pages 139–143.

[59] Guy Perrier and Bruno Guillaume. Frigram: a French Interaction Grammar. *Journal of Language Modelling Vol*, 3(1):265–316, 2015.

[60] Rajeev Agarwal and Lois Boggess. A simple but useful approach to conjunct identification. In *Proceedings of the 30th annual meeting on Association for Computational Linguistics*, pages 15–21. Association for Computational Linguistics, 1992.

[61] Sadao Kurohashi and Makoto Nagao. A syntactic analysis method of long Japanese sentences based on the detection of conjunctive structures. *Computational Linguistics*, 20(4):507–534, 1994.

[62] Sunil Issar and Wayne Ward. CMU robust spoken language understanding system. In *Third European Conference on Speech Communication and Technology*, 1993.

[63] Harald Aust, Martin Oerder, Frank Seide, and Volker Steinbiss. The Philips automatic train timetable information system. *Speech Communication*, 17(3-4):249–262, 1995.

[64] John Dowding, Jean Mark Gawron, Doug Appelt, John Bear, Lynn Cherny, Robert Moore, and Douglas Moran. Gemini: A natural language system for spoken-language understanding. In *Proceedings of the 31st annual meeting on Association for Computational Linguistics*, pages 54–61. Association for Computational Linguistics, 1993.

[65] Hiyan Alshawi, David M Carter, Jan van Eijck, Robert C Moore, Douglas B Moran, and Stephen G Pulman. Overview of the Core Language Engine. In *FGCS*, pages 1108–1115, 1988.

[66] Gokhan Tur, Dilek Hakkani-Tür, and Larry Heck. What is left to be understood in ATIS? In *2010 IEEE Spoken Language Technology Workshop*, pages 19–24. IEEE, 2010.

[67] Tobias Kuhn and Rolf Schwitter. Writing support for controlled natural languages. In *Australasian Language Technology Association Workshop 2008*, volume 6, pages 46–54, 2008.

[68] Susana Palmaz, Montse Cuadros, and Thierry Etchegoyhen. Statistically-guided controlled language authoring. In *International Workshop on Controlled Natural Language*, pages 37–47. Springer, 2016.

[69] Krasimir Angelov and Michal Boleslav Měchura. Editing with search and exploration for controlled languages. In *Controlled Natural Language*, pages 1–10. IOS Press, 2018.

[70] Feikje Hielkema, Chris Mellish, and Peter Edwards. Evaluating an Ontology-Driven WYSIWYG Interface. In *Proceedings of the Fifth International Natural Language Generation Conference*, pages 138–146, 2008.

[71] Yannis Haralambous and Tian Tian. Tailoring a controlled language out of a corpus of maintenance reports. In *Proceedings of the Seventh International Workshop on Controlled Natural Language (CNL 2020/21)*, 2021.

[72] Michael S Hsiao. Automated Program Synthesis from object-oriented natural language for Computer Games. In *Proceedings of the Sixth International Workshop on Controlled Natural Language*, pages 71–74, 2018.

[73] Michael Hsiao. Multi-phase context vectors for generating feedback for natural-language based programming. In *Proceedings of the Seventh International Workshop on Controlled Natural Language (CNL 2020/21)*, 2021.

[74] Bo Chen, Le Sun, Xianpei Han, and Bo An. Sentence rewriting for semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 766–777, Berlin, Germany, August 2016. Association for Computational Linguistics.

[75] Teruko Mitamura and Eric Nyberg. Automatic rewriting for controlled language translation. In *The Sixth Natural Language Processing Pacific Rim Symposium (NLPRS2001) Post-Conference Workshop, Automatic Paraphrasing: Theories and Applications*, 2001.

[76] Charles J Fillmore and Collin F Baker. Frame semantics for text understanding. In *Proceedings of WordNet and Other Lexical Resources Workshop, NAACL*, volume 6, 2001.

[77] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642, 2016.

[78] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[79] Aaron Dutle, César Muñoz, Esther Conrad, Alwyn Goodloe, Laura Titolo, Ivan Perez, Swee Balachandran, Dimitra Giannakopoulou, Anastasia Mavridou, and Thomas Pressburger. From Requirements to Autonomous Flight: An Overview of the Monitoring ICAROUS project. In Matt Luckcuck and Marie Farrell, editors, Proceedings Second Workshop on *Formal Methods for Autonomous Systems,* Virtual, 7th of December 2020, volume 329 of *Electronic Proceedings in Theoretical Computer Science*, pages 23–30. Open Publishing Association, 2020.

[80] Anastasia Mavridou, Hamza Bourbouh, Dimitra Giannakopoulou, Thomas Pressburger, Mohammad Hejase, Pierre-Loic Garoche, and Johann Schumann. The ten Lockheed Martin cyber-physical challenges: Formalized, Analyzed, and Explained. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pages 300–310. IEEE, 2020.

[81] Marie-Catherine De Marneffe and Joakim Nivre. Dependency grammar. *Annual Review of Linguistics*, 5:197–218, 2019.

[82] Rongjie Yan, Chih-Hong Cheng, and Yesheng Chai. Formal consistency checking over specifications in natural languages. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1677–1682. IEEE, 2015.

[83] Zifan Nan, Hui Guan, Xipeng Shen, and Chunhua Liao. Deep NLP-based co-evolvement for synthesizing code analysis from natural language. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 141–152, 2021.

[84] Aishwarya Chhabra, Amit Sangroya, and C Anantaram. Formalizing and Verifying natural language system requirements using Petri nets and context based reasoning. In *MRC@ IJCAI*, pages 64–71, 2018.

[85] Daniel Gildea. Corpus variation and parser performance. In *Proceedings of the 2001 Conference on Empirical Methods in Natural Language Processing*, 2001.

[86] ARM. *AMBA 3 AXI Protocol Checker User Guide*, 2006.

[87] ARM. *AMBA 4 ACE and ACE-Lite Protocol Checkers User Guide.*, 2012. https://developer.arm.com/docs/dui0576/b/ace-and-ace-lite-protocol-assertion-descriptions.

[88] Jacob Gorban. *UART IP core specification Architectures.*, 2002. http://www.isy.liu.se/edu/kurs/TSEA44/OpenRISC/UART_spec.pdf.

[89] Srikanth Vijayaraghavan and Meyyappan Ramanathan. SVA for Memories. In *A practical guide for SystemVerilog assertions*, chapter 5, pages 191–232. Springer US, Boston, MA, 2006. https://doi.org/10.1007/0-387-26173-7_6.

[90] Rahul Krishnamurthy and Michael S Hsiao. BINGO: A dependency grammar framework to understand hardware specifications written in English. In *Proceedings of the Sixth International Conference on Dependency Linguistics (Depling, SyntaxFest 2021)*, pages 68–80, 2021.

[91] *ARM,AMBA 4 AXI4, AXI4-Lite, and AXI4-Stream Protocol Assertions User Guide*, 2010.