Extraction of ABNF Rules from RFCs to Enable Automated Test Data Generation

Markus Gruber, Phillip Wieser, Stefan Nachtnebel, Christian Schanes, Thomas Grechenig

Extraction of ABNF Rules from RFCs to Enable Automated Test Data Generation

Markus Gruber, Phillip Wieser, Stefan Nachtnebel,
Christian Schanes, and Thomas Grechenig

Research Group for Industrial Software, Vienna University of Technology,
1040 Vienna, Austria
{markus.gruber,phillip.wieser,stefan.nachtnebel,
christian.schanes,thomas.grechenig}@inso.tuwien.ac.at

http://security.inso.tuwien.ac.at/

Abstract. The complexity of IT systems and the criticality of robust IT systems is constantly increasing. Testing a system requires consideration of different protocols and interfaces, which makes testing hard and expensive. Test automation is required to improve the quality of systems without cost explosion. Many standards like HTML and FTP are semi–formally defined in RFCs, which makes a generic algorithm for test data generation based on RFC relevant. The proposed approach makes it possible to automatically generate test data for protocols defined as ABNF in RFCs for robustness tests. The introduced approach was shown in practice by generating SIP messages based on the RFC specification of SIP. This approach shows the possibility to generate data for any RFC that uses ABNF, and provides a solid foundation for further empirical evaluation and extension for software testing purposes.

1 Introduction

Security testing is an important and at the same time also expensive task for developing robust and secure systems. Costs of software testing increase due to the complexity and interconnection of modern software systems. Different interfaces, protocols and standards are used which requires much test effort to cover all aspects. Test automation can eliminate repetitive and time–consuming manual testing tasks and therefore reduce costs. Test data is required to test a System Under Test (SUT), and good test data might increase confidence in software quality, e.g., by testing more parts of the software [8].

Many Internet standards are commonly defined in a document called Request For Comments (RFC). The Internet Engineering Task Force (IETF) describes the purpose of RFC as:

"Memos in the RFC document series contain technical and organizational notes about the Internet. They cover many aspects of computer networking, including protocols, procedures, programs, and concepts, as well as meeting notes, opinions, and sometimes humor." [11]

Augmented Backus Naur Form (ABNF), a metalanguage to describe the syntax of parsable structures, is often used in RFCs to describe formal specifications, e.g., protocol specifications or flow definitions. These ABNF rules are usually hidden in informal descriptions.

Certain ABNF rules within an RFC are used to specify protocols or technologies. Examples are Hyper Text Transfer Protocol (HTTP) or Session Initiation Protocol (SIP), which are defined using ABNF rules. Based on these ABNF rules, test data can be generated in order to test different aspects of the interface, e.g., the conformance of a SUT to an RFC.

In this work, an approach to semi–automatically generate test data based on an RFC specification is presented. The process of extracting ABNF rules out of an RFC and the transformation from ABNF to XML Schema Definition (XSD) makes it possible to generate test data in Extensible Markup Language (XML) format. For test data generation, a number of existing frameworks and scientific test data generation algorithms can be used. XML is widely used in web applications and enables test data transformation to various other formats. The application of this approach is shown by generating test data for SIP messages based on the specification of SIP in RFC 3261 [20]. The possibility to semi–automatically generate test data based on an RFC might greatly reduce the time and effort needed to efficiently test a SUT.

Fenner [9] has developed a simple heuristic extractor as part of his ABNF parser. The solution proposed in this paper is based on this parser, but the workflow is adapted and additional features to generate a valid and self-contained set of ABNF rules are implemented. Valid and self-contained ABNF rulesets do not contain validation errors, e.g., syntax errors or missing rules. These rulesets can be validated by other ABNF parsers, e.g., Bill's ABNF Parser[1], to prove syntactic and semantic validity. Additionally, the ability to transform ABNF rules to XSD is introduced. For automated test data generation we use our approach presented in a previous work [21] which operates on an XSD model.

