Automated Suggestions Framework for Processing Hardware Specifications Written in English

Rahul Krishnamurthy
Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, Virginia, USA 24061
rahulk4@vt.edu

Michael S. Hsiao
Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, Virginia, USA 24061
hsiao@vt.edu

***Abstract*—Automatic creation of formal models from natural language specifications can help reduce design time and manual errors. However, the accuracy of the translation may not be high due to the ambiguous, incomplete, and inconsistent nature of natural language. Specifications written in a controlled natural language (CNL) can overcome the problems associated with natural language translation and deliver similar benefits in automation. However, a user has to learn to write in a CNL. We propose a suggestions framework that provides automatic feedback to assist users in writing specifications in CNL. Our feedback generates different ways of writing CNL acceptable sentences when the input sentence is not understood. We developed a ranking scheme to ensure the semantics of generated suggestions are closer to the input specification's intent. We evaluated the 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.**

***Index Terms*—controlled natural language, Assertion-based verification, Suggestion generation**

I. Introduction

One of the root causes for the difficulties in hardware design validation is the informal nature of specification documents [1], [2]. Specification documents contain the intended behavior of the individual IP blocks and the overall SoC design in natural language, charts, and tables. The absence of standardization in writing specifications leads to inconsistencies, ambiguities, and even errors in requirement documents [1]. The design validation process is further delayed when these erroneous specifications are used as an authoritative guide for planning various design activities [1].

Standardized specifications not only have the benefits of being clear, coherent, and unambiguous but can also allow for automatic verification of the design. Automating verification can improve verification time and reduce manual errors in creating testbench and assertions [3].

Recent work on standardizing specifications and automatic formal code generation in [4], [5] manually translates specifications to a formal modelling 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. In contrast, writing specifications in standardized English language is easier and much more intuitive. However, standardized English is a subset of natural language with limited vocabulary and restricted grammar rules. A user may not be aware of syntactic and semantic constructs allowed in the standardized English language to write specifications. Therefore, it would be extremely useful if immediate feedback to any mistakes to a spec could be provided to the user.

As a motivating example, the following spec "CDREADY should be high on 3 cycles after assertion of CDVALID" may seem correct to the user but is actually ambiguous and cannot be translated. The ambiguity is about the time of triggering the event. It is unclear if the CDREADY signal should be high after or exactly on 3 cycles. A user has to be informed automatically about these and other issues and given suggestions to fix the sentence that can be accepted by the system.

We present a suggestion generation framework that provides feedback to users on writing specifications that comply with the grammar and vocabulary of a controlled English language. Specifications written according to an underlying grammar are parsed and translated to either semantic frames or system verilog assertions. The framework is built on top of the BINGO model [6] that consist of a grammar and a parser to understand specs written in English.

Our Contributions are summarized as follows:

- The generated suggestions are ensured to be syntactically and semantically accurate.
- A ranking scheme is developed to generate top ranked suggestions closer to the syntactic and semantic structures of the input specification.
- An input sentence can be rewritten by adding new words, removing existing words and re-ordering the words at any location in the input spec.
- We can infer design variables like signals and registers from the input spec. using contextual analysis at the end of suggestion generation. As a result, a user can write their design specific variable names in specs without explicitly declaring them a priori.
- Our suggestion framework is only dependent on the grammar rules of BINGO model. We can port our suggestion

supported in part by NSF grant 2101021.

mechanism to any domain by simply creating domain specific grammar rules in the BINGO format.

The rest of the paper is organized as follows. Section 2 discusses some previous authoring support techniques for CNLs and earlier work in rewriting sentences. In Section 3, a brief overview of BINGO framework is presented. Section 4 presents the methodology of our suggestions framework. In Section 5, we discuss the evaluation of our work. Finally, a concluding summary is presented in Section 6.

II. Related Work

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 [7] performs auto-completion using a lookahead analysis based on Prolog DCG grammar rules and parser. The predictive editor in [8] 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. CNL editor proposed in [9] 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 [10], users can be overwhelmed by many options in a drop-down menu. In [11], [12] context vectors obtained from parsing of the erroneous sentence are used to generate suggestions.

In [13], 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 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 [14], 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.

We propose a suggestion framework that can generate alternate ways of writing sentences when an input sentence is not completely understood. Our approach can lead to the insertion of new words, removal, and re-ordering of the existing words of the input specification to make it acceptable to the CNL system. In our work, we leverage our existing syntactic-semantic grammar rules to rewrite sentences instead of creating additional rewriting rules. Our approach doesn't require parsing of all possible rewritten sentences to ensure the generation of CNL acceptable sentences. 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. Our 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.

