Automating the Translation of Assertions Using Natural Language Processing Techniques

Mathias Soeken[1,2] Christopher B. Harris[3] Nabila Abdessaied[2] Ian G. Harris[4] Rolf Drechsler[1,2]
[1] Department of Mathematics and Computer Science, University of Bremen, Germany
[2] Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
[3] Department of Electrical and Computer Engineering, University of California, Irvine, United States
[4] Department of Computer Science, University of California, Irvine, United States
{msoeken,nabile,drechsle}@cs.uni-bremen.de christopher.harris@uci.edu harris@ics.uci.edu

***Abstract*—In order to verify natural language assertions from a specification automatically, they need to be translated into formal representations. This process is error-prone and can lead to a product that does not meet the initial intentions. We automate this process by first partitioning all assertions into subsets based on sentence similarity and then providing a translation template for each subset which must be completed by the designer. Since many assertions are described by similar sentences, the number of manual translation steps can be decreased significantly. We evaluated our approach by translating English constraint sentences from an industrial specification into SystemVerilog assertions.**

***Index Terms*—Design automation, Natural language processing.**

I. Introduction

Verification is not easy. As system complexity has increased with the advent of *Systems-on-Chip* (SoC) we have seen functional verification move from an informal activity completed by design engineers, to a distinct activity undertaken by verification engineers, to an activity that often dominates the design cycle. It is widely accepted that verification activities can take up to 60% of the design cycle in a modern SoC. However, while the systems we must verify are getting more complex the cost of getting it wrong continues to increase. The continued increase in manufacturing costs for integrated circuits provide strong incentive to find as many design errors as possible before a lithographic mask set is generated. The problem is tougher and the stakes are higher than every before.

One technique used to address this growing problem is *Assertion Based Verification* (ABV). Assertions have been used for years to help verify complex software systems. In recent years, verification languages such as SystemVerilog have been extended to include support for hardware assertions which allow greater observability, controllability, and can reduce verification effort. One study reported up to a 50% reduction in verification effort through the use of ABV [1].

However, ABV alone is not a panacea for the verification problem. As important as *how* to verify a design is the question of *what* to verify. The accurate conversion of a design specification into a set of assertions can result in the detection of hard to catch bugs. The manual translation of large numbers of requirements into formal assertion statements can be an error prone process. Malformed assertions can result in lost time and cause verification errors [2].

To assist in the process of efficiently generating well defined assertions we propose a technique to automatically extract assertion-type information written in English from a system specification. Even though a specification contains substantial linguistic variation, sentences containing *Natural Language Assertions* (NLAs) tend to exhibit grammatical similarities based on the type of property check the sentence is describing.

These NLAs can be automatically grouped into clusters of sentences of similar linguistic structure, where each NLA in the cluster can be described by an archetypical SystemVerilog Assertion (SVA). This archetypical assertion template can then be used to automatically generate a specific and correct assertion statement for each NLA in a cluster. Consider the following sentences: “REQUEST is only permitted to change from HIGH to LOW when ACKNOWLEDGE is HIGH”, and “ACKNOWLEDGE is only permitted to change from LOW to HIGH when REQUEST is HIGH”. Although the specifics differ, both NLAs display a similar sentence structure and both describe an assertion check where a signal is only allowed to toggle from one specified state to another when a different signal is asserted.

Using our technique, a verification engineer tasked with generating assertions for dozens of requirements in a typical specification would only need to generate a small number of archetypical assertion templates which are then used to automatically generate a full set of assertions. This not only reduces the potential for errors in the translation of large numbers of English language requirements to formal assertion statements, but notably reduces verification effort. It also has the benefit of allowing a verification engineer to expend a larger percentage of time generating checks for difficult to verify conditions such as corner cases, thus improving overall verification quality.

