RDF Doctor: A Holistic Approach for Syntax Error Detection and Correction of RDF Data

Ahmad Hemid[1], Lavdim Halilaj[2], Abderrahmane Khiat[1] and Steffen Lohmann[1]
[1]*Fraunhofer IAIS, Sankt Augustin, Germany*
[2]*Robert Bosch Corporate Research, Stuttgart, Germany*

Keywords: RDF, Error Detection, Error Correction, Syntax Validation.

Abstract: Over the years, the demand for interoperability support between diverse applications has significantly increased. The Resource Definition Framework (RDF), among other solutions, is utilized as a data modeling language which allows for encoding the knowledge from various domains in a unified representation. Moreover, a vast amount of data from heterogeneous data sources are continuously published in documents using the RDF format. Therefore, these RDF documents should be syntactically correct in order to enable software agents performing further processing. Albeit, a number of approaches have been proposed for ensuring error-free RDF documents, commonly they are not able to identify all syntax errors at once by failing on the first encountered error. In this paper, we tackle the problem of simultaneous error identification, and propose RDF-Doctor, a holistic approach for detecting and resolving syntactic errors in a semi-automatic fashion. First, we define a comprehensive list of errors that can be detected along with customized error messages to allow users for a better understanding of the actual errors. Next, a subset of syntactic errors is corrected automatically based on matching them with predefined error messages. Finally, for a particular number of errors, customized and meaningful messages are delivered to users to facilitate the manual corrections process. The results from empirical evaluations provide evidence that the presented approach is able to effectively detect a wide range of syntax errors and automatically correct a large subset of them.

1 INTRODUCTION

The growing size of data represented in RDF (Zeng et al., 2013) requires scalable and efficient tools to ensure the correct processing of RDF documents with respect to syntax. Ontologies, as a one of the cases where RDF is used as a modeling language, usually are built in a collaborative environment with the involvement of several stakeholders. When trying to comprehend the concrete encoding or perform a bunch of modifications or insertions at the same time, ontology engineers often use plain text editors (Petersen et al., 2016) where they can easily introduce a number of syntax errors after each modification.

One fundamental pre-condition to consume ontologies encoded in RDF documents by different stakeholders is that they are syntactically correct. Existing tools used for syntax checking assert that the input is error-free. If a particular ontology has one or more errors, these tools naturally stop parsing and report only the first error to the user with some additional information about the error, such as an error message, row and column number, and where the error occurred. This process becomes tedious for engineers if not all of the errors are detected at once: each time the tool detects an error, the engineer has to correct it and reprocess the corrected ontology version. Next, the used tool checks again for syntax errors and reports when any other errors encountered. Moreover, it might happen that, during the correction phase, engineers introduce new errors. Therefore, this procedure is time-consuming, error-prone, and tedious to all engineers involved in the ontology development process.

Over the years, several approaches have been presented for parsing RDF documents and finding syntactic errors. Well-known tools, such as the Jena API (McBride, 2002), the RDF4J API (RDF,), or the N3 Parser (Verborgh,), implemented as Java-based toolkits or as a Javascript-based parser, respectively, are used to process RDF in N-Triple and Turtle formats (or in other serialization formats). Commonly, these tools utilize an exception handling mechanism to catch any potential syntax error, and are able to rec-

Hemid, A., Halilaj, L., Khiat, A. and Lohmann, S.
RDF Doctor: A Holistic Approach for Syntax Error Detection and Correction of RDF Data.
DOI: 10.5220/0008493205080516
In *Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2019)*, pages 508-516
ISBN: 978-989-758-382-7

ognize only one error at a time.

In this paper, we present RDF-Doctor, a comprehensive approach for error detection and correction in RDF documents. The motivation of this work was mainly encouraged by the tremendous RDF data generation and usage in both Turtle and N-Triples serialization formats, respectively. RDF-Doctor is capable of detecting an exhaustive number of syntactic errors and automatically correct a subset of them. Although those two formats are used as study cases in this research, the approach can be easily extended for supporting other serialization formats by specifying the respective grammar.

RDF-Doctor is fully operational and is currently integrated within the VoCol platform[1]. VoCol (Halilaj et al., 2016b) leverages the fundamental principles of Git as a version control system to support ontology development in distributed scenarios. RDF-Doctor can be used a standalone tool as well, and the source code is openly available at https://github.com/ahemaid/RDF-Doctor.

The main contributions of this work are: 1) definition of a set of grammar rules with the objective of covering an exhaustive list of syntactic errors; 2) enabling the continuation of the parsing procedure after errors occurrence; 3) identifying multiple errors in the same line or subsequent statements; 4) automatic correction of a subset of errors; and 5) improving conflict resolution via user-friendly messages.

This paper is organized into the following sections: Section 2 presents related work summarizing relevant approaches for syntax checking. Section 3 provides a detailed description of our approach. Section 4 describes a scenario for error detection and correction by RDF-Doctor. The approach is evaluated in various scenarios in Section 5. Section 6 concludes our work and provide an outlook for potential extensions.

