RZ 3098 (# 93144) 01/11/99
Computer Science/Mathematics 13 pages

Research Report

Automatic Generation of Java Stubs for ASN.1

Andreas Schade

IBM Research Division
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

IBM Research Division
Almaden · Austin · Beijing · Haifa · T.J. Watson · Tokyo · Zurich

Automatic Generation of Java Stubs for ASN.1

Andreas Schade

IBM Research Division, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

This paper describes the *Snacc for Java* project, aiming at developing an ASN.1 stub compiler for Java, a tool which generates Java stub classes for ASN.1 data type definitions. The paper illustrates the basic object model implemented by the generated Java code. The Java stub classes are based on an ASN.1 run-time system which provides the basic encoding/decoding functionality for different ASN.1 encoding rules. The structure and the properties of the generated Java classes are described in detail.

1 Introduction

ASN.1 specifications play an important role in security-related communication protocols. ASN.1 must also be integrated into Java, since Java is rapidly spreading as the new implementation language of choice for Internet applications such as electronic commerce.

ASN.1 compilers already exist for a number of target languages for which stubs can be generated. These stubs can then be used by applications implemented in the target language for decoding, accessing and encoding ASN.1 data. *Snacc for Java* also belongs to this family of tools with Java (JDK 1.1) [1] as the target language of code generation.

The Java code generated by *Snacc for Java* is based on an ASN.1 coding module.This run-time system provides Java classes for encoding and decoding ASN.1 data according to different encoding rules. It also contains classes that are used as Java representations for certain ASN.1 types. The scenario is depicted in figure 1.

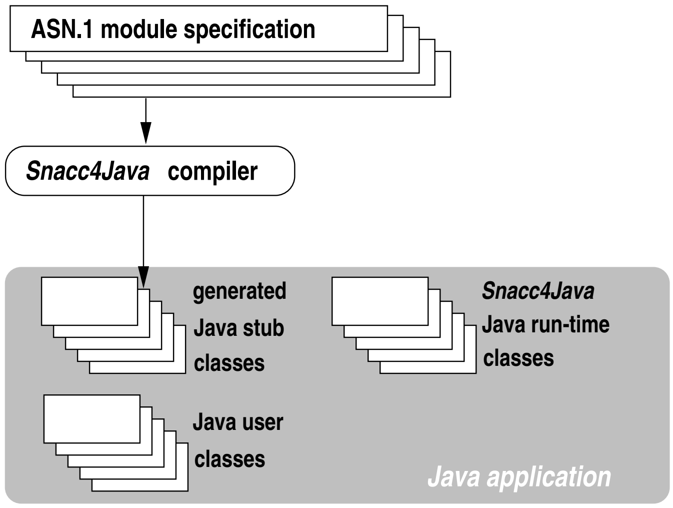

Figure 1: Mode of operation

The remainder of this documentation is structured as follows. Section two describes the object model implemented by the generated Java classes. The two interfaces provided by these Java classes are outlined in section three and four. Section five compares the two data type models supported by *Snacc for Java*. Section concludes the paper listing current limitations of *Snacc for Java* as well as directions of further work.

2 The stub object model

For each ASN.1 type definition that is not an alias definition a Java class with the same name will be generated. This class is called the *Java stub class*.

Alias definitions are those ASN.1 type definitions which only introduce another name for an existing type without specifying any additional (tagging) information. Since these definitions are redundant, the compiler will recursively resolve aliases to their corresponding base type definition. This also holds for type definitions imported from other modules: As long the imported type definition refers to an alias, the corresponding import clause will be suppressed as there is no corresponding Java stub class generated for the exporting module.

The generated Java classes act as mediators between the ASN.1 data and any Java application that wishes to access data encoded in ASN.1. All Java stub classes implement the same Java interface `com.ibm.asn1.ASN1Type` which defines two conceptual interfaces (access points) that are supported by each Java stub class:

- a programming interface to be used by the Java application to access ASN.1 data, and
- a decoding/encoding interface that can be used to read/write ASN.1 data from/to a data stream.

This is illustrated in figure 2.