In Section 2 we will outline the overall flow of our technique. Sections 3 and 4 will discuss the dependency representation used to characterize sentences in terms of their grammatical structure, and our information extraction algorithm which utilizes triple store databases to identify and extract property information from target NLAs. Section 5 details the implementation of our assertion translation technique while Section 6 evaluates that implementation using a design document for a modern bus specification. We discuss our

results in Section 7 and compare our technique to related work in Section 8 before summarizing our conclusions in Section 9.

II. Proposed Flow

Fig. 1 illustrates the general flow of the proposed approach. The starting point is a set of NLAs given in terms of English sentences (indicated in the figure by zigzag lines). These assertions are automatically partitioned into subsets of *high abstraction level* and *low abstraction level* assertions in the first step. The idea is that high level assertions contain implicit and imprecise information which impedes automatic translation and therefore need to be translated manually as in the conventional flow. Consequently, high level assertions are not further considered in the remainder of the flow.

In the second step the determined low level assertions are partitioned into clusters of similar sentences. For this purpose, a metric for sentence similarity is defined based on the grammatical structure of the sentence and the semantics of some words. Each cluster is represented by a graph structure that represents the general structure of the sentences and stores the words which are variable for all sentences in the cluster.

The third step is a manual step in which the designer has to define transformation rules for each cluster. The transformation rule can be described in terms of a formal property (indicated by solid lines) that also contains variables similar to those in the general structural graph of the cluster. Once such a rule is defined, all sentences of the corresponding cluster can automatically be translated to assertions by extracting the variable words from the sentence and inserting them into the respective placeholders in the formal property.

The proposed flow can be as good as or better than the conventional flow. It cannot be worse when comparing the number of translation steps that must be performed manually. In the worst case the proposed flow does not detect any low level assertion or the partition contains only clusters of exactly one element. In this case each assertion needs to be translated manually as in the conventional flow.

We conjecture that the proposed flow is robust although the given set of initial assertions can vary significantly depending on the input specification. However, although it is likely that different specifications have individual writing styles for describing assertions, the style usually does not vary significantly within the same specification. Consequently, the likelihood of getting a favorable partition of low level assertions is high.

III. Dependency Representation

Our approach uses the grammatical structure of each sentence in order to partition the sentences into similar groups, and to extract key information from each sentence which is used to create SVAs. In order to represent the grammatical structure of each sentence we use the Stanford typed dependency representation [3] which is generated automatically by the Stanford Natural Language Parser [4]. A dependency

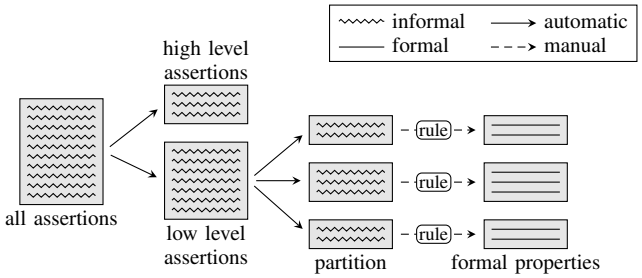


Fig. 1. Proposed flow

(a) Textual form (b) Graphical form

Fig. 2. Dependency representation

representation of a sentence is a set of binary ordered grammatical dependencies between pairs of words. The representation was designed to capture dependency relations which are semantically meaningful for a range of information extraction problems.

An example of a dependency representation is shown in Fig. 2(a) and Fig. 2(b) which show the representation of the sentence "X must be HIGH" in textual and graphical forms respectively. The sentence uses a *copular verb* "be" to relate the sentence *subject*, "X", to the complement of the verb, "HIGH". The sentence also contains an *auxiliary* "must" which modifies the verb. The three dependencies in Fig. 2(a) show all of these grammatical relationships. The first dependency *cop*(HIGH, be) is the copula relationship between the complement of a copular verb and the copular verb. The second dependency *nsubj*(HIGH, X) relates the complement of the copular verb with the subject of the sentence. The third dependency *aux*(HIGH, must) relates the complement to the auxiliary.