2 RELATED WORK

In this section, we discuss the related work to our problem, i.e., research that has been realized in the field of RDF syntax parsing and checking. During our literature review, we focused on the following three aspects: 1) parsing tools for different RDF serializations; 2) types of error messages generated after error encountering; and 3) the error recovery.

[1]https://github.com/vocol/vocol

Table 1: Comparison between RDF-Doctor and other RDF syntax checking tools.

Feature	Jena (McBride, 2002)	ShEx.js (Tolle, 2000)	VRP (Prud'hommeaux et al, 2014)	IDLab Validator IDLab, 2019	RDF Doctor
Multiple error detection per scan	✗	✗	✓	✗	✓
Error correction	✗	✗	✗	✗	✓
User-friendly error messages	✓	✗	✓	✓	✓
Grammar based approach	✓	✓	✓	✓	✓

2.1 Parsing Tools

Several tools for validating RDF documents use the Another RDF Parser (ARP) parser of Jena (McBride, 2002) such as W3C RDF validation tool (Prud'hommeaux,), Jena RDF toolkit.

These tools can commonly detect only the first error while consecutively parsing input from the start point to the end. Therefore, ontology engineers are struggling whilst debugging their RDF documents, and need alternative tools that could be more helpful. To the best of our knowledge, only the Validating RDF Parser (VRP) (Tolle, 2000) proposed by K. Tolle can detect multiple errors at the same time. However, this work is limited only to RDF/XML serialization. Other tools such as ShEx.js (Prud'hommeaux et al., 2014), Jena API (McBride, 2002), RDF Validator (Myb,), N3Parser (Verborgh,), IDLab Validator (IDL,), and TurtleEditor (Petersen et al., 2016) are fault-intolerant, therefore not able to detect multiple errors simultaneously.

2.2 Types of Error Messages

Releasing user-friendly and meaningful error messages is of a great benefit to help the user to easily identify and correct the errors. Practically, parsing tools under the Shape Expressions approach, like ShEx.js (Prud'hommeaux et al., 2014), show less expressive and unfriendly error messages. On other hand, tools which utilize an ARP-parser-dependable approach like Jena API (McBride, 2002), RDF Validator (Myb,) or an N3-parser-dependable approach, like N3Parser (Verborgh,), IDLab Turtle Validator (IDL,) and TurtleEditor (Petersen et al., 2016) present more expressive and user-friendly error messages including its location.

2.3 Error Recovery Approaches

Automatic error recovery is a crucial feature in ontology development process as well as for RDF data reuse (Halilaj et al., 2016a). Our survey of research

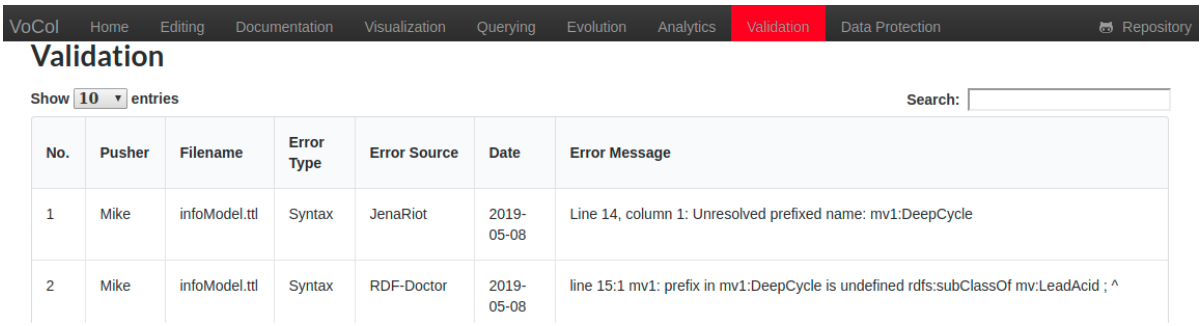

Figure 1: Example of a generated report for Turtle syntax validation by RDF-Doctor as an integrated component within VoCol platform (Halilaj et al., 2016b).

papers and tools related to auto-correction of RDF syntax errors shows that such a feature does not exist either theoretically or practically. Several approaches already exist in software development to help on automatic error correction. Balachandran reports about a Review Bot tool (Balachandran, 2013) that can automatically review a source code using static analysis output and correct encountered syntax errors using the parse tree. This approach is applicable in our case. However, RDF-Doctor corrects a common subset of syntax errors by matching these errors with predefined error messages.