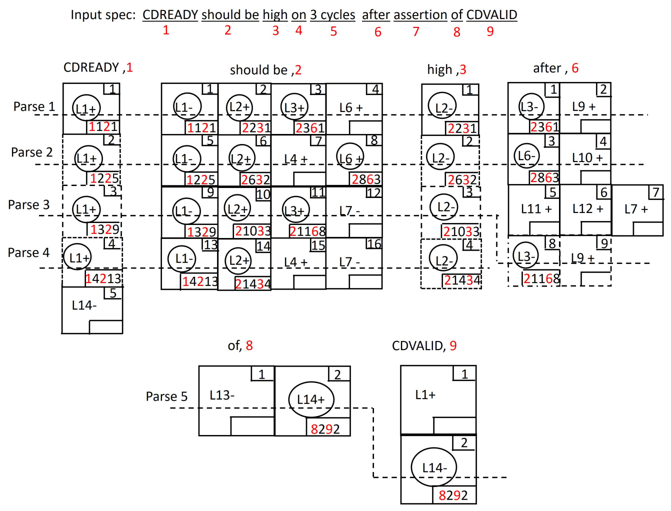


Fig. 1. Incomplete set of charts at the end of BINGO parsing.

III. BINGO Framework

This section provides a brief overview of the BINGO framework and terminology that will be useful in explaining the proposed suggestion generation.

BINGO model: In the BINGO model [6], 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 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 2 illustrates grammar rules for nodes used to create charts in Figure 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.

The nodes in grammar rules that are indicated in brackets '$<$ ', '$>$' represents rules for a category. For example, '$<$signal$>$' rules are assigned to all signal category nodes during parsing.

Chart Creation: The link rules of '$<$signal$>$' node consist of two disjunct links L_1 and L_{14}. The disjunct links of

'<signal>' are represented by two separate rows in the chart of the node 'CDREADY' in Figure 1. The remaining dashed cells in the node 'CDREADY' chart are created by replicating the first row link L_1 during parsing.

The conjunct links of a node's grammar rules are represented by a single row in the node's chart. For example, the conjunct links L_1,L_2,L_3 and L_6 in the rule of the node 'should be' is translated to a single row in the chart of 'should be' in Figure 1. The self links like L_5 and L_8 in the grammar rules are not shown in the charts. These self link rules are used to manipulate the semantics of frames on the node and do not connect words in dependency relations.

Marking chart cells: Figure 1 illustrates charts of nodes at the end of parsing. The marking in these charts is indicated by circling the links in the cells. The links in the cells correspond to the syntactic-semantic links of the grammar rules that are shown in Figure 2. The links with the same label but with opposite direction polarity of '+' and '-' and opposite dependency polarity of 'head' and 'child' are considered matched links and are marked with the same connection id. The connection id is shown at the bottom right corner of the marked cells.

Two cells with the same connection id indicate a connection between two nodes in a parse tree. For example, connection id '1121' in the chart of nodes 'CDREADY' and 'should be' represents linking between these nodes through links L_1+ and L_1- in a parse tree.

Chart Cell identifiers: As illustrated in Figure 1, 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 cell 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 'CDREADY' chart, the cell connection id of the L_1 link in the first row is 1121 (node 1 cell 1 and node 2 cell 1) and is different from the L_1 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.

Complete Parse or BINGO row: A node is understood unambiguously when only one row of the node's chart has all the mandatory cells marked. A single complete parse tree is obtained when all the nodes of the input spec are understood unambiguously. 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 1, the chart of some nodes have incomplete marking in their rows and can never 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 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 1 are shown in Figure 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 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'. The interpretation of grammar link rules is explained in [6].

