Opportunities for Analyzing Hardware Specifications with NLP Techniques

Alejandro Rago*†1, Claudia Marcos*‡2, J. Andrés Diaz-Pace*†3
*Instituto Superior de Ingeniería de Software (ISISTAN-UNICEN)
Tandil, Buenos Aires, Argentina
†CONICET, Argentina
‡CIC, Buenos Aires, Argentina
1 2 3{arago,cmarcos,adiaz}@exa.unicen.edu.ar

***Abstract*—Hardware design is a mature discipline that heavily relies on complex models to create the blueprints of a system and special notations to describe the expected behavior of its components. However, hardware engineers frequently have to go through multiple specifications written in natural language to identify components, constraints and assertions and translate them to more formal expressions in order to enable automated verifications and consistency checks. For this reason, computer-assisted tools capable of processing and understanding hardware documentation can be of great help to assist and guide engineers in difficult and otherwise error-prone activities. In previous works, we have explored several *Natural Language Processing* (NLP) techniques for the analysis of requirements and architecture specifications with promising results. In this article, we report on some interesting applications we developed for inspecting Software Engineering documentation and discuss their potential applications to automated hardware design.**

I. Introduction

In the last decade, there has been an increasing demand of software systems encoded as hardware circuits. This kind of development is commonly referred to as Systems-on-Chip (SoC) or Application Specific Integrated Circuits (ASIC). Since the production of SoC is often in the number of thousands units, making mistakes in early engineering phases can have negative consequences in the quality of the product. For reducing bugs in the design and keep manufacturing costs on level, engineers must verify the design models and test the circuits before constructing a lithographic mask of the chip. In fact, verification activities are said to take up to 60% of the design cycle in a modern SoC [1]. However, most documentation of SoCs specified at early stages is written in natural language [2]. This brings some challenges for engineers, who have to carefully read hardware specifications in order to build “the right” design model (e.g., using VHDL or HDL, among others) or to determine the “correct” assertions that need to be verified (e.g., expressed in CTL or SystemVerilog).

It is this context that Natural Language Processing (NLP) can play an interesting role in assisting engineering tasks and streamlining design and verification activities. NLP techniques have been recently adopted in many engineering activities for analyzing textual documentation and reducing development costs by improving the overall quality of end products. Examples of disciplines currently taking advantage of NLP are: software development, medical advising, recommender systems, psychological assessments, among others. We believe that the hardware community can benefit from NLP techniques for automating hardware design processes. Some researchers have already explored these technologies by analyzing hardware descriptions, specifications and comments for diverse tasks, such as: transforming informal sentences into formal models [3], generating VHDL snippets out of natural language [4], deriving assertions from requirements [1], and generating verification properties expressed in formal languages [5], among others. However, these works have only used a fraction of technologies available today for understanding written text and we think there are many opportunities left unexplored.

In this article, we report our experiences of applying NLP techniques for understanding textual documentation produced as a byproduct of a software development process. Our research team has been working for the last five years trying to find deficiencies in requirements specifications and software architecture documents, tackling problems like uncovering latent concerns in use case specifications [6], identifying duplicate functionality in textual requirements [7], and recovering traceability links between requirements and architecture documentation [8]. Our position is that the tooling developed to this end, such as the text processing infrastructure and the information extraction mechanisms can be used either directly or with some adaptations to analyze hardware specifications.

The rest of the paper is organized into 3 sections. Section II introduces the text analyses infrastructure to deal with software development problems, consisting of an NLP pipeline and two querying languages. Section III presents three applications we have successfully implemented on top of that infrastructure. Section IV reviews existing work for automating hardware design activities using NLP techniques. Finally, Section V gives the conclusions and discusses some opportunities for advancing the state-of-the-art in hardware automation.