A *Typed Dependencies Graph* (TDG) G can be formally defined as a 4-tuple (V, E, r, s) where V is a set of vertices which represent the words in a sentence, and E is a set of directed edges which represent the dependencies between the words. Each edge $e = (g, d) \in E$ has a *governor* g which is its predecessor vertex, and a *dependent* d which is its successor vertex. Each edge $e \in E$ is also assigned a relation type $r(e)$ which is the type of dependency represented by the edge. Each vertex $v \in V$ is associated with a string $s(v)$ which is the word in the sentence represented by the set of vertices. The function tdg yields a TDG G for a sentence S and we write $G = \mathrm{tdg}(S)$.

One reason that we use the Stanford dependency representation is that the relations it contains are easy to understand and useful in implementing the partitioning and information extraction tasks which we perform. Sentences can be grouped into partitions based on the number of dependencies which

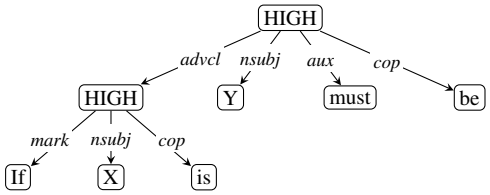


Fig. 3. Canonical dependency representation for two different sentences

their representations have in common, and key information can be extracted by examining the arguments of selected relations. For example, the sentence "X must be HIGH" should generate an assertion containing a comparison operation "==" between X on the left-hand side and 1 on the right-hand side. Sentences of this general form can be converted into SystemVerilog by using the second argument of the *nsubj* relation on the left-hand side of the assertion, and using the first argument of the *nsubj* relation to compute the right-hand side.

Another important reason for using the Stanford dependency representation is its canonicity in the presence of a large degree of linguistic variation. The techniques used to generate the dependency representations are robust and can identify dependencies even when the grammatical structures of the sentence are reordered. In other words, there can exist two sentences $S_1 \neq S_2$ such that $\mathrm{tdg}(S_1) = \mathrm{tdg}(S_2)$. An example can be seen in the representations of the two sentences $S_1 =$ "If X is HIGH, Y must be HIGH" and $S_2 =$ "Y must be HIGH if X is HIGH.". These two sentences have the same semantic meaning but the ordering of the phrases, "X is HIGH" and "Y must be HIGH" is different. The dependency representations of these two sentences is the same, independent of the ordering of their constituents, as illustrated in Fig. 3. The ability of the Stanford dependency representation to capture grammatical relations independent of ordering is essential to process a wide range of writing styles used by different possible authors. In order to further increase the robustness of our algorithms with respect to linguistic variation we apply preprocessing steps to both the sentences and resulting typed dependency graphs.

IV. Database Based Information Extraction

In order to implement our algorithm we additionally propose an information extraction technique that makes use of databases. The idea is that for a given sentence a database is extracted that contains a variety of linguistic information about the sentence. We make use of triple store databases which are represented as sets of 3-tuples, where each 3-tuple represents how two entities relate to each other. Information extraction is then performed by applying a query to the database which captures the type of information that should be extracted. The specified information can then be extracted from the query result.

Example 1: Given the three sentences "X must be HIGH", "Y must be LOW", and "X is not equal to Y", the task is to extract signal names and their required value from the first and second sentence. The third sentence does not match.

Natural language processing techniques, e.g. those provided by the Stanford Parser are used to construct the database.

Example 2: The database for the first sentence of Example 1 contains the following triples:

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

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

The task described in Example 1 could have easily been done with regular expressions as well, however, in this case it would not be as robust compared to our proposed solution. This is due to the canonicity of typed dependency representations as illustrated in the previous section. We chose the *SPARQL Protocol and RDF Query Language* (SPARQL, [5]) in order to query the generated databases. SPARQL queries consist of triples as they are found in the database but allow to use variables and additional constraints on these variables.

