Chapter 3
Generation of Verification Artifacts from Natural Language Descriptions

Ian G. Harris and Christopher B. Harris

3.1 Introduction

The integrated circuit (IC) design process has evolved greatly, from the manual layout of a small number of components to the automated design of ICs containing billions of transistors. To accommodate the dramatic increases in design complexity, the field of electronic design automation (EDA) was born, starting with simple schematic capture tools and culminating in the complex automation tools available today. EDA tools depend on the existence of a well-defined behavioral model, or model of computation, which can be used by EDA tools to perform synthesis and verification tasks. Over time, the abstraction level of the behavioral models in use has risen to efficiently capture more complex behaviors. EDA tools have proven effective in supporting synthesis and verification tasks, but the initial behavioral model must be generated manually by human experts. The process of manually creating an accurate and complete behavioral description has always been a central bottleneck in the design process which EDA tools seek to alleviate. Manually generating a behavioral description is expensive, requiring significant time and a large number of well-trained design and verification engineers. A large part of the verification process is devoted to detecting and fixing design errors created during the processes of creating a behavioral description.

Developing a natural language specification of design behavior is a well-accepted precondition for generating a formal behavioral model. Natural language specifications are the first concrete behavioral description which is the basis for the

I. G. Harris (✉)
Department of Computer Engineering, University of California Irvine, Irvine, CA, USA
e-mail: harris@ics.uci.edu

C. B. Harris
Department of Electrical and Computer Engineering, Auburn University, Auburn, Alabama, USA
e-mail: cbharris@auburn.edu

M. Soeken, R. Drechsler (eds.), *Natural Language Processing for Electronic Design Automation*, https://doi.org/10.1007/978-3-030-52273-5_3

manually generated formal behavioral model. Natural language is preferred as the initial description method mainly because it is much simpler for a designer to use than existing hardware description languages. Natural language specifications also have the advantage that they can be used to communicate behavioral information with non-technical stakeholders, such as a client for whom the design is being made, or a high-level manager. Design and verification engineers use the specification as the main source of behavioral information, to generate a formal behavioral description and to identify corner cases and expected responses for verification. The task of interpreting natural language specifications has been exclusively manual because, generally speaking, only humans with expert design knowledge have the ability to properly interpret specification documents.

3.1.1 Verification Artifacts

Simulation-based verification involves applying test vectors to a system under test and verifying the correctness of the responses. An enormous number of test results must be evaluated when verifying complex systems, so the task of evaluating test results must be automated. The evaluation of test responses is typically automated using one of two approaches.

- **Assertion-Based Verification** An assertion is a program invariant which is evaluated automatically during hardware simulation to perform response evaluation. Assertion-based verification is a well-used hardware verification approach [14].
- **Transaction-Level Modeling** Transaction-level modeling is a high-level behavioral modeling approach which separated computation from communication [6]. The high level of abstraction enables simulation models to be created more efficiently than models at lower abstraction levels such as register-transfer level.

Both result evaluation approaches require a tedious and error-prone manual step to generate assertions or generate simulatable transaction models. The task of formally specifying the behavior in terms of assertions or transaction models can be nearly as difficult as the design task itself.

3.1.2 Verification from Natural Language

We present approaches to generate assertions and transactor models directly from natural language descriptions of the system behavior. The benefit of our work is to simplify the verification process by reducing the amount of manual effort required, as well as the time required to debug the assertion framework itself.

It is common for a specification to contain sentences which express constraints on the legal behavior of a system. We present an approach to generate SystemVerilog assertions directly from constraint sentences such as these. An example of the

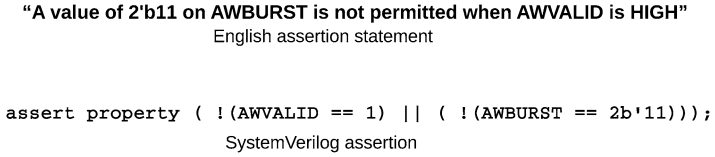

Fig. 3.1 Generation of an assertion

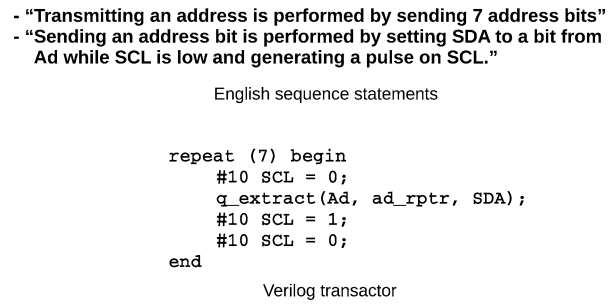

Fig. 3.2 Generation of a transactor

goal of this project is seen in Fig. 3.1 which shows an English assertion statement together with the equivalent SystemVerilog assertion which our approach generates.

Specifications often describe event sequences which implement features of the behavior. These event sequences are often named (i.e., "write transaction") and may be expressed over multiple sentences. Formally capturing event sequences is essential in order to create a transaction model which will provide a "golden model" of the transaction for response checking. Capturing sequences has the additional complexity that information contained in multiple related sentences must be combined to create a complete model. An example of transaction generation is shown in Fig. 3.2 which shows two English sentences which are part of a larger sequence description and the corresponding portion of the transactor written in Verilog.

3.1.3 Chapter Organization

The remainder of the chapter is organized as follows. Related work in the application of natural language processing is presented in Sect. 3.2. Section 3.3 outlines some of the key issues that must be considered when generating a formal description from natural language. Our use of semantic parsing to address issues related to linguistic variation is described in Sect. 3.4. Sections 3.5 and 3.6 describe our work in assertion generation and transactor generation, respectively. Section 3.7 presents conclusions from our results and suggests future work.

3.2 Related Work

Tasks related to the processing of natural language documents traditionally fall into the NLP field [26]. NLP includes *information extraction* as well as many related subproblems such as part-of-speech tagging and syntactic parsing. An information extraction approach which we will leverage for understanding hardware specifications is semantic parsing which maps natural language sentences to formal meaning representations using syntactic parsing [37]. Existing semantic parsing approaches can be categorized based on the restrictions on the text which they process and the meaning representation used. In all previous work, the structure of the text is restricted in some way. Restrictions may be very strict, such as insisting on a *subject, verb, direct object, and indirect object* pattern in each sentence [9, 34] or restricting verb tense and the number of independent clauses in a sentence [29]. Several semantic parsers accept spoken dialog but restrict the conversation domain [4, 24]. Research in processing software requirements frequently accepts semi-structured use case textual descriptions [9, 29, 34, 42]. Semantic parsing systems are domain-specific, so the meaning representation must be appropriate to contain information in the chosen domain. Several earlier semantic parsers relied on semantic frames [36], which are similar to structures which contain a set of fields to describe the attributes of each structure instantiation[4, 24]. Message sequence charts are often used as meaning representations for systems which process software specifications [19, 29, 42]. Other meaning representations used include hidden understanding models [35] and abstract state machines [30].

An alternative approach for information extraction is the use of machine learning techniques in the form of text mining [12, 22]. Machine learning approaches use statistical techniques to extract association rules involving words groups. Machine learning has several applications in NLP including document retrieval, text summarization [47], and semantic role labeling [18, 44].

3.2.1 NLP for Hardware and Software Design

Natural language processing has been applied to several different hardware design problems in the past [10]. Researchers have developed a natural language interface to search through circuit simulation results [40, 41]. The simulation process produces a results file containing a set of triples of the form $(signal, voltage, time)$, and natural language queries are used to specify constraints on each parameter.

Researchers have generated partial hardware designs from natural language specifications [20, 21] 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 [7] defines a grammar to parse natural language expressions and generates VHDL snippets. More recent efforts have improved on the sophistication

of the analysis by relying on the semi-formal structure of test scenarios described by acceptance tests [43]. 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.

NLP has been applied in the software engineering field to support various problems related to program comprehension. Information has been extracted from software artifacts including source code and code comments [15, 45, 48] and development emails [2]. Program comprehension tasks which have been addressed include concern location, aspect mining [15], traceability, artifact summarization [2], program rule extraction, and code search [45, 46]. The work presented in [45] on program rule extraction is most related to our work since program rules are essentially the same as assertions. The work in [45, 48] extracts information from comments, while our work extracts information from hardware specifications. *iComment* uses "rule templates" to match pattern in sentences, while our approach uses an attribute grammar approach.

3.3 Issues in Formalizing Natural Language for Hardware

When developing approaches to formalize natural language descriptions of hardware, there are several overarching concerns which fundamentally impact the process and must be considered at the outset. We describe two of the most important concerns in order to motivate and justify our approaches.

3.3.1 Computational Models

The information extracted from a specification document must be represented using a formal, unambiguous **computational model**. Several types of computational models may be used since the most efficient representation for each type of information may be different. We broadly classify the types of information into three classes, *structural* information which describes the physical objects referred to in the text, *behavioral* information which defines constraints on events occurring on objects in the ontology, and *behavioral constraints* which express general limits on legal behavior.

3.3.1.1 Structural Representation

In a hardware specification, the most basic objects which will be part of the structural description are wires, state elements, and hierarchical combinations of the two. Hardware specifications will also refer to structural blocks in the design which have associated behaviors. Events occurring on wires and state elements

define the behavior of the system, and the purpose of a specification is to define allowable relationships between these events. It is possible to define a specification by referring to only the input and output wires, without explicitly defining any internal state elements. However, it is common for hardware specifications to reference some key state elements, and sometimes internal wires, which are assumed to exist in the design.