The remainder of this paper is structured as follows. An overview of related work is given in Sect. 2. Section 3 describes ABNF and the concept of ABNF model extraction. Section 4 covers test data generation based on transformation from ABNF rules to XSD. Section 5 covers the results and the lessons learned by developing and applying the approach for test data generation of SIP systems. The paper finishes with a conclusion and ideas for further work in Sect. 6.

2 Related Work

The IETF regularly publishes RFCs which describe Internet standards. Other organizations like International Organization for Standardization (ISO) and World Wide Web Consortium (W3C) publish standards in computer science as well.

ABNF is only one metalanguage to describe parsable structures, other widely used formal metalanguages are Backus Naur Form (BNF) [15], Wirth Syntax Notation (WSN) [25] or Extended Backus Naur Form (EBNF) [22].

[1] https://code.google.com/p/bap/

Concerning test data generation, one can distinguish between random and dynamic test data generation [17]. Random data generation techniques do not require (but may take into account) an interface– or protocol specification of the SUT. While some authors state that random data generation produces test data efficiently [2], [12], others have come to the conclusion that most data is rejected by the SUT [16], [19]. Dynamic data generation approaches analyze the execution of test data against a SUT and try to generate new data based on the obtained knowledge. One approach tries to adapt test data so that critical software regions are tested more thoroughly [6], while another employs dynamic binary analysis [5].

The proposed approach generates data in the generic format XML. Several authors show the transformation of XML to other commonly used formats [24], [13], [10], [14]. Specific applications of test data generation from XSD have been proposed by several authors. A simple XML data generator based on defined rules has been proposed in [1]. Another software called TAXI generates XML documents based on an XSD [4], while ToxGene described in [3] is a template–based generator of synthetic XML documents. For the presented approach in this work our test data generation approach presented in [21] is used, which allows the generation of XML data based on XSD input.

3 Concept of ABNF Model Extraction from RFC

This section presents an introduction to the usage of ABNF in RFCs followed by the description of the process of the specification extraction approach. This is an iterative process of improving the quality of the extracted ruleset. This section also describes the ABNF error classes which can occur during this process.

3.1 Introduction to the Usage of ABNF in RFCs

ABNF is a metalanguage based on BNF and defined in RFC 5234 [7]. Both are notations for context–free grammars, used to describe the syntax of parsable structures, e.g., communication protocols. Most RFCs use ABNF to describe formal specifications. These ABNF rules, however, are usually embedded in informal descriptions, as seen in Fig. 1. In addition to the informal description at the top of this example, this ABNF rule consists of the rule name (HTTP-Version) and the rule definition on the right side. ABNF rules in one RFC can also reference ABNF rules in other RFCs. A mutual dependence of each ABNF rule can be described with a dependency tree, as shown in [23].

The ABNF specification is a set of derivation rules. These rules can be seen as a tree of rules and operators. Several options exist for creating this tree. One possible option is the *top–down parsing strategy*, which consists of taking a single element of interest, designating it as root of the tree and adding the dependent rules iteratively. Another possibility is the *bottom–up parsing strategy*. Here, all rules are considered and topologically sorted based on their dependencies, resulting in a tree with multiple roots. The approach presented in this paper

```
The version of an HTTP message is indicated by an
HTTP-Version field in the first line of the message.

    HTTP-Version    = "HTTP" "/" 1*DIGIT "." 1*DIGIT
```

Fig. 1. Example of an ABNF rule in RFC 2616

follows the second approach, since it is more flexible and supports our goal to extract all ABNF rules, e.g., including referenced rules in other RFCs. A drawback of the second approach is, that the developer most likely will have to deal with rules that are not relevant, since only some rules (or a single rule including dependencies) are in scope of interest.