Example 3: In order to extract the signal/value pairs as described in Example 1, the following query is created and evaluated on each database that is generated for each sentence:

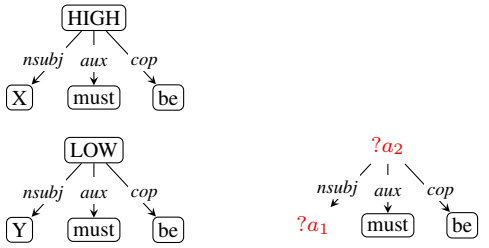
```
SELECT ?signal ?value WHERE {
  ?w4 aux ?w2. ?w4 cop ?w3. ?w4 nsubj ?w1.
  ?w2 word "must". ?w3 word "be".
  ?w1 word ?signal. ?w4 word ?value. }
```

Six variables are used in this query where only two of them are *global* (`?signal` and `?value`) appear in the result set which is obtained after evaluating the query. The other three variables are *locally* used and represent the corresponding word items in this case. An evaluation algorithm tries to determine values for the variables such that the triples can be found in the queried database. The variables are used to link the triples.

If a query matches, a result set is returned that consists of one or more assignments in which global variables are mapped to precise values. In our work we use this information to derive formal representations for natural language assertions.

V. Algorithm

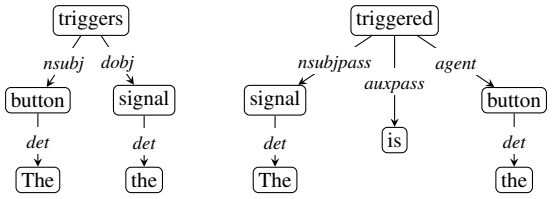
The three steps of the proposed algorithm are described in the next three subsections.



(a) Two typed dependency graphs

(b) Representative dependency graph

Fig. 4. Partitioning of dependency graphs



(a) Active voice

(b) Passive voice

Fig. 5. Writing active voice into passive voice

A. Abstraction Level Classification

A good heuristic to determine the abstraction level of natural language assertions heavily depends on the writing style of the specification. As a result, a general heuristic cannot be provided. Instead, we propose a classification method that only requires one SPARQL query as input by the designer. After a brief inspection of some assertions in the specification the designer can determine common characteristics of low level assertions, e.g. the use of some particular words or a special formatting. The observations are described in terms of a SPARQL query which is applied to each assertion. A knowledge of the structure of the database and typed-dependency graphs is necessary formalize such queries. An assertion is classified low level, if and only if the query matches the assertion. Furthermore, we require that an assertion contains only one sentence.

B. Partitioning based on Sentence Similarity

1) Representative Dependency Graph: In order to determine similar sentences we make use of a *Representative Dependency Graph* (RDG) which is a generalized description of a set of TDGs. The generalization is performed by allowing a subset of vertices to represent variables which can represent any word. An RDG is a TDG $G = (V, E, r, s)$ whose set of vertices V is partitioned into two set, W, which represents words in a sentence, and A, which are variables and may represent any string.

Example 4: Fig. 4(a) shows the TDGs of the sentences "X must be HIGH" and "Y must be LOW". Fig. 4(b) shows an RDG which describes the graphs of both sentences. The RDG contains two variables $?a_1, ?a_2 \in A$ which match the name and value, respectively.

That is, two sentences S_1 and S_2 are *similar* if they have the same TDG when disregarding the words that are associated to the vertices. In other words, given the sentences' TDGs $G_1 = (V_1, E_1, r_1, s_1) = \mathrm{tdg}(S_1)$ and $G_2 = (V_2, E_2, r_2, s_2) = \mathrm{tdg}(S_2)$, we have $G_1 \simeq G_2$, i.e. there exists a graph isomorphism $f : V_1 \to V_2$ where additionally the dependency relations need to be preserved, i.e., for each $e_1 = (u, v) \in E_1$ the property $r(e_1) = r(e_2)$ holds, where $e_2 = (f(u), f(v)) \in E_2$.