The most appropriate method to represent a structure, and the most commonly used method in previous work in software requirements analysis [28, 31], is the use of a class hierarchy. Each type of object defined in the structure can be represented as a class with attributes which indicate the basic wires and other classes which are aggregated together. A structural example can be seen in the following sentence from the IEEE 1500 standard specification document [23], "The WPP terminals consist of the wrapper parallel input (WPI) terminal(s), wrapper parallel output (WPO) terminal(s), and wrapper parallel control (WPC) terminals." This sentence defines a class called **WPP** with the attributes **WPO**, **WPI**, and **WPC**. Both WPO and WPI are simple wires, but WPC is a grouping of wires defined elsewhere in the specification document, so WPC would be defined as another class.

3.3.1.2 Behavior Representation

There are many well-accepted formal models for hardware behavior [16, 25, 32, 33]. *Process-based* models represent behavior as a set of concurrent processes which are internally described in an imperative form using a sequential programming model. Each concurrent process can be represented by a control-dataflow graph model. *State-based* models describe system behavior by defining a set of states and transitions between states.

The type of model chosen to capture meaning in a natural language specification depends on the style in which the behavior is described. We demonstrate the relationship between the writing style and the behavioral model used by comparing the following two descriptions of a generic data transfer operation.

Description 1: A data transfer is initiated when the sender asserts the REQ signal. The sender then waits until the ACK signal is asserted before transmitting the data byte.

Description 2: When in the READY state, the sender asserts the REQ signal and transitions to the WAIT state. The sender remains in the WAIT state until the ACK signal is asserted, which causes it to enter the SEND state. In the SEND state, the sender transmits the data byte.

Both of the specifications above describe the same behavior in different styles. The first description clearly outlines a sequence of events and could most naturally be captured with a process-based model as shown in Fig. 3.3a, using a single process and a sequential program to implement the behavior. The second description is explicitly written in terms of states and transitions, so it could be easily represented using a state-based model as shown in Fig. 3.3b.

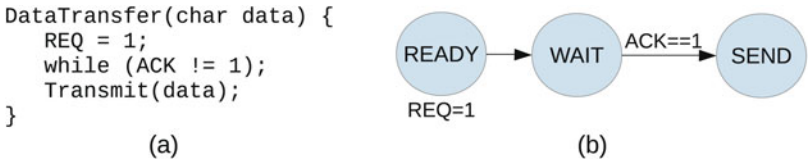

Fig. 3.3 Models of behavior, (**a**) process-based, (**b**) state-based

Representing the meaning of natural language specifications requires the features of both process-based and state-based models, so the best model would incorporate both. There are several appropriate models to choose from including SpecCharts [38] and SpecC [17].

3.3.1.3 Behavioral Constraints

In addition to explicit descriptions of individual features, hardware specifications also contain *behavioral constraints* which limit particular aspects of behavior more generally. The following statement taken from the I2C bus protocol specification document [39] is an example, "The data on the SDA line must be stable during the HIGH period of the clock." This statement does not describe a sequence of events associated with a particular behavioral feature. Instead, it is a general constraint on almost all explicit behaviors, including read and write transactions. As a result, the statement is not easily represented using either a process-based or state-based representation. Behavioral constraints are a common part of hardware specifications and are often used to express properties for model checking and assertions for checking during simulation.

To accommodate the representation of behavioral constraints, it is convenient to use models which are already accepted for specifying properties and assertions. Different forms of first-order predicate calculus, such as CTL and LTL, are effective for specifying boolean and temporal relationships between events. Constraint satisfaction programming (CSP) [8] can express arithmetic constraints as well. Efficient solvers exist for CTL properties and CSP formulations. Hardware verification languages, such as SystemVerilog and E, allow assertions to be expressed with additional flexibility including hierarchy and timing precision. The complexity of assertions makes them unsuitable for automatic theorem proving, but they are useful for result checking during simulation.

3.3.2 Linguistic Variation

Linguistic variation describes the aspect of a language which enables a single concept to be expressed in multiple ways in the language.

Morphological – Morphological variation is employed when two homonyms, words with the same meaning, are used in two different sentences.

```
I hate cats.
I detest felines.
```

These two sentences display morphological variation because they have the same meaning but vary only in the choice of homonym used in each position in the sentence. The words "hate" and "cats" in the first sentence are replaced with their homonyms "detest" and "felines," respectively.

Syntactic – Syntactic variation describes the use of different sentence structures to express a single concept.

```
Joe hates cats.
Cats are hated by Joe.
```

The two sentences use largely the same words but the word order is changed from an active voice in the first sentence to a passive voice in the second sentence.

Pragmatic – Pragmatic variation describes two sentences with different literal meanings, but the same connotation.

```
Your breath stinks.
You might want to try using this toothbrush.
```

These two sentences have different literal meanings, but they both convey the same meaning to most listeners: your breath stinks. The first sentence is direct, while the second is suggestive, allowing the listener to infer the true meaning.

The chief problem associated with linguistic variation is to ensure that the computational models generated from two semantically equivalent sentences are themselves equivalent, independent of any linguistic variation present.

3.3.2.1 Linguistic Variation in Hardware Descriptions

Morphological Variation Morphological variation does occur in hardware descriptions. For example, the verbs "set" and "assign" are often used interchangeably. This type of variation can be modeled in a straightforward way using a thesaurus to identify words with the same meaning. Each word is associated with some object in the computational model, and a thesaurus enables all words with the same meaning to be associated with the same model. The modeling of morphological variation is shown in Fig. 3.4 which shows possible models generated from the sentences "I hate cats" and "I hate felines."

The entity-relationship diagram (ERD) in Fig. 3.4a relates the words "cat" and "feline" to a single object "cat_type" using the **IS-A** relation. The logic expression in Fig. 3.4b contains a predicate "cat_type" which is used to describe both cats and felines. By using a thesaurus to define the scope of the nodes in the ERD of Fig. 3.4a and the domain of the "cat_like" predicate in Fig. 3.4b,

Fig. 3.4 Modeling morphological variation, (**a**) entity-relationship diagram, (**b**) predicate calculus

we can ensure that the two sentences with the same meaning have identical computational models.

Syntactic Variation English grammar has a rich syntax providing for many ways to express a single idea. This is true in the domain of hardware specifications, for example, the sentences "Assign X to one" and "Signal X is asserted," which both describe the same signal assignment. In order to generate equivalent computational models from sentences exhibiting syntactic variation, the meaning of the sentence must be extracted in a way which is robust in the presence of different word orderings.

Central to the meaning of any sentence is the verb (or verbs) which it contains. Each declarative sentence contains *actions* or *states* involving one or more participants, and the verbs in the sentence describe the action/state. Declarative sentences describe events, as in "Signal X is asserted," and states, such as "Signal Y is low." In order to use information in a sentence, it is essential to detect the event or state, as well as the participants in the event or state. The verb (or verbs) is a sentence describing the type of action or state. Each participant is said to have a *semantic role* in the sentence, with respect to the action as usually described by the verb. A simple example can be seen in the sentence, "Joe loves cats" where loves describes the action, "Joe" has the role of the *loving thing*, and "cats" has the role of the *loved thing*. Each sentence is assumed to match a *semantic frame* [13] which is a template to describe the action of a sentence and the semantic roles involved in the action. The sentence "Ian loves cats" can be described with a semantic frame which describes the act of loving and contains two participants, the *loving thing* and the *loved thing*.

The use of semantic frames to represent information in a way which is syntactically neutral is a well-accepted approach in artificial intelligence research. We perform parsing to fill a semantic frame for each English sentence, allowing the identification of key words which fill each semantic role.

Pragmatic Variation – Pragmatic variation exists when context indicates that the intended meaning of a sentence is different from the apparent meaning. For example, if two people are at a store buying a toothbrush and one person says, "You might want to try using this toothbrush," then the meaning should be taken literally. However, if the same sentence is spoken by a person who is forced to be in close physical proximity with another person, then the meaning of the sentence is likely to be an insult about the breath of the listener.

Although pragmatic variation can occur in English, it would never be expected in a hardware description. We expect that a hardware description is always stated in a direct manner for a single purpose, specifying system behavior.

3.4 Semantic Parsing

One technique that we will use to extract important elements from a sentence is *semantic parsing* which uses a syntactic parser to process each sentence according to the rules defined by a context-free grammar (CFG). Each sentence is associated with one or more semantic frames, and the key words in the sentence which perform each semantic role defined by the frame are easily located in the resulting parse tree. Although natural languages are not context-free languages, the use of syntactic parsers is well-accepted in the NLP community due to the existence of efficient parsing algorithms. A CFG is defined to capture the English subset of interest, and the parser generates a parse tree representation of the sentence in which each node represents a production used in the parse. An example of a syntactic parse is shown using the example sentence "Set P to one" and the CFG shown in Fig. 3.5. The symbols used in the productions of Fig. 3.5 describe standard constituents in English grammar which include sentence (S), verb phrase (VP), and noun phrase (NP). The parse tree resulting from using this grammar is shown in Fig. 3.6a.