2.4 RDF-Doctor vs. Other Tools

Table 1 summarizes this section by providing a comparison of RDF syntax checking tools including RDF-Doctor. Obviously, it can be seen that RDF-Doctor is the sole tool that provides an error correction for RDF syntax errors, whereas others are not. Additionally, both RDF-Doctor and VRP (Tolle, 2000) are supporting multiple error detection while the former currently supports Turtle and N-Triples and is willing to support more serializations in the future work; meanwhile, the latter is designed to handle only RDF/XML serialization and no further development to extend the tool can be identified. In common with other tools, RDF-Doctor rises user-friendly error messages and has a grammar based approach upon which its parser was auto-generated.

3 RDF-Doctor

In this section, we present RDF-Doctor[2], an approach with the objective of the realization of a comprehensive parser for syntax validation and recovery.

[2]All study material of RDF-Doctor as well as as standalone version including grammar rules, can be found at https://github.com/ahemaid/RDF-Doctor.

RDF-Doctor is able to identify more than one syntax error at the same time and correct a subset of them. In addition, it provides meaningful and customisable error messages to facilitate the manual correction. Currently, RDF-Doctor is integrated within the VoCol platform (see Figure 1), an integrated environment that supports the development of vocabularies using Version Control Systems[3](Halilaj et al., 2016b). In the following, we describe the main components of RDF-Doctor.

3.1 ANTLR for Parser Generation

The core component inside RDF-Doctor is the ANTLR framework. It is used to automatically generate a parser based on our developed grammar, including predefined error production rules to match sequences of tokens that contains RDF syntax errors. Due to lack of availability of grammar that defines error production rules, we created RDF-Doctor grammar based on Turtle serialization. Thus, RDF-Doctor can parse both Turtle and N-Triple since the latter can be considered as a special case of the former.

The ANTLR framework has several important features: 1) a given grammar can be equipped with error production rules and when a sequence of tokens match such rules, the parser will fire a notification error; 2) it uses a parse tree to parse input tokens and the view of this parse tree can be delivered as one of the outputs to the user at the end of the parsing process; and 3) it supports the auto-generation of parsers in different programming languages.

3.2 Errors Detection and Recovery

Figure 2 illustrates the main steps of a typical workflow to detect and recover syntax errors of RDF documents. These steps are described in the following:

[3]https://github.com/vocol/vocol

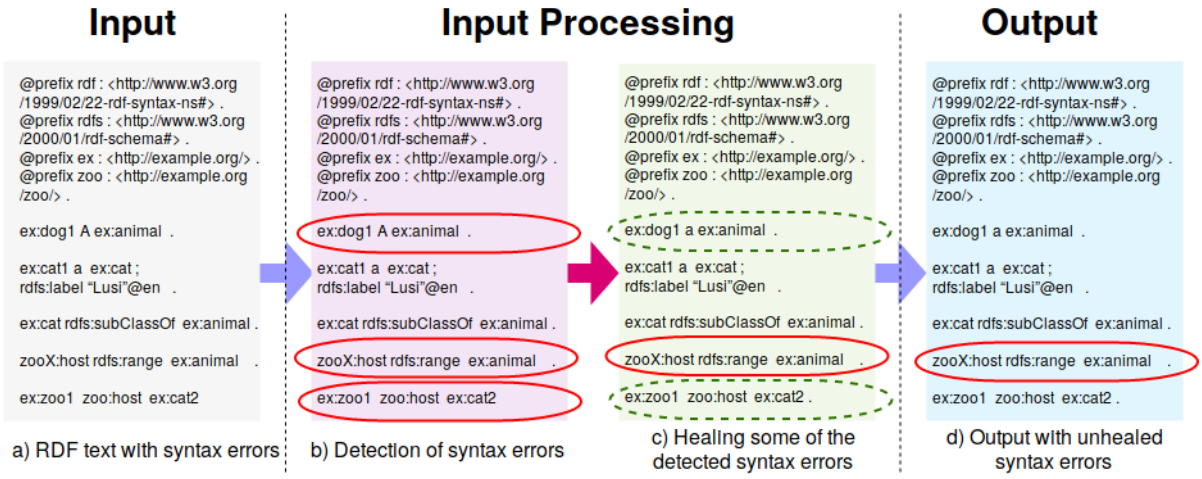

Figure 2: RDF-Doctor workflow. The user modifies an RDF file, then sends it to RDF-Doctor for parsing. RDF-Doctor receives the file as input however, if the file is large, RDF-Doctor splits it into multiple chunks. Next, each chunk or the original file is parsed individually. In case of encountering any syntax error, RDF-Doctor tries to recover a subset of detected errors, whenever the automatic error recovery feature is enabled. Finally, RDF-Doctor outputs a parse tree, an error report, a correction report, and the RDF file after error recovery.

3.2.1 Reading RDF File

During this step, RDF-Doctor receives an RDF file as input. In case that the file is too large (e.g., more than 1 million triples), then it is segmented into two or more chunks based on the number of triples. Each chunk is parsed and processed separately from others.