2) Pre- and Postprocessing: The partitioning step aims at getting an as small as possible number of clusters. We discovered that sometimes sentences are similar when reading them but not by means of the definition above. In order to still associate these sentences to the same subset we apply a preprocess on the sentence and a postprocess on the TDG.

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

Graph rewriting in the postprocess step allows for removing unimportant words or transforming structures that are semantically equivalent. In our implementation we applied different postprocessing rules. One rule removes auxiliary words, e.g. "the signal remains LOW" is treated equivalent to "the signal must remain LOW". Performing this step on the graph level is advantageous compared to sentence level because the word's dependencies can be taken into account. As an example, an auxiliary word is only removed when no other words depend on it which is equivalent to being a sink vertex in the TDG. A more complex postprocessing rule rewrites phrases in passive voice into active voice as illustrated in Fig. 5. After the auxiliary word in Fig. 5(b) has been removed it remains to relate the edges *nsubj* and *dobj* of the graph representing the active voice to the edges *agent* and *nsubjpass* of the graph representing the passive voice, respectively.

Given a sentence S in the following we combine the preprocessing step pre and postprocessing step post as $\mathrm{process}(S) = (\mathrm{pre} \circ \mathrm{tdg} \circ \mathrm{post})(S)$.

3) Partitioning and Construction of the RDG: For the partitioning and construction of the RDG we make use of a data structure

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

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

starting at the root nodes and then recursively compare their children. Since these graphs are typically small and also very heterogeneous, i.e. varying node labels and edge weights, the check for isomorphism is efficient.

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

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

```
SELECT ?a1, ..., ?aℓ WHERE {
  ?src(e) r(e) ?dest(e).   // for each e ∈ E.
  ?v word s(v).            // if ∃ w ∈ W : w = s(v)
  ?v word ?ak. }           // if ∄ w ∈ W : w = s(v)
```

where $(V, E, r, s) = \text{process}(S_1)$ and $W = \{w_1, \ldots, w_m\}$. That is, first the TDG structure is resembled, secondly all common words are asserted, and thirdly all variable words are related to the remaining vertices. Applying this SPARQL query to a triple store obtained from a sentence of the cluster directly returns the non-common variable words in the sentence.

C. Assertion Generation

Assertion generation occurs in two stages. First, an assertion template is generated for each cluster. This template contains variables in the positions where assertion specific information such as signal names, logic levels, or numerical constants would normally appear. If we recall the RDG from Fig. 4(b) we can see that a generalized sentence can be intuitively constructed from the information in the graph. This generalized sentence is representative of all sentences in a cluster. Using this generalized sentence a designer or verification engineer can manually design an appropriate SystemVerilog Assertion template.

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

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

VI. Experimental Evaluation

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

We applied the algorithm to the *AMBA 3 AXI Protocol Checker* [6] user guide that consists of 145 natural language assertions for the *AMBA AXI 3 Protocol* [7]. The next subsection describes implementation decisions and the main results of the evaluation.

Example 5: To illustrate the experimental evaluation we make use of the following four example sentences from the AMBA specification:

- S1 AWID must remain stable when AWVALID is asserted and AWREADY is LOW.
- S2 A write transaction with burst type WRAP has an aligned address.
- S3 AWVALID is LOW for the first cycle after ARESETn goes HIGH.
- S4 BRESP remains stable when BVALID is asserted and BREADY is LOW.

1) Abstraction Level Classification: We have prepared the data for the experimental evaluation by first classifying all assertions manually. These expected values were then compared to the result of the classifier from which we computed the accuracy $A = \frac{\text{TP+TN}}{\text{TP+TN+FP+FN}}$, the recall $R = \frac{\text{TP}}{\text{TP+FN}}$, the precision $P = \frac{\text{TP}}{\text{TP+FP}}$ and the F-measure $F = \frac{2 \cdot P \cdot R}{P+R}$, where TP, FP, TN, and FN refer to the number of *true positives*, *false positives*, *true negatives*, and *false negatives*, respectively.