Semantic parsing uses a CFG containing symbols which are associated with well-defined semantic interpretations. To demonstrate the semantic parsing process, Fig. 3.6b shows the parse tree generated for the example sentence using the grammar

Fig. 3.5 Simple context-free grammar for English

$$
\begin{aligned}
S &\rightarrow VP \\
VP &\rightarrow VB\ \ NN\ \ PP \\
VB &\rightarrow \text{“set”} \\
PP &\rightarrow IN\ \ NN \\
IN &\rightarrow \text{“to”} \\
NN &\rightarrow \text{“P”} \\
NN &\rightarrow \text{“one”}
\end{aligned}
$$

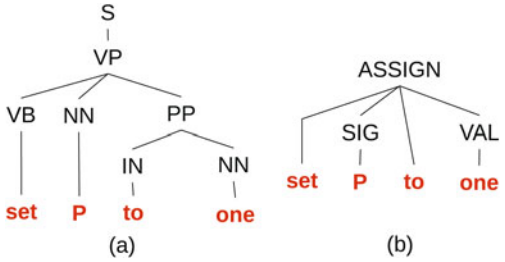

Fig. 3.6 Parse trees generated from (**a**) an English syntactic grammar, (**b**) a semantic grammar

Fig. 3.7 Semantic grammar

$$ASGN \rightarrow \text{“set”}\ SIG\ \text{“to”}\ VAL$$
$$SIG \rightarrow \text{“P”}$$
$$VAL \rightarrow \text{“one”}$$

in Fig. 3.7. The semantic grammar includes the SIG symbol indicating a signal name, the VAL symbol indicating a signal value, and the ASGN signal indicating a signal assignment.

The key observation of semantic parsing is that it performs semantic role labeling by highlighting relevant domain-specific information in the parse tree. For example, in order to find the name of the signal being assigned, the parse tree can be searched for the SIG symbol, and the signal name is its child. This signal assignment can now be represented using a process-based representation as an assignment statement of the form, *SIG = VAL;*, where the SIG role is the signal being assigned and the VAL role is the value to which it is assigned.

We have developed semantic grammars to perform semantic role labeling for all behavioral concepts which we consider. We accommodate the wide range of linguistic variation by extending the grammar to capture all variations which are common in hardware descriptions.

3.5 Generating Assertions

We present an approach to generate SystemVerilog assertions from behavioral constraints expressed in English. This work depends only on the English text and does not require the existence of a simulatable model of the design. Our approach is based on the use of an *attribute grammar* to define the formal semantics of a subset of assertion descriptions in English. Attribute grammars were originally developed by Knuth [27] as a simple yet powerful formalism for expressing the semantics of programming languages. We define an attribute grammar which associates key natural language structures with semantically equivalent SystemVerilog code. We use a parser to recognize important natural language structures in assertion descriptions. The evaluation rules which are part of the attribute grammar are used to generate SystemVerilog code for each structure.

3.5.1 Attribute Grammars

An attribute grammar is a context-free grammar enhanced with *attribute values* and *evaluation rules* to compute the attribute values as a function of the attributes of adjacent nodes in the parse tree. They were originally developed by Knuth [27] for expressing the semantics of programming languages. We define an attribute

Fig. 3.8 Attribute grammar

$$
\begin{aligned}
ASGN \rightarrow & \text{“set”}\ SIG\ \text{“to”}\ VAL \\
& [ASGN.v = SIG.v + \text{“=”} + VAL.v + \text{“;”}] \\
SIG \rightarrow & \text{“P”} \\
& [SIG.v = \text{“P”}] \\
VAL \rightarrow & \text{“one”} \\
& [VAL.v = \text{“1”}]
\end{aligned}
$$

Fig. 3.9 Extension to an attribute grammar

$$
\begin{aligned}
ASGN \rightarrow & SIG\ \text{“must be assigned to”}\ VAL \\
& [ASGN.v = SIG.v + \text{“=”} + VAL.v + \text{“;”}]
\end{aligned}
$$

grammar to capture the semantics of English assertions as the attribute values of the symbols. The attribute value of each symbol of our grammar is a string of SystemVerilog assertion code.

Attribute values can be passed from a node to its parent using a synthesized attribute or from a node to its child using an inherited attribute. The grammar that we define in this paper is said to be *S-attributed* because it only uses synthesized attributes. Each production is associated with an evaluation function which computes the attribute value of the symbol on the left-hand-side of the production. The attribute value of the root node of the parse tree is the semantic meaning of the parsed string. In our notation, each production is followed by its attribute evaluation rule in square brackets. We also use the “+” symbol to represent the string concatenation operation.

As a demonstrative example, we modify the semantic grammar in Fig. 3.7 to produce an attribute grammar shown in Fig. 3.8. The attribute values in this grammar are equivalent strings of C code. When the attribute grammar is used to parse the sentence, “Set P to one,” the attribute value of the ASGN symbol is “P = 1;” which is the C code equivalent to the sentence.

An important property of attribute grammars is the fact that they can be easily extended to consider linguistic variation. As an example, consider an English statement which declares that signal P should be assigned to the value 1. The grammar shown in Fig. 3.8 will parse the sentence “Set P to one,” but it will not parse, “P must be assigned to one” which uses the passive voice. However, by including the production shown in Fig. 3.9 the sentence written in the passive voice can be parsed as well. The attribute grammar can be constructed in a general way to accommodate all of the linguistic variation required.

3.5.2 System Overview

Figure 3.10 shows the structure of our system and the flow of data between its components. The system starts with the *English Assertion* at the top left. The

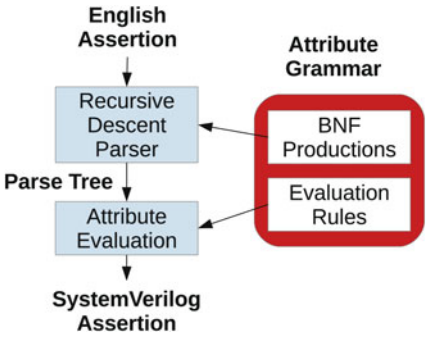

Fig. 3.10 System structure

first processing step uses the *Recursive Descent Parser* to generate a *Parse Tree*. The second processing step performs *Attribute Evaluation* to generate the *SystemVerilog Assertion*, which is the attribute of the root node of the parse tree. Both processing steps use the *Attribute Grammar* which we present in this paper. The recursive descent parser uses Backus-Naur Form (BNF) productions of the grammar to perform parsing. The attribute grammar includes evaluation rules, associated with each production, which are evaluated to generate semantically equivalent SystemVerilog assertions.

The recursive descent parser which we use is an off-the-shelf component taken from the open source Natural Language Toolkit [3]. Attribute evaluation is well understood, and we implement an existing technique using a left-to-right depth-first traversal of the parse tree [11].

3.5.3 Attribute Grammar

We present an attribute grammar which parses a class of assertions descriptions written in English and produces SystemVerilog assertions which are semantically equivalent to the English descriptions. The following subsections present the productions of the grammar grouped based on the hardware description concepts which the productions recognize. Linguistic variation in English ensures that there are several ways to express each concept. In each subsection we describe a set of ways in which each concept is described in English, and we describe the features of our grammar which captures each method of expression.

Productions in the grammar are associated with an attribute value, labeled with the suffix *.sv*, which is a string of equivalent SystemVerilog code. The attribute value of the root node of each parse tree is the SystemVerilog assertion which is equivalent to the parsed sentence.

Fig. 3.11 Productions for constants

$$
\begin{aligned}
CST &\rightarrow \text{“0”} \\
&\quad [CST.sv = \text{“0”}] \\
CTR &\rightarrow CST \\
&\quad [CTR.sv = CST.sv] \\
CTR &\rightarrow DET \text{“value” “of”} CST \\
&\quad [CTR.sv = CST.sv] \\
DET &\rightarrow \text{“a”} \mid \text{“an”} \mid \text{“the”} \\
&\quad [DET.sv = \emptyset]
\end{aligned}
$$

3.5.3.1 Constants

Constants may be referred to directly by their names or indirectly by referencing their value. An example of a direct reference to the constant “1” would be “V is assigned to *1*” and an indirect reference example would be “V is assigned to *the value of 1*.” In Fig. 3.11, the productions for the CST symbol define all constant names, although only the definition of the constant “0” is shown. The symbol CTR captures indirect constant references.

3.5.3.2 Signals and Storage Elements

The SN symbol captures all valid signal names in the system. It is common practice to provide a list of all key signals and storage elements in any hardware specification, so we assume that such a list is provided and we generate productions for the SN signal to recognize signal names. Although all signals and storage elements are described by SN productions, only one is shown in Fig. 3.12 for the “awvalid.” The attribute value of each SN production is the name of the signal or storage element.

References to signals and storage elements can be either direct or indirect. A direct reference may use only the signal name, such as $OPCODE$ in the sentence, “*OPCODE* must be reset.” A direct reference may also include a determiner and a label specifying what type of storage element is being referred to, such as “*The OPCODE register* must be reset.” The SL symbol describes the possible labels, the SLR symbol describes direct references with labels, and the SDI symbol captures all direct signal references.

Indirect signal references indicate the value of the signal rather than the signal itself, such as “The value of the OPCODE register must be reset.” The IND symbol describes the “the value of” string used to identify indirect signal references. The SDE symbol captures all indirect signal references, and the SR symbol captures all signal references, both direct and indirect.

