BINGO: A Dependency Grammar Framework to Understand Hardware Specifications Written in English

Rahul Krishnamurthy
Bradley Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, VA, USA
rahulk4@vt.edu

Michael S. Hsiao
Bradley Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, VA, USA
mhsiao@vt.edu

Abstract

Automatic understanding of specifications containing flexible word order and expressiveness close to natural language is a challenging task. We address this challenge by modeling semantic parsing as a game of BINGO with dependency grammar. In this model, the rows in a BINGO chart of a word represent distinct interpretations, and the columns describe the constraints required to complete each of these interpretations. BINGO parsing considers the context of each word in the input specification to ensure high precision in the creation of semantic frames. We encode contextual information of the hardware verification domain in our grammar by adding semantic links to the existing syntactic links of the link grammar. We also define semantic propagation operations as declarative rules that are executed for each dependency edge of the parse tree to create a semantic frame. We used hardware design specifications written in English to evaluate the framework. Our results showed that the system could translate highly expressive specifications. It also demonstrated the ease of creating rules to generate the same semantic frame for specifications with the same meaning but different word order.

1 Introduction

Automatic understanding of natural language specification documents for hardware and software has numerous benefits such as reduced verification efforts for debugging the design, detection of incomplete specifications, reduced time to fabricate the chip (Ray et al., 2016), etc. The formal output generated by a semantic parser is 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. As evident in (Gu et al., 2016; Lin et al., 2018), machine learning-based semantic parsing approaches require thousands of input-output examples to achieve high accuracy. Unavailability of large number of examples resulted in many rule-based translation works like (Dutle et al., 2020; Giannakopoulou et al., 2020; Mavridou et al., 2020). These rule-based approaches achieve high accuracy in understanding specifications by imposing strict restrictions on the order of words in the input specifications.

This paper presents a dependency grammar-based framework to understand hardware specifications written in English. The grammar is not as rigid as the grammars in the existing works and allows flexibility in the word order variations 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.

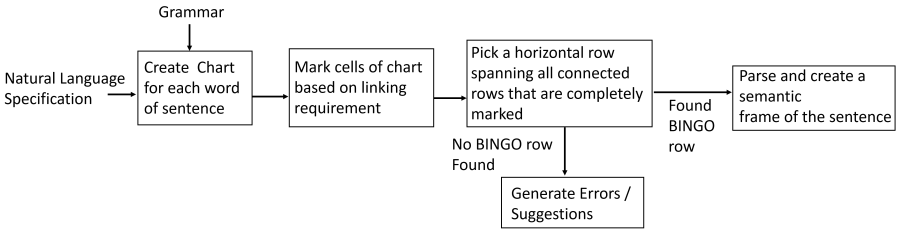


Figure 1: Framework for parsing and translating hardware specification to Semantic Frames.

The declarative rules component is inspired by the syntactic link rules of the link grammar. 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.

Figure 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 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.

The rest of the paper is organized as follows. Section 2 discusses the previous works that have employed dependency structures to understand specifications. To better explain the working of our framework, we introduce the data structure used in our framework in section 3. In section 4, we present different components of grammar. Section 5 covers the parsing methodology of the framework. In section 6, we discuss the evaluation of our work. Finally, a concluding summary with the future work is described in section 7.

2 Related Work

Dependency parse trees are useful in extracting semantic relations between entities and have close correspondence to the semantic representation of the sentence (De Marneffe and Nivre, 2019; Covington, 2001). Driven by these advantages, dependency parsing has been used as the core component in the recent works (Ghosh et al., 2016; Yan et al., 2015; Soeken et al., 2014; Nan et al., 2021; Zhang et al., 2020; Chhabra et al., 2018) 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 literature (Gildea, 2001) that the accuracy of syntactic parser may reduce when applied on text outside its training corpus. Work in (Ghosh et al., 2016; Zhang et al., 2020; Chhabra et al., 2018) 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 down-stream natural language understanding application. As pointed out in (Bajwa et al., 2012), 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. More recently, in (Hsiao, 2018), (Hsiao, 2021), domain-specific context knowledge is used to parse specifications for video games, and suggestions and warning messages are given for sentences that could not be handled.

