A DOMAIN SPECIFIC LANGUAGE FOR THE AUTOMATIC GENERATION OF PARSER CLASSES FOR TEXT PROTOCOLS

Thomas Kistel, Ralf Vandenhouten

Abstract

ABNF is a language for the definition of the formal syntax of technical specifications and is widely used for the definition of textual protocol messages of many internet protocols. The automatic generation of parser classes for ABNF specifications is currently very limited, because ABNF only defines the transfer syntax of the text messages and does not define names for the set of production rules. The lack of name definitions within ABNF rules does not allow to automatically generate expressiveness and meaningful program code for ABNF specifications. In this paper we present X-ABNF, which is a domain-specific language (DSL) for the definition of name-bindings for ABNF rules. The name-bindings with X-ABNF facilitates to generate a concise and meaningful code for an ABNF specification. Additionally, we show that the name-binding can also be used for language extensions through macro programming to dynamically access ABNF encoded text data within source code. We have used Xtext and Xtend for the implementation of the language grammar of ABNF and X-ABNF, which provide good tool support and code generation capabilities.

Zusammenfassung

ABNF ist eine Sprache zur Definition einer formalen Syntax für technische Spezifikationen und wird häufig zur Beschreibung textueller Nachrichten von Internetprotokollen eingesetzt. Die Möglichkeiten der automatischen Generierung von Parser-Klassen aus ABNF-Spezifikationen sind derzeit sehr begrenzt, da ABNF lediglich die Transfersyntax und Produktionsregeln von Textnachrichten beschreibt. Die fehlende Definition von Variablennamen innerhalb einer ABNF-Spezifikation ermöglicht es nicht, sinnvollen und ausdrucksstarken Programmcode zu generieren, der von einem Programmierer verwendet werden kann. In diesem Artikel stellen wir X-ABNF vor, eine domänenspezifische Sprache (DSL) zur Definition von Variablennamen für ABNF-Regeln. Dies ermöglicht die Generierung von ausdrucksstarkem und lesbarem Programmcode aus ABNF-Spezifikationen. Des Weiteren zeigen wir, dass dieser Ansatz auch für Spracherweiterungen mithilfe von Makroprogrammierung genutzt werden kann. Dies ermöglicht die dynamische Instanziierung von ABNF-Textnachrichten im Programmcode. Wir verwenden Xtext und Xtend zur Implementierung der Sprachgrammatik von ABNF und X-ABNF, was eine gute Werkzeugunterstützung gewährleistet und Codegenerierung ermöglicht.

I. INTRODUCTION

The Augmented BNF for Syntax Specification (ABNF) (IETF, RFC 5234, 2008) is a grammar language for defining the formal syntax of technical specifications. ABNF is widely used for the specification of textual messages of many internet protocols. Although ABNF is very popular and a relatively compact language there is insufficient tool support (e.g. advanced editing, syntax highlighting, validation and error reporting) for software development. Furthermore, an important task is to automatically generate a program code for different general purpose languages (GPL) like Java or C++ to decode/encode text messages according to an ABNF specification.

Besides many IETF Internet protocols ABNF is also applicable for different vertical industry notations like NMEA[1] or EDIFACT[2], for example. The integration of a NMEA or EDIFACT parser into software applications is mostly done through specific software libraries, which provides a set of API functions and classes to decode, encode and manipulate NMEA, EDIFACT, or other text messages. The drawbacks are that software libraries typically also include much functionality, which may not be required by the software application or make use of sophisticated design patterns that have influences on, or more seriously, violate the overall software architecture. Another option for the integration of ABNF parsers into software applications is to create an individual parsing library that best fits to the surrounding software application. For text-based protocols, the creation of a specific parser can be a tedious and error prone task.

The main problem of transforming ABNF into source code of a GPL is that ABNF defines the transfer syntax of text-based messages. Therefore ABNF only consists of a set of production rules, and the context-free grammar of ABNF does not define any names (except for the rule itself), whereas the context-free grammar of a GPL defines names for class, variable, or method declarations. This naming issue leads to a semantic gap between ABNF and a GPL. Therefore existing parser generators for ABNF like APG (APG – ABNF Parser Generator) or aParse (aParse - parser generator for Augmented BNF)

[1] www.nmea.org
[2] www.unece.org/cefact/edifact/welcome.html

only provide raw parsing capabilities of ABNF files and generic access methods.