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 1 are 'CDREADY','high','3 cycles','assertion' and 'CDVALID'.

```
<signal> =  S+;signal_expr:signal_node:node;child  or
                              L1
           J-;change_expr:change_what_signal:node;child
                              L14
should be =  S-; signal_expr:signal_node:node;head  &
                              L1
                O+; signal_expr:signal_value_is:node;head &
                              L2
                ( MV+;occur_expr:occur_when_after_clock:slot:same;head or
                              L3
                 MV+;occur_expr:occur_when_before_clock:slot:same;head    ) &
                              L4
                 self;occur_expr:occur_what:SE:signal_expr;self &
                              L5
               ( MV+;if_expr:ante:slot:same;head      or
                              L6
                 MV-;if_expr:ante:slot:same;head   )  &
                              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 &
                              L3
          J+;occur_expr:occur_when_after_clock:node;head ) or
                              L9
         ( MV-;if_expr:ante:slot:same;child &  J+;if_expr:ante:slot:same;head )  or
                              L6                          L10
         ( J+;if_expr:ante:slot:same;head & {prep_comma+;;head} &
                              L11                   L12
           MV+;if_expr:ante:slot:same;child )
                              L7
of   =   Mf-;change_expr:change_what_signal:slot:same;child &
                              L13
          J+;change_expr:change_what_signal:node;head
                              L14
```

Fig. 2. Grammar rules for creating charts shown in Figure 1.

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 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 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 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.

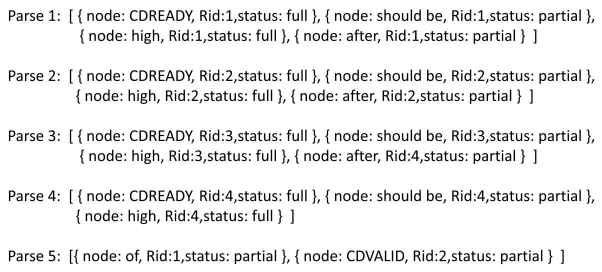
missing_input_non_concept_node_in_parse (MIN): MIN i for a Parse i is a list of non_concept nodes of the input spec that are not present in the Parse i tree. In Figure 4, MIN 1 list contains the nodes 'on' and 'of' since these non_concept nodes are not part of Parse 1 tree. Similarly, MIN 5 contains 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 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 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.

IV. Methodology

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



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 }]

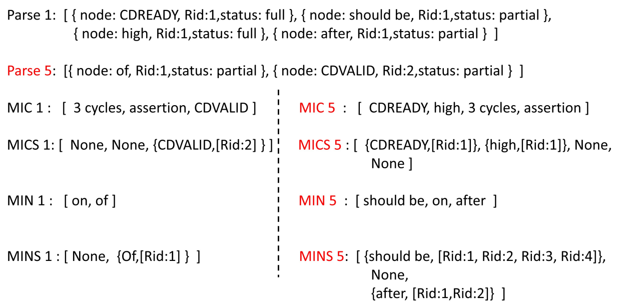
Parse 2: [{ node: CDREADY, Rid:2,status: full }, { node: should be, Rid:2,status: partial }, { node: high, Rid:2,status: full }, { node: after, Rid:2,status: partial }]

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 }]

Fig. 3. Incomplete parse tree information extracted from Fig 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]}]

Fig. 4. Parameters of incomplete Parse 1 and Parse 5 from Fig. 1.

Figure 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 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 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

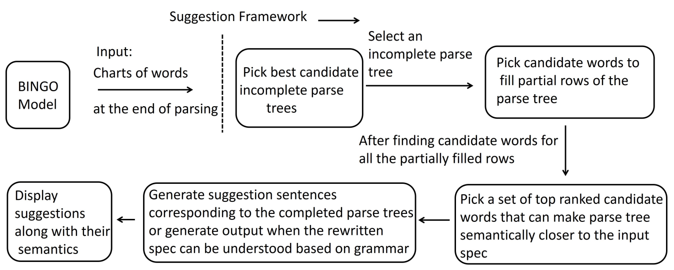

Suggestion Framework
BINGO Model
Input: Charts of words at the end of parsing
Pick best candidate incomplete parse trees
Select an incomplete parse tree
Pick candidate words to fill partial rows of the parse tree
After finding candidate words for all the partially filled rows
Pick a set of top ranked candidate words that can make parse tree semantically closer to the input spec
Generate suggestion sentences corresponding to the completed parse trees or generate output when the rewritten spec can be understood based on grammar
Display suggestions along with their semantics

Fig. 5. Main blocks of suggestion framework on top of BINGO model.

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 1.

A. *Selecting Incomplete Parse trees*

As illustrated in Figure 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. 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 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.