II. Reusable Assets for Analyzing Textual Specifications

In the last years, our research group has explored the application of modern NLP techniques to help software analysts at early development stages. This endeavor led to the instantiation of an scalable text processing architecture called UIMA, integrating diverse NLP techniques for understanding

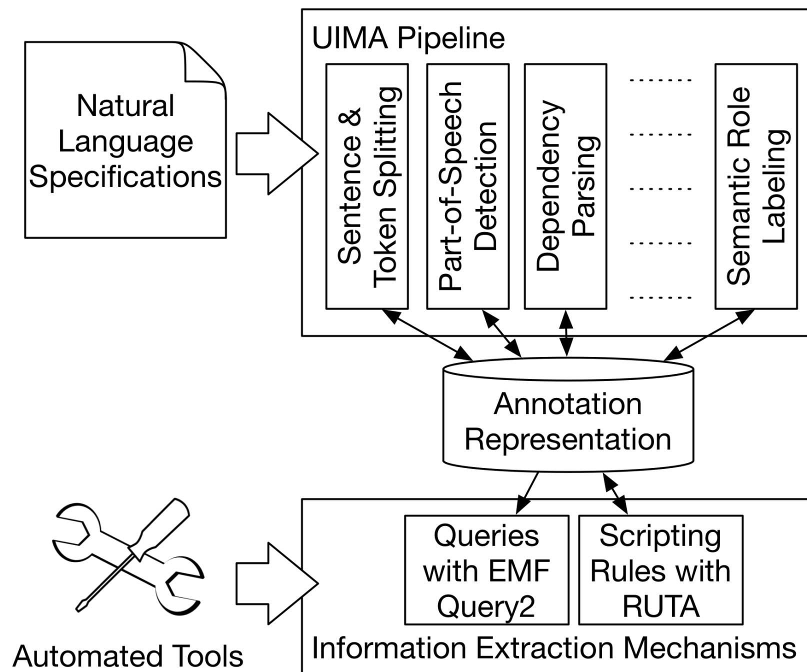

Figure 1. Schema of our NLP infrastructure

textual software artifacts. Furthermore, we successfully integrated *information extraction* techniques for taking advantage of the results of text processing modules, such as an SQL-like and a rule-based scripting language called EMF Query2 and RUTA, respectively. Figure 1 sketches the components for analyzing textual documents and developing automated tools.

A. Analyzing Text with the UIMA Pipeline

The first step we took towards the analyses of documents produced in SE was defining an extensible but yet powerful pipeline of text processing modules. To this end, we leveraged the UIMA framework[1] [6], [7]. UIMA is an extensible architecture for building analytic applications that process unstructured information to discover relevant knowledge. Among the UIMA features that support semantic search, we can mention: detection of the language of a specific document (e.g., English, Spanish), language-dependent linguistic processing (e.g., tokenization, lemmatization, etc.), and the discovery of entities and relations in the text. Still, UIMA does not provide a text processing pipeline out-of-the-box. Instead, the framework only defines the structure and communication protocols for text analytic modules, leaving the implementation of individual modules up to the developers. For an in-depth discussion of UIMA, the reader is referred to [9].

Our instantiation of UIMA supports the linguistic analysis of both textual use cases and architectural documentation. Furthermore, we implemented a set of modules that make extensive use of the annotation mechanisms provided by UIMA. An annotation identifies and labels (i.e., annotates) a specific region of a text document. For instance, an annotation can label a noun as "object" or a verb as an "action" in a sentence. The building blocks of a UIMA application are the so-called *annotators*. An annotator is a module that iterates over an artifact (e.g., a textual document) in order to discover new annotation types based on existing ones, and updates a shared representation structure. Different arrangements of annotators can be configured to produce an end-to-end analysis.

Figure 2 shows a linguistic analysis of one use case step excerpted from a requirements specification. Each rectangle is an annotation that points either to the base text or to other annotations. The annotations of level 1 correspond to tokens. For example, the rectangles labeled as "Token" identify each word within the use case step, including information such as if those words are verbs ("computes") or nouns ("system" and "report"), among other properties. The annotations of levels 2 and 3 provide richer information, such as the predicate structure. In the figure, the rectangles labeled as "Predicates" and "Argument" recognize the semantic role of a syntactical structure, whereas the rectangles labeled as "DomainAction" recall the intention of the action of the sentence.