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

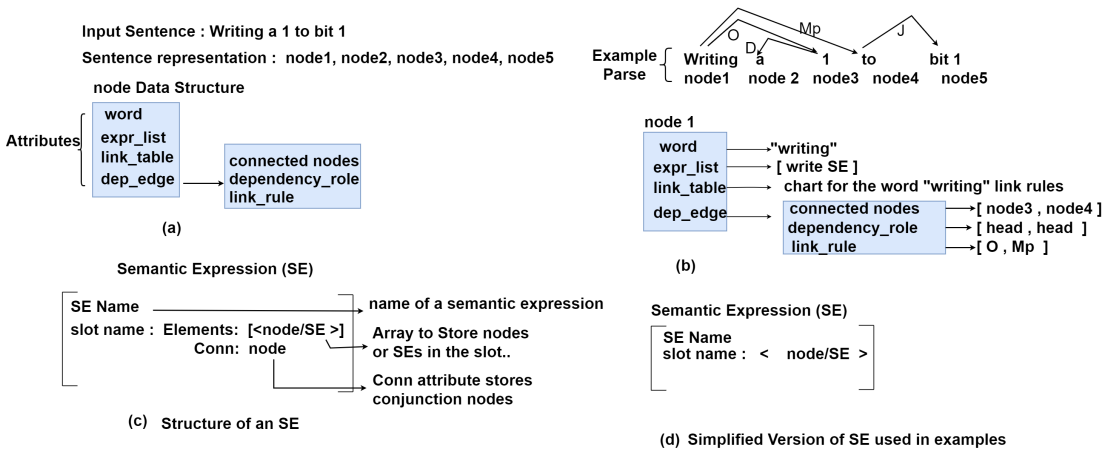


Figure 2: (a) A node data structure to store word and its dependency and semantic information. (b) An example that shows the representation of a node for the word "writing". (c) A semantic expression (SE) structure to store elements in the slot of an SE. (d) A simplified version of SE without Elements and Conn attributes.

inspired by the link grammar formalism (Sleator and Temperley, 1993) 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 (Sleator and Temperley, 1993). 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.

Earlier work on chart parser in (Nasr and Rambow, 2004) extended a CKY parser of context-free grammar to parse a dependency grammar. The chart items of the CKY parser contained finite state machines, and the chart parsing produced a dependency tree from a packed parse forest in two steps. In the first step, binary syntagmatic trees were extracted from the packed parse forest, and in the second step, each syntagmatic tree was transformed into a dependency tree. In contrast to (Nasr and Rambow, 2004), cells in our chart contained links that connect two words of the input sentence. In our chart parser, a fully connected dependency tree is extracted from the chart in a single step that involved searching for a BINGO row. A BINGO row provided the linkages that connect all the input sentence words in a dependency relation.

3 Data Structure

In our framework, the words of an input specification are represented as nodes of a tree. A node data structure consists of four attributes as shown in Figure 2 (a). The purpose of these attributes are as follows: (1) a word attribute is needed to store the node's word as a string, (2) expr_list keeps an array of semantic expressions that are either created at the node or propagated to the node in a dependency tree, (3) link_table contains a chart for the node's link rules, (4) a depedge stores the dependency edge information for the node. Figure 2 (b) illustrates the node representation for the word "writing". In Figure 2 (c), the structure of the Semantic Expression (SE) is shown for the ease of explaining semantic frames in the subsequent sections. An SE is similar to a semantic frame and has semantic slots. A semantic slot has two attributes Elements and Conn. Elements is an array to store slot values that are either a node or another SE. Conn of the slot contains conjunction nodes like 'and' ,'or' that connects values of the elements array. In the examples of the subsequent sections, we will use a simplified version of SE shown in Figure 2 (d) that does not explicitly mention Elements and Conn.