Special classes of ABNF rules are *prose rules*, *semantic pseudo rules* and *stub rules*. A prose rule is enclosed by < >. Prose rules are informal definitions of rules. Figure 2 shows an example of a prose rule in RFC.

```
LOALPHA = <any US-ASCII lowercase letter "a".."z">
```

Fig. 2. Example of a prose rule

Semantic pseudo rules use operators to describe semantic relations, which will lead to an ABNF syntax error. Figure 3 shows an example of a syntax error in an ABNF rule.

```
response_is_fresh = (freshness_lifetime > current_age)
```

Fig. 3. Example of a semantic pseudo rule, leading to a syntax error

Stub rules, in contrast to semantic pseudo rules, are semantically incorrect. Figure 4 shows an example of a stub rule, i.e., a Message Digest 5 (MD5) checksum found in RFC 1864 [18]. It is very likely that this MD5 checksum will not represent a valid checksum of the generated sample.

3.2 Process of the Specification Extraction Approach

Based on the extractor developed by Fenner [9], additional features were implemented, e.g., namespacing, case escaping or detecting and separating prose rules, to *heuristically* extract all defined conditions from the RFC. Not all RFCs could be parsed automatically, because in some cases conflicts could not be resolved automatically in order to get semantically correct ABNF rulesets. To increase the quality of the approach, it was decided to use a semi–automatic approach to extract ABNF rules from RFCs.

```
md5-digest = "Q2hlY2sgSW50ZWdyaXR5IQ=="
```

Fig. 4. Example of a stub rule

Figure 5 describes our approach to extract a valid and self-contained ABNF ruleset of an RFC. The approach starts with the choosing of an arbitrary RFC, or multiple RFCs, one wants to have an ABNF ruleset for. After the initial configuration, it is an iterative process, fixing one problematic rule after another, until the full set of ABNF rules is generated.

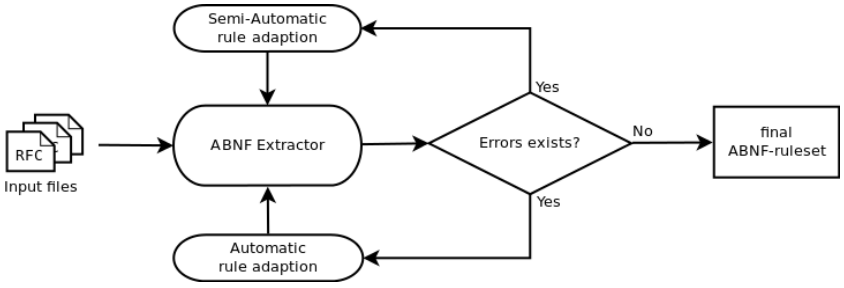

Fig. 5. Process of the ABNF extraction approach

The ABNF extractor automatically processes the following steps in order to get a final ruleset free of any errors:

1. Handle multiple input files as a single set of rules
2. Detect prose rules for semi–automatic processing
3. Expand `#` operators to valid constructs in ABNF syntax, because `#` is an originally unsupported rule in ABNF but defined later in some RFCs as `#rule`
4. Replace widely–used (but actually forbidden) characters "`_`" by "`-`" and "`|`" by "`/`"
5. Strip comments from rules
6. Unify the rules and remove redundancies
7. Replace rules that are defined multiple times (same name and same definition) by a single occurrence
8. Generate a dependency tree
9. Topologically sort rules based on the dependency tree

The topological sorting of rules is necessary, because if a rule is referenced before defined some parsers may throw errors. Additionally to the automatically processed tasks, the following rule adaption possibilities to clean the ABNF rules in order to get a valid and self-contained ruleset exist:

– Blacklist definitions for all rules which should be ignored

- Namespace transformations for rules with the same name in different RFCs
- Replace rule names with case–insensitive rule names
- Replace invalid rule definitions with customized rules

Our approach proposes a solution for each error, but the final choice of the rule adaption must be accomplished manually in order to avoid semantic errors.

3.3 Validating Validity and Self-Containedness of a Rule Set

The syntactic validity of ABNF rules can be tested using ABNF parsers, which throw an error if invalid ABNF syntax is used.