The first part of the pipeline consists of standard NLP annotators, namely: i) sentence splitting, for identifying sentence boundaries; ii) token splitting, for extracting tokens from sentences; iii) stopwords detection, for discarding irrelevant tokens; and iv) stemming, for reducing each token to its lexical root. In addition, Part-of-Speech (POS) tagging is used for identifying the linguistic category of each token (e.g., noun, verb, adjective, participle, pronoun, preposition, etc.). Implementations of these five techniques are already

[1]http://uima.apache.org/

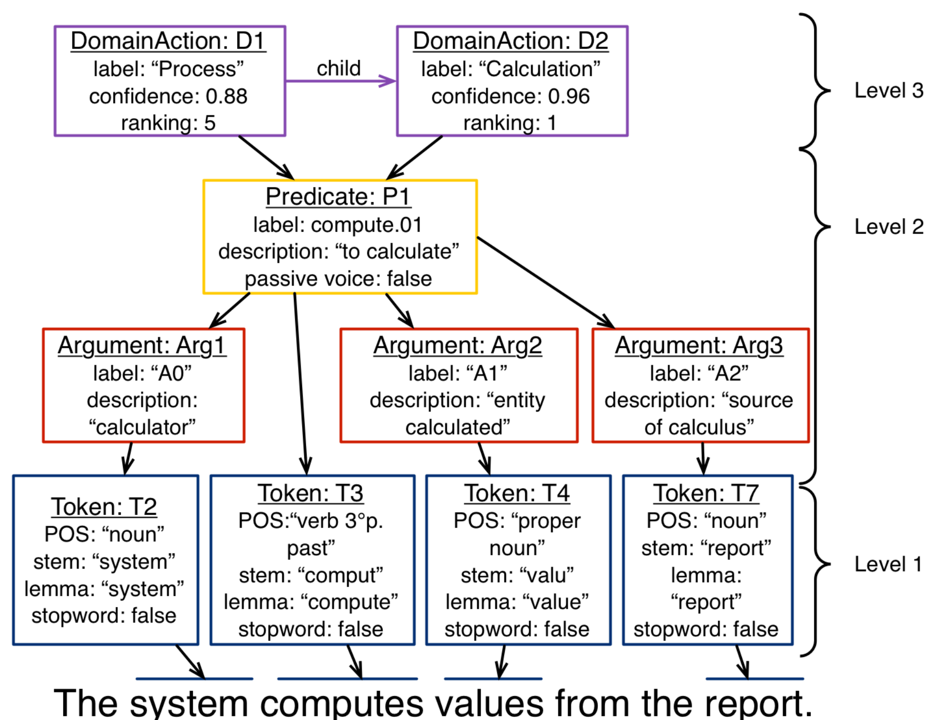

Figure 2. Linguistic Analyses made with the UIMA Framework

available. Well-known packages include: OpenNLP[2], Stanford CoreNLP[3], and Mate-Tools[4]. For example, OpenNLP provides algorithms for sentence splitting, token splitting and POS tagging. Stanford CoreNLP provides similar algorithms but also supports dependency parsing.

In the second part of the pipeline we included more advanced NLP tasks from the field of computational linguistics, such as lemmatization, dependency parsing and semantic role labeling (SRL) [10]. Lemmatization is a technique that helps to identify the morphological root of a token (instead of trimming words like in stemming). Dependency parsing, in turn, focuses on the grammatical structure by identifying syntactic relationships between words. For our purposes, lemmatization and dependency parsing are prerequisites for applying SRL. This last task is instrumental in our pipeline, because SRL recognizes the semantic arguments associated with a predicate (or verb) and classifies these arguments into specific roles (e.g., the agent, the patient, the manner, the time, the location, etc.). The predicate annotation in Figure 2 (level 2) is an example of SRL (see Predicate P1). Typical NLP applications using SRL include: question answering, machine translation, or information extraction. Moreover, SRL allow us to derive more complex abstractions, such as recurrent interactions in use cases that we called "domain actions". Existing packages, such as Mate-Tools, provide algorithms for lemmatization, dependency parsing, and SRL.