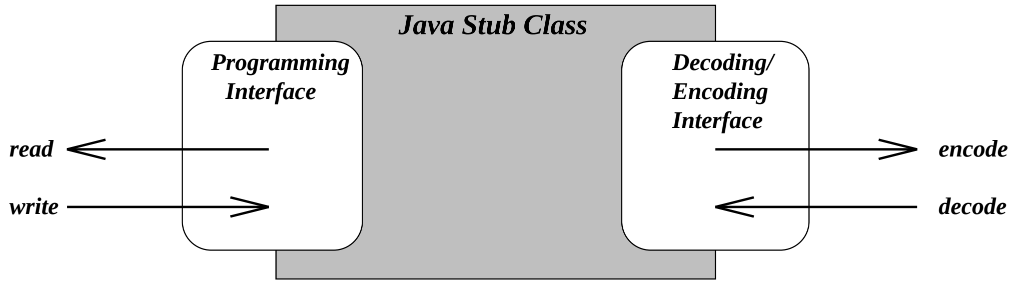

Figure 2: Interfaces provided by a Java stub class.

The decoding/encoding interface is independent of the particular ASN.1 type definition for which the Java stub class has been generated. The abstract base class `ASN1Type` defines the two methods `encode` and `decode` which resemble the encoding/decoding interface.

The programming interface of a Java stub class depends on the corresponding ASN.1 type definition. *Snacc for Java* distinguishes between six different sorts of ASN.1 type definitions:

1. simple types,
2. `SEQUENCE` types,
3. `SET` types,
4. `CHOICE` types,
5. `SET OF` types, and
6. `SEQUENCE OF` types.

The Java class generated provides access to the ASN.1 data using different schemes. These will be described in detail in section 4.

The generated Java stub classes belong to a package that has the same name as the ASN.1 module which contained the corresponding type definitions. If this module also contains value definitions (constants), an additional Java interface is generated that has the same name as the package. Instead of implementing `com.ibm.asn1.ASN1Type` directly, the Java stub classes implement the package interface which inherits from `com.ibm.asn1.ASN1Type` whereby the constant definitions are made available package-wide.

3 Use of the run-time system

The decoding/encoding interface of each Java class consists of two functions:

```
public void decode(com.ibm.asn1.ASN1Decoder dec);
public void encode(com.ibm.asn1.ASN1Encoder enc);
```

The class `com.ibm.asn1.ASN1Decoder` represents the abstract base class from which all decoder classes for a particular set of encoding rules are derived. Such concrete classes currently exist for DER [7], BER [3], [7] and IndefiniteBER. Since the BER and DER decoding methods do not differ, there are two decoder classes provided by the run-time system: `com.ibm.asn1.BERDecoder` and `com.ibm.asn1.BERIndefiniteDecoder`. The set of encoding rules according to which the decoding is performed depends on the concrete object which the user supplies to the decoding functions.

The same scheme is applied for encoding ASN.1 data. There are three subclasses of `com.ibm.asn1.ASN1Encoder`, one for each supported set of encoding rules:

- `com.ibm.asn1.DEREncoder`,
- `com.ibm.asn1.BEREncoder`, and
- `com.ibm.asn1.BERIndefiniteEncoder`.

The encoder and decoder objects are assigned with an output or input stream through which data are read and written. This association is established at construction time of these objects and has to be initiated by the application programmer.

Assuming that some data is BER-encoded according to some ASN.1 specification `Spec`, the following code sample illustrates an application skeleton[1] decoding the BER-encoded data given in a file and writing a DER-encoded output file:

```
public class MyApp {
  public static void main(String[] args) {
    // Instantiate stub classes generated by Snacc4Java,
    Spec spec = new Spec();
    // create a BERDecoder object reading from a file "in",
    BERDecoder dec = new BERDecoder(new FileInputStream("in"));
    // and create a DEREncoder object.
    DEREncoder enc = new DEREncoder();

    // Populate stub objects
    spec.decode(dec);
    // and do some work with the ASN.1 data in spec.

    // Encode ASN.1 data
    spec.encode(enc);
    // and write then to file "out".
    enc.writeTo(new FileOutputStream("out"));
  }
}
```

[1]Note that this code will not compile since there is no exception handling.

4 The Programming Interface

The Java classes created for each ASN.1 type definition are subclasses of `com.ibm.asn1.ASN1Type`. This abstract base class defines the properties common to all generated Java classes. Aside from the decoding and encoding methods these are:

- a default constructor,
- a copy constructor,
- a print method allowing variable indentation, and
- a default print method for debugging purposes.