In this paper we present X-ABNF, an approach to close the semantic gap between ABNF and GPLs by using domain specific modelling. X-ABNF is a domain-specific language (DSL) that we have built on top of an ABNF parsing framework. X-ABNF allows to define variable names for ABNF constructs and a mapping mechanism to GPL procedures. We have used the Xtext framework (Xtext Project Website) and the Xtend language (Xtend project website) for the implementation of our approach. We have developed a code generator of X-ABNF for the Java programming language so far, but code generation for other languages can be included as well. In summary our solution makes the following contributions:

1. **IDE functions:** Provision of common IDE functions like advanced editing capabilities for ABNF and X-ABNF files with syntax highlighting, code completion and validation rules.

2. **Code Generation:** The generation of Java code for ABNF data is highly adaptable through the X-ABNF file. In the X-ABNF file each rule and rule reference can be mapped to specific names. Therefore the resulting program code is very concise and expressive by making less usage of library code.

3. **Macro programming:** Macro programming is known from the Lisp programming language and can be used for language extensions at compile time. We have implemented macro programming with Active Annotations, a language feature of the most recent version (2.4) of the Xtend language. With X-ABNF and macro programming it is possible to access ABNF encoded text data at compile time, which is a very useful mechanism for software testing with real protocol data.

In the next section we introduce the main concepts of ABNF and outline the aspects which are important for our presented approach. In section III we describe our solution in detail. In section III.I we present the IDE features for editing ABNF and X-ABNF files, in section III.II we explain how the code generation is done with name bindings (see section III.III), and in section III.IV we introduce the details of the macro programming mechanism. In section IV we compare our solution with other ABNF parsing frameworks, and finally, section V concludes this paper and outlines future work.

II. THE ABNF SPECIFICATION

ABNF is a context-free grammar that is a modified version of the Backus-Naur Form (BNF). ABNF allows the specification of so called *Rules*. On the left-hand side, each rule has an identifying name. The right-hand side of a rule is the rule's definition that can be a *Concatenation, Alternation, Repetition* or an *Element*. Elements can also be grouped or optional. An element is either a name reference to another rule, a character or terminal value. Terminals can be specified as binary, decimal or hexadecimal values and are mapped to a specific encoding. For example the ABNF specification (IETF, RFC 5234, 2008) outlines that the terminal values `%d13` or `%x0D` specify the decimal and hexadecimal representations of the carriage return in US-ASCII.

According to (IETF, RFC 5234, 2008) the differences between BNF and ABNF involve the naming of rules, repetitions, alternatives, order-independence, and value-range. Repetitions in BNF, for example, have to be specified via recursion. In BNF it is not possible to specify an upper bound to explicitly terminate the recursion. ABNF in contrast allows the definition of repetitions with upper and lower bounds. For example the rule `IPv4address = 1*3DIGIT "." 1*3DIGIT "." 1*3DIGIT "." 1*3DIGIT` specifies the syntax of an IPv4 address, where each part can contain one (lower bound) to three (upper bound) digits. In contrast to the rule `IPv4address` the rule `Hostname = 1*ALPHA` has a repetition with no upper bound, i.e. infinitive number of ALPHA's. The rules `DIGIT` and `ALPHA` in turn can be specified with an alternation (`DIGIT = "0" / "1" / "2" / … "8" / "9"` or `ALPHA = "a"/ "b" / … "z" / "A" / "B" / … „Z"`) that is also possible in BNF. ABNF additionally allows the specification of value ranges (i.e. `DIGIT' = %x30-39` and `ALPHA' = %x41-5A / %x61-7A`). The hexadecimal value representations are mapped into a specific character set (e.g. US-ASCII). Therefore the above rules are equivalent: DIGIT ↔ DIGIT' and ALPHA ↔ ALPHA'. The value ranges of ABNF are similar to regular expressions, but less expressive.

III. SOLUTION

The transformation of an input source into a target language, i.e. the transformation of an ABNF document into Java source code, is a typical compiler construction task. The first phase of compiler construction is characterized by the creation of a parser that transforms the input sequence into an abstract syntax tree (AST). The AST can then be traversed to generate code. In this chapter we explain the creation of the parser for ABNF documents and the surrounding IDE support. Then we describe how code generation is done with the X-ABNF mapping language. In the third part we explain how the implemented parsing and code generation technology with X-ABNF can be used for macro programming. We use the NMEA 0183 standard, whose syntax can be described with ABNF, to demonstrate the implementation details.