[2]http://opennlp.sourceforge.net/projects.html
[3]http://nlp.stanford.edu/software/corenlp.shtml
[4]http://code.google.com/p/mate-tools/

B. Extracting Information with EMF/Query2 and RUTA

After gathering diverse types of meta-information from textual documents by using NLP techniques, we needed to have some sort of mechanism to consume and filter the data in order to simplify the analysis of information and tool development. For this reason, we explored two alternatives. The first idea was to complement the UIMA pipeline with searching queries by using a special language to browse/navigate the annotations. The second idea was to incorporate a rule-based engine able to evaluate conditions in the annotation schema and execute actions as a result (for example, creating new annotations or modifying existing ones).

The search queries implemented in [6] are built on top of the EMF Query2[5] project, which serves as an SQL-like language for searching through EMF models. The query syntax is simple to understand and powerful enough to express complex recovery searches. A query is composed of three parts, namely: the selection, the origin and the conditions. The selection, expressed in a query with the keyword "select", tells the query engine which information we want to obtain as a result. The origin, expressed with the keyword "from", defines the annotations the engine should browse to obtain the results. Finally, the conditions, expressed with the keyword "where", describe the clauses to filter the data according to the properties in the annotations. Conditions can be seen as predicates that evaluate the annotations.

[5]http://www.eclipse.org/modeling/emf/downloads/?project=query2

```
select S from
    [#Sentence#] as S,
    [#Token#] as T
where for T(
    stem = 'commun' or lemma = 'interaction' or
    lemma = 'internet' or lemma = 'external' or
    lemma = 'separate' or lemma = 'online' or
    lemma = 'server' or lemma = 'offline'
    or stem = 'connect'
)
where T.begin > S.begin where T.end < S.end
```

Figure 3. NLP-based query

In addition, we have developed an abstraction layer that allows analysts to seamlessly incorporate UIMA-generated annotations in the queries. This allow us to avoid inputting complex namespaces for the annotations and figuring out the correct domains on the fly. For instance, instead of having to enter "select S from edu.isistan.uima.unified.typesystems.nlp.Sentence ..." we can just type "select S from [#Sentence#]". Figure 3 shows an example of an NLP-enriched query for finding special concepts related to distribution and connectivity concerns. The query searches for sentences containing tokens whose stems or lemmas match a particular keyword.

Alternatively, the rule-base engine is basically the integration of the existing NLP pipeline and its annotations with the UIMA RUTA (Rule-based Text Annotation) project[6]. RUTA is an imperative rule language extended with scripting elements. The main idea behind a rule is to define a pattern of annotations with additional conditions, and if the pattern is matched, then be able to execute a set of actions.

A rule is composed of four parts, namely: a matching condition, an optional quantifier, a list of inner conditions, and a list of actions. The matching condition is typically an NLP annotation by which the engine narrows the scope to the covered text. The quantifier is essentially describes if it is necessary to match the annotation and how often. Inner conditions make possible to check additional properties of the annotations that need to be fulfilled, such as their properties. Actions define the consequences of the rule, which commonly end up being the creation of new annotations or the modification of existing annotations.

Figure 4 shows an example of a RUTA rule for finding a modifiability architectural tactic in design documents. In the rule, the first line declares a temporal annotation called *ReduceCoupling*. Next, we load a number of words associated to the reduce coupling tactic and store them in a list. Then, we mark the text with the *ReduceCoupling* annotation for those words present in the list. Then, we look for sentences that contain those words and create a new *DesignDecision* annotation. Finally, we remove the temporal annotations.

[6]https://uima.apache.org/ruta.html

```
DECLARE ReduceCoupling;
STRINGLIST relatedWords = { encapsulate, API, application programming interface, intermediary, wrapper, restrict, dependencies, refactor, abstract common services, bean, ejb };
Document{ -> MARKFAST(ReduceCoupling, relatedWords, true) };
Sentence{ CONTAINS(ReduceCoupling) -> CREATE( DesignDecision, "kind" = "Modifiability", "typex" = "reduce coupling tactic") };
ReduceCoupling{ -> UNMARK(ReduceCoupling) };
```

Figure 4. NLP-based rule

III. Addressing Software Engineering Problems with Automated Tools

The text processing infrastructure and the searching languages presented before were employed as basis for the development of automated tools for SE problems, such as *REAssistant* [6], *ReqAligner* [7], and a tool that retrieves design decisions [8] for improving the recovery of traceability links between requirements and the architecture. The following sub-sections give more details about each of the applications.