4.1 Simple ASN.1 Types

Simple ASN.1 type definitions are only mapped if they add specific tagging information to their base type definitions. If so, a Java class is created which provides a public member variable `value` of the aliased type.

4.1.1 Example

For an ASN.1 type definition

```
SerialNumber ::= [APPLICATION 99] IMPLICIT INTEGER
```

contained in a module X509 the following Java class is generated.[2]

```
package X509;

public class SerialNumber implements X509 {

  public BigInteger value;

  /** default constructor */
  public SerialNumber() {}

  public SerialNumber(BigInteger arg) {
    value = arg;
  }

  /** copy constructor */
  public SerialNumber (SerialNumber arg) {
    value = arg.value;
  }

  /** encoding method. */
  public void encode (ASN1Encoder enc) throws ASN1Exception {
    enc.nextIsImplicit(enc.makeTag(enc.APPLICATION_TAG_CLASS,44));
```

[2]The Java files also contain import clauses for classes provided by JDK 1.1 and the ASN.1 run-time system. These clauses as well as all *javadoc*-related comments are not shown in all Java code presented in this paper.

```
    enc.encodeInteger(value);
  }

  /** decoding method. */
  public void decode (ASN1Decoder dec) throws ASN1Exception {
    dec.nextIsImplicit(dec.makeTag(dec.APPLICATION_TAG_CLASS,44));
    value = dec.decodeInteger();
  }

  /** print method (variable indentation) */
  public void print (PrintStream os, int indent) {
    os.print(value.toString());
  }

  /** default print method (fixed indentation) */
  public void print (PrintStream os) {
    print(os,0);
  }
}
```

4.2 SEQUENCE Types

The Java stub classes generated for ASN.1 SEQUENCE types contain public member variables representing the elements of the sequence. These variables have the same name as their corresponding ASN.1 element.

The initialization of these variables depends on their particular specification. Normal elements are initialized using the default constructor of their corresponding Java class. Optional variables are initialized with null[3]. The members representing ASN.1 data with default values are initialized with these defaults.

4.2.1 Example

The ASN.1 SEQUENCE type definition

```
Certificates ::= SEQUENCE {
    certificate        Certificate,
    certificationPath ForwardCertificationPath OPTIONAL
}
```

contained in a module X509 is mapped to the following Java stub class

```
package X509;

public class Certificates implements X509 {

  /** member variable representing the sequence member
      certificate of type Certificate */
  public Certificate certificate = new Certificate();
  /** member variable representing the sequence member
```

[3] For the ASN.1 types such as BOOLEAN and REAL (currently not supported) which are mapped to scalar Java types, an auxiliary member is introduced indicating the presence of the data.

```
      certificationPath of type ForwardCertificationPath */
  public ForwardCertificationPath certificationPath = null;

  /** default constructor */
  public Certificates() {}

  /** copy constructor */
  public Certificates (Certificates arg) {
    certificate = arg.certificate;
    certificationPath = arg.certificationPath;
  }

  /** encoding method. */
  public void encode (ASN1Encoder enc) throws ASN1Exception {
    int seq_nr = enc.encodeSequence();
    certificate.encode(enc);
    if (certificationPath != null) {
      certificationPath.encode(enc);
    }
    enc.endOf(seq_nr);
  }

  /** decoding method. */
  public void decode (ASN1Decoder dec) throws ASN1Exception {
    int seq_nr = dec.decodeSequence();
    certificate.decode(dec);
    if (!dec.nextIsOptional(dec.makeTag(dec.UNIVERSAL_TAG_CLASS,16))) {
      certificationPath = new ForwardCertificationPath();
      certificationPath.decode(dec);
    }
    dec.endOf(seq_nr);
  }

  /** print method (variable indentation) */
  public void print (PrintStream os, int indent) {
    os.println("{ -- SEQUENCE --");
    indent(os, indent+2);
    os.print("certificate = ");
    certificate.print(os, indent+2);

    os.println(',');
    if (certificationPath != null) {
      indent(os, indent+2);
      os.print("certificationPath = ");
      certificationPath.print(os, indent+2);
    }

    os.println();
    indent(os, indent);
    os.print('}');
  }
}
```

4.3 SET Types

The elements of an ASN.1 `SET` type can be encoded in an unordered manner under certain encoding rules. However, from the point of view of the programming interface ASN.1 `SET` types correspond to `SEQUENCE` types in which all elements are labeled optional. The only difference is that in sequences there must be at least one element which is not optional since the ASN.1 standard [2] does not allow empty sequences. Because of this similarity an example is omitted here.