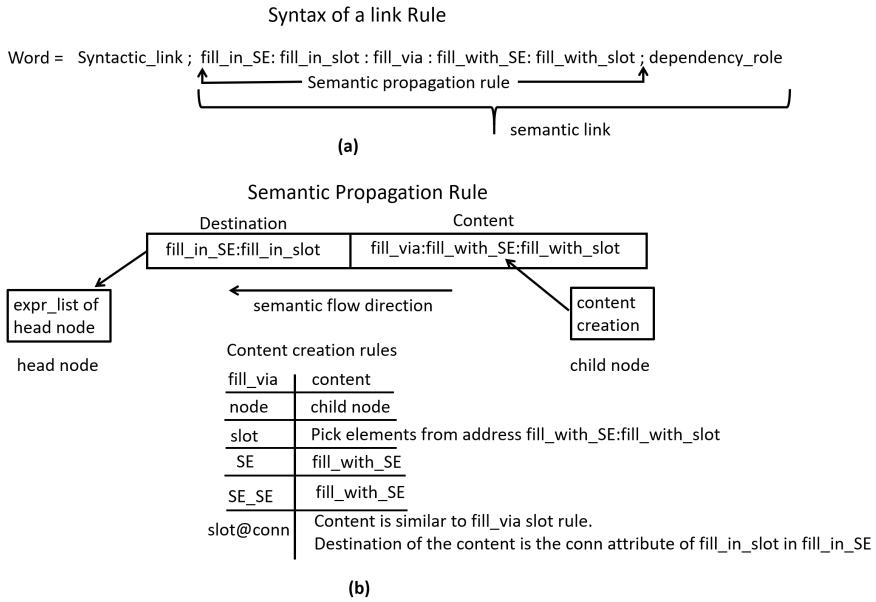


Figure 3: Structure of (a) a grammar link rule and (b) a semantic propagation rule.

4 Grammar

A link rule in our proposed grammar consists of a syntactic link and a semantic link. Figure 3 (a) shows a general structure of a link rule in the grammar. The syntactic link is taken from the link grammar (Sleator and Temperley, 1993). The meaning of syntactic link rules can be found in (Sleator and Temperley, 1991).

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 different 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. The syntactic 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.

A semantic propagation rule in a semantic link consist of five parameters separated by ":" as shown in figure 3 (a). A semantic propagation rule is needed to perform two tasks. The first task is to define the

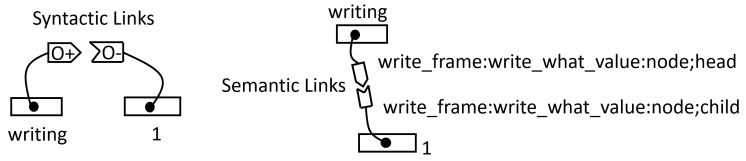


Figure 4: Syntactic links shows syntactic requirement of words in a sentence. Semantic links represent similar linking requirement between the head and child words in a dependency relation.

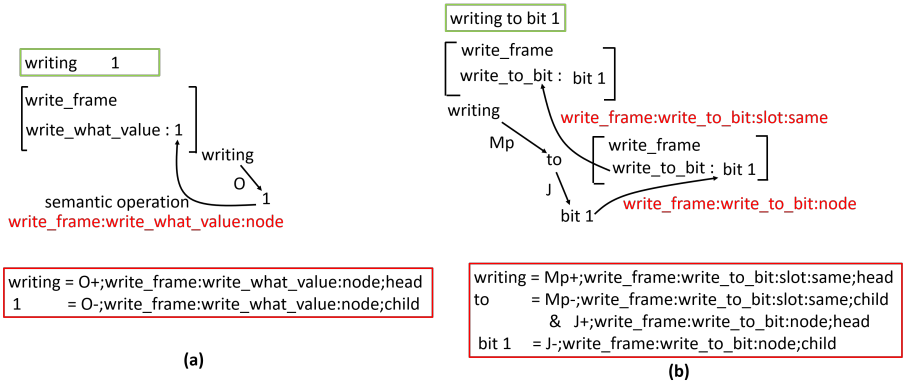


Figure 5: (a) Example of Semantic propagation operation when fill_via is node. (b) Example of Semantic propagation operation when fill_via is slot.

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 3 (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 the Figure 3 (b). When the 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.

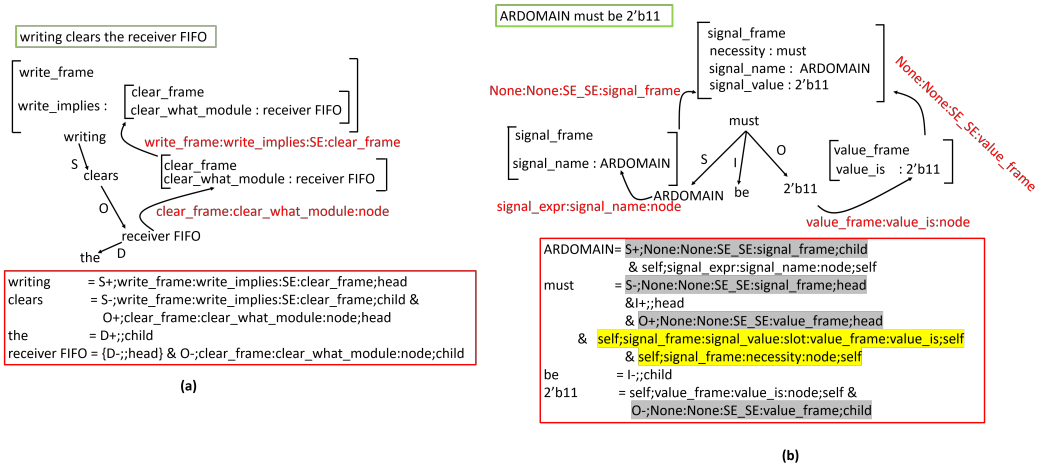


Figure 6: (a) Example of Semantic propagation operation when fill_via is SE. (b) Example of Semantic propagation operation when fill_via is SE_SE.