A. App#1: Revealing Crosscutting Concerns

The first application where we employed our NLP infrastructure was devoted to the analysis of use cases. Use cases normally have textual specifications that describe the interactions between the system and external actors. However, since use cases are specified from a functional perspective, concerns that do not fit well this decomposition criterion are kept away from the analysts' eye and might end up intermingled in multiple use cases. These *crosscutting concerns* (CCCs) are generally relevant for analysis, design and implementation activities and should be dealt with from early stages. Unfortunately, identifying such concerns by hand is a cumbersome and error-prone task, mainly because it requires a semantic interpretation of textual requirements.

To ease the analysis of CCCs, we developed an automated tool called *REAssistant* that is able to extract high-level properties and localize quality-attribute information from textual use cases to reveal candidate CCCs, helping analysts to reason about them before making important commitments in the development. Analysts can define concern-specific queries in terms of the NLP annotations to search for CCCs. The queries take advantage of use-case-specific annotations called "Domain Actions" to extract not only CCCs but also their crosscutting relations (i.e., requirements affected by CCCs). Domain actions are a taxonomy of domain-neutral classes applicable to use cases that abstract common types of interactions, such as input/output operations, information transmission, or data handling in the system, among others. The *REAssistant* tool is implemented as a set of Eclipse plugins that provide special views for visualizing CCCs at different levels of granularity.

REAssistant comes loaded with a predefined ruleset of CCCs, but can be easily customized by analysts. The queries

DIRECT QUERY #1
select *S* **from** [#*Sentence*#] **as** *S*, [#*Token*#] **as** *T*
where for *T* (*lemma* = 'response' **or** *lemma* = 'second' **or** *lemma* = 'time' **or** *stem* = 'delay' **or** *lemma* = 'throughput' **or** *lemma* = 'latency' **or** *lemma* = 'deadline')
where *T.begin* >= *S.begin* **where** *T.end* <= *S.end*
INDIRECT QUERY #2
select *S* **from** [#*Sentence*#] **as** *S*, [#*DomainAction*#] **as** *DA*
where for *DA* (*label* = 'Calculation')
where *DA.begin* >= *S.begin* **where** *DA.end* <= *S.end*
INDIRECT QUERY #3
select *S* **from** [#*Sentence*#] **as** *S*, [#*DomainAction*#] **as** *DA*, [#*Token*#] **as** *T*
where for *DA* (*label* = 'Process')
where for *T* (*lemma* = 'result' **or** *lemma* = 'value')
where *T.begin* >= *S.begin* **where** *T.end* <= *S.end*
where *DA.begin* >= *S.begin* **where** *DA.end* <= *S.end*

Figure 5. Queries for Searching a Performance Concern

codify knowledge about concerns and how they relate semantically to natural language expressions, and were defined by experienced analysts to cover a wide range of software domains. There are two types of queries: i) *direct queries*, responsible for detecting a CCC; and ii) *indirect queries*, for detecting domain actions that are potentially related to that concern. Direct queries are focused in localizing explicit references to a particular CCC, for example, the word "server" or "database". Complementary, indirect queries are focused in finding more subtle associations that come from a semantic interpretation of the use cases. Figure 5 illustrates a PERFORMANCE rule composed of three queries. Query #1 would find parts of the text related to PERFORMANCE through the analysis of token lemmas such as "response" and "second", similarly to keyword-based approaches. Queries #2 and #3 make use of domain actions to reveal indirect impacts, looking for actions such as "calculation" and "process". For more information about the architecture of the tool, the NLP pipeline and the concern ruleset, the reader is referred to [6]. In this publication, we also report on the results of an empirical evaluation of *REAssistant* with three case-studies.