4.4 CHOICE Types

Similar to `SEQUENCE` types, the Java stub classes `CHOICE` types also contain public member variables representing the CHOICE elements. In addition, they also contain an integer constant `<element_name>_CID` for each element, and a selector variable `choiceId` whose value is set to one of these constants indicating which of the CHOICE elements is present. The constants are generated with names that comply with the Java coding standards, that is upper-case only.

4.4.1 Example

The ASN.1 `CHOICE` type definition

```
GenName ::= CHOICE {
    printable PrintableString,
    teletex   TeletexString
}
```

contained in a module X509 is mapped to the following Java stub class:

```
package x509;

public class GenName implements X509 {

  static public final int PRINTABLE_CID = 0;
  static public final int TELETEX_CID   = 1;

  int tag_list[] = { PRINTABLE_CID, TELETEX_CID };

  public int choiceId;
  public String printable = null;
  public String teletex = null;

  /** default constructor */
  public GenName () {}

  /** copy constructor */
  public GenName (GenName arg) {
    choiceId = arg.choiceId;

    switch(choiceId) {
    case PRINTABLE_CID:
      printable = arg.printable;
```

```
    break;
  case TELETEX_CID:
    teletex = arg.teletex;
    break;
  }
}

/** encoding method. */
public void encode (ASN1Encoder enc) throws ASN1Exception {
  enc.encodeChoice(choiceId, tag_list);
  switch(choiceId) {
  case PRINTABLE_CID:
    enc.encodePrintableString(printable);
    break;
  case TELETEX_CID:
    enc.encodeTeletexString(teletex);
    break;
  }
}

/** decoding method. */
public void decode (ASN1Decoder dec) throws ASN1Exception {
  int tag = dec.decodeChoice(tag_list);
  if (tag == dec.makeTag(dec.UNIVERSAL_TAG_CLASS,19)) {
    printable = new String();
    printable = dec.decodePrintableString();
    choiceId = PRINTABLE_CID;
  }
  if (tag == dec.makeTag(dec.UNIVERSAL_TAG_CLASS,20)) {
    teletex = new String();
    teletex = dec.decodeTeletexString();
    choiceId = TELETEX_CID;
  }
}

/** print method (variable indentation) */
public void print (PrintStream os, int indent) {
  os.println("{ -- CHOICE --");
  switch(choiceId) {
  case PRINTABLE_CID:
    for(int ii = 0; ii < indent+2; ii++) os.print(' ');
    os.print("printable = ");
    os.print(printable.toString());
    break;
  case TELETEX_CID:
    for(int ii = 0; ii < indent+2; ii++) os.print(' ');
    os.print("teletex = ");
    os.print(teletex.toString());
    break;
  }
  for(int ii = 0; ii < indent; ii++) os.print(' ');
  os.print("}");
```

```
  }
}
```

4.5 SET OF Types

SET OF types in ASN.1 represent sets in the mathematical sense. Their Java stub classes both implement the common stub interface *and* inherit from the java.util.Hashtable class provided by JDK [1]. The encoding is based on the hash table's enumeration facility, which does not preserve the order in which elements are entered. Because of the inheritance relationship, Java stub classes for ASN.1 SET OF types provide the same interface as Hashtable objects. The encode and decode method are still mandatory as the ASN1Type interface is inherited by X509.