Fig. 3.12 Productions for signals and storage elements

$$
\begin{aligned}
SN &\rightarrow \text{“awvalid”} \\
&\quad [SN.sv = \text{“awvalid”}] \\
SL &\rightarrow \text{“signal”} \mid \text{“wire”} \mid \text{“register”} \mid \text{“bus”} \\
&\quad [SL.sv = \emptyset] \\
SLR &\rightarrow \text{“the”}\ SN\ \ SL \\
&\quad [SLR.sv = SN.sv] \\
SDI &\rightarrow SN \\
&\quad [SDI.sv = SN.sv] \\
SDI &\rightarrow SLR \\
&\quad [SDI.sv = SLR.sv] \\
IND &\rightarrow \text{“the”}\ \ \text{“value”}\ \ \text{“of”} \\
&\quad [IND.sv = \emptyset] \\
SDE &\rightarrow IND\ \ SN \\
&\quad [SDE.sv = SN.sv] \\
SDE &\rightarrow IND\ \ SLR \\
&\quad [SDE.sv = SLR.sv] \\
SR &\rightarrow SDE \\
&\quad [SR.sv = SDE.sv] \\
SR &\rightarrow SDI \\
&\quad [SR.sv = SDI.sv]
\end{aligned}
$$

3.5.3.3 Events

Hardware descriptions refer to events on signals in order to place constraints on those events. We allow two types of event references, *transition references* and *assignment references*. A transition reference can indicate either a rising edge, a falling edge, or a transition of any kind. These transitions are captured by the symbols TU, TD, and TA in Fig. 3.13. The attribute values for each transition use the SystemVerilog functions for transition detection, $\$rise$, $\$fell$, and $\$stable$.

An assignment reference uses a signal assignment as a noun phrase in a sentence. One type of assignment reference refers to a constant and uses a prepositional phrase to indicate the signal being assigned. An example is “a value of 1 on v,” where the prepositional phrase “on v” indicates the signal being assigned. The other type of assignment reference is the use of a gerund phrase to indicate the signal being assigned. An example is “assigning v to 1” where the subject of the gerund “assigning” is the signal name and the prepositional phrase “to 1” indicates the value to which the signal is assigned. The symbol AR captures both types of assignment references, and the ER symbol captures both assignment and transition references.

Fig. 3.13 Productions for events

$$
\begin{aligned}
&TU \rightarrow \text{“a”}\ \ \text{“rising”}\ \ \text{“edge”}\ \ \text{“on”}\, SN \\
&\quad [TU.sv = \text{“\$rise(”} + \ SN.sv\ + \text{“)”}] \\
&TD \rightarrow \text{“a”}\ \ \text{“falling”}\ \ \text{“edge”}\ \ \text{“on”}\, SN \\
&\quad [TD.sv = \text{“\$fell(”} + SN.sv + \text{“)”}] \\
&TA \rightarrow \text{“a”}\ \ \text{“transition”}\ \ \text{“on”}\, SN \\
&\quad [TA.sv = \text{“!\$stable(”} + SN.sv + \text{“)”}] \\
&TR \rightarrow TU \\
&\quad [TR.sv = TU.sv] \\
&TR \rightarrow TD \\
&\quad [TR.sv = TD.sv] \\
&TR \rightarrow TA \\
&\quad [TR.sv = TA.sv] \\
&AG \rightarrow \text{“assigning”} \,|\, \text{“setting”} \\
&\quad [AG.sv = \emptyset] \\
&AR \rightarrow AG\ \ SDI\ \text{“to”}\ CST \\
&\quad [AR.sv = SDI.sv + \text{“ == ”} + CST.sv] \\
&AR \rightarrow CTR\ \text{“on”}\ SN \\
&\quad [AR.sv = SN.sv + \text{“ == ”} + CTR.sv] \\
&ER \rightarrow TR \\
&\quad [ER.sv = TR.sv] \\
&ER \rightarrow AR \\
&\quad [ER.sv = AR.sv]
\end{aligned}
$$

3.5.3.4 Comparison Operations

Constraints in hardware descriptions are frequently specified using comparison operations. We accept the following comparisons and their complements: *equal*, *greater than*, and *less than*. Examples include “v is greater than 1” and “v must be equal to w.” A comparative relation is typically indicated in text by the use of a form of the verb “to be” sometimes together with a modal verb such as “must” or “can.” The productions for comparisons are shown in Fig. 3.14 which capture equality statements, Fig. 3.15 which capture inequality statements, and Fig. 3.16 which capture magnitude comparison statements. The RL symbol describes all of the verbs which indicate a comparative relation, and RN describes their negations. The symbols EQ (EN), GR (GN), and LS (LN) capture the comparisons of the type *equal* (*not equal*), *greater than* (*not greater than*), and *less than* (*not less than*), respectively. The attribute values of comparison productions use the appropriate comparison operators built into SystemVerilog.

Fig. 3.14 Productions for equality statements

$$
\begin{aligned}
&RL \rightarrow \text{“is”} \mid \text{“must” “be”} \mid \text{“remains”} \\
&RL \rightarrow \text{“must” “remain”} \\
&\quad [RL.sv = \emptyset] \\
&RN \rightarrow \text{“is” “not”} \mid \text{“must” “not” “be”} \\
&RN \rightarrow \text{“cannot” “be”} \mid \text{“must” “not” “remain”} \\
&\quad [RN.sv = \emptyset] \\
&EQ \rightarrow SR\ \ RL\ \ CST \\
&\quad [EQ.sv = SR.sv + \text{“ == ”} + CST.sv] \\
&EQ \rightarrow SR\ \text{“equals”}\ CST \\
&\quad [EQ.sv = SR.sv + \text{“ == ”} + CST.sv] \\
&EQ \rightarrow SR\ \ RL\ \text{“equal” “to”}\ CST \\
&\quad [EQ.sv = SR.sv + \text{“ == ”} + CST.sv] \\
&EQ \rightarrow CST\ \ RL\ \ SDE \\
&\quad [EQ.sv = SDE.sv + \text{“ == ”} + CST.sv] \\
&EQ \rightarrow CST\ \ RL\ \text{“equal” “to”}\ SDE \\
&\quad [EQ.sv = SDE.sv + \text{“ == ”} + CST.sv]
\end{aligned}
$$

Fig. 3.15 Productions for inequality statements

$$
\begin{aligned}
&EQN \rightarrow SR\ \ RN\ \ CST \\
&\quad [EQN.sv = SR.sv + \text{“ != ”} + CST.sv] \\
&EQN \rightarrow SR\ \ RN\ \text{“equal” “to”}\ CST \\
&\quad [EQN.sv = SR.sv + \text{“ != ”} + CST.sv] \\
&EQN \rightarrow SR\ \ RL\ \text{“does” “not” “equal”}\ CST \\
&\quad [EQN.sv = SR.sv + \text{“ != ”} + CST.sv] \\
&EQN \rightarrow CST\ \ RN\ \ SDE \\
&\quad [EQN.sv = SDE.sv + \text{“ != ”} + CST.sv] \\
&EQN \rightarrow CST\ \ RN\ \text{“equal” “to”}\ SDE \\
&\quad [EQN.sv = SDE.sv + \text{“ != ”} + CST.sv]
\end{aligned}
$$

3.5.3.5 Event Constraints

Hardware descriptions can constrain the possible events which can occur, both transition events and assignment events. We accept event constraints which are exclusionary, indicating that an event cannot occur. Transition events can be constrained in a positive sense by stating that a signal must remain stable, such as “V must be stable.” Figure 3.17 shows the productions for the symbol ST which is used to capture statements of stability and the symbol EX which captures exclusionary event constraints.

Fig. 3.16 Productions for magnitude comparison statements

$$
\begin{aligned}
GR &\rightarrow SRRL \text{ “greater” “than” } CST \\
&\quad [GR.sv = SR.sv + \text{ “ > “ } + CST.sv] \\
GR &\rightarrow SR_1 \ RL \text{ “greater” “than” } SR_2 \\
&\quad [GR.sv = SR_1.sv + \text{ “ > ” } + SR_2.sv] \\
GN &\rightarrow SR \ RN \text{ “greater” “than” } CST \\
&\quad [GN.sv = SR.sv + \text{ “ } leq \text{ ” } + CST.sv] \\
GN &\rightarrow SR_1 \ RN \text{ “greater” “than” } SR_2 \\
&\quad [GN.sv = SR_1.sv + \text{ “ } leq \text{ ” } + SR_2.sv] \\
LS &\rightarrow SR \ RL \text{ “less” “than” } CONST \\
&\quad [LS.sv = SR.sv + \text{ “ < ” } + CST.sv] \\
LS &\rightarrow SR_1 \ RL \text{ “less” “than” } SR_2 \\
&\quad [LS.sv = SR_1.sv + \text{ “ < ” } + SR_2.sv] \\
LN &\rightarrow SR \ RN \text{ “less” “than” } CST \\
&\quad [LN.sv = SR.sv + \text{ “ } \geq \text{ ” } + CST.sv] \\
LN &\rightarrow SR_1 \ RN \text{ “less” “than” } SR_2 \\
&\quad [LN.sv = SR_1.sv + \text{ “ } \geq \text{ ” } + SR_2.sv]
\end{aligned}
$$

Fig. 3.17 Productions for event constraints

$$
\begin{aligned}
SW &\rightarrow \text{“stable”} \mid \text{“constant”} \mid \text{“fixed”} \\
ST &\rightarrow SR \ RL \ SW \\
&\quad [ST.sv = \text{“\$stable(”} + SR.sv + \text{“)”}] \\
EX &\rightarrow ER \text{ “is” “not” “permitted”} \\
&\quad [EX.sv = \text{“!(”} + ER.sv + \text{“)”}] \\
EX &\rightarrow ER \text{ “is” “not” “allowed”} \\
&\quad [EX.sv = \text{“!(”} + ER.sv + \text{“)”}]
\end{aligned}
$$

3.5.3.6 Boolean Logic

Constraints can be combined using boolean constructs such as “V is greater than 1 *and* w is equal to 1.” We allow the use of the words “and” and “or” to indicate the boolean combination of constraints. The symbol CB captures all basic constraints which do not include boolean constructs. Figure 3.18 presents productions for CB which match all arithmetic comparison and event constraints. The CA (CO) symbol describes the conjunction (disjunction) of any set of basic constraints. The CL symbol captures all boolean combinations of basic constraints involving “and” and “or” operations.