In order for a ruleset to be self-contained, all referenced rules need to be defined in the current ABNF document, meaning that there are no rules that are not defined. A simple algorithm as seen in Fig. 6 uses all extracted ABNF rules as input. It can be used to generate two sets: One contains all defined rules, and the other contains all referenced rules. By verifying that every referenced rule is part of the set of defined rules, one can verify that there are no missing rules.

```
for rule in rules
    defined_rules.insert(rule.name)
    for element in rule.body
        if typeof(element) is REFERENCE
            referenced_rules.insert(element)
    endfor
endfor
```

Fig. 6. Pseudocode: Constructing set of definitions and set of references

3.4 Error Classes of ABNF Generation

During the automated derivation of a model with the ABNF rules from one or more RFCs different problems can arise. The three main classes of problems are missing rule errors, syntax errors and double rule errors. By using multiple RFCs, e.g., referencing definitions in other RFCs, it can occur, that rules are defined more than once. If not all RFCs are given as input to the parser, missing rule errors can occur. Due to failures during the parsing process, syntax errors can arise, e.g., wrong definitions in the RFC or problems of parsing the ABNF rules from the RFC text. Figure 7 shows the errors which can occur during the extraction process.

Missing Rule Errors A missing rule is a rule, that is referenced, but is not defined. For example the rules A and B are defined, rule A references rules B

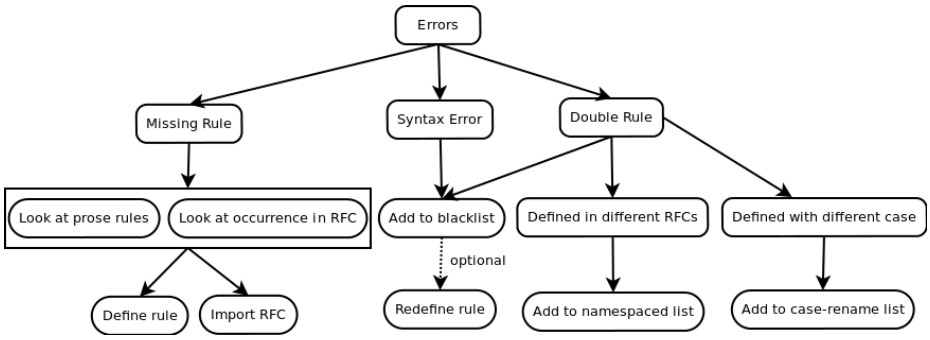

Fig. 7. Error Classes of ABNF Generation and Countermeasures.

and C, but rule C is not defined. This could happen if a rule is defined as prose or defined in a referenced document.

If a rule may be referenced from another RFC, these references are informally defined either in a prose rule, or in the surrounding textual description. If this occurs, one can either import the complete referenced RFC or copy the specific rule. Another problem could occur, when some RFCs include some core rules (as defined in RFC 5234 [7]), while other core rules are missing. Core rules are basic rules that are in common use. Some authors include all ABNF core rules directly in implementations of scanners or parsers. For the sake of simplicity, it is suggested to include them in a separate file. Concerning duplicate – or missing – rules, they would be treated equally to extracted rules.

A special kind of errors are non–validation errors. A ruleset which contains non–validation errors contains valid ABNF syntax, but is semantically incorrect. Semantic errors can only be detected by so–called validating parsers (in contrast to non–validating parsers). One reason could be, that some rules are only stub rules. This issue can not be solved in ABNF, but only by some processor on a higher level that takes semantic aspects like data dependencies into account.

Another reason of non–validation errors could be *blind text* as part of a rule. Blind text is meant as additional information or description, but parsers may consider it part of the rule. An example of blind text in an ABNF rule can be seen in Fig. 8. Because of its indent, the blind text will be treated as part of the rule, which will result in a syntax error although the text is only meant as a comment.

```
credentials = auth-scheme #auth-param
  Note that many browsers will only recognize Basic and will↩
     require that it be the first auth-scheme presented. ↩
    Servers should only include Basic if it is minimally ↩
    acceptable.
```

Fig. 8. Example of blind text in an ABNF rule

Syntax Errors The second error class are syntax errors. Mostly, semantic pseudo rules (as explained in Sect. 3.1) occur in addition to regular rule definitions, or are referenced by another semantic rule. Although it is not usable in ABNF directly, it could be parsed as additional information. This information could be used at a later time for a format supporting it, e.g., XSD generation. Because of the lack of typing, and describing semantic relations in ABNF, these rules have to be stubbed. Another possibility is, transforming the ABNF to a different grammar, which enables supporting the specific aspect of the subjected prose rule.