4.5.1 Example

The ASN.1 SET OF type definition

```
CrossCertificates ::= SET OF Certificate
```

contained in a module X509 is mapped to the following Java stub class:

```
package x509;

public class CrossCertificates extends Hashtable implements X509 {

  /** default constructor */
  public CrossCertificates() {}
  /** copy constructor */
  public CrossCertificates (CrossCertificates arg) {
    for (Enumeration e = arg.elements(); e.hasMoreElements();) {
      Certificate tmp = (Certificate )(e.nextElement());
      put(tmp,tmp);
    }
  }

  /** encoding method. */
  public void encode (ASN1Encoder enc) throws ASN1Exception {
    int set_of_nr = enc.encodeSetOf();
    for (Enumeration e = elements(); e.hasMoreElements();) {
      ((Certificate)(e.nextElement())).encode(enc);
    }
    enc.endOf(set_of_nr);
  }

  /** decoding method. */
  public void decode (ASN1Decoder dec) throws ASN1Exception {
    int set_of_nr = dec.decodeSetOf();
    while (!dec.endOf(set_of_nr)) {
      Certificate tmp = new Certificate();
      tmp.decode(dec);
      put(tmp,tmp);
```

```
    }
  }

  /** print method (variable indentation) */
  public void print (PrintStream os, int indent) {
    boolean nonePrinted = true;
    os.println("{ -- SET OF --");
    for (Enumeration e = elements(); e.hasMoreElements();) {
      if (nonePrinted == false)
        os.println(',');
      nonePrinted = false;
      for(int ii = 0; ii < indent+2; ii++) os.print(' ');
      ((Certificate)(e.nextElement())).print(os, indent+2);
      if (!e.hasMoreElements())
        os.println();
    }
      for(int ii = 0; ii < indent; ii++) os.print(' ');
      os.print('}');
  }
}
```

4.6 SEQUENCE OF Types

ASN.1 SEQUENCE OF types correspond to vectors since the order of their elements is preserved. Therefore, similar to the stub classes for SET OF types, the SEQUENCE OF Java stub classes inherit from the `java.util.Vector` class. An example is omitted.

Note that for ASN.1 BOOLEAN and REAL types the "objectified" representations of their corresponding Java types are stored in the vector/hashtable, that is `Boolean` and `Double` respectively.

5 ASN.1 Data Type Models

Snacc for Java supports two different data type models

- the *basic type model*, and
- the *advanced type model*.

An ASN.1 module definition can be considered as a set of type hierarchies (trees). The leaf nodes of these trees represent the basic ASN.1 types as specified in [2]. If the basic type model is applied, only these leaf nodes are mapped directly to Java types[4]. The mapping rules used are shown in table 1.

The basic model has the disadvantage that the useful ASN.1 data type such as `UTCTime` or `PrintableString` are mapped to ASN.1 OCTET STRING types and are therefore inconvenient to use in an application. If the advanced model (default) is used, the ASN.1 useful types are also considered as leaf nodes and are mapped directly to Java types as shown in table 2.

[4] The ASN.1 specification of the useful types must be compiled using the basic model.

Basic ASN.1 Type	Java Type
BOOLEAN	boolean (Boolean for SET OF/SEQ. OF
ENUMERATED	*Java stub class with an int member as for simple types*
REAL	double (Double for SET OF/SEQ. OF)
OCTET STRING	byte[ ]
INTEGER	java.math.BigInteger
BIT STRING	com.ibm.util.BitString
OBJECT IDENTIFIER	com.ibm.asn1.ASN1OID
NULL	com.ibm.asn1.ASN1Type
ANY	com.ibm.asn1.ASN1Any

Table 1: Type conversion scheme for ASN.1 base types

Useful ASN.1 Type	Java Type
UCTTime	java.util.Calendar
GeneralizedTime	java.util.Calendar
ObjectDescriptor	java.lang.String
NumericString	java.lang.String
PrintableString	java.lang.String
TeletexString	java.lang.String
T61String	java.lang.String
VideotexString	java.lang.String
IA5String	java.lang.String
GraphicString	java.lang.String
VisibleString	java.lang.String
ISO646String	java.lang.String
GeneralString	java.lang.String

Table 2: Advanced type conversion scheme for ASN.1 useful types

Snacc for Java supports compiler directives which indicate that the type tree is to be pruned at a certain position. In this case the ASN.1 type at this position would be mapped to an already existing Java class type. This Java class would have to be written by the application programmer. The pragma scheme provides greater flexibility than the advanced data type model.

Compiler directives can also be used to modify the compiler's behavior when generating constructors, encode and decode methods for Java stub classes. The name of the Java package the generated code will be part of can also be specified using pragmas.