Fig. 3.18 Productions for boolean logic

$$
\begin{aligned}
&CB \rightarrow EQ \\
&\quad [CB.sv = EQ.sv] \\
&CB \rightarrow EQN \\
&\quad [CB.sv = EQN.sv] \\
&CB \rightarrow GR \\
&\quad [CB.sv = GR.sv] \\
&CB \rightarrow GN \\
&\quad [CB.sv = GN.sv] \\
&CB \rightarrow LS \\
&\quad [CB.sv = LS.sv] \\
&CB \rightarrow LN \\
&\quad [CB.sv = LN.sv] \\
&CL \rightarrow CB \\
&\quad [CL.sv = CB.sv] \\
&CL \rightarrow CA \\
&\quad [CL.sv = CA.sv] \\
&CL \rightarrow CO \\
&\quad [CL.sv = CO.sv] \\
&CA \rightarrow CB \text{“and”} CL \\
&\quad [CA.sv = CB.sv + \text{“ \&\& ”} + CL.sv] \\
&CO \rightarrow CB \text{“or”} CL \\
&\quad [CO.sv = CB.sv + \text{“ || ”} + CL.sv]
\end{aligned}
$$

3.5.3.7 Implication

Implication is a concept commonly expressed in hardware descriptions. There are several ways in which implication is expressed in English, using the key words “if,” “then,” and “when.” Examples of implications include “V must be equal to 1 when W is asserted” or “If W is asserted then V must be equal to 1.” The productions for the symbols for the antecedent (AN) and consequent (CN) of an implication are shown in Fig. 3.19. Both antecedents and consequents match any boolean combination of basic constraints. The productions for symbol CI which captures implication constraints are also shown in Fig. 3.19.

$$
\begin{aligned}
&AN \rightarrow CL \\
&\quad [AN.sv = CL.sv] \\
&CN \rightarrow CL \\
&\quad [CN.sv = CL.sv] \\
&CI \rightarrow \text{“if”}\ AN \text{“then”}\ CN \\
&\quad [CI.sv = \text{“!(”} + AN.sv + \text{“)} \parallel \text{(”} + CN.sv + \text{“)”}] \\
&CI \rightarrow \text{“when”}\ AN \text{“,”}\ CN \\
&\quad [CI.sv = \text{“!(”} + AN.sv + \text{“)} \parallel \text{(”} + CN.sv + \text{“)”}] \\
&CI \rightarrow CN\ \text{“when”}\ AN \\
&\quad [CI.sv = \text{“!(”} + AN.sv + \text{“)} \parallel \text{(”} + CN.sv + \text{“)”}]
\end{aligned}
$$

Fig. 3.19 Productions for implications

Fig. 3.20 Productions for assertion sentences

$$
\begin{aligned}
&S \rightarrow CL \\
&\quad [S.sv = \text{“assert property (”} + CL.sv + \text{“);”}] \\
&S \rightarrow CI \\
&\quad [S.sv = \text{“assert property (”} + CI.sv + \text{“);”}]
\end{aligned}
$$

3.5.3.8 Assertion Sentences

Each assertion is assumed to be expressed in a single sentence. A sentence is the basic unit of text which is parsed, and the sentence symbol *S* is the root node of any parse tree. We assume that each sentence being parsed is the expression of a constraint on system signals and storage elements. A sentence can be either a boolean combination of basic constraints or an implication. Figure 3.20 shows the productions for the *S* symbol.

3.5.4 Experimental Results

The system was implemented in Python using the API provided in the Natural Language Tool Kit [3] to create a recursive descent syntactic parser. All results were generated on an Intel Core i5 processor, 3.2 GHz, with 8 GB RAM.

3.5.4.1 Benchmark Set

To evaluate our system it was necessary to identify a set of assertions for a real system, specified in English. As a benchmark set of assertions, we have used the assertions developed by ARM Inc. for the verification of AXI protocol

```
assert property (!(awvalid == 1) ||
                 (!(awburst == 2b'11)));
```

Fig. 3.21 SystemVerilog assertion created from an AXI assertion

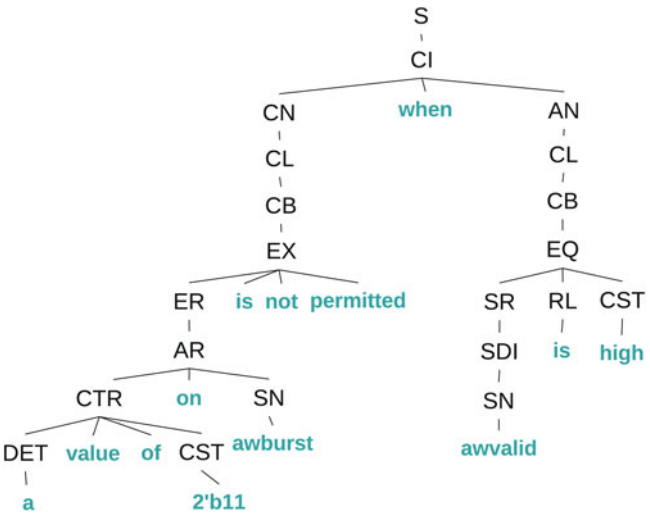

Fig. 3.22 Parse tree created from an AXI assertion

implementations [1]. We evaluated all assertions checking the write and read channels including write/read address channel checks, write/read data channel checks, and write response channel checks. The benchmark set of assertions consists of 117 individual assertions, expressed in English.

3.5.4.2 Results on Benchmarks

Our tool successfully generated SystemVerilog assertions for 52 out of 117, 44% of all assertions. The total CPU time required to generate all SystemVerilog assertions was 199 s, which is 3.82 s per assertion, on average.

Figure 3.22 shows the parse tree resulting from the AXI assertion, "A value of 2'b11 on awburst is not permitted when awvalid is high" [1]. The SystemVerilog assertion generated from this English assertion is shown in Fig. 3.21. Near the top of the parse tree the CI symbol indicates that an implication was recognized, of the form *CONSEQUENT* "when" *ANTECEDENT*. The consequent and antecedent are the subtrees under the CN and AN symbols, respectively. The antecedent is the phrase "awvalid is high" which is the subtree under the EQ symbol, indicating that it represents an equality constraint. The antecedent and consequent appear in the SystemVerilog assertion as the terms "awvalid == 1" and "!(awburst == 2b'11)."

3.5.4.3 Limitations of Assertion Generation

The range of English expression which our system can process is limited by the attribute grammar which we have defined. In order to better understand the practical limits of the grammar, we have examined the English assertions in our benchmark set which our grammar failed to parse. We have identified the three features of the unparsed assertions which caused our system to fail.

Sequential Constraints: Our system cannot parse sequential constraints which describe properties spanning multiple clock cycles. An example of such a constraint in our benchmark set is, "Recommended that wready is asserted within MAXWAITS cycles of WVALID being asserted."

Object Hierarchy: Our system only accepts constraints directly on the values of signals and storage elements. However assertions may constrain a more abstract object or event. An example in our benchmark set is, "The size of a read transaction must not exceed the width of the data interface." This assertion refers to the abstract "read transaction" event which is defined elsewhere in the specification, and it constrains the "size" property of this event.

Multiple Sentences: Our system assumes that each assertion is expressed in a single sentence, but this is sometimes not convenient in practice. A sample assertion in our benchmark set which requires multiple sentences is, "The number of write data items matches awlen for the corresponding address. This is triggered when any of the following occurs: ...." Accepting assertions expressed across multiple sentences introduces several referencing issues including *anaphora resolution* which enables the word "This" in the second sentence to be related to the constraint defined in the first sentence. Although we did not address them here, reference problems, including anaphora resolution, have been well studied in the field of natural language processing.

In order to address these limitations, it was necessary to expand our work from processing assertions describing an instant in time to entire transactions which model sequences of events over a span of time. Considering transactions is the topic of the following section.

3.6 Generating Transactors

We present an approach to automatically generate simulatable bus transactions directly from natural language bus protocol specifications. Our technique employs semantic parsing to produce Verilog transactors with high timing fidelity [5]; all significant events of the protocol are modeled explicitly. We identify a set of *transaction concepts* which are ideas commonly used in natural language description of bus protocols to express different aspects of a transaction. Each transaction concept is recognized in the natural language description using a set of context-free grammar (CFG) productions which we define. The resulting parse

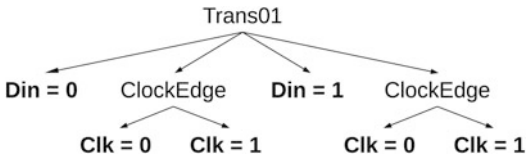

Fig. 3.23 Transaction graph representation, shifting '01' into a shift register

trees are scanned to locate each transaction concept and generate appropriate Verilog code. Automating the generation of bus transactors reduces design and verification time, eliminating the need to manually design and verify each transactor.