Double Rule Errors The third error class are double rule errors. If the semantic pseudo rule is a duplicate of a regular rule, the semantic pseudo rule could be simply ignored.

The ABNF standard defines rule names as case–insensitive, which is not always the case in RFCs. Sometimes, rules with the same name (and different casing) are intended to be different. The presented approach solves this by heuristically replacing the upper–cased letter with a lower–cased and some additional ABNF–compatible information tagging it as escaped. Figure 9 shows an example of escaping the upper–cased letters. This circumstance also often occurs, when importing referenced RFCs.

```
'Foo' would be escaped to '---f--oo'
```

Fig. 9. Example of escaped upper–case letter 'F'

Another case of double rule errors could occur when importing a referenced RFC, or processing multiple RFCs. Then, it may happen, that rules are defined in multiple RFCs. Similar rules with the same name and same body can be resolved by ignoring all duplicated rules. It may also happen, that the two rules mean two completely different things in the different scopes of the RFCs. In this case, it is required to namespace them in an ABNF–compatible fashion. One option would be to set a prefix to the affected rules to avoid duplicated rules.

4 Test Data Generation Based on Transformation from ABNF Rules to XSD

This section presents the approach of test data generation based on transformation from ABNF rules to XSD. An XSD file describes the structure of an XML document, and was introduced by the W3C. The reason for choosing XSD as the destination format is that test data generation in XML format, which is a generic format and can be transformed to many other formats, is easily possible and existing data generation algorithms can be used. Several differences and similarities of ABNF and XSD need to be taken into account, which are described in this section.

4.1 Differences Between ABNF and XSD

Each rule itself can be transformed to a valid XSD representation, but the result of the combination of the rules does not produce a valid XSD. In ABNF, literals, references and ranges may all be used as a part of the rule definition, but those constructs all need to be transformed to XSD differently.

Well–formed XSD documents are well–formed XML documents themselves. Not all characters are valid in XML, but it is required to be able to encode all bytes from `0x00` to `0xFF`. Therefore one of the binary types had to be picked. The type *hexBinary* was chosen in favor of *base64Binary*, because it is easier to use with a regular expression pattern. When using hexBinary to encode a text, each letter is represented by two hexadecimal characters, and can be changed individually without the need to re–encode the rest of the string.

Literals, strings and ranges can be expressed either as patterns using a regular expression, or as an enumeration. It was decided to take the regular expression pattern approach, because the representation as regular expression is more compact and therefore in our opinion more readable. Especially when a large number of data instances are possible for an element, an enumeration is not a viable option. Existing regular expression parsers can be used to generate instances that conform to a regular expression.

Additionally a string literal in ABNF is case–insensitive. This means "foo" could produce "fOo", "FOo", etc. For the sake of simplicity, "foo" will only produce "foo" after transforming to XSD.

4.2 Mapping ABNF to XSD

To use our generic approach of data generation as described in our previous work [21], which allows the generation of XML data based on XSD input, an ABNF ruleset needs to be transformed to XSD. An example of the expected result, i.e., for the transformation of the `DIGIT rule`, is seen in Fig. 10.