3.2.2 Detecting of Syntax Errors

Next, RDF-Doctor parses a given RDF file based on predefined rules of syntax errors. Once a rule is matched with a sequence of input tokens, it sends a notification error to the *Error Listener* module, where a list of all encountered errors is stored. Reading of input tokens continues until the end of the file (or chunk) while searching for any match in order to identify potential syntax errors. We used a parse tree to represent how the RDF-Doctor process with detecting syntax errors. Figure 3 shows how each sub-tree of the root node (acts as the global rule) is the main rule containing all other rules which represent either correct or incorrect syntactic forms. Each of these rules initiates either a sub-tree of non-terminals (grammar rules) and terminals (lexical forms of input tokens) nodes. Each sub-tree of non-terminals can either have zero, one or more nodes. On the other hand, a sub-tree of terminals can only have one or more nodes.

A sub-tree produces a syntax error, if all of its terminals (cf. Figure 3) formulate an error production rule based on the predefined grammar. Hence, both second and fourth sub-trees from left-to-right in Figure 3 with terminals in red color are rules with a sequence of tokens producing syntax errors. Meanwhile other sub-trees: first, third, and fifth with terminals in green color contain correct syntactic forms.

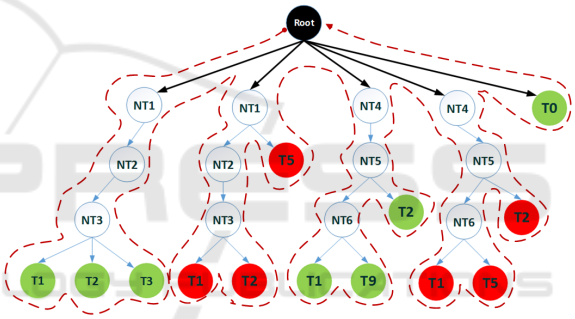

Figure 3: Detection of syntax errors while traversing the parse tree. The root node is the head of the first rule in the grammar and the head of the parse tree. All the children of the root are either non-terminal nodes represented by "NT" followed by a number of terminal ones shown by "T" succeeded. An error is detected on a non-terminal node when all of its terminals represent a sequence of tokens for a statement which includes a syntax error. Red terminals represent error-inclusive statements and green ones represent error-free statements.

3.2.3 Healing a Subset of Syntax Errors

The syntax error recovery currently focuses on a certain type of errors which has a predefined solution for each error in order to be recovered. Examples of such errors are: missing a dot at the end of a triple, missing a semi-colon after multiple predicates sharing the same subject, or missing a comma after multiple objects having the same subject and predicate.

Algorithm 1: The pseudo-code of RDF-Doctor.

```
Data: inputText, correctSyntaxRules, incorrectSyntaxRules,
      CorrectionIsSelected
Result: foundSyntaxErrors and recoveredSyntaxErrors
foundSyntaxErrors = [ ];
recoveredSyntaxErrors = [ ];
syntaxRules ← correctSyntaxRules + incorrectSyntaxRules;
while token in inputText && inputText ≠ EOF do
    currentTokens += inputText[token];
    ruleToBeMatched = getLexicalForm(currentTokens);
    if syntaxRules contains ruleToBeMatched then
        if incorrectSyntaxRules contains ruleToBeMatched then
            foundSyntaxErrors.push(currentTokens);
            if isCorrectionSyntaxErrorSelected then
                if canErrorBeRecovered(ruleToBeMatched) then
                    if recoverSyntaxError(currentTokens) then
                        recoveredSyntaxErrors.push(currentTokens);
                        foundSyntaxErrors.pop(currentTokens);
        currentTokens ← "";
        ruleToBeMatched ← "";
end
return foundSyntaxErrors, recoveredSyntaxErrors;
```

3.2.4 Producing of the Output

During this step, RDF-Doctor returns as output a file containing information about errors already recovered as well as those ones which are not corrected along with a user-friendly description. Commonly, errors that are not recovered by RDF-Doctor without a user intervention are those errors with several solutions. For example, a literal with multiple language tags like *"me"@en@de* which shows a string with two language tags where one solution might be a removal of one of them but the intention of the user is unknown in these cases. In addition, there exist errors with undefined or unknown recovery solution, e.g. missing a user-defined prefix declaration for a certain local namespace which cannot be guessed since it is only known by the user.

Algorithm 1 presents the abstract behaviour of RDF-Doctor. It initializes with the *syntaxRules* variable which combines both correct syntax rules and incorrect ones. The main *while loop* carries on until reaching the end of the current file or chunk. The *ruleToBeMatched* variable encloses one rule from either correct or incorrect production rules formed by the supplied tokens whereas the *currentTokens* variable stores a sequence of given tokens to check to which rule they belong to. Since a crucial step is to identify syntax errors, *ruleToBeMatched* is evaluated to check if its content matches with the *incorrectSyntaxRules*. In case of matching, the content of *currentTokens* is considered as a syntax error, and the parser sends a notification error to any method subscribed to *Error Listener* module, in order to further process the collected errors. The algorithm continues with the automatic correction in case it is activated where the Error Correction Module traverses the entire list of identified errors and based on the error message, it assesses if a particular error can be corrected or needs user intervention. At the end of the correction phase, a report containing the corrected errors will be delivered to the user.