3.6.1 Transactions and Transactors

We describe a transaction as a hierarchical sequence of events on a set of signals. Each event may write a value to a signal and read a value from one or more signals. Figure 3.23 shows the **transaction graph** representation of a transaction to shift the value '01' into a two bit shift register. We assume that the shift register has two single bit inputs *Din*, the data input, and *Clk*, the clock input. A bit is shifted into the register by assigning *Din* to the appropriate value and causing a rising edge on the *Clk* input. The transaction, referred to as *Trans01*, is described hierarchically and drawn as a directed acyclic graph in Fig. 3.23. The leaf nodes in the graph are assignments to input signals of the shift register. Each non-leaf node is a transaction which is defined by the sequence of its successor nodes. For example, the *ClockEdge* transaction is defined by the sequential execution of the assignments *Clk=0* and *Clk=1*.

3.6.1.1 Bus Transactors

A bus transactor implements a transaction defined in a protocol, acting as the link between a transaction generator and a bus. The generic transactor interface which we assume is shown in Fig. 3.24. In the figure, the transactor is shown to interface with the bus signals on the right. The transactor is implemented as a Verilog *task*, so a transaction generator invokes the transactor task with a set of arguments defined by the transactor interface. The interface includes the three signals shown in bold on the left side of Fig. 3.24. Many bus protocols include a unique address for each device on the bus, so the *Address* signal is defined to hold the address of the receiver. *Tx* contains the data to be transmitted during a write transaction, and *Rx* contains the data received during a read transaction. The width of the *Address*, *Tx*, and *Rx* signals must be declared in the specification document.

References to the *Address (Ad)*, *Tx*, and *Rx* signals in the specification document are considered differently than normal signal references. We assume that the Ad and Tx data will be transmitted on the bus and that the Rx data is read from the bus. We

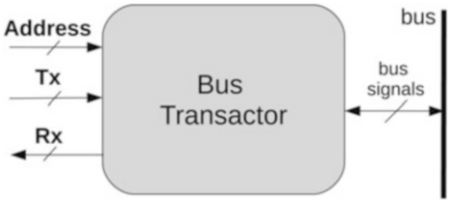

Fig. 3.24 Transactor interface

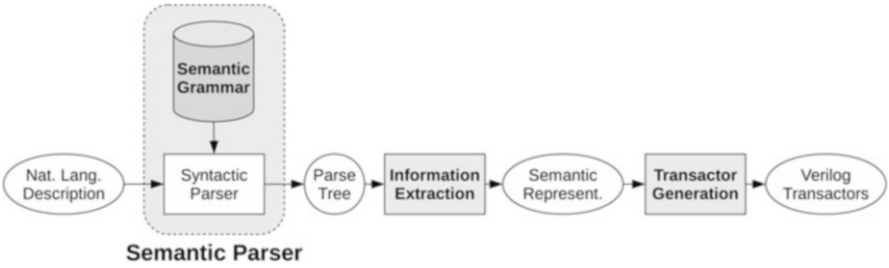

Fig. 3.25 Structure of the natural language transactor generation system

assume that Ad and Tx are stored in transmit queues, so when data is read from them the data is removed from the queue. We also assume that Rx is stored in a receive queue so that when data is written to Rx, it does not overwrite existing data in the queue.

3.6.2 System Overview

Figure 3.25 depicts the structure of the proposed system for transactor generation from a natural language description. The natural language description is processed using a *Semantic Parser* to generate a parse tree. The semantic parser is built using an off-the-shelf syntactic parser, the Natural Language Toolkit [3], and a semantic grammar which we define for this application. *Information Extraction* is applied to the resulting parse tree to generate a semantic representation which contains all behavioral information about each transaction. The semantic representation is used to perform *Transactor Generation* and generate a set of Verilog transactors which accurately model the behavior of the specified transactions.

3.6.3 Transaction Concepts

We have defined a semantic grammar to identify the expression of transaction concepts in natural language descriptions of bus transaction protocols. The grammar which we present is not sufficient to parse all legal descriptions; however, it is

broad enough to parse a useful subclass of all descriptions. In this section we define the subclass of descriptions which can be parsed by our grammar. We present the syntactic patterns which are recognized by our grammar and a subset of the production rules used to parse each pattern. The productions we use to recognize standard English grammatical constructs (i.e., noun phrase, verb phrase, etc.) are a subset of those presented in [26].

We present a set of *transaction concepts* which are ideas used in the natural language description to express different aspects of a transaction. The transaction concepts are expressed in the natural language specification to describe the protocol, and each transaction concept can be mapped directly to Verilog constructs. Each transaction concept is recognized in the natural language description using a set of CFG productions which we define.

- **Signal Definitions** – All input and output signals must be declared in the natural language document. The SIGDEF grammatical symbol captures each signal declaration.
- **Transaction References** – Each transaction must be referred to in the text using some noun phrase as an unique identifier. The symbol TRANSREF is used to capture transaction references.
- **Sequence Descriptions** – Each transaction is composed of a sequence of other transactions which are lower in the transaction hierarchy. The three symbols FULLSEQUENCE, PREFIXSEQUENCE, and SUFFIXSEQUENCE are used to capture sequence descriptions.
- **Signal Reading/Writing** – Transactions at the lowest hierarchical level must directly interact with signals. The symbols ASSIGNDIRECT, ASSIGNSTRUCT, and ASSIGNIMPL are used to capture signal reading and writing.

The transaction concepts and their associated CFG productions are described in the following sections.

3.6.3.1 Signal Definitions

We assume that all input and output signals are declared in the document. The top production used to capture signal definitions is shown in Fig. 3.26.

The symbol SIGNAME matches any string and is assumed to be a proper noun which is used to reference the signal. The DIRECTION symbol matches one of the following values: “input,” “output,” and “input/output.” The WIDTH symbol is a numeral indicating the bitwidth of the signal.

SIGDEF → SIGNAME “is an” DIRECTION “signal,” WIDTH “bit(s) wide”

Fig. 3.26 Productions to recognize signal definitions

Fig. 3.27 Productions to recognize transaction references

TRANSREF → SNP | GNP
GNP → VG NP
VG → "sending" | "transmitting" | "receiving"
SNP → NP

3.6.3.2 Transaction References

We assume that transactions are referred to in one of two ways in the document, either as a simple noun phrase (SNP) or a gerundive noun phrase (GNP). A simple noun phase is a noun phrase which contains a head noun, a set of modifiers for the noun, and a determiner. An example of a simple noun phrase references is "a write transaction," with the head noun "transaction," the modifier "write," and the indefinite article "a" which acts as a determiner. A gerundive noun phrase uses a verb (i.e., "send") in gerund form (i.e., "sending") as the first modifier for the head noun. For example, "sending a data byte" refers to a transaction which transmits a byte of data. The gerund used in a gerund noun phrase describes the movement of information, so we assume that it is either "sending," "transmitting," or "receiving." The gerund indicates the direction of data flow with respect to the bus master. Figure 3.27 shows a subset of the productions used to recognize transaction references and to identify their head noun and modifiers. In the figure, the symbol TRANSREF represents a transaction reference, and the symbol NP represents a noun phrase.

In a gerundive noun phrase, the head noun may be plural to indicate iterative execution. For example, in the sentence "Sending a byte is performed by sending 8 bits," the transaction "sending bits" is plural and describes the sending of a single bit eight times. Recognition of plurals is well understood in natural language processing research. We add productions to identify plural head nouns and to identify the numeral representing the number of iterations.

3.6.3.3 Sequence Descriptions

The relationship between a transaction and the sequence of sub-transactions which compose it can be expressed with several syntactic patterns. In order to recognize sequence descriptions we define a set of **cues**, multiword substrings which we expect to find in a sequence description. Cues are included in the production rules of our grammar to allow us to identify the elements of a sequence. We define three different syntactic patterns which indicate that a sentence contains sequence information.

The *full sequence pattern* matches sentences which define the entire sequence of sub-transactions composing a transaction. An example of a sentence which matches the full sequence pattern is the following: "Sending a byte is performed by sending 8 bits and receiving an acknowledge bit." In this sentence, "Sending a byte" is the

Fig. 3.28 Productions to recognize the full sequence pattern

S → FULLSEQUENCE
FULLSEQUENCE → TRANSHEAD
    CUE_FULL_SEQ TRANSLIST
TRANSHEAD → TRANSREF
TRANSLIST → TRANSREF
TRANSLIST → TRANSREF "and" TRANSREF
TRANSLIST → TRANSREF "," TRANSLIST
CUE_FULL_SEQ → "is performed by" |
    "is transmitted by" | "is executed by" |
    "is sent by"

Fig. 3.29 Productions to recognize the prefix and suffix patterns

S → PREFIXSEQUENCE
S → SUFFIXSEQUENCE
PREFIXSEQUENCE → TRANSHEAD
    CUE_PREFIX TRANSLIST
SUFFIXSEQUENCE → TRANSHEAD
    CUE_SUFFIX TRANSLIST
CUE_PREFIX → "begins with" | "starts with"
CUE_SUFFIX → "ends with"

head transaction, "is performed by" is the cue substring, "sending 8 bits" is the first sub-transaction, and "receiving an acknowledge bit" is the second sub-transaction.