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 5 and Figure 6. In these figures, we have represented the example phrase in top green box and the corresponding grammar rules in the red box at the bottom of the figure.

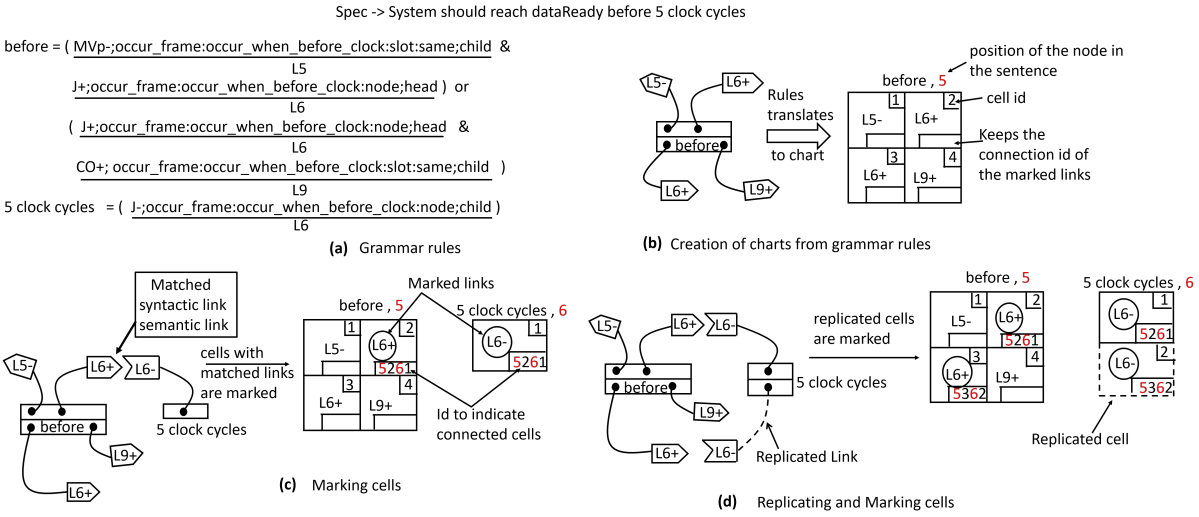


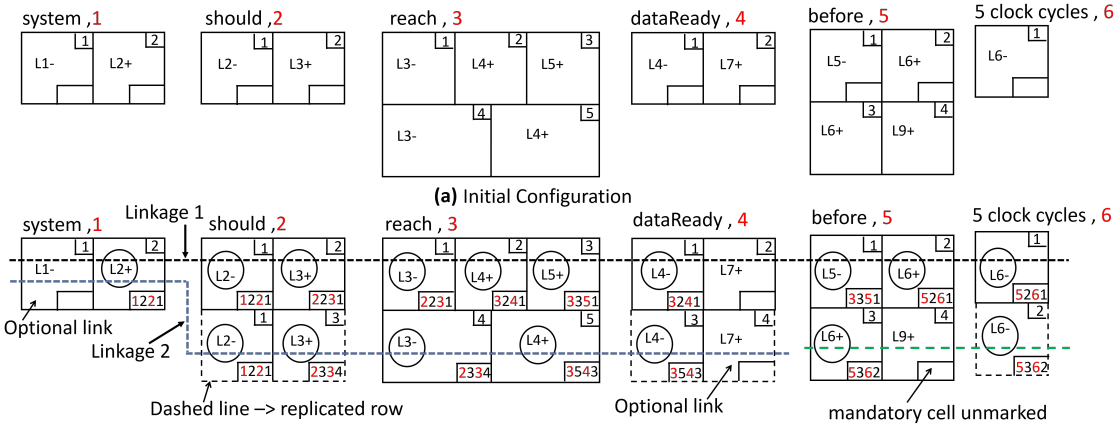
Figure 7: (a) Grammar link rules for words ‘before’ and ‘5 clock cycles’ to parse ‘ system should reach dataReady before 5 clock cycles’.(b) Translating grammar rules to BINGO chart. (c) Marking cells of links that have matched syntactic and semantic links without violation link order constraints. (d) Link replication and marking cells of replicated link.

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 5 (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.