3.3 Categories of RDF Syntax Errors

A significant part of this work is the ability to identify syntax errors which should be known in advance in order to be defined.

We divide types of syntax errors into multiple categories as shown in Appendix Table [4]. These categories are taken from (Tur, 2013) with some modification to become more expressive. The table shows each category with one error sample and an error position where the actual syntax error is located. This facilitated the definition of the grammar rules as well as the investigation of which of them can be automatically corrected.

4 APPLICATION SCENARIO

In this section, we present an application scenario that illustrates the detection and correction of a syntax error with RDF-Doctor. The scenario starts with a Turtle example which has no syntax errors. Then, a syntax error is introduced to show the process of handling it with RDF-Doctor.

Listing 1: RDF example in Turtle serialization format.

```
@prefix  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix  rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix  ex:  <http://example.org/> .
@prefix  zoo:  <http://example.org/zoo/> .
         ex:dog1  rdf:type  ex:animal .
         ex:cat1  rdf:type  ex:cat ;
                  rdfs:label  "Lusi"@en .
         ex:cat  rdfs:subClassOf  ex:animal .
         zoo:host  rdfs:range  ex:animal .
         ex:zoo1  zoo:host  ex:cat2 .
```

Listing 1 shows a Turtle example without syntax errors. The first four lines are Prefix declarations whereas the following are a couple of triples. For that reason, our grammar is initialized with the topmost node **start** which describes the coming rules by zero or more **statement**(s). In addition, each **statement** is either a directive or a triple as it is represented in Listing 2.

Listing 2: Starting rules in the grammar file.

```
start
 : statement*  EOF
 ;
statement
```

[4] https://github.com/ahemaid/RDF-Doctor/blob/master/List_of_Errors.pdf

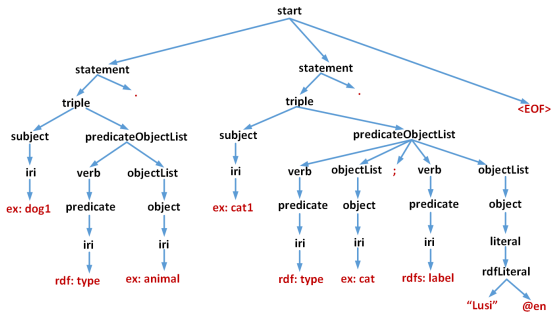

Figure 4: A view of parse tree for listing 1. is illustrated. It shows the sub-trees of the parent node **start** in the parse tree. Those nodes written in *black* are non-terminals or heads of the grammar rules. The remaining nodes are terminals or matched patterns of input tokens.

```
: directive
| triples '.'
;
```

To shed light on the generated parse tree for Listing 1, Figure 4 demonstrates a view of the parse tree for the first 3 lines after prefixes: lines 5, 6, and 7, as well as *EndOfFile* (EOF) sequence. Line 5 matches *triple* with only one subject *ex:dog1*, but lines 6, and 7 match *triple* with multiple predicates and objects and share the same subject *ex:cat1*, same is applied on the parse tree.

After showing a part of the grammar rules and views of the parse tree, a real use case is presented in the following. The use case shows an error production rule that matches a syntax error pattern for detecting and resolving it.

Missing a Dot at the End of a Triple: In the Turtle syntax, a triple must end with a dot. In Listing 3, the head rule, i.e., a **statement**, can either be a directive or a triple both ending with a dot. Equally important, the last line which shows that triples without a dot can also be a sub-goal of this rule. This sub-goal is considered as a normal path in a parse tree, but once the parser detects it, a notification is sent to the *Error Listener* module with an error message *"Missing a '.' at the end of a directive prefix and/or triple"*.

Once such an error is saved by the *Error Listener* module, the role of *Error Detection* module is finished. The next phase starts in the *Error Correction* module to correct the error. Since the list of syntax errors is shared with *Error Correction* module, it iterates all of these errors messages. If one of these messages in the error list matches, then it applies a predefined function to recover the error. For instance, the function *addDot(lineNumber, columnNumber)* is applied, since it matches the same error message and similarly, in the case of *"Missing '.' at the end of Prefix directive"* message. The task of this function is to reach the line which misses the dot, identify the line number and column number, then finally automatically correct the error by adding a dot at the end of the given triple.

Table 2: Evaluation summary of RDF-Doctor for both correct and incorrect syntactic forms. "Detected" for "Correct Syntax" means that RDF-Doctor recognizes them as correct syntactic forms, whereas not correctly recognized ones are classified as "Undetected". Similarly, "Detected" for "Incorrect/Bad Syntax" refers to recognized syntactic forms as incorrect forms with releasing corresponding error messages, meanwhile "Undetected" specifies incorrect syntactic forms which are not recognized and might generate false positives.