B. *Finding Candidate Words*

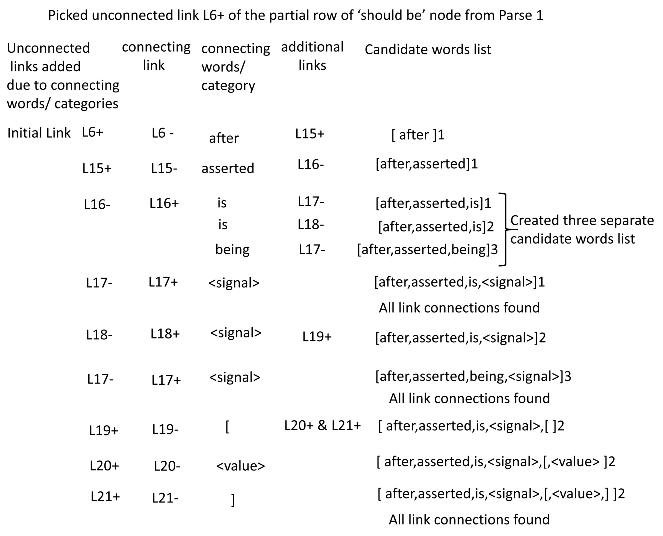
After selecting the incomplete parse trees, we begin the search for candidate words that can be connected in the parse tree to create a meaningful and parsable sentence.

A parse of a spec is incomplete due to the existence of partially filled rows, as illustrated in the Figure 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 6. The complete set of labeled link rules for the candidate words/categories are shown in Figure 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 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.

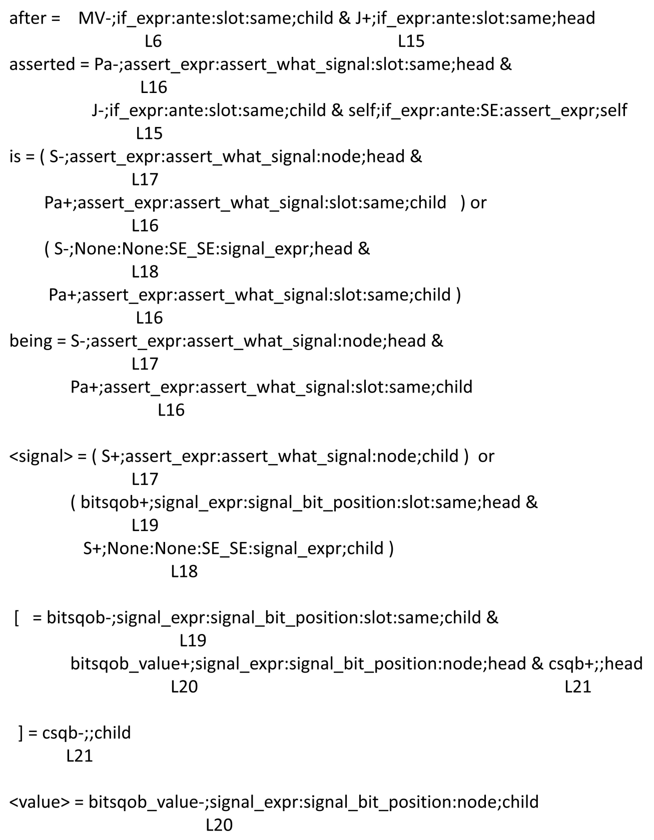
We can also have more than one word in the grammar that can satisfy the link requirement of an unconnected link. For example, $L_{16}-$ link can be matched with $L_{16}+$ link that belongs to three different link rules of two words, 'is' and 'being'. As shown in Figure 6, we add all three links and their words in different set of candidate word lists. Each set of candidate words list represents a group of words that can separately fulfill all the link requirements of the partial row in the Parse 1 tree. The process of searching for matching links and adding new candidate words and new links in our parse tree continues until all the links in the parse tree are marked. The end of this process is indicated in Figure 6 by "All link connections found" comment for each candidate word list.



Picked unconnected link L6+ of the partial row of 'should be' node from Parse 1

	Unconnected links added due to connecting words/ categories	connecting link	connecting words/ category	additional links	Candidate words list	
Initial Link	L6+	L6 -	after	L15+	[after]1	
	L15+	L15-	asserted	L16-	[after,asserted]1	
	L16-	L16+	is	L17-	[after,asserted,is]1	Created three separate candidate words list
			is	L18-	[after,asserted,is]2	
			being	L17-	[after,asserted,being]3	
	L17-	L17+	<signal>		[after,asserted,is,<signal>]1 All link connections found	
	L18-	L18+	<signal>	L19+	[after,asserted,is,<signal>]2	
	L17-	L17+	<signal>		[after,asserted,being,<signal>]3 All link connections found	
	L19+	L19-	[	L20+ & L21+	[after,asserted,is,<signal>,[ ]2	
	L20+	L20-	<value>		[after,asserted,is,<signal>,[,<value>]2	
	L21+	L21-	]		[after,asserted,is,<signal>,[,<value>,]]2 All link connections found	