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 5 (b) illustrates 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 fill_via node semantic operation. This content is then transferred to the node ‘writing’ using 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 parameters. 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.

Figure 6 (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 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 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.

Figure 6 (b) shows that the grammar can also express the composition operation of two SE’s. 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 fill_via SE_SE operation as highlighted in the grey color in the red box. 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



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 8: (a)Initial config. of Charts based on grammar rules given in Figure 9. (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.

and fill_in_slot parameters of the rule. Since the SE's 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 6 (b). We created self link rules in the grammar that allows to manipulate SE's 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.

5 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 are called linkages in (Sleator and Temperley, 1991). Figure 8(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 7 (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 chart as shown in the Figure 7 (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 covers only marked cells. A BINGO row represents a linkage that has satisfied all the syntactic-semantic constraints to build the parse tree.

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 7, 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 7, 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 7 (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.

Table 1: Grammar size for specifications tested

Total Words	Total Chart	Total rows
417	261	821

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 7 (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 7, 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 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.

Step2 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 8 illustrates the output after each step of the parsing for the specification 'system should reach dataReady before 5 clock cycles'. In Figure 8 (a) , the initial charts are shown with a subset of grammar rules of Figure 9.

Figure 8 (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 8 (c), where the final linkage is represented by the links covered by the BINGO row of the Black dashed line (Linkage 1).

Semantic creation: BINGO row provides linkages after taking into account the context of each node in the sentence. Nodes are connected with links in a transition-based dependency parsing framework like (Nivre, 2003). When a link is created between two nodes using either left-arc or right-arc transitions, then the semantic propagation rule associated with that link is also executed. The execution of semantic propagation rules for every transition arc between nodes results in the creation of a final SE at the root node of the parse tree. The resulting SVA translated from the root node SE is shown in Figure 8 (d).

6 Evaluation

The BINGO framework is written in JavaScript and is executed in 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