III.I ABNF PARSER AND WORKBENCH

As mentioned above, we have used the Xtext framework (Xtext Project Website) for the implementation of an ABNF parser and surrounding tool support for the Eclipse workbench. Xtext is a metamodel-based framework for the implementation of DSL's like ABNF or ASN.1 (ITU, X.680, 11/2008; Kistel, Vandenhouten 2013). The corner stone of implementing a language with the Xtext framework is to define an EBNF-like grammar file. Xtext uses this grammar file to generate Parser, Lexer, an EMF Ecore metamodel, Editor, and other Eclipse workbench functions. The underlying parser/lexer technology of Xtext is ANTLR (ANTLR website; Parr op. 2007), a two phase LL(*) parser, which creates the concrete

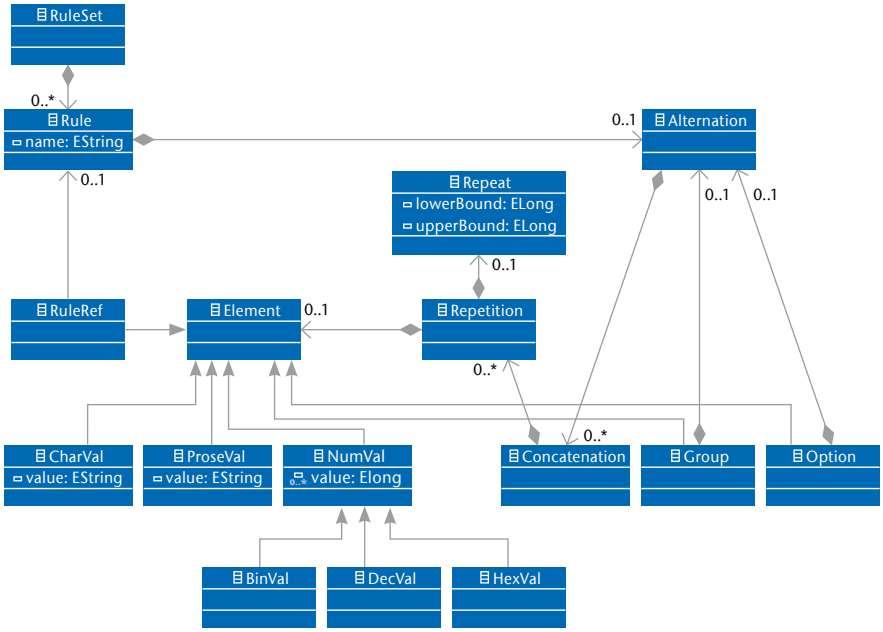

Fig. 1) Metamodel of ABNF

syntax of the language. The Xtext framework translates the Xtext grammar file into a grammar description of ANTLR. In Xtext, the parsing result, i.e. the abstract syntax tree, is represented by an EMF metamodel. **Figure 1** shows the metamodel of the abstract syntax of ABNF.

The entry point of the metamodel is a `RuleSet` which contains a set of `Rule` elements. Each `Rule` contains `Alternation` elements, an `Alternation` itself contains `Concatenation` elements which concatenate `Repetition` elements. A `Repetition` refers to an ABNF `Element` and optionally has a `Repeat` variable. An ABNF `Element` can either be a `CharVal` (a String), `ProseVal`, `NumVal` (i.e. binary, decimal or hexadecimal values), a reference to another rule (`RuleRef`), an `Option` or a `Group`. `Option` and `Group` elements refer to `Alternation`.

Optional elements in ABNF can be declared in two ways, either by an `Option` or a `Repetition` element with lower/upper bound set to 0/1. Therefore `Rule1 = 0*1"optional"` and `Rule2 = ["optional"]` are equivalent, but have a different representation in the abstract and concrete syntax (i.e. `Rule1` is represented by a `Repetition`, whereas `Rule2` is represented by an `Option` element). In the backend phase of a compiling process, e.g. for code generation (see section III.II), these ambiguities must be resolved.

Besides lexing and parsing, i.e. the transformation of tokens into an abstract syntax tree, another important step in the front end phase of a compiler is the semantic analysis of the language. With the Xtext framework this can be done through validation rules. We have implemented different validation rules. For example, we check for unique rule names and validate the input sequences of the `NumVal` elements.

nmea.abnf

```
1  ; General specifications
2  Float = *DIGIT "." *DIGIT
3  ; NMEA-specifications
4  NMEA =  "$" (GPGLL / GPGGA / GPRMC) CRLF
5
6  ; Geographic Position - Latitude/Longitude
7  GPGLL = "GPGLL,"            ; Prefix
8          [Float] ","         ; Latitude
9          ["N" / "S"] ","     ; N or S (North or South)
10         [Float] ","         ; Longitude
11         ["E" / "W"] ","     ; E or W (East or West)
12         [Float] ","         ; Universal Time Coordinated (UTC)
13         [("A" / "V")] "*"   ;  A - Data Valid, V - Data Invalid
14         [2*2(ALPHA)]        ; Checksum
15
```