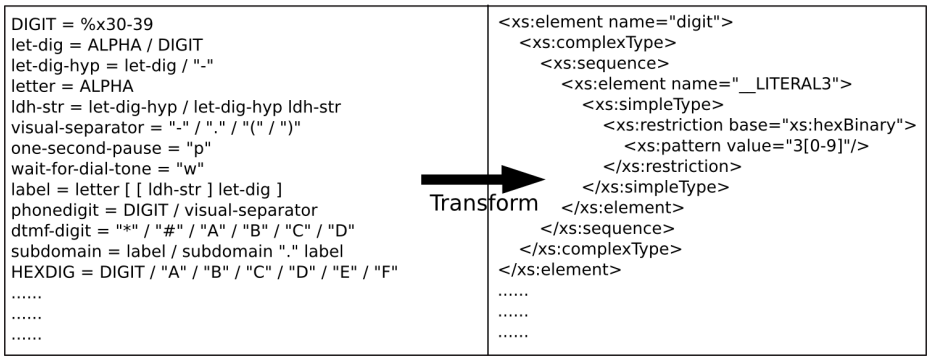

Fig. 10. Example of ABNF to XSD transformation

Our approach uses a transformation matrix, as seen in Fig. 11, from ABNF rules to XSD. These mapping rules require additional transformation logic to correctly generate a valid XSD element tree. The proposed approach derives transitions from ABNF rules to XSD elements by comparing the valid children of the enclosing element with the valid parents of an enclosed element, until a transition from the enclosing to the enclosed element is detected.

To reduce complexity, classes are combined, as defined in the first column of Fig. 11, of (in this case) compatible ABNF rules:

- $Choice, Group, Repetition \equiv CGR$
- $Literal, String, Range \equiv LSR$

A repetition is transformed to XSD using a (single–content) sequence with attributes minOccurs and maxOccurs. Therefore it is in the same class as group.

However, the following transformations are used to transform ABNF rule classes to XSD trees. XSD follows certain restrictions concerning element positioning in the tree, which need to be considered for the transformation.

1. $Rule \rightarrow CGR$
2. $Rule \rightarrow Reference$
3. $Rule \rightarrow LSR$
4. $CGR \rightarrow CGR$
5. $CGR \rightarrow Reference$
6. $CGR \rightarrow LSR$

Classes irrelevant to our approach have been omitted.

ABNF-Class	**Operator**	**Parameter**	**XSD-Element**
Rule	= =/	name value	<element name="name"> value <\element>
Choice	|	value []	<choice> value [] <\choice>
Group	( )	value []	<sequence> value [] <\sequence>
Repetition	n*m * [ ]	value min max	@minOccurs="min" @maxOccurs="max"
Reference		name	<element ref="name" />
Literal	"foo"	value	<pattern value="someregex" />
String	%xA.B...N %dA.B...N %bA.B...N	value type	<pattern value="someregex" />
Range	%xN–M %dN–M %bN–M	type min max	<pattern value="someregex" />

Fig. 11. Basic ABNF to XSD mapping rules.

4.3 Description of XML Test Data Generation from XSD

After transformation of the ABNF rules to XSD, a generic data generation framework is used to generate the required test data in XML format. For this purpose, the test data generation features of an existing fuzzing framework called fuzzolution[2] were used, which are also described in [21].

The XSD document generated in the previous step serves as an input to the framework. Data is generated based on two features of the generated XSD, which are structural information and data–based restrictions.

Structural information is represented by elements like `choice`, `sequence` as well as attributes like `minOccurs`. The used framework takes these restrictions into account, and generates XML instances that conform to the given XSD. Data–based restrictions describe the data within the elements. In this example, hexadecimal values are generated for several elements based on the information available in the XSD.

Using the transformed XSD, the framework generates test data in XML format. These XML files contain the test data and might be used either directly by executing it against a SUT or by transforming it to another format first.

5 Experiences with Generation of Test Data for SIP Systems

In this section, the applicability of the proposed approach for a specific RFC is presented. For this purpose, test data for SIP messages as defined in RFC 3261 [20] is generated and validated. Since the number of Voice over IP (VoIP) systems and SIP users is constantly increasing, attackers have more incentive to attack SIP systems. This shows the necessity to automatically test SIP systems to find and resolve robustness errors.