We have discovered that most of the low level assertions contain one of the signal names that were all listed in a table in the specification. Further we discovered that many local parameters are constrained. In Example 5, S2 is a high level assertion, whereas all other ones are low level. We have extended the triple store generation algorithm by storing whether a word is a signal name in the database. For this purpose, the predicate `isSignalName` has been used that relates each word item to a truth value. The SPARQL query provided to the classifier algorithm then checks whether a signal name is contained or whether the word "parameter" occurs. It reads:

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

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

2) Partitioning based on Sentence Similarity: For the partitioning we have implemented all pre- and postprocessing rules as described in Sect. V-B. This led to a partition of 11 clusters

TABLE I
EVALUATION OF THE CLASSIFIER

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

for the 100 assertions that have been asserted low level in the previous step, i.e. each cluster contains of approximately 9 sentences on average. (Some more details about the clusters are listed in the appendix.) In Example 5 the sentences S1 and S4 belong to the same cluster which becomes evident after the auxiliary words have been removed in the post processing step. Sentence S3 belongs to a different cluster. In order to understand the effect of the classification of the previous step to the clustering we have also computed the partition based on all 145 assertions. This lead to 50 clusters and therefore about 3 sentences per cluster on average.

3) Assertion Generation: After partitioning the 100 NLAs into 11 distinct clusters a set of 11 corresponding SystemVerilog assertion templates were generated. These assertion templates are presented in Table II with the variables highlighted in red. For each cluster mapping functions were also generated. Two types of mapping functions were used. The "signal name" mapping function simply passes the input value to the output. The other mapping function used was the "logic level" mapping function. This function maps an abstract logic level such as *HIGH* or *LOW* to a logical 1 or 0 respectively. While these mapping functions might at first seem superfluous, they allow our technique much greater flexibility. By utilizing these mapping functions our method can not only handle *HIGH*, and *LOW* but also words such as *asserted*, *deasserted* and their synonyms.

4) Assertion Translation Walkthrough: In this section we will walk through an end-to-end example of our algorithm as applied to a portion of a real data set. Recall Example 5 where four natural language assertions from [6] are presented. Upon application of step 1 of the algorithm the sentences are separated into high and low abstraction sentences. The low abstraction sentences *S1*, *S3*, and *S4* are supplied to the next stage of the algorithm while the high abstraction sentence *S2* is discarded from the translation set.

In step 2 the sentences are partitioned into clusters and an RDG for each cluster is generated. The three remaining sentences in our example are partitioned into two clusters based on sentence similarity as determined by the RDG. Sentences *S1* and *S4* are in one cluster while sentence *S3* is in a second cluster. In step 3 we will look at the translation of sentences *S1* and *S4* while remembering that the steps outlined will be performed for each cluster in turn.

Example 6: SVA Translations T1 and T4 which correspond to sentences S1 and S4:

```
T1 assert property (@(posedge clock)
   ((AWVALID == 1) && (AWREADY == 0))
   |->
   $stable(AWID));

T4 assert property (@(posedge clock)
   ((BVALID == 1) && (BREADY == 0)) |->
   $stable(BRESP));
```

We have now ascertained that sentences *S1* and *S4* are in the same cluster and share an RDG, which was generated in the previous step. We now manually construct the representative sentence "a_1 remain stable when a_2 is a_3 and a_4 is a_5" for our cluster of interest. We also construct an appropriate assertion template (see entry 8 of Table II). In this final stage of the algorithm we read the variables for sentences *S1* and *S4* from their individual TDGs. The variables are applied to the mapping functions and combined with the assertion template resulting in the two fully realized SVAs shown in Example 6.

VII. THREATS TO VALIDITY

What happens if an assertion is classified wrongly? There are two cases in which an assertion has been wrongly classified. If it has been classified high level although it is low level, it needs to be manually classified after performing our proposed flow together with the other high level assertions. If it has been classified low level although it is high level, it will likely end up as a single sentence in a cluster. Writing a translation rule for a cluster with only one sentence is equal to translating a high level assertion since no common words can be extracted. Hence, a wrongly classified assertion cannot cause any harm.