Fig. 2) Specification of an NMEA message

The created workbench provides reasonable IDE features for editing ABNF files. We customized the default implementation of the generated Xtext framework for custom syntax highlighting, code completion, code formatting, outline view and, most importantly, syntax and error validation. Through the implementation of a linking mechanism, it is possible to split an ABNF specification into multiple files. This allows to refer an ABNF rule that is defined in a separate file in order to avoid duplicate code. **Figure 2** shows the specification of the NEMA message GPGLL (Geographic Position - Latitude/Longitude) with the created workbench editor. We will refer to this message specification in the next sections.

III.II CODE GENERATION

Code generation is a part of the backend phase of a compiler, which usually creates machine or interpreter code. In our implementation the code generation phase creates Java source code. In MDSE terms, code generation is also referred to as a model to text (M2T) transformation (Flores Beltran et al. 2007), because generally a M2T generator may not only create programming code, but also other artefacts like configurations, database or user-interface scripts.

Model to text transformation can have different strategies (Hemel et al. 2008; Völter, Kolb 10/11/2006). **Figure 3** shows two different strategies. In strategy 'a' the source model (the metamodel of ABNF) is directly transformed into the target model (Java code). In this strategy, it is difficult to add optimization procedures. Therefore many compilers create an intermediate model (strategy 'b') before the target model is created. **Figure 4** shows the intermediate model of the ABNF code generator. Each ABNF `Rule`

is represented by an `AbnfRuleClass`, e.g. the ABNF rule `GPGLL` of the NMEA example (**see Figure 2**) is transformed to an instance of `AbnfRuleClass` with `className = "GPGLL"`. Every `Element` (**see Figure 1**) will be translated into an `AbnfElementField`. An `AbnfElementField` stores a repeat variable (`AbnfRepeat`) with lowerBound and upperBound. The default value is 1/1, optional elements will be stored as 0/1, `Repeat` elements are directly mapped to `AbnfRepeat`.

An `AbnfElementField` is an abstract class, the concrete subclasses are `AbnfCharValue` and `AbnfAttribute`. An `AbnfCharValue` is just a representation of `CharValue` (see metamodel in **Figure 1**). An `AbnfAttribute` stores a reference to another rule in the `type` variable, the `name` variable can be either derived from the type or from a name binding, which we will describe in section **III.III.** For example, the rule reference *Float* (line 8 in the NMEA example in **Figure 2**) is translated into an `AbnfAttribute(type=Float, name=float)`. Alternations in ABNF are transformed into instances of `AbnfAlternation`, which is a subclass of `AbnfAttribute`. An `AbnfAlternation` contains a list of values of type `AbnfElementField`. Alternations that are references to other rule definitions will be transformed to super class instances of an `AbnfRuleClass`. In the NMEA example of **Figure 2** the `NMEA` class will be the super class of `GPGLL`, `GPGGA` and `GPRMC`. Furthermore, objects of type `NumVal` in the metamodel (i.e. binary, hexadecimal or decimal values) will be translated into `AbnfNumVal` of the intermediate model.

In summary, the intermediate model (**Figure 4**) is an optimized structure of the ABNF metamodel (**Figure 1**). Optional elements and repetitions of the metamodel are transformed into repeat elements in the intermediate model, group elements are completely removed because they are only necessary for structuring ABNF data. The intermediate model is used to generate the code artefacts of the target programming language. **Table 1** shows how the main elements of the intermediate model are represented in Java code.