6 Generated Files

Snacc for Java generates a directory tree in which the generated Java files and the makefiles will be located. The root directory of this tree is called `TARGET_DIR`. The target directory can be specified by the command line option `-d`. If omitted the target directory will be set to the current working directory. variable.

The target directory contains three subdirectories `src`, `classes`, and `docs`. For n ASN.1 modules specified on the command line *Snacc for Java* generates $n+1$ makefiles. The top-level makefile is named `Makefile` and is located in the target directory. This makefile invokes the package makefiles (named `<package name>.mk`. The Java files and the makefile for each package will be located in a directory `TARGET_DIR/src/<package qualifier>/<package name>`. The tree structure is depicted in figure 3.

```
TARGET_DIR
 |
 |-- src
 |    |--- <package qualifier>
 |    |                  |-- <package name>
 |    |___ ...           |__ ...
 |-- classes
 |    |--- <package qualifier>
 |    |                  |-- <package name>
 |    |___ ...           |__ ...
 |__ docs
      |--- <package qualifier>.<package name>
      |___ ...
```

Figure 3: Generated directory tree.

The package qualifier can be specified either through the `-P` command line option or using the `packagePrefix` compiler directive (using the Java dot notation in both cases. The package name is the name is derived from the ASN.1 module name by converting all upper-case characters to lower case.

The makefile provides three goals `all` (default goal), `docs` and `clean` which can be used to build the entire application, only the class documentation (using `javadoc`) or the delete all generated files.

7 Conclusions

Snacc4Java is a flexible ASN.1 compiler that generates Java stub code. The stub code provides one Java class per ASN.1 type definition. It supports two different data type models which can be further modified using compiler directives. The generated code is based on an run-time system for encoding and decoding ASN.1 data which was written entirely in Java. The run-time system currently allows to encode and decode three different ASN.1 coding standards (DER, BER, IndefiniteBER).

The compiler part currently handles ASN.1 specifications according to the 1988 ASN.1 standard [2]. Syntactical add-ons such as those defined by [4] or special macros as defined in X.683 [6] cannot be handled. Other limitations are that default values of types other than `OBJECT IDENTIFIER`, `INTEGER`, and `BOOLEAN` are currently not supported. The ASN.1 `REAL` type is not implemented by the run-time system since there has been no need to do so yet. These limitations, as well as supporting additional encoding rules such as CER [7] or PER [8], also represent the directions of further work.

Snacc4Java is one of the first ASN.1 compilers that produce Java target code. Although the compiler part itself was not written in Java it was ported to all major UNIX platform such as AIX, LINUX, and Solaris, as well as to Win32. The code size of the compiler part is around 19.5 kloc, the size of the run time part around 8.7 kloc. The tool was first used and tested extensively in a number of internal security-related projects. In the meantime it was licensed to some interested companies and universities. *Snacc4Java* is freely available on IBM's *Alphaworks* page `www.alphaworks.ibm.com`.

References

[1] Gossling J. and Arnold K.: The Java Programming Language. Addison-Wesley Publishing Company, Reading MA, 1996.

[2] International Telecommunication Union: Open Systems Interconnection – Model and Notation – Specification of Abstract Syntax Notation One (ASN.1). ITU-T Recommendation X.208, 1988.

[3] International Telecommunication Union: Open Systems Interconnection – Model and Notation – Specification of basic encoding rules for Abstract Syntax Notation One (ASN.1). ITU-T Recommendation X.209, 1993.

[4] International Telecommunication Union: Open Systems Interconnection – Model and Notation – Specification of Abstract Syntax Notation One (ASN.1). ITU-T Recommendation X.208, 1993.

[5] International Telecommunication Union: Information Technology – Abstract Syntax Notation One (ASN.1) – Specification of basic notation. ITU-T Recommendation X.680, 1994.

[6] International Telecommunication Union: Information Technology – Abstract Syntax Notation One (ASN.1) – Parameterization of ASN.1 specifications. ITU-T Recommendation X.683, 1994.

[7] International Telecommunication Union: Information Technology – ASN.1 encoding rules – Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), andDistinguished Encoding Rules (DER) . ITU-T Recommendation X.690, 1994.

[8] International Telecommunication Union: Information Technology – ASN.1 encoding rules – Specification od Packed Encoding Rules (PER). ITU-T Recommendation X.691, 1995.