B. App#2: Identifying Duplicate Functionality

A second application of our NLP infrastructure to use cases is presented in [7]. In this publication, we tackled the problem that in spite of existing guidelines for writing use cases, industrial use cases do not often meet the standards of what it is considered a "good" use case model and often exhibit signs of unwanted/unnecessary redundancy. Duplicating functionality is the action of repeating the description of some interactions between the system and actors [11]. Several factors contribute to this phenomenon, such as: too many requirements, inexperienced analysts, changing requirements, or copy/paste abuse, among others. Although duplication is not always a quality defect, and it might be there for the sake of readability of non-technical stakeholders, the lack of modularity and abstraction can have a profound (negative) effect on the developers conducting activities such as effort estimation, project planning , architectural design, change impact analyses and evolution management. Unfortunately, finding duplicate functionality in multiple specifications is a cumbersome, arduous and error-prone activity for the analysts. For this reason, we developed a tool called *ReqAligner* that helps analysts to identify duplicate behaviors in use cases and provides guidelines to mend those defects (duplications) in an automated fashion. Similarly to *REAssistant*, *ReqAligner* leverages on a domain-specific classifier of *semantic actions* and, based on such knowledge, employs a *sequence alignment* technique for finding suspiciously similar functionalities in the use cases. Moreover, the tool is also able to suggest UML relationships that can help analysts to remove duplications and ultimately improve the overall requirements model.

Internally, *ReqAligner* assembles sequences (or chains) of domain actions after the semantic analyses of use-case steps is completed. The resulting sequences synthesize a summarized view of a use case scenario, which enables the comparison of scenarios based on their overall semantics (that is, according their intention and meaning). At this point, the different sequences are processed and matched (pairwise) by means of a customized *sequence alignment* (SA) technique [12]. The sequence alignment basically compares the constituent elements of two different sequences (also referred to as symbols or characters in the bioinformatics jargon) and aims at finding a sub-chain that maximizes a given similarity function. When two sequences go under analysis, the comparison depends on two predefined parameters [12]: the substitution matrix and the penalization table, which we adapted to the use cases domain. On one hand, the substitution matrix defines the similarities between the use-case steps (according to their intention). On the other hand, the penalization table defines adjustments (i.e., penalties) the aligner should apply to the similarity score when there are gaps between the matched sequences. If some textual portions of a pair of use cases were successfully aligned, it means that we have one or more candidate duplications in the functional specification. Based on these results, the approach applies simple heuristics to determine the kind of problem detected and the most likely UML relationship that can help to refactor (and thus, improve) the use cases. For more information about the tool, the adaptation of the sequence alignment technique to use cases, and the refactoring heuristics, the reader is referred to [6]. In this publication, we also report on the results of an empirical evaluation of *ReqAligner* with five publicly available case-studies that produce promising results.

C. App#3: Retrieving Traceability Links

We are also working on a solution that takes advantage of NLP pipeline and the scripting language for recovering traceability links between architectural documentation and requirements documentation. We are interested in this problem because in some software domains the satisfaction of quality attributes is critical, and the consequences of a failure may

cause financial losses or endanger human lives (e.g., automotive systems, aircraft control systems, spaceships controllers and medical machinery, among others) [13]. These systems often have to go through rigorous controls to ensure that the implementation fulfills the requirements. In that context, having traceability records between the requirements and the architecture is a must for assessing software quality [14]. Unfortunately, these kind of traces are hard to recover and maintain. One impediment is the ambiguities and complex semantics of natural language, which is commonly used to specify requirements and describe the architecture. Another setback is the size of the documents, which contain irrelevant information that hinder the discovery of traces.

Essentially, our solution initially finds portions of the documents related to quality attributes (e.g., crosscutting concerns and design decisions) and then uses a *latent semantic analysis* (LSA) technique for identifying potential traces. To filter the documents, we take advantage of the NLP results and RUTA to codify rules able to spot design decisions in the text using a well-known taxonomy of architecture patterns and tactics [15] (see Figure 4). Then, using the output of *REAssistant* for use case specifications, we transform the resulting sentences into a vector space model. Afterwards, we use LSA to recognize potential traces by comparing the sentences pairwise and using the cosine distance to compute their similarity. We are currently writing an article to report the design of the tool and some experiments carried out in three case-studies.