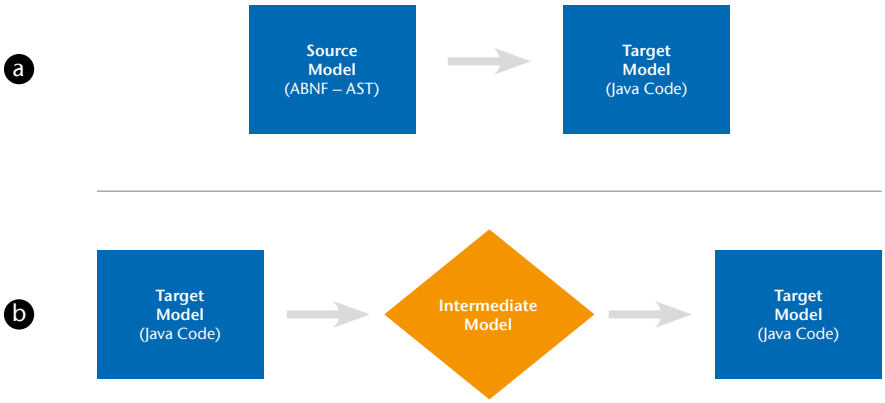

Fig. 3) Strategies of a code generator

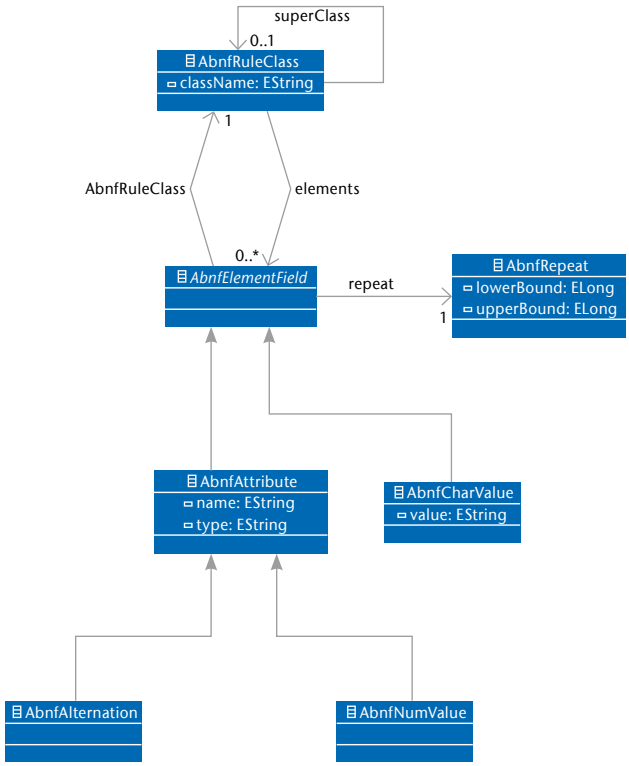

Fig. 4) Intermediate model of the code generator

Intermediate model	Java representation
AbnfRuleClass	Object class
AbnfAttribute	Class field with setter and getter method
AbnfAlternation	Enum class

Tab. 1) Transformation of intermediate model to code

The code generator creates a Java class for each AbnfRuleClass. A generated Java class consists of a field section, where each AbnfAttribute will become a Java field with a corresponding setter- and getter-method. If the upper bound of the AbnfRepeat is greater than one, the type of the Java field will be java.util.List. An AbnfAlternation is transformed to a Java Enum class, where each value of the AbnfAlternation will be an Enum value. For each AbnfElement-Field (i.e. AbnfAttribute, AbnfAlternation, AbnfNumValue and AbnfCharValue) a read/write entry is generated in the read/write method. **Listing 1** shows a snippet of the generated Java classes for the NMEA and GPGLL rules in the NMEA example of **Figure 2**.

```
public class GpGll extends AbstractNmeaObject {
      private Float latitude;
      // other fields ...
      public Float getLatitude() {
            return latitude;
      }
      public void setLatitude(Float latitude) {
            this.latitude = latitude;
      }
      // other setter/getter
      // and read/write methods ...
}
```

Listing 1) Generated Java class for GPGLL message

III.III NAME BINDING WITH X-ABNF

An important requirement on code generation is that the resulting source code is understandable for the programmer, i.e. the source code must have meaningful names for classes, methods, and fields. In contrast, the ABNF grammar does not define names (except for the rules), but only the transfer syntax. In the NMEA example of figure **Figure 2** the GPGLL rule defines a Float variable, the semantic information of the name of that variable cannot be derived from the ABNF specification, which would lead to a semantic gap between the ABNF specification and the generated source code.

To close this gap, we have developed X-ABNF that is a DSL for defining name bindings for ABNF. X-ABNF allows the definition of a *Generator model* and a *Binding model*. The *Generator model* configures the overall code generation process, and the *Binding model* allows defining specific name bindings for ABNF elements. **Table 2** explains the elements of the generator and binding model of X-ABNF.

Model element	Description
file	A reference to the ABNF file.
language	The target language that should be generated (currently only Java is supported).
package	An optional specification of the package of the resulting code.
Encoding	An optional specification of the encoding that should be used (default is US-ASCII).
RuleBinding	A binding of an ABNF rule to a specific class name. A RuleBinding can contain several AttributeBindings.
AttributeBinding	A binding of an ABNF element (i.e. an AbnfAttribute of the intermediate model) to a specific field name in the target class.
ClassBinding	A binding of an ABNF rule to a target library class.