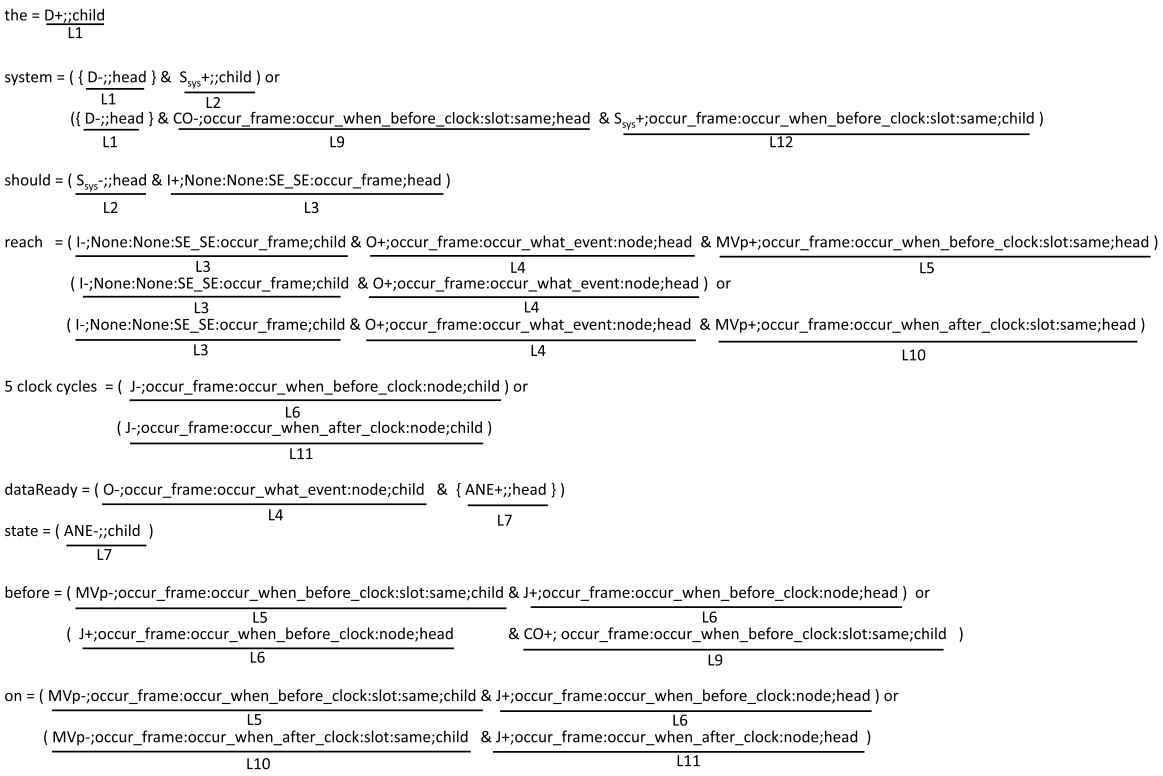


Figure 9: A small set of grammar rules to parse specifications

Table 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

framework by creating the grammar with syntactic and semantic link rules that can parse specifications found in documents. As shown in Table 1, our grammar consisted of a vocabulary of 417 words. We created 261 charts and 821 rows to represent syntactic and semantic connections of the words in our grammar. We present the framework's performance in Table 2 in terms of the type of specifications and number of specifications parsed, and the average time taken to parse each specification.

Similar to earlier approaches in (Harris and Harris, 2016; Zhao and Harris, 2019; Keszocze and Harris, 2019; Krishnamurthy and Hsiao, 2019), we picked specifications of ARM's AMBA protocol from (ARM, 2012) and (ARM, 2006) documents that have the names of all the signals and registers needed to generate an SVA code. In Table 2, RTL Spec type under the Spec type column refers to these low abstraction level 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

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 10: Specifications with RTL information are translated to SVA code.

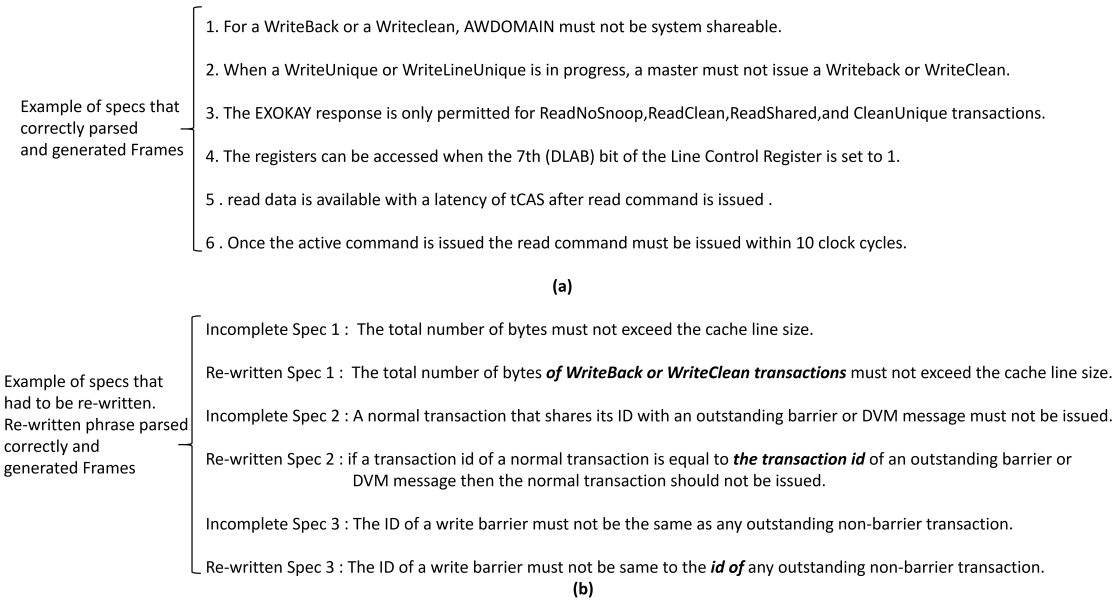


Figure 11: (a) Examples of specifications that were correctly parsed. (b) Example of specifications that had to be re-written for correct translations to semantic frames.