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

What if the number of sentences per cluster is too small? If in the worst case each cluster contains only one sentence, the proposed translation flow again performs equally compared to the conventional one. In order to obtain larger clusters one can inspect the assertions and add new pre- and postprocessing rules to the partitioning step. However, one needs to assure that sentences in same clusters still have the same meaning and can be translated using a common translation pattern.

VIII. RELATED WORK

Related work can be divided into two groups, research in assertion generation, and research in the use of natural language processing for hardware design.

A. Assertion Generation

Approaches have been presented to automatically extract assertions based on simulation traces using machine learning techniques [8], [9]. The assertion is presented to the user (design/verification engineer) to determine whether the assertion is a good candidate to make it into the design or whether the

TABLE II
ASSERTION TEMPLATES

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

test should include additional cases to put the design in that state via a new scenario case. The main challenge is in the difficulty in learning all the sequences over time.

Reference [10] presents a methodology that uses the failing assertion, counterexample, and mutation model to produce alternative properties that are verified against the design and serve to make possible corrections as they provide insight into the design behavior and the failing assertion. The results show that this process is effective in finding high quality alternative assertions for empirical instances. However, the process is not yet automated.

B. Natural Language Processing for Hardware

Researchers have generated partial hardware designs from natural language specifications [11], [12] by identifying a set of concepts expressed, together with a textual pattern for each concept. Any sentence which matches a textual pattern can be mapped to a structures in a design data structure defined by the authors. The approach taken in [13] defines a grammar to parse natural language expressions, and generates VHDL snippets. In [14] an approach is presented that translates English specification sentences into temporal logic formulas for the SMV model checker. Lightweight formal methods for the validation of natural language requirements have been used in [15] which allows consideration of partial specifications and partial properties and therefore allows for a higher scalability. The work in [14] and [15] focuses on the automatic translation of English text to formal representations using natural language processing techniques. However, in our work natural language processing techniques are used to cluster requirements and the actual translation remains a manual task.

More recently researchers have improved on the sophistication of NLP based analysis by relying on the semi-formal structure of test scenarios described by acceptance tests [16]. A UML class diagram is generated based on the entities referred to in the scenario, and a UML sequence diagram is generated from the sequence of operations described. Assertions have also been generated from natural language using a special-purpose attribute grammar [17]. These previous works directly address the hardware verification problem by producing usable verification artifacts, namely UML sequence diagrams and assertions. The work which we present in this paper is distinguished from previous work [16], [17] in that our use of clustering together with assertion templates exploits the regularity in writing style commonly found in natural language specification.

IX. Conclusions

We have presented an algorithm that automates the translation of natural language assertions into SystemVerilog Assertions using natural language processing techniques. Instead of manually translating each assertion separately, our approach first splits assertions based on their abstraction level and then partitions low level assertions into clusters based on sentence similarity. Since typically a common writing style is being used in one specification, the numbers in these clusters are large, i.e. multiple assertions can be translated using the same template, which significantly decreases the verification effort. In order to implement our approach we have additionally proposed an information extraction algorithm based on databases. In an evaluation we have applied our approach to modern bus specifications.

In future work we want to consider how the automatically generated RDGs can be extended to be more generic. This is particularly useful when reusing the translation templates for other projects. As a result, once an assertion template has manually been generated for a set of sentences in one specification it can be reapplied in future specifications which additionally decreases the verification effort.

Acknowledgment

This work was supported by the German Federal Ministry of Education and Research (BMBF) (01IW13001) within the project SPECifIC and by the German Research Foundation (DFG) (DR 287/23-1) within a *Reinhart-Koselleck* project.