IV. Related Work

Using NLP in hardware documentation can bring several benefits to the building process of SoC-based designs and improve the quality of Application Specific Integrated Circuits (ASIC). On one hand, text analytics modules are helpful to translate hardware requirements described in natural language into design models that can be used as a starting point by engineers. On the other hand, several NLP techniques can help hardware designers to derive constraints and assertions out of comments and textual specifications in order to automate the verification of the design via model checking. The work of Granacki et al. was one of the first attempts to use NLP to generate partial hardware designs by analyzing natural language specifications [3]. Their idea consisted of identifying a set of fixed concepts with pattern matching techniques, and then map those concepts to pre-defined design structures. Another interesting proposal given by Cyre et al. aims to the derivation of VHDL (VHSIC Hardware Description Language) snippets from informal hardware descriptions [4]. They presented a proof-of-concept prototype that parses sentences written in English and builds a conceptual graph for them (including processes, conditions and signals). Then, the graph is analyzed and a Process Model Graph and VHDL code is generated. However, this idea was never implemented in an actual assistance tool.

More recent publications have centered their attention in streamlining the verification of hardware designs using NLP. In [1], Soeken et al. proposed an advanced technique for extracting assertion-type information from a system specification written in English by exploiting grammatical similarities of the sentences. Their research is an attempt to reduce the effort of defining hardware assertions by hand. This kind of assertions are commonly used in Assertion Based Verification (ABV) methods to verify complex software systems, for instance, using languages like SystemVerilog. At the outset, the technique classifies the sentences into high and low abstraction level assertions and filters the former because these do not have enough detail. The remaining sentences are then grouped into clusters based on their grammatical structure and the semantics of the words. Particularly, the technique makes extensive use of a typed dependency representation for comparison purposes and the SPARQL querying language for extracting information from the sentences. Each of the clusters is then converted into formal properties (i.e., assertions) by using transformation rules. The idea is that each cluster can be described by an archetypical SystemVerilog Assertion (SVA) template, and rules have to only fill up the template. This solution requires to generate a small number of SVA to automatically generate a full set of assertions. Harris et al. also explored the use of NLP for automating verification tasks in the hardware design cycle [5]. They developed a formal attribute grammar for translating inline comments of HDL (Hardware Description Language) code specifications to syntactically-correct verification properties in CTL (Computation Tree Logic). These properties can be directly used to verify the design via model checking techniques. An attribute grammar allows grammatical symbols to be replaced by an attribute which is then evaluated in terms of the attributes of other symbols. The customized grammar used in this work contains over a hundred unique rules to recognize things like: top-level attributes, implications, wait until semantics, signal values, signal assignments, signal names, among others productions. The sentences are gathered from HDL code comments and parsed using a descent parser and the attribute grammar. Then, the resulting parse tree is evaluated for creating a set of CTL properties.

In summary, despite some interesting tools for automating hardware design exist, we think that the underlying NLP techniques used are rather basic and can be enhanced with newer algorithms. An important drawback we noticed is that most tools are developed in a ad-hoc fashion, making the reuse and evolution of the underlying techniques harder. We believe that these limitations can be overcome by using a text infrastructure like ours for SE, and that existing text modules can be adapted to hardware specifications with little work.

V. Discussion and Open Challenges

Many essential tasks in modern hardware development, such as hardware design and testing, can be largely enhanced by using automated tools able to assist engineers. Some initial research has been done by integrating smart natural language analyzers, achieving promising results and leading to effort/time savings. However, we believe that existing approaches still have room for improvements by using more

advanced and semantic-aware NLP modules. Along this line, using an extensible and configurable text processing infrastructure like ours may help to create better tools for processing hardware specifications and deriving useful models. Some direct advantages are the following. First, different types of hardware documentation can be analyzed by reconfiguring the NLP pipeline, allowing to process code comments or informal requirements with the same modules. Second, the retrieval mechanisms provided by RUTA are powerful enough to replace complex techniques like attribute grammars and parse trees and accomplish the same goals. Third, enhancing the results produced by an automated tool that applies searching queries or rules is straightforward because the languages are easy to understand and modify.