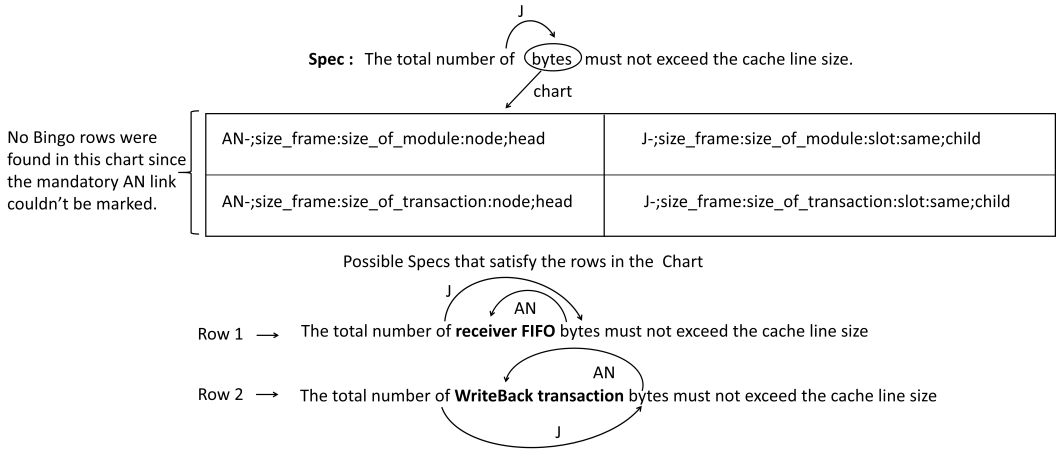


Figure 12: A BINGO row does not exist for incomplete specifications and we cannot create semantic frames for these specifications.

SVA is shown in Figure 10. However, as shown in (Krishnamurthy and Hsiao, 2020), these specifications are concise and lack variations in their sentence structures.

In order to evaluate the framework on different types of sentence structures, we created rules to generate semantic frames for high-level abstraction specs of the AMBA 4 ACE protocol checker document (ARM, 2012) . In the second row of Table 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. We parsed a total of 113 high-level specs from AMBA 4 ACE (ARM, 2012) document. We further evaluated the tool by manually extracting and re-writing 40 memory controller specifications from (Vijayaraghavan and Ramanathan, 2006) and 40 specifications from UART (Gorban, 2002) documents. A small set of these specifications and the high-level specs of AMBA 4 ACE protocol that were translated to semantic frames are shown in Figure 11 (a).

In Table 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 (ARM, 2012):

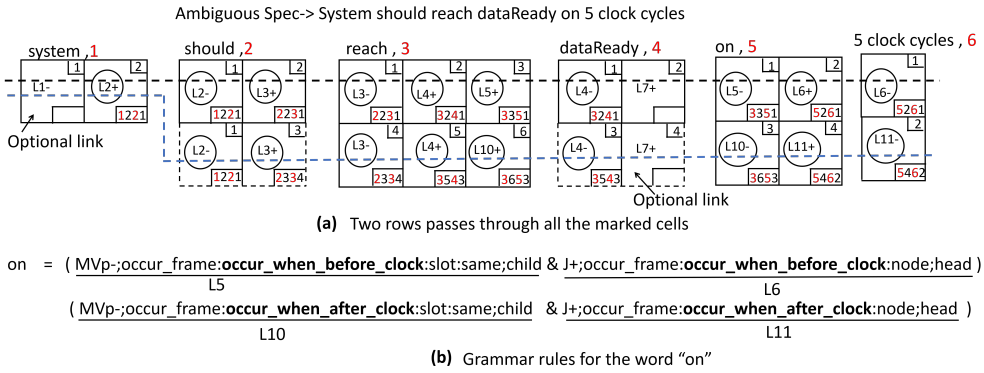


Figure 13: Ambiguous spec generates more than one BINGO row.

"A slave must not give an Isshared (RESP[3] = 'b1) response to a readsnoop, readunique, cleanunique, cleaninvalid, makeinvalid or makeunique transaction" takes 2 seconds to parse.

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 12 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 12, 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 linkage requirements of AN- link 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 re-write the incomplete specifications with the complete information needed for accurate understanding. In Figure 11 (b), we have illustrated examples of incomplete specifications of (ARM, 2012) that we re-wrote with additional details. The re-written specs were accurately parsed and translated to semantic frames.

In our framework, ambiguous sentences can be detected when more than one BINGO row with complete coverage is generated in the parsing stage. Figure 13 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 13 (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.

7 Conclusions and Future Work

In this paper, we demonstrated a dependency grammar-based framework to process hardware design specifications written in English. Our grammar is inspired from the syntactic link grammar. We have introduced contextual information and semantic propagation rules to the grammar using semantic links. We successfully evaluated the framework on specifications for a different range of hardware assertions taken from documents of four types of hardware architectures. We further modified some spec statements to test the robustness of the framework on handling different sentence structures. Our future work will further investigate the automatic detection of incomplete and ambiguous specifications and the generation of suggestions that can assist a user in writing specifications according to grammar.

Acknowledgement

The research is supported in part by National Science Foundation grant #2101021.

References

ARM, 2006. *AMBA 3 AXI Protocol Checker User Guide*. https://developer.arm.com/documentation/dui0305/b/Assertion-Descriptions.

ARM, 2012. *AMBA 4 ACE and ACE-Lite Protocol Checkers User Guide.* https://developer.arm.com/docs/dui0576/b/ace-and-ace-lite-protocol-assertion-descriptions.