File Content	Detected	Undetected	Total
Correct Syntax	185	25	210
Incorrect Syntax	53	12	65
Total	238	37	275

Listing 3: Grammar rules for detecting of syntax error of missing dot at the end of a triple.

```
statement
 : directive
 | triples
 | triples {notifyErrorListeners("Missing '.'
    at the end of a triple")} ;
```

5 EXPERIMENTAL STUDY

Three different experiments were conducted to check for effectiveness and efficiency of RDF-Doctor.

5.1 Experiment I

The target of this experiment is to measure the efficiency of RDF-Doctor while testing with the W3C Turtle Test Suite.

5.1.1 Procedure and Measure

We used the Test Suite recommended by W3 Consortium (W3C) in (Tur, 2013). The total number of files of the given suite is 275, which we divided into two parts: 1) files that are syntactically correct; and 2) files that are syntactically incorrect. The first subset contains 210 files, whereas the second one contains 65 files. We computed Precision, Recall, and Accuracy values using equations (1), (2), and (3), accordingly.

$$Precision = \frac{t_p}{t_p + f_p}; \qquad \begin{array}{l} t_p = \text{number of true positives} \\ f_p = \text{number of false positives} \end{array} \tag{1}$$

$$Recall = \frac{t_p}{t_p + f_n}; \qquad \begin{array}{l} t_p = \text{number of true positives} \\ f_n = \text{number of false negatives} \end{array} \tag{2}$$

Table 3: Evaluation of RDF-Doctor for detection of incorrect syntactic forms in the Turtle Test Suite (Tur, 2013).

Classification of Error Types	Detected	Undetected	Total
Bad String Escape	0	4	4
Bad Keywords	5	0	5
Bad Language Tag	2	0	2
Bad Local Namespace in Prefixed IRI	2	3	5
Bad Prefix Label in Prefixed IRI	2	0	2
Bad Syntax from N3 Notation	11	1	12
Bad Prefix Label in Directives	2	0	2
Bad Number as a Literal	5	0	5
Bad Directive	4	0	4
Bad String	6	1	7
Bad Structure	12	0	12
Bad IRI	2	3	5
Total	53	12	65

$$Accuracy = \frac{t_p + t_n}{t_p + t_n + f_p + f_n}; \quad \begin{array}{l} t_p = \text{number of true positives} \\ t_n = \text{number of true negatives} \\ f_p = \text{number of false positives} \\ f_n = \text{number of false negatives} \end{array} \tag{3}$$

5.1.2 Result and Discussion

As we can observe from results given in Table 2, the majority of syntactically correct files i.e. 88% are categorized correctly (as free of errors) while the remaining 12% is considered syntactically incorrect. On the other hand, when dealing with incorrect syntax, results show that 81.5% of files which include incorrect syntax are detected and the appropriate error message is given, whereas the rest 18.5%, are not recognized as errors, thus no error message is given. In terms of Precision, Recall, and F-measure, RDF-Doctor reaches a Precision of 0.82, Recall equals to 0.68, and Accuracy equals to 0.78.

Table 3 shows evaluation results with respect to the above-mentioned input. In order to evaluate the automatic error correction feature of RDF-Doctor, we used the results of this experiment. The outcome shows that 5 types of errors can be faultlessly corrected: 1) using of 'A' as a predicate instead of 'a'; 2) missing a colon in prefix or base declarations; 3) missing a dot at the end of a triple; 4) adding more than one dot at the end of the triple; and 5) ending a triple with a semi-colon instead of a dot, correspondingly.

5.2 Experiment II

The objective of this experiment is to simulate a scenario where an ontology engineer develops an ontology for a particular domain using a plain text editor. The engineer may introduce a number of syntactic errors, which he/she has to identify and correct before being able to share his/her contribution with other team members.

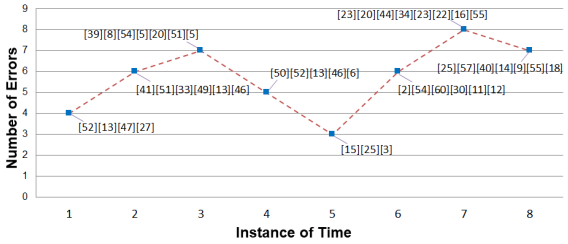

Figure 5: Distribution of syntax errors. The number and types of syntax errors between brackets for an interval of eight instances of time. A Poisson Distribution and a Uniform Distribution generate a number of errors per instance of time and types of errors as listed in Appendix Table, respectively. For example, at the 5th instance, three error of different types are introduced, i.e., [3,15,25].