Additionally, we think particular tasks of the related work might be enhanced with our NLP pipeline and the information extraction mechanisms developed for software artifacts. Some ideas worth investigating include: (i) incorporating special modules for identifying time expressions and their relations[7] for enriching the analysis of temporal constraints, (ii) using anaphora resolution modules for avoiding problems with pronouns and referring entities (only understood by context) commonly used in informal descriptions, (iii) using robust parsing techniques such as finite state transducers able to tolerate syntactic and grammatical mistakes in the text, (iv) using RUTA for filling SVA templates and producing assertions, and (v) converting the attribute grammar to RUTA rules for improving its evolution over time.

Overall, we are confident that the hardware community can take advantage of the knowledge of NLP techniques learned in other disciplines, such as Software Engineering. Especially, we think that they should exploit the synergies with other researchers so as to allow the tools to improve rapidly and achieve better performance.

[7] http://nlp.stanford.edu/projects/time.shtml

References

[1] M. Soeken, C. Harris, N. Abdessaied, I. Harris, and R. Drechsler, "Automating the translation of assertions using natural language processing techniques," in *Specification and Design Languages (FDL), 2014 Forum on*, vol. 978-2-9530504-9-3, Oct 2014, pp. 1–8.

[2] I. Harris, "Extracting design information from natural language specifications," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, June 2012, pp. 1252–1253.

[3] J. J. Granacki, A. C. Parker, and Y. Arena, "Understanding system specifications written in natural language," in *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, ser. IJCAI'87, vol. 2. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, pp. 688–691.

[4] W. R. Cyre, J. Armstrong, M. Manek-Honcharik, and A. J. Honcharik, "Generating vhdl models from natural language descriptions," in *Proceedings of the European Design Automation Conference*, ser. EURO-DAC '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 474–479.

[5] C. Harris and I. Harris, "Generating formal hardware verification properties from natural language documentation," in *Semantic Computing (ICSC), 2015 IEEE International Conference on*, Feb 2015, pp. 49–56.

[6] A. Rago, C. Marcos, and A. Diaz-Pace, "Assisting requirements analysts to find latent concerns with REAssistant," *Automated Software Engineering*, June 2014.

[7] ——, "Identifying duplicate functionality in textual use cases by aligning semantic actions," *Software and Systems Modeling*, August 2014.

[8] ——, "Uncovering quality-attribute concerns in use case specifications via early aspect mining," *Requirements Engineering*, vol. 18, no. 1, pp. 67–84, March 2012.

[9] D. Ferrucci and A. Lally, "UIMA: an architectural approach to unstructured information processing in the corporate research environment," *Natural Language Engineering*, vol. 10, no. 3-4, pp. 327–348, 2004.

[10] M. Palmer, D. Gildea, and N. Xue, *Semantic Role Labeling*, ser. Synthesis Lectures on Human Language Technologies. Morgan & Claypool, 2010.

[11] A. Ciemniewska and J. Jurkiewicz, "Automatic detection of defects in use cases," Master's thesis, Poznan University of Technology - Faculty of Computer Science and Management - Institute of Computer Science, 2007.

[12] A. Polanski and M. Kimmel, *Bioinformatics*. Springer, 2007. [Online]. Available: http://books.google.com.ar/books?id=oZbR3GEdmVMC

[13] J. Cleland-Huang, O. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability: Trends and future directions," in *36th ACM/IEEE International Conference on Software Engineering (ICSE'14)*, ser. Workshop on the Future of Software Engineering, Hyderabad, India, June 2014, pp. 55–69.

[14] M. Mirakhorli and J. Cleland-Huang, "Detecting, tracing, and monitoring architectural tactics in code," *IEEE Transactions on Software Engineering*, 2015.

[15] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., ser. SEI Series in Software Engineering. Addison-Wesley Professional, October 2012.