Fig. 6. Connecting L6+ link of 'should be' node from Parse 1.



```
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
```

Fig. 7. Link rule used for completing the connection of L6+ link of Fig 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
```

Fig. 8. Scoring candidate words based on parse tree parameters coverage.

C. Ranking Mechanism

Many lists of candidate words can be generated through the process illustrated in Figure 6. The ranking mechanism assists in selecting candidate words list that can generate 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 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 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.

We combine these five different scores and generate a single score for a suggestion using a weighted equation shown in Figure 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

```
/*   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
}
```

Fig. 9. Computing a single score for a candidate words list.

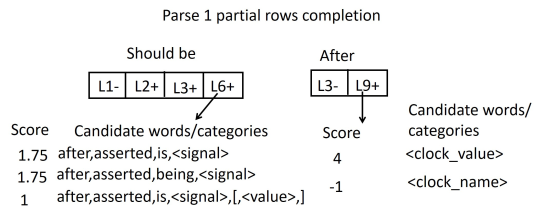

Parse 1 partial rows completion
Should be
L1-
L2+
L3+
L6+
After
L3-
L9+
Score
Candidate words/categories
1.75
after,asserted,is,<signal>
1.75
after,asserted,being,<signal>
1
after,asserted,is,<signal>,[,<value>,]
Score
Candidate words/ categories
4
<clock_value>
-1
<clock_name>

Fig. 10. Final set of candidate words for partial row's unconnected links and their scores.

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.

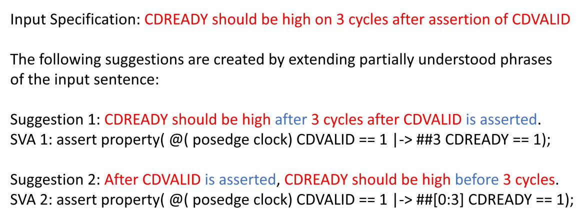
Because we want to generate concise suggestions, we have used the count of new words taken from vocabulary as the denominator in the combined_score metric for the suggestion.

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 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 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.

D. Generating Suggestions

At the end of ranking mechanism, we have the top-ranked candidate words and their links that can satisfy the unmarked links of partial row. 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 11. The red color words in the suggestions are taken



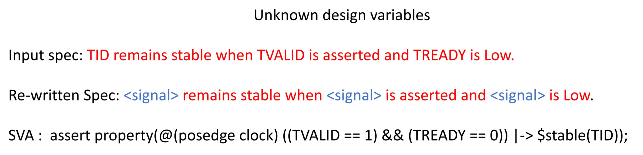
Input Specification: CDREADY should be high on 3 cycles after assertion of CDVALID

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.
SVA 1: assert property(@(posedge clock) CDVALID == 1 |-> ##3 CDREADY == 1);

Suggestion 2: After CDVALID is asserted, CDREADY should be high before 3 cycles.
SVA 2: assert property(@(posedge clock) CDVALID == 1 |-> ##[0:3] CDREADY == 1);

Fig. 11. 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));

Fig. 12. Signal name inference & SVA generation based on re-written spec.

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.

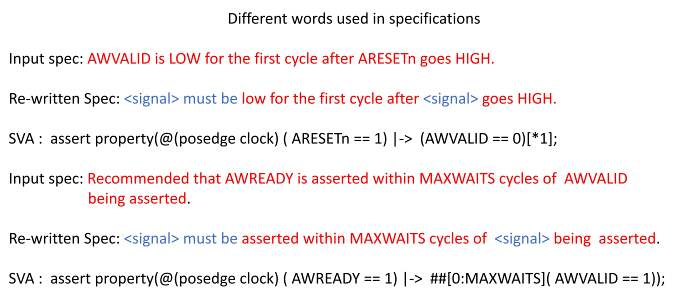
The framework generates four suggestions for the input spec but chose only two suggestions for display. The final list of suggestions are selected based on the semantics generated by the suggestions. The suggestions that can generate unique semantics in few words are selected in the final output. As shown in Figure 11, the System Verilog Assertion (SVA) semantics of the generated suggestions are different and hence they were selected to be displayed.

When the rewritten spec 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 12.

V. 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 and generate SVA if the re-written spec is equivalent in semantics to input specification.

We created grammar rules and vocabulary based on specifications taken as it is from AMBA ACE [15] document. We manually wrote specifications in English based on the semantics of assertions verifying memory controller architecture given in [16]. 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.



Different words used in specifications

Input spec: AWVALID is LOW for the first cycle after ARESETn goes HIGH.