Tab. 2) Elements of the X-ABNF Generator and Binding model

The specification of a RuleBinding allows the definition of class names for ABNF rules. In our NMEA example (see **Listing 1**) the rule NMEA is bound to the name AbstractNmeaObject because this class is an abstract super class of GPGLL, GPGGA and GPRMC. An AttributeBinding allows the definition of a specific name for an ABNF rule reference (e.g. the first Float variable of the GPGLL is bound to the name latitude, because it specifies the latitude of a GPGLL message).

The ClassBinding can be used to statically bind an ABNF rule to an existing library class. The ClassBinding definition also defines a method binding that provides the object conversion. In the target source code the method that is bound in the class binding is called to convert the input sequence (i.e. a String). This way the rule Float of the NMEA example is bound to the Java class Float with the method binding java.lang.Float.valueOf(java.lang.String). The code generator ensures that every ABNF rule Float is then converted into an object of java.lang.Float. This mechanism allows to define predefined classes for specific ABNF rules that can be reused for different ABNF specifications. We have also implemented static validation rules that ensure that the class bindings are valid and refer to existing classes and methods in the library path. **Figure 5** shows the X-ABNF specification for the NMEA example that is used to generate the code in **Listing 1**.

In the code generation progress the generator consults the X-ABNF specification to derive the name of a specific ABNF rule or rule reference. The name bindings are derived when the intermediate model is transformed into the target code.

III.IV MACRO PROGRAMMING

Macro programming is a language-extension mechanism and gained popularity with the Common Lisp programming language. OpenJava (Tatsubori et al. 2000) introduced macro programming as a language extension mechanism for Java at compile time, which uses some features of OpenC++ (Chiba 1995). Erdweg (Erdweg 2013) discussed macro programming in the context of domain-specific programming/languages and compared different approaches (also OpenJava, among others). Macro programming is a useful mechanism for several purposes. (Xtend project website) and (Tatsubori et al. 2000) claim

```
nmea.xabnf
1  GeneratorModel {
2      file = "nmea.abnf"
3      language = Java
4      package = "de.thwildau.tm.moses.nmea"
5      encoding = "US-ASCII"
6  }
7
8  Bindings {
9      rule Float toClass "java.lang.Float" toMethod "valueOf(java.lang.String)"
10     rule NMEA to "AbstractNmeaObject"
11     rule GPGLL to "GpGll" {
12         ref "field1" Float to "latitude"
13         ref "field2" LatitudeOrientation to "latitudeOrientation"
14         ref "field3" Float to "longitude"
15         ref "field4" LongitudeOrientation to "longitudeOrientation"
16         ref "field5" Float to "utcTime"
17         ref "field6" to "status"
18     }
19     // other bindings ...
20 }
```

Fig. 5) X-ABNF specification of the NMEA

that macro programming can help to substitute redundant "boilerplate" code that is necessary to use design patterns (Gamma et al. 1995). This is because many design patterns require the implementation of a skeleton of several classes that interact in a specific way. For example the *Observer pattern* (Gamma et al. 1995) requires the implementation of an Observer interface that can be attached to a *subject class*. The *subject class* notifies a concrete Observer implementation on internal state changes. Usually the Observer pattern requires the implementation of at least four different classes. This programming overhead can be reduced with macro programming.

We argue that macro programming is also very useful for protocol development, because the implementation of a communication protocol for a software application also requires testing the implementation and the conformance to the protocol specification. Implementation and conformance tests often require the development of *Mock Objects*, which are dummy classes that emulate real behavior (Mackinnon et al. 2001). The development of *Mock Objects* for protocol messages that behave like real messages can be a complicated and tedious task.

The recent version (2.4) of the Xtend language (Xtend project website) allows the specification of *Active Annotations*. The *Active Annotations* are an implementation of macro programming for the Xtend language and are similar to OpenJava (Tatsubori et al. 2000). The Xtend language is built on top of the Java programming language and therefore has many features of

```
@Target(ElementType.TYPE)

@Active(AbnfDataProcessor.class)
public @interface AbnfData {


      String file();

      String data();

}
```

Listing 2) Java Annotation AbnfData

Java (e.g. declarations of classes, methods, and fields). The Xtend compiler transforms an Xtend class into a Java class and creates Java source code. *Active Annotations* can be declared in Xtend source files like regular Java annotations. The *Active Annotations* specify a processor class, which acts as a call-back class for the Xtend compiler.

The Xtend compiler calls the processor class to extend the Xtend language with a macro at compile time. Therefore the processor reads the content of a declared *Active Annotation* and provides specific language extensions for the given class. **Listing 2** shows the definition of the `AbnfData` annotation that can be declared as an *Active Annotation* on Xtend type declarations.