Figure 3.28 lists the productions added to our grammar to recognize the full sequence pattern and its components. The TRANSHEAD symbol is a reference to the transaction being defined in the sentence. The TRANSLIST symbol represents an unbounded sequence of transaction references. CUE_FULL_SEQ matches the set of substrings which identify the full sequence pattern. CUE_FULL_SEQ is defined as the following set of substrings: "is performed by," "is transmitted by," "is executed by," and "is sent by."

The second and third syntactic patterns to describe sequence are the *prefix pattern* and the *suffix pattern*. The only difference between the three sequence syntactic patterns is the cues used. The cues used for the prefix pattern are "begins with" and "starts with." The cue for the suffix pattern is "ends with." The productions added to recognize the prefix and suffix patterns are shown in Fig. 3.29.

3.6.3.4 Reading and Writing Signals

Each transaction is a hierarchical sequence of events on a set of signals. We define three types of signal assignment descriptions, a *direct signal assignment*, a *structured signal assignment*, and an *implicit signal assignment*.

A direct signal assignment is expressed with a gerundive noun phrase as in the sentence, "Transmitting a bit is performed by setting X to 1." In this sentence, the

Fig. 3.30 Direct signal assignment production

$$\text{ASSIGNDIRECT} \rightarrow \text{CUE\_DIRECT SIGNAME } \text{“to”} \text{ SIGVALUE}$$

$$\text{ASSIGNSTRUCT} \rightarrow \text{CUE\_STRUCT } \text{“on”} \text{ SIGNAME}$$

Fig. 3.31 Structured signal assignment production

Fig. 3.32 Implicit signal assignment production

$$\text{ASSIGNIMPL} \rightarrow \text{SIGASSIGN } \text{“while”} \text{ SIGCOND}$$

gerundive noun phrase “setting X to 1” indicates the direct signal assignment. The top production for a direct signal assignment is shown in Fig. 3.30.

In Fig. 3.30, CUE_DIRECT is the set of direct assignment cues which we recognize: “setting” and “assigning.” SIGNAME matches any string and should match the name of a signal defined in another sentence. SIGVALUE matches the set of values to which a signal can be assigned. For single-bit signals, the SIGVALUE set includes “0,” “1,” and “Z.”

We also allow the description to apply the gerund “releasing” to a signal to indicate that it should be assigned to the value Z. In this case, the form of the signal assignment is *“releasing” SIGNAME*, and no SIGVALUE is required.

The description can also specify a *structured signal assignment* which is a predefined sequence of assignments on a signal. Structured signal assignments are assignments which would be commonly understood by any designer without having to be explicitly defined in the specification. We define three structured signal assignments, generating a pulse, generating a rising edge, and generating a falling edge. The top production for a structured signal assignment is shown in Fig. 3.31. In Fig. 3.31, CUE_STRUCT matches the following multiword strings: “generating a pulse,” “generating a rising edge,” and “generating a falling edge.”

The transaction description may require a signal assignment to be performed without explicitly declaring the assignment. This type of *implicit signal assignment* can occur when the word “while” is used to specify a condition on a signal. An example of an implicit signal assignment can be seen in the following sentence: “Sending a bit is performed by setting SDA to 1 while SCL is low.” In this sentence, the gerundive noun phrase “setting SDA to 1” indicates an assignment to signal SDA. The phrase “while SCL is low” implies that the SCL signal must be assigned to 0 before the SDA signal assignment occurs. The production for an implicit signal assignment is shown in Fig. 3.32. SIGASSIGN matches either a direct or structured signal assignment, as shown in Figs. 3.30 and 3.31. SIGCOND is a condition on a signal which is assumed to be in the following form, *SIGNAME “is” SIGVALUE*.

3.6.4 Information Extraction

The information extraction stage analyzes the parse tree to find all information needed to generate a simulatable bus transactor. The output of this stage is a computational model containing all extracted information. We present a set of classes which define the model. Information extraction instantiates these classes to generate a set of objects which contain the extracted information.

3.6.4.1 Class Structure

We define two main classes to store information, the **Signal** class and the **Transaction** class. The **Signal** class contains three attributes: *SignalName*, the name of the signal; *Width*, the bit width of the signal; and *Direction*, the input/output direction of the signal.

The **Transaction** class has two subclasses, **TerminalTransaction** and **NonTerminalTransaction**. The **TerminalTransaction** class defines transactions which directly writes or reads signals. Objects of the TerminalTransaction class represent one event, either a signal read event or a signal write event. A TerminalTransaction does not describe a sequence of transactions. The TerminalTransaction class has the following attributes:

- *R/W* – This indicates whether the transaction is a read event or a write event.
- *Signal* – This a reference to the signal which is being read (read transaction) or written (write transaction).
- *Value* – This attribute only has meaning for a write transaction. This attribute is either the value being assigned to a signal {0, 1, Z} or the name of the input queue being read from, {Address, Tx}.

The **NonTerminalTransaction** class defines transactions which are defined as a sequence of other transactions. A NonTerminalTransaction does not directly assign values to signals. The NonTerminalTransaction class has the following attributes:

- *TransID* – This is the unique name of the transaction.
- *TransList* – This is an ordered list of transactions.

Each element of the *TransList* is a member of the **TransListElt** class which has the following attributes:

- *Transaction* – This is a reference to a single transaction object.
- *Iterations* – This indicates the number of times the associated transaction must be repeated. This attribute is used when a transaction is modified by a numeral in the specification document. For example, the phrase “sending 8 bits” indicates that the transaction to send a bit should be repeated 8 times.

3.6.4.2 Extraction Process

Information extraction is performed by scanning the parse trees of each sentence to find symbols which have well-defined semantic meanings. When such a symbol is found an object is added to the semantic representation to capture the meaning of the symbol. The attributes of the object are defined by examining the subtree of the parse tree whose head is the semantic symbol.

A **Signal** object is created for each SIGDEF symbol found. The SIGNAME, DIRECTION, and WIDTH symbols, which are children of the SIGDEF symbol, are used to define the attributes of the **Signal** object.

A **Transaction** object is created when any of the sequence description symbols are found: FULLSEQUENCE, PREFIXSEQUENCE, and SUFFIXSEQUENCE. The *TransID* attribute of the transaction object is generated from the transaction reference associated with the TRANSHEAD symbol of the sequence description. The *TransList* is created from the TRANSLIST symbol of the sequence description. A **TransListElt** object is created for each TRANSREF symbol in the TRANSLIST. A TRANSLIST may also contain signal read/write events. A **TransListElt** object is created for each signal read/write also.

The result of information extraction is a hierarchy of transaction objects whose *TransList* attributes refer to the objects in the next lower level of the hierarchy.

3.6.5 Transactor Generation

The transaction graph generated by information extraction is used to create a Verilog description of a simulatable bus transactor. Transactor generation involves a depth-first traversal of the transaction graph, starting at the top-level node. When a leaf node is reached, Verilog code is added to the transactor which performs the operation corresponding to the leaf node. Each leaf node is an assignment involving constants, signals, and the input/output queues Tx, Rx, and Ad. If the signal assignment does not involve a queue then an appropriate Verilog dataflow assignment statement is generated to perform the assignment. If the assignment involves a queue, special-purpose Verilog tasks are used to access queues. We have defined the **q_extract** task to extract a number of bits of a queue, and the **q_insert** task to add bits to a queue. These two tasks maintain the read and write pointers associated with the queue.

Information about iterative execution is contained in each element of the *TransList* of each **Transaction** object. Iterative execution is modeled using the *repeat* construct in Verilog. When a **TransListElt** which has $Iterations > 1$, a *repeat* construct is generated. The scope of the *repeat* construct includes the entire subgraph.

1. Sending a write transaction is performed by sending a start condition, sending a write header byte, sending a data byte, and sending a stop condition.
2. Sending a start condition is performed by setting SDA to 0 and setting SCL to 0.
3. Sending a write header byte is performed by sending an address, sending a bit whose value is 0, and receiving an acknowledge bit.
4. Transmitting an address is performed by sending 7 address bits.
5. Sending an address bit is performed by setting SDA to a bit from Ad while SCL is low and generating a pulse on SCL.
6. Sending a bit is performed by setting SDA to a value while SCL is low and generating a pulse on SCL.
7. Receiving an acknowledge bit is performed by releasing SDA and generating a pulse on SCL.
8. Sending a data byte is performed by sending 8 data bits and receiving an acknowledge bit.
9. Sending a data bit is performed by setting SDA to a bit from Tx while SCL is low and generating a pulse on SCL.
10. Sending a stop condition is performed by setting SDA to 0, setting SCL to 1, and generating a rising edge on SDA.

Fig. 3.33 Natural language specification, I^2C write transaction

3.6.6 Experimental Results

The transactor generation system was implemented in Python using the API provided in the Natural Language Tool Kit [3] to create a recursive descent syntactic parser. All results were generated on a 2 GHz Intel Core 2 Duo processor.

To evaluate our system we have generated a bus transactor for the I^2C serial protocol developed by Philips [39] to support onboard communication. The protocol uses two wires, the data line SDA and the clock line SCL whose rising edges synchronize data transmission. We have explored a subset of the protocol involving a single Master node, 7-bit addressing, and no use of the repeated start condition. We use the 10 sentence natural language specification of the write transaction shown in Fig. 3.33. To focus on the more interesting part of the example, we have omitted the sentences defining the signals.