5.1 Generation of SIP Test Data Based on the RFC

The process of generating test data in this approach can be summarized by these steps:

1. Generate ABNF rules out of the RFC using the presented ABNF extractor approach
2. Transform the ABNF ruleset to an XSD tree
3. Generate XML test data based on the XSD using the test data generation framework
4. Transform XML test data to SIP messages and validate them

Using the presented approach, a large amount of valid and invalid test data for SIP systems based on RFC 3261 could be generated. While the XSD contains restrictions which describe valid instances, invalid instances can be generated by

[2] http://security.inso.tuwien.ac.at/esse-projects/fuzzolution/

violating the constraints defined in the XSD, e.g., violating regular expression patterns or violating structural restrictions, e.g., omitting required elements.

To validate the generated valid data, a custom stand–alone tool was developed that transforms XML messages back to raw SIP messages. The resulting SIP messages were validated using APG[3], an ABNF parser. Part of the set of tools is a set of ABNF rules for SIP messages. These rules were used to show that the validity of the generated SIP messages. Provided that the external ABNF ruleset is valid, it was thus shown that the generated SIP messages are valid with respect to the RFC.

5.2 Learnings and Limitations of the Proposed Approach

Compared to the ABNF extraction of HTTP (RFC 2616), which needed a quite large amount of rule adaption iterations, SIP (RFC 3261) only needed a couple of iterations. As in RFC 2616, an informal note was parsed as part of one rule and had to be redefined. In contrast to RFC 2616 no syntactical errors were detected in rules. Also in contrast to RFC 2616, having quite a lot of dependencies, is was only necessary to import two other RFCs to fix missing rule errors. RFC 2806 could be imported directly, and RFC 1035 was written in BNF and had to be transformed to ABNF.

It was observed that the generated XSD file did not include all desired structural restrictions. A very specific example is that it is not possible to set required and optional message headers for different SIP methods (e.g., REGISTER and INVITE) individually. Instead, the message header elements and the SIP method are independent choice elements in XSD. This means that all permutations of those two groups are allowed. However, this is not a restriction of this approach, because the ABNF rules are not more restricted in the RFC.

5.3 Discussion of Test Data Generation from RFC

The framework used for data generation, fuzzolution, makes it possible to generate a large number of data files with little risk of memory shortages. The problem, however, lies in the large possibilities of combinations of possible test data instances.

The XSD file for the test data generation of SIP messages contains about 8000 lines of code. In its most basic configuration, the framework generates all possibilities, i.e. each combination of possible structural and data–related instances is generated. Because this schema file contains many choice indicators and optional elements, the number of possibilities of valid XML files is very large. Recursive structures are used (an element might contain itself), so it is not even possible to generate all instances because an infinite number of possibilities exists. All those problems are considered and resolved in our generic data generation approach [21] by configuration of the framework.

[3] http://www.coasttocoastresearch.com/apg

Missing ABNF rules have to be reviewed manually, deciding whether to import, stub or write the rule. This could be partially automated by parsing the text for referenced RFCs. The proposed approach includes fetching potentially interesting RFCs, searching the RFCs and ordering based on distance heuristics and importing of the top–rated RFC, with eventual user intervention.

6 Conclusion and Further Work

In this paper, an approach of generating test data from RFCs was presented. This is done by extracting an ABNF model from an existing RFC. Based on this model, an XSD file is generated, which in turn is the input for a data generation software which generates data in XML format. This test data might be used for testing a SUT. For the extraction of an ABNF model, a heuristic extractor by Fenner [9] is extended to get a valid and self-contained ruleset. Using the proposed approach makes it possible to semi–automatically generate test data based on an RFC. The application of the approach was shown using SIP and proved to be able to generate valid and invalid test data to test a system.

Future areas of work include the stateful representation of SIP in this approach, and the improvement of the transformation from ABNF to XSD in order to test more aspects of the system. In the SIP example, this means to make a distinction between allowed message headers for each allowed message.