Imran Sarwar Bajwa, Mark Lee, and Behzad Bordbar. 2012. Resolving syntactic ambiguities in natural language specification of constraints. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 178–187. Springer.

Aishwarya Chhabra, Amit Sangroya, and C Anantaram. 2018. Formalizing and verifying natural language system requirements using petri nets and context based reasoning. In *MRC@ IJCAI*, pages 64–71.

Michael A. Covington. 2001. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102.

Marie-Catherine De Marneffe and Joakim Nivre. 2019. Dependency grammar. *Annual Review of Linguistics*, 5:197–218.

Aaron Dutle, César Muñoz, Esther Conrad, Alwyn Goodloe, Laura Titolo, Ivan Perez, Swee Balachandran, Dimitra Giannakopoulou, Anastasia Mavridou, and Thomas Pressburger. 2020. 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.

Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. 2016. Arsenal: automatic requirements specification extraction from natural language. In *NASA Formal Methods Symposium*, pages 41–46. Springer.

Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. 2020. Generation of formal requirements from structured natural language. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 19–35. Springer.

Daniel Gildea. 2001. Corpus variation and parser performance. In *Proceedings of the 2001 Conference on Empirical Methods in Natural Language Processing*.

Jacob Gorban, 2002. *UART ip core specification Architectures.* http://www.isy.liu.se/edu/kurs/TSEA44/OpenRISC/UART_spec.pdf.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642.

Christopher B Harris and Ian G Harris. 2016. 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.

Michael S Hsiao. 2018. Automated program synthesis from object-oriented natural language for computer games. In *Proceedings of the Sixth International Workshop on Controlled Natural Language (CNL)*, pages 71–74.

Michael S Hsiao. 2021. Multi-phase context vectors for generating feedback for natural-language based programming. In *Proceedings of the Seventh International Workshop on Controlled Natural Language (CNL)*.

Oliver Keszocze and Ian G Harris. 2019. Chatbot-based assertion generation from natural language specifications. In *2019 Forum for Specification and Design Languages (FDL)*, pages 1–6. IEEE.

Rahul Krishnamurthy and Michael S Hsiao. 2019. Controlled natural language framework for generating assertions from hardware specifications. In *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*, pages 367–370. IEEE.

Rahul Krishnamurthy and Michael S Hsiao. 2020. Transforming natural language specifications to logical forms for hardware verification. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 393–396. IEEE.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. 2018. 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)*.

Anastasia Mavridou, Hamza Bourbouh, Dimitra Giannakopoulou, Thomas Pressburger, Mohammad Hejase, Pierre-Loic Garoche, and Johann Schumann. 2020. The ten lockheed martin cyber-physical challenges: formalized, analyzed, and explained. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pages 300–310. IEEE.

Zifan Nan, Hui Guan, Xipeng Shen, and Chunhua Liao. 2021. 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.

Alexis Nasr and Owen Rambow. 2004. A simple string-rewriting formalism for dependency grammar. In *Proceedings of the Workshop on Recent Advances in Dependency Grammar*, pages 17–24.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the eighth international conference on parsing technologies*, pages 149–160.

Sandip Ray, Ian G Harris, Goerschwin Fey, and Mathias Soeken. 2016. Multilevel design understanding: from specification to logic. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6. IEEE.

Daniel Sleator and Davy Temperley. 1991. Parsing english with a link grammar. carnegie mellon university computer science technical report cmu-cs-91-196. `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/link/pub/www/papers/ps/tr91-196.pdf`.

Daniel D Sleator and David Temperley. 1993. Parsing english with a link grammar. In *Proceedings of the Third International Workshop on Parsing Technologies*, pages 277–292.

Mathias Soeken, Christopher B Harris, Nabila Abdessaied, Ian G Harris, and Rolf Drechsler. 2014. 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.

Srikanth Vijayaraghavan and Meyyappan Ramanathan. 2006. Sva for memories. In *A practical guide for SystemVerilog assertions*, chapter 5, pages 191–232. Springer US, Boston, MA. `https://doi.org/10.1007/0-387-26173-7_6`.

Rongjie Yan, Chih-Hong Cheng, and Yesheng Chai. 2015. Formal consistency checking over specifications in natural languages. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1677–1682. IEEE.

Shiyu Zhang, Juan Zhai, Lei Bu, Mingsong Chen, Linzhang Wang, and Xuandong Li. 2020. 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.

Junchen Zhao and Ian G Harris. 2019. Automatic assertion generation from natural language specifications using subtree analysis. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 598–601. IEEE.