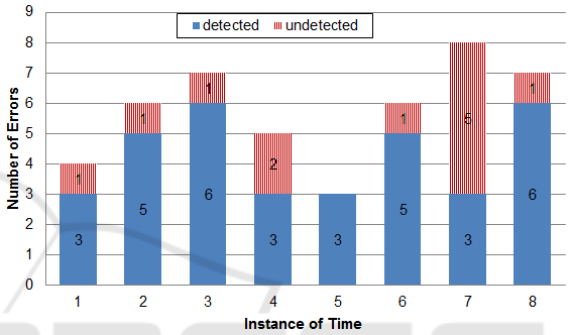

Figure 6: Error detection when syntax errors are randomly distributed. For each instance of time, a number of errors which are detected or not detected by RDF-Doctor are shown.

5.2.1 Procedure

In this experiment, we assume that the ontology engineer contributing to the DBpedia ontology by performing several modifications, e.g. insert, edit or delete during a period of eight instances of time, i.e., each hour. The Poisson and Uniform Distribution are used to simulate the number and the type of errors occurred in each instance of time, respectively. The average number of syntax errors per instance of time is represented by λ parameter. For example, $\lambda = 5$, means that an average number of five syntax errors occur per hour. The Uniform Distribution is used to randomly select the error types from a comprehensive list of errors given in Appendix Table[5]. Moreover, the exact location for injecting syntax errors within the RDF file is realized using the Uniform Distribution where a random line number is generated. The error

[5]https://github.com/ahemaid/RDF-Doctor/blob/master/List_of_Errors.pdf

is successfully applied if the randomly selected location is appropriate for the error type; otherwise, one of the lines from its neighborhood are selected. If the error is related to the header (the top of the file where directives are normally located), then the injection is applied in the header of the file.

5.2.2 Result and Discussion

Experimental results illustrated in Figure 5 show that RDF-Doctor in a majority of the cases is able to correctly recognize syntax errors. More specifically, Figure 6 shows that in half of the time instances, apart from one syntax error which is not recognized, all of them are identified. In time instance number 5, all errors are detected. However, there are cases where more than one error is not recognized, such as time instance number 7 with five undetected syntax errors. The reason behind this, is that most the of randomly inserted errors are those known as unrecognized errors by RDF-Doctor during that time instance. A solution for such an issue can be achieved by developing additional mechanisms that detect nested and conflicting rules.

5.3 Experiment III

In this experiment, we checked the effect on the behaviour and the performance of our approach whenever the number of errors increases and the size of the ontologies is changed, to simulate real-world scenarios where ontologies continue growing over the time.

5.3.1 Procedure

We used two types of ontologies; small and medium-sized. The Friend of a Friend Vocabulary (FOAF) is used as a small ontology with a total number of 631 triples. As a medium-size ontology, we used the DBpedia version 2016-10 with a total number of 30,790 triples. In both ontologies, three different numbers of syntax errors are introduced i.e., 10, 30, 61. These syntax errors are randomly injected using the same way as in the previous experiment where the Uniform Distribution is utilized. Additionally, to avoid any impact on the current overload of the processor during the execution time, we ran our experiment five times for each case and calculated the average processing time.

5.3.2 Result and Discussion

Figure 7 illustrates the performance with respect to the number of errors and the ontology size. After introducing 10 syntax errors, 9 out of 10 are detected and one is not detected for both FOAF and DBpedia ontology, respectively. In the second case, 25 out of 30 injected errors are detected in FOAF whereas in DBpedia 26 out of 30. Finally, from a total number of 61 injected errors, 6 in FOAF and 11 in DBpedia are not detected. In all the cases, for each error that is detected, an expressive message is given to facilitate resolving procedure.

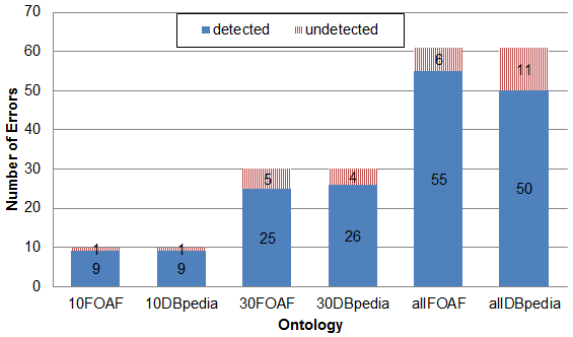

Figure 7: Impact of the number of errors and the size on RDF-Doctor. FOAF and DBpedia ontologies are used to evaluate RDF-Doctor, 10foaf, 30foaf, allfoaf are modes of FOAF ontology, including with 10, 30, 61 random syntax errors, respectively; same is applicable for DBpedia. Detected errors are the errors which were properly identified by RDF-Doctor matched error messages released, while undetected errors are those which were not correctly recognized.