The ABNF rules are only a small part of RFCs in comparison to the text length. They mostly consist of natural language descriptions discussing the field of interest. This information could be helpful constructing test data, or distinguish between valid and invalid variations. These include for example semantic relations, constraints, examples and references to other RFCs. The approach proposed in this paper might support additional research in implementing a parser looking for those natural language patterns.

With the presented approach, an automated extraction of a model for the generation of test data is possible. With the presentation of the transformation from ABNF to XSD, a generic data generation approach for different protocols is possible. This allows clear separation of concerns for test tools and a focus on a robust test generation logic.

References

1. A. Aboulnaga, J. F. Naughton, and C. Zhang. Generating synthetic complex-structured XML data. In *In Proc. 4th Int. Workshop on the Web and Databases (WebDB 2001)*, 2001.
2. A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Trans. Softw. Eng.*, 38(2):258–277, Mar. 2012.
3. D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a template-based data generator for XML. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 616–616, New York, NY, USA, 2002. ACM.

4. A. Bertolino, J. Gao, E. Marchetti, and A. Polini. TAXI–a tool for XML-based testing. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 53–54, Washington, DC, USA, 2007. IEEE Computer Society.
5. J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329, New York, NY, USA, 2007. ACM.
6. M. Carbin and M. C. Rinard. Automatically identifying critical input regions and code in applications. In *ISSTA*, pages 37–48, 2010.
7. E. Crocker and P. Overell. Augmented bnf for syntax specifications: Abnf, 2008.
8. C. Ebert and R. Dumke. *Software Measurement: Establish - Extract - Evaluate - Execute*. Springer, 1 edition, Aug. 2007.
9. B. Fenner. Bill fenner's abnf extractor. `http://code.google.com/p/bap/source/browse/trunk/aex`. [accessed: 2013-01-21].
10. J. Fong, F. Pang, and C. Bloor. Converting relational database into XML document. pages 61 –65, 2001.
11. I. E. T. Force. Overview of rfc decument series. `http://www.rfc-editor.org/RFCoverview.html`. [accessed: 2013-01-26].
12. W. Gutjahr. Partition testing vs. random testing: the influence of uncertainty. *Software Engineering, IEEE Transactions on*, 25(5):661–674, Sept./Oct. 1999.
13. ITU-T. X.693 information technology ASN.1 encoding rules: XML encoding rules (XER), Nov. 2008. Identical standard: ISO/IEC 8825-4:2008 (Common).
14. M. Jacinto, G. Librelotto, J. Ramalho, and P. Henriques. Bidirectional conversion between XML documents and relational databases. pages 437 – 443, 2002.
15. D. D. McCracken and E. D. Reilly. Backus-naur form (bnf). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley and Sons Ltd., Chichester, UK.
16. P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.
17. C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, 27(12):1085–1110, Dec. 2001.
18. J. Myers and M. Rose. The content-md5 header field, 1995.
19. J. C. B. Ribeiro, M. A. Zenha-Rela, and F. F. de Vega. Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software. *Information and Software Technology*, 51(11):1534 – 1548, 2009.
20. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. RFC 3261: SIP - Session Initiation Protocol.
21. C. Schanes, F. Fankhauser, S. Taber, and T. Grechenig. Generic data format approach for generation of security test data. In *The Third International Conference on Advances in System Testing and Validation Lifecycle, October 2011, Barcelona, Spain*. IEEE Computer Society Press, Oct. 2011.
22. E. Standard. EBNF: ISO/IEC 14977: 1996 (E). *URL http://www.cl.cam.ac.uk/mgk25/iso-14977.pdf*.
23. T. Stefanec and I. Skuliber. Grammar-based sip parser implementation with performance optimizations. In *Telecommunications (ConTEL), Proceedings of the 2011 11th International Conference on*, pages 81 –86, June 2011.
24. D. Van Deursen, C. Poppe, G. Martens, E. Mannens, and R. Walle. XML to RDF conversion: A generic approach. pages 138 –144, Nov. 2008.
25. N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, Nov. 1977.