Re-written Spec: <signal> must be low for the first cycle after <signal> goes HIGH.

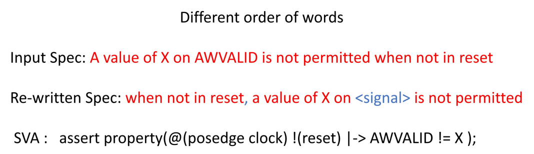
SVA : assert property(@(posedge clock) (ARESETn == 1) |-> (AWVALID == 0)[*1];

Input spec: Recommended that AWREADY is asserted within MAXWAITS cycles of AWVALID being asserted.

Re-written Spec: <signal> must be asserted within MAXWAITS cycles of <signal> being asserted.

SVA : assert property(@(posedge clock) (AWREADY == 1) |-> ##[0:MAXWAITS](AWVALID == 1));

Fig. 13. Analysis of rewritten spec led to SVA generation of specs with unknown sentence structures/words.



Different order of words

Input Spec: A value of X on AWVALID is not permitted when not in reset

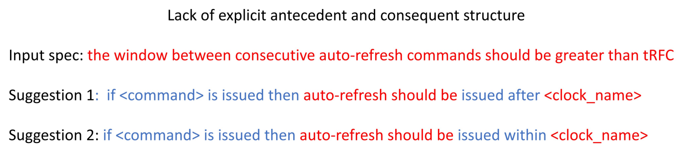
Re-written Spec: when not in reset, a value of X on <signal> is not permitted

SVA : assert property(@(posedge clock) !(reset) |-> AWVALID != X);

Fig. 14. 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 [17], and AMBA AXI4 Stream Protocol Assertions [18] documents. These spec documents have semantics similar to the specs of [15] but have variations in sentence structures. We took natural language specifications from [16] 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.



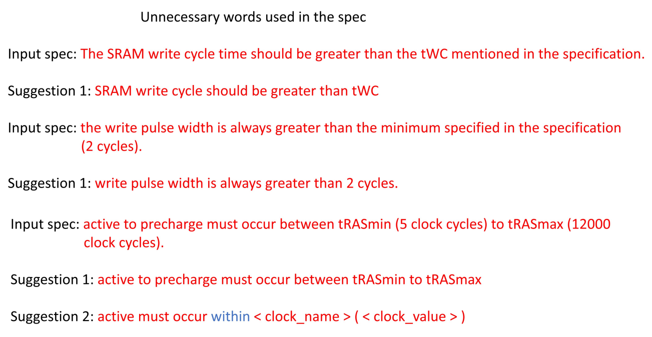
Lack of explicit antecedent and consequent structure

Input spec: the window between consecutive auto-refresh commands should be greater than tRFC

Suggestion 1: if <command> is issued then auto-refresh should be issued after <clock_name>

Suggestion 2: if <command> is issued then auto-refresh should be issued within <clock_name>

Fig. 15. Suggestion generation for the spec without standard antecedent-consequent structure.



Unnecessary words used in the spec

Input spec: The SRAM write cycle time should be greater than the tWC mentioned in the specification.

Suggestion 1: SRAM write cycle should be greater than tWC

Input spec: the write pulse width is always greater than the minimum specified in the specification (2 cycles).

Suggestion 1: write pulse width is always greater than 2 cycles.

Input spec: active to precharge must occur between tRASmin (5 clock cycles) to tRASmax (12000 clock cycles).

Suggestion 1: active to precharge must occur between tRASmin to tRASmax

Suggestion 2: active must occur within < clock_name > (< clock_value >)

Fig. 16. Unnecessary words removed from original spec in suggestions.

There were a total of 94 specifications in [17] and [18] that had different sentence structures and words, but their semantics were similar to the specifications in [15]. We used these 94 specs to test our framework.

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 specs where it expected a signal name, as shown in Figure 12. 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.

The test specifications contained non_concept words different from the words used in our grammar. An example is illustrated in Figure 13. The system rewrote the input spec with non_concept words of our system as shown in 'Re-written Spec'. Since 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, the spec was considered semantically equivalent to the input, and an SVA code was generated instead of a suggestion.

Similarly, in Figure 14, the system generated a rewritten spec that contained all the input concept and non_concept words with the same semantic roles. An SVA was generated for this spec with different order of words.

Results Analysis: All 94 specs had design variables unknown to our system. Of these specs, 41% had different order of phrases as shown in Figure 14 and 59% of specs had different words as shown in Figure 13. By manual validation, we verified that all the suggestions were correct and semantically close to the intent of the input suggestion.