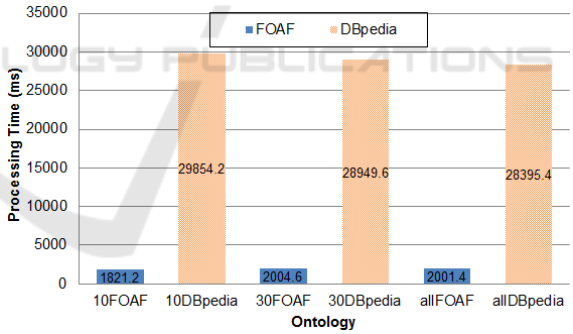

Figure 8: Performance evaluation when a number of errors and ontology size is changed, individually. Performance is calculated with respect to the required processing time, i.e., in milliseconds (ms) in both cases: 1) 10, 30, and 61 injected syntax errors; 2) FOAF ontology with 631 triples and DBpedia ontology, with 30,790 triples.

The performance of RDF-Doctor is reported in Figure 8. It summarizes the impact of both, the number of errors and the ontology size. As we can observe, the numbers of errors has a limited influence on the performance, i.e., the processing time is similar in either FOAF or DBpedia while changing the number of errors. Additionally, the increasing number of syntax errors has a slightly oscillatory effect on the processing time. This is due to the different system

behavior when dealing with false positive errors since RDF-Doctor needs to recover until it finds a matched grammar rule formed from input tokens which their number frequently varies. On the other hand, the size of ontologies has a high impact on the overall performance, i.e, processing DBpedia consumes about 29000ms on average; whereas, FOAF is parsed in about 2000ms. In conclusion, approximately more than 90% of injected syntax errors are detected, as shown in Figure 7.

6 CONCLUSION

This paper presents RDF-Doctor, an approach for fault-tolerant error detection and automatic error recovery for RDF data. To enable a comprehensive error detection, a set of grammar rules covering an exhaustive list of syntactic errors occurring commonly in RDF data are predefined. The ANTLR framework is used to generate the parser based on the given grammar. For each detected error, RDF-Doctor provides a friendly and precise error message helping users to easily locate and correct them. Furthermore, using an internal mechanism for automatic error correction, RDF-Doctor is able to recover a subset of syntax errors without user intervention. We performed three different empirical evaluations to asses the effectiveness and efficiency of RDF-Doctor in various scenarios. The achieved results provide evidence that the effectiveness of RDF-Doctor is higher for each error already defined in the grammar. On the other hand, the efficiency is not impacted by the number and the type of errors, but from the size of the ontology, which is expended since an increasing number of triples to be parsed RDF-Doctor consumes more time.

As future work, we aim to improve syntax error detection of RDF data by extending the rules to cover a wide range of different tokens. We plan also to evaluate the scalability of RDF-Doctor by increasing number of errors. Furthermore, a rule-based approach must be defined to handle conflicts of syntax error rules. Finally, a comprehensive error recovery can be achieved by employing supervised learning methods allowing RDF-Doctor to learn from training data and be more effective and precise in the error correction.

ACKNOWLEDGEMENTS

This work has been supported by the German Federal Ministry of Education and Research (BMBF) in the context of the projects “LUCID” (grant no. 01IS14019C) and “Industrial Data Space Plus” (grant no. 01IS17031). It has also been supported by the Fraunhofer Cluster of Excellence “Cognitive Internet Technologies” (CCIT).

REFERENCES

IDLab Turtle Validator .

Programming with RDF4J.

RDF Validator and Converter .

(2013). Turtle Test Suite. W3c test suite, World Wide Web Consortium (W3C).

Balachandran, V. (2013). Fix-it: An extensible code autofix component in review bot. pages 167–172.

Halilaj, L., Grangel-González, I., Coskun, G., Lohmann, S., and Auer, S. (2016a). Git4voc: Collaborative vocabulary development based on git. *International Journal of Semantic Computing*, 10(02):167–191.

Halilaj, L., Petersen, N., Grangel-González, I., Lange, C., Auer, S., Coskun, G., and Lohmann, S. (2016b). *VoCol: An Integrated Environment to Support Version-Controlled Vocabulary Development*, pages 303–319. Springer International Publishing, Cham.

McBride, B. (2002). Jena: A semantic web toolkit. *IEEE Internet Computing*, 6(6):55–59.

Petersen, N., Coskun, G., and Lange, C. (2016). Turtleeditor: An ontology-aware web-editor for collaborative ontology development. In *2016 IEEE Tenth International Conference on Semantic Computing (ICSC)*, pages 183–186.

Prud'hommeaux, E. RDF Validation Service.

Prud'hommeaux, E., Labra Gayo, J. E., and Solbrig, H. (2014). Shape expressions: an rdf validation and transformation language. In *Proceedings of the 10th International Conference on Semantic Systems*, pages 32–40. ACM.

Tolle, K. (2000). Analyzing and Parsing RDF. Master's thesis.

Verborgh, R. Lightning-fast rdf in JavaScript. Blog post.

Zeng, K., Yang, J., Wang, H., Shao, B., and Wang, Z. (2013). A distributed graph engine for web scale rdf data. *Proc. VLDB Endow.*, 6(4):265–276.