References

[1] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, "FoCs: Automatic generation of simulation checkers from formal specifications," in *CAV*, 2000, pp. 538–542.
[2] L. Kof, "Natural language processing: Mature enough for requirements documents analysis?" in *NLDB*, 2005, pp. 91–102.
[3] M.-C. de Marneffe and C. D. Manning, "The stanford typed dependencies representation," in *CrossParser*, 2008, pp. 1–8.
[4] M.-C. de Marneffe, B. MacCartney, and C. D. Manning, "Generating typed dependency parses from phrase structure parses," in *LREC*, 2006, pp. 449–454.
[5] *SPARQL 1.1 Query Language*, W3C, Mar. 2013.
[6] *AMBA 3 AXI Protocol Checker User Guide*, r0p1 ed., ARM, Jun. 2009.
[7] *AMBA AXI and ACE Protocol Specification*, ARM, Oct. 2011.

[8] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "GoldMine: Automatic assertion generation using data mining and static analysis," in *DATE*, 2010, pp. 626–629.
[9] P.-H. Chang and L.-C. Wang, "Automatic assertion extraction via sequential data mining of simulation traces," in *ASP-DAC*, 2010, pp. 607–612.
[10] B. Keng, S. Safarpour, and A. Veneris, "Automated debugging of SystemVerilog assertions," in *DATE*, 2011, pp. 323–328.
[11] J. J. Granacki and A. C. Parker, "PHRAN-SPAN: a natural language interface for system specifications," in *DAC*, 1987, pp. 416–422.
[12] J. J. Granacki, A. C. Parker, and Y. Arena, "Understanding system specifications written in natural language," in *IJCAI*, 1987, pp. 688–691.
[13] W. R. Cyre, J. Armstrong, M. Manek-Honcharik, and A. J. Honcharik, "Generating VHDL models from natural language descriptions," in *EURO-DAC*, 1994, pp. 474–479.
[14] A. Holt, "Formal verification with natural language specifications: guidelines, experiments and lessons so far," *South African Computer Journal*, vol. 24, pp. 253–257, 1999.
[15] V. Gervasi and B. Nuseibeh, "Lightweight validation of natural language requirements," *Software – Practice and Experience*, vol. 32, no. 2, pp. 113–133, 2002.
[16] M. Soeken, R. Wille, and R. Drechsler, "Assisted behavior driven development using natural language processing," in *TOOLS*, 2012, pp. 269–287.
[17] I. G. Harris, "Capturing assertions from natural language descriptions," in *NaturaLiSE*, 2013, pp. 17–24.

APPENDIX

A. Representatives for Low-level Clusters

The following list shows representatives for each of the 11 clusters that has been determined in the experimental evaluation. Variable words are marked in bold font. The numbers after the sentence refer to the number of sentences in the corresponding cluster.

1) Parameter **AWUSER_WIDTH** must be greater than or equal to 1. *[10]*
2) A value of **X** on **RDATA** valid byte lanes is not permitted when **RVALID** is **HIGH**. *[1]*
3) When **AWVALID** is asserted then it remains asserted until **AWREADY** is **HIGH**. *[5]*
4) A value of **X** on **AWVALID** is not permitted when not in **reset**. *[13]*
5) A value of **X** on **AWID** is not permitted when **AWVALID** is **HIGH**. *[29]*
6) **AWVALID** is **LOW** for the first cycle after **ARESETn** goes **HIGH**. *[5]*
7) When **AWVALID** is **HIGH** and **AWCACHE[1]** is **LOW** then **AWCACHE[3:2]** are also **LOW**. *[1]*
8) **AWID** must remain stable when **AWVALID** is asserted and **AWREADY** is **LOW**. *[28]*
9) When **ARVALID** is **HIGH**, if **ARCACHE[1]** is **LOW** then **ARCACHE[3:2]** must also be **LOW**. *[1]*
10) Recommended that **AWREADY** is asserted within **MAXWAITS** cycles of **AWVALID** being asserted. *[5]*
11) **CSYSREQ** is only permitted to change from **HIGH** to **LOW** when **CSYSACK** is **HIGH**. *[4]*