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 15, the spec does not have explicit antecedent and consequent phrases. The system was able to generate suggestions explaining the required conditional structure, as shown in Figure 15. Moreover, the system was able to remove unnecessary words from specifications to generate semantically accurate suggestions, as shown in Figure 16.

Results Analysis: Of all the 38 specs, 55% had unknown words and 45% had different sentence structures. Our approach generated suggestions for all the 38 specs. Manual inspection of suggestions has showed that 60% of the suggestions were semantically close to user's intent like the suggestions in Figure 16. In the other 40%, the system was able to explain the sentence structural requirements to the user as shown in Figure 15.

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.

VI. Conclusions

We presented a framework for automatically generating suggestions by rewriting the input spec into CNL acceptable sentences with semantics closer to the user's intent. The framework was evaluated on specifications that had words and the order of words not defined in our grammar. The system generated useful suggestions for 87% of the input specs. Moreover, the rewritten specs generated by the suggestions framework provided additional analysis capability to improve the flexibility in understanding specifications.

References

[1] S. Ray, I. G. Harris, G. Fey, and M. Soeken, "Multilevel design understanding: from specification to logic," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–6.

[2] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, "Challenges and trends in modern soc design verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, 2017.

[3] B. Keng, S. Safarpour, and A. Veneris, "Automated debugging of systemverilog assertions," in *2011 Design, Automation & Test in Europe*. IEEE, 2011, pp. 1–6.

[4] E. Ebeid, F. Fummi, and D. Quaglia, "Hdl code generation from uml/marte sequence diagrams for verification and synthesis," *Design automation for embedded systems*, vol. 19, no. 3, pp. 277–299, 2015.

[5] M. Leite and M. A. Wehrmeister, "System-level design based on uml/marte for fpga-based embedded real-time systems," *Design Automation for Embedded Systems*, vol. 20, no. 2, pp. 127–153, 2016.

[6] R. Krishnamurthy and M. 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)*, 2021, pp. 68–80.

[7] T. Kuhn and R. Schwitter, "Writing support for controlled natural languages," in *Australasian Language Technology Association Workshop 2008*, vol. 6, 2008, pp. 46–54.

[8] S. Palmaz, M. Cuadros, and T. Etchegoyhen, "Statistically-guided controlled language authoring," in *5th International Workshop, CNL 2016, Aberdeen, UK, July 25-27, 2016*, 2016, pp. 37–47.

[9] K. Angelov and M. B. Mechura, "Editing with search and exploration for controlled languages," in *Proceedings of the Sixth International Workshop, CNL*, 2018, pp. 1–10.

[10] F. Hielkema, C. Mellish, and P. Edwards, "Evaluating an ontology-driven wysiwym interface," in *Proceedings of the Fifth International Natural Language Generation Conference*, 2008, pp. 138–146.

[11] M. S. Hsiao, "Automated program synthesis from object-oriented natural language for computer games," in *Proceedings of the Sixth International Workshop, CNL*, 2018, pp. 71–74.

[12] M. 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.

[13] B. Chen, L. Sun, X. Han, and B. An, "Sentence rewriting for semantic parsing," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 766–777. [Online]. Available: https://aclanthology.org/P16-1073

[14] T. Mitamura and E. 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.

[15] 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.

[16] S. Vijayaraghavan and M. Ramanathan, "Sva for memories," in *A practical guide for SystemVerilog assertions*. Boston, MA: Springer US, 2006, ch. 5, pp. 191–232, https://doi.org/10.1007/0-387-26173-7_6.

[17] ARM, *AMBA 3 AXI Protocol Checker User Guide*, 2006.

[18] *ARM, AMBA 4 AXI4, AXI4-Lite, and AXI4-Stream Protocol Assertions User Guide.*, 2010.