The Annotation `@Active(AbnfDataProcessor.class)` specifies the call-back processor class for the Xtend compiler. The content of the `AbnfData` annotation is the specification of a `file` attribute that must refer to an XABNF file. The `data` attribute specifies concrete data, whose content is defined by an ABNF grammar. **Figure 6** shows an example for an `AbnfData` annotation with an NMEA message.

In the `NmeaMock` example class the `file` attribute refers to the *nmea.xabnf* file which must be located in the same directory as the Xtend file. The `data` attribute contains a concrete *GPGLL* message that is part of the NMEA-ABNF specification. The `AbnfDataProcessor` class (see **Listing 2**) parses the *GPGLL* message with the information of the nmea.xabnf file. The processor class uses Java reflection to dynamically instantiate the required parser classes (`GpGll.java` in our example) that were generated by the code generator (see section III.II). The processor creates extension methods for all *get-methods* of a parser class. The result of a *get-method* is determined by invoking the corresponding method of the parser class via Java reflec-

```
*NmeaMock.xtend
4  @AbnfData(
5      file = "nmea.xabnf",
6      data = "$GPGLL,1131.330,N,04344.089,E,160029.77,A*0C"
7  )
8  class NmeaMock {
9
10     def static void main(String[] args) {
11         val mock = new NmeaMock()
12         println("UTC-Time: " + mock.utcTime)
13         println("Longitude: " + mock.longitude)
14         println("Latitude: " + mock.l
15     }
16 }
17
18
19
```

```
latitude : Float - NmeaMock.getLatitude()
latitudeOrientation : LatitudeOrientation -
longitude : Float - NmeaMock.getLongituc
longitudeOrientation : LongitudeOrientatic
```

Fig. 6) Class NmeaMock with AbnfData annotation

tion. Informally spoken, the processor class of `AbnfData` annotation passes the NMEA message of the annotation to the generated parser classes by invoking the parser classes via Java reflection. The *get-methods* and the results of this invocation is dynamically inserted into the resulting Java code of the Xtend class. This mechanism allows the programmer to instantiate the methods of the parser class within the Xtend class. In our example (see **Figure 6**) the result of `mock.getLatitude()` is the float value "1131.330", which is retrieved from the data attribute of the `AbnfData` annotation.

IV. RELATED WORK

As indicated in the introduction section, the ABNF grammar only describes the transfer syntax of text-based messages instead of their abstract syntax and encoding rules. Existing parsers for ABNF like APG (APG – ABNF Parser Generator) or aParse (aParse - parser generator for Augmented BNF) therefore only provide generic access and simple validation methods for ABNF documents. A similar approach to generic parsing solutions is the generation of regular expressions from ABNF grammars (abnf2regex). Schulz (Schulz 2004) discusses the issue of the mixture of abstract and transfer syntax in ABNF in the context of conformance testing of text-based protocols. He solves this problem with a derivation of an abstract protocol type definition by mapping ABNF rules to TTCN-3[3] types. This solution has the drawback that the mapping also has to be specified, and it depends on a different technology (i.e. TTCN-3).

V. CONCLUSIONS

In this paper we have presented X-ABNF, which is a domain-specific language (DSL) for the definition of name-bindings of a given ABNF specification. The definition of name-bindings for ABNF through X-ABNF allows to handle ABNF like a programming language with variable identifiers. We have shown that the name-binding is particularly useful for generating ABNF parser classes for ABNF specifications that are understandable for the programmer. Additionally we outlined how the name-binding of X-ABNF can be used with macro programming for language extensions at compile time. We have used Active Annotations, a language feature of Xtend, to implement the language extension. This mechanism enables the programmer to access ABNF data in Xtend classes. Our implementation of the ABNF and X-ABNF grammar with the Xtext framework also provides a state-of-the-art tool support and good integration with Eclipse EMF.

In the future, we plan to improve the implementation of the code generator. Currently the transformation of the ABNF metamodel to the intermediate model is implemented with Xtend. This transformation can also be done by a model transformation language like ATL (Jouault, Kurtev 2006). Furthermore, we plan to extend the use of the `AbnfData` not only to type definitions, but also to method definitions. This allows to specify ABNF encoded text messages on method level, which would improve the implementation of unit test classes.

The source code of the described project is available online at code.google.com/p/moses-project.

[3] www.ttcn-3.org

ACKNOWLEDGMENTS

This work was funded by the Federal Ministry of Education and Research (BMBF) in the project MOSES - Modellgetriebene Software-Entwicklung von vernetzten Embedded Systems (FKZ 17075B10).