An unique **Transaction** object is created to capture the information in each sentence of the natural language description, forming an object hierarchy with the *Write transaction* as the top-level object. Figure 3.34a shows the partial object hierarchy representing the *Transmitting an address* transaction generated from sentence 4 in Fig. 3.33. Figure 3.34a includes the *Sending an address bit* transaction from sentence 5 which is annotated with the number 7 to represent the number of iterations as specified in sentence 4. The **q_extract** task call removes a bit from the *Ad* queue and assigns SDA to the bit value. The resulting Verilog code for *Transmitting an address* is shown in Fig. 3.34b.

The bus transactor generation process was performed in 61.5 s of CPU time. The resulting transactor, named **i2c_write**, is composed of 32 lines of Verilog code. The first portion of the simulation result of invoking **i2c_write** to write to address 7'0011001 is shown in Fig. 3.35. The portions of the write transaction shown include the *Start condition* from sentence 2 (START), *Transmitting an address* from sentence 4, and the 0 bit (R/W) referred to in sentence 3.

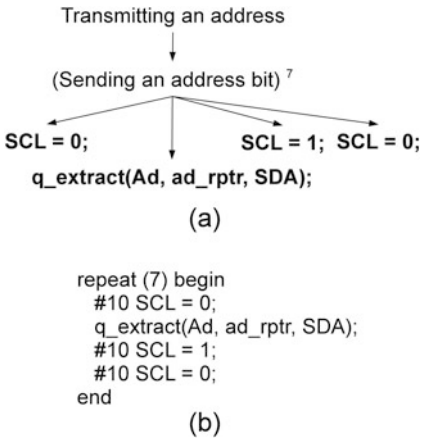

Fig. 3.34 "Transmitting an address" transaction, (**a**) transaction graph, (**b**) Verilog code

Fig. 3.35 Simulation waveform from **i2c_write**

3.7 Conclusions

We have presented approaches to generate verification artifacts, assertions, and transactors, directly from natural language text found in a hardware description. The assertions are generated in SystemVerilog and can be used for response checking as part of a standard verification flow. The transactors are simulatable Verilog and can also be used with any Verilog simulator. Our approaches are based on the use of semantic context-free grammars which can be easily extended to accept any desired degree of linguistic variation which might be present in the English hardware description. In its present form, our work can significantly reduce the amount of manual labor which is traditionally devoted to creating and debugging a hardware verification environment.

Acknowledgments This material is based upon work supported by the National Science Foundation under Grant No. 1813858.

References

1. ARM Ltd. *AMBA 3 AXI Protocol Checker User Guide* (2009)
2. A. Bacchelli, T. Dal Sasso, M. D'Ambros, M. Lanza, Content classification of development emails, in *Proceedings of the 2012 International Conference on Software Engineering*, ICSE (2012)

3. S. Bird, E. Klien, E. Loper, *Natural Language Processing with Python — Analyzing Text with the Natural Language Toolkit* (O'Reilly Media, 2009)
4. D. Bobrow, *GUS, A Frame Driven Dialog System* (Morgan Kaufmann Publishers Inc., 1986)
5. M. Burton, J. Aldis, R. Gunzel, W. Klingauf, Transaction level modeling: A reflection on what TLM is and how TLMs may be classified, in *Forum on Specification Design Languages (FDL)*, pp. 92–97, 2007
6. L. Cai, D. Gajski, Transaction level modeling: An overview, in *International Conference on HW/SW Codesign and System Synthesis (CODES-ISSS)*, pp. 19–24, 2003
7. W.R. Cyre, J. Armstrong, M. Manek-Honcharik, A.J. Honcharik, Generating VHDL models from natural language descriptions, in *Proceedings of the conference on European design automation*, EURO-DAC '94, 1994
8. R. Dechter, *Constraint Processing* (Morgan Kaufmann Publishers, 2003)
9. J. Drazan, V. Mencl, Improved processing of textual use cases: Deriving behavior specifications, in *In Proceedings of SOFSEM 2007* (Springer, 2007)
10. R. Drechsler, M. Soeken, R. Wille, Automated and quality-driven requirements engineering, in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2014
11. J. Engelfriet, Attribute grammars: Attribute evaluation methods, in *Methods and Tools for Compiler Construction* (Cambridge University Press, New York, 1984), pp. 103–138
12. R. Feldman, I. Dagan, Knowledge discovery in textual databases (kdt), in *In Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95)* (AAAI Press, 1995), pp. 112–117
13. C. Fillmore, Frame semantics, in *Linguistics in the Morning Calm*, Linguistic Society of Korea (Hanshin Publilshing Company, 1982)
14. H. Foster, Applied assertion-based verification: An industry perspective. Found. Trends Electron. Des. Autom. **3**(1) (2009)
15. Z.P. Fry, D. Shepherd, E. Hill, L. Pollock, K. Vijay-Shanker, Analysing source code: Looking for useful verb-direct object pairs in all the right places. Software, IET **2**(1) (2008)
16. D.D. Gajski, S. Abdi, A. Gerstlauer, G. Schirner, *Embedded System Design: Modeling, Synthesis and Verification* (Springer Publishing Company, Incorporated, 2009)
17. D.D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, S. Zhao, *SpecC: Specification Language and Methodology* (Kluwer Academic Publishers, 2000)
18. D. Gildea, D. Jurafsky, Automatic labeling of semantic roles. Comput. Linguist. **28**, 245–288 (2001)
19. M. Gordon, D. Harel, Generating executable scenarios from natural language, in *Proceedings of the 10th International Conference on Computational Linguistics and Intelligent Text Processing* (Springer, 2009)
20. J.J. Granacki, A.C. Parker, PHRAN-SPAN: A natural language interface for system specifications, in *Proceedings of the 24th ACM/IEEE Design Automation Conference*, 1987
21. J.J. Granacki, A.C. Parker, Y. Arena, Understanding system specifications written in natural language, in *Proceedings of the 10th International Joint Conference on Artificial Intelligence – Volume 2*, IJCAI'87, 1987
22. M.A. Hearst, Untangling text data mining, in *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, ACL '99, pp. 3–10, 1999
23. IEEE Press, *IEEE Std. 1500-2005, IEEE Standard for Embedded Core Test*, 2005
24. S. Issar, W. Ward, Cmu's robust spoken language understanding system, in *EUROSPEECH'93*, pp. 2147–2150, 1993
25. A. Jantsch, *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation* (Morgan Kaufmann, 2004)
26. D. Jurafsky, J.H. Martin, *Speech and Language Processing An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 2nd edn (Pearson Education, 2009)
27. D.E. Knuth, Semantics of context-free languages. Theory Comput. Syst. **2**(2), 127–145 (1968)

28. L. Kof, Natural language processing: Mature enough for requirements documents analysis, in *10th International Conference on Applications of Natural Language to Information Systems (NLDB* (Springer, 2005), pp. 91–102
29. L. Kof, Scenarios: Identifying missing objects and actions by means of computational linguistics, in *In Proceedings 15th RE*, pp. 121–130, 2007
30. L. Kof, Translation of textual specifications to automata by means of discourse context modeling, in *Proceedings of the 15th International Working Conference on Requirements Engineering: Foundation for Software Quality* (Springer, 2009)
31. S.J. Korner, T. Brumm, Natural language specification improvement with ontologies. Int. J. Semant. Comput. **3**, 445–470 (2009)
32. L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, Models of computation for embedded system design, in A. Jerraya, J. Mermet (eds.) *System-Level Synthesis* (Kluwer Academic Publishers, 1998), pp. 45–102
33. P. Marwedel, *Embedded System Design* (Springer, 2006)
34. V. Mencl, Deriving behavior specifications from textual use cases, in *Oesterreichische Computer Gesellschaft*, pp. 3–85403, 2004
35. S. Miller, Hidden understanding models of natural language, in *In Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pp. 25–32, 1994
36. M. Minsky, A framework for representing knowledge. Technical report, Massachusetts Institute of Technology, Cambridge, 1974
37. R.J. Mooney, Learning for semantic parsing, in *In Computational Linguistics and Intelligent Text Processing: Proceedings of the 8th International Conference*, 2007
38. S. Narayan, F. Vahid, D.D. Gajski, System specification with the speccharts language. IEEE Des. Test **9**(4) (1992)
39. NXP Semiconductors, *I2C-bus specification and user manual*, rev. 03 edition, June 2007. UM10204
40. T. Samad, S. Director, Natural-language interface for cad: A first step. IEEE Des. Test **2**(4) (1985)
41. T. Samad, S.W. Director, Towards a natural language interface for CAD, in *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, 1985
42. L.M. Segundo, R.R. Herrera, K. Yeni Perez Herrera, Uml sequence diagram generator system from use case description using natural language, in *Proceedings of the Electronics, Robotics and Automotive Mechanics Conference* (IEEE Computer Society, 2007), pp. 360–363
43. M. Soeken, R. Wille, R. Drechsler, Assisted behavior driven development using natural language processing, in *TOOLS (50)*, 2012
44. R.S. Swier, S. Stevenson, Unsupervised semantic role labelling, in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2004
45. Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: bugs or bad comments?*/. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, 2007.
46. S.H. Tan, D. Marinov, L. Tan, G.T. Leavens, @tcomment: Testing javadoc comments to detect comment-code inconsistencies, in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, 2012
47. I.H. Witten. *Practical Handbook of Internet Computing*, chapter Text Mining (Chapman & Hall/CRC Press, 2005)
48. J. Yang, L. Tan, Inferring semantically related words from software context, in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, June 2012