LITERATURE

ANTLR website. Available online at http://www.antlr.org, checked on 9/7/2012.

Xtend project website. Available online at http://www.eclipse.org/xtend, checked on 7/23/2012.

ITU X.680, 11/13/2008: Abstract Syntax Notation One (ASN.1) - Specification of basic notation.

IETF RFC 5234, January 2008: Augmented BNF for Syntax Specifications: ABNF. Available online at http://tools.ietf.org/html/rfc5234.

Chiba, Shigeru (1995): A metaobject protocol for C++. In SIGPLAN Notices 30 (10), pp. 285–299.

Coast to Coast Research, Inc: APG – ABNF Parser Generator. Available online at http://www.coasttocoast-research.com, checked on 6/10/2013.

Eclipse Foundation: Xtext Project Website. Available online at http://www.eclipse.org/Xtext, checked on 7/15/2012.

Erdweg, Sebastian Thore (2013): Extensible Languages for Flexible and Principled Domain Abstraction. Dissertation. Philipps-Universität Marburg, Marburg. Department of Mathematics and Computer Science.

Flores Beltran, Juan Carlos; Holzer, Boris; Kamann, Thorsten; Kloss, Michael; Mork, Steffen A.; Niehues, Benedikt; Thoms, Karsten (2007): Modellgetriebene Softwareentwicklung. MDA und MDSD in der Praxis. Edited by Jens Trompeter, Georg Pietrek. Frankfurt [Main]: Entwickler.press.

Gamma, Erich; Helm, Richard; Johnson, Ralph E. (1995): Design patterns. Elements of reusable object-oriented software. Reading, Mass: Addison-Wesley.

Hemel, Zef; Lennart C. L. Kats; Visser, Eelco (2008): Code Generation by Model Transformation. In : Proceedings of the 1st international conference on Theory and Practice of Model Transformations. Zurich, Switzerland: Springer-Verlag, pp. 183–198.

Jouault, Frédéric; Kurtev, Ivan (2006): Transforming Models with ATL. In Jean-Michel Bruel (Ed.): Satellite Events at the MoDELS 2005 Conference, vol. 3844: Springer Berlin Heidelberg (Lecture Notes in Computer Science), pp. 128-138.

Kistel, Thomas; Vandenhouten, Ralf (2013): A metamodel-based ASN.1 editor and compiler for the implementation of communication protocols. In Wissenschaftliche Beiträge TH Wildau, pp. 61–66.

Mackinnon, Tim; Freeman, Steve; Craig, Philip (2001): Endo-testing: unit testing with mock objects. In Giancarlo Succi, Michele Marchesi (Eds.): Extreme Programming examined. Boston: Addison-Wesley (The XP series), pp. 287–301.

Parr, Terence (op. 2007): The Definitive ANTLR reference guide. Building domain-specific languages. Raleigh [etc.]: The Pragmatic bookshelf.

parse2: aParse - parser generator for Augmented BNF. Available online at http://www.parse2.com, checked on 6/10/2013.

Schulz, Stephan (2004): Derivation of Abstract Protocol Type Definitions for the Conformance Testing of Text-Based Protocols. In Roland Groz, RobertM Hierons (Eds.): Testing of Communicating Systems, vol. 2978: Springer Berlin Heidelberg (Lecture Notes in Computer Science), pp. 177-192.

Tatsubori, Michiaki; Chiba, Shigeru; Killijian, Marc-Olivier; Itano, Kozo (2000): OpenJava: A Class-Based Macro System for Java. In Walter Cazzola, RobertJ Stroud, Francesco Tisato (Eds.): Reflection and Software Engineering, vol. 1826: Springer Berlin Heidelberg (Lecture Notes in Computer Science), pp. 117-133.

Thomson, Martin: abnf2regex. Available online at https://github.com/martinthomson/abnf2regex, checked on 6/10/2013.

Völter, Markus; Kolb, Bernd (2006): Best Practices for Model-to-Text Transformations. Eclipse Summit 2006, 10/11/2006.

AUTHORS

Thomas Kistel, M. Eng
Fachgebiet Telematik
Fachbereich Ingenieur- und Naturwissenschaften
Technische Hochschule Wildau [FH]
T +49 3375 508-615
thomas.kistel@th-wildau.de

Prof. Dr. rer. nat. Ralf Vandenhouten
Fachgebiet Telematik
Fachbereich Ingenieur- und Naturwissenschaften
Technische Hochschule Wildau [FH]
T +49 3375 508-359
ralf.vandenhouten@th-wildau.de