Automatic Generation of Java Stubs for ASN.1

Andreas Schade
IBM Research Division, Zurich Research Laboratory
Säumerstrasse 4, 8803 Rüschlikon, Switzerland
Tel.: +41 1 724 84 05, Fax.: +41 1 710 89 53
email: san@zurich.ibm.com

Abstract

This paper documents *Snacc for Java*, as 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.

This paper documents a tool, *Snacc for Java*, which is a Java stub compiler for ASN.1 specifications. 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) [Jav95] as the target language of code generation.

The Java code generated by *Snacc for Java* is based on the ASN.1 Coding Module [Eir96a]. This run-time system provides 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 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 six explains the programming environment that is provided by *Snacc for Java*.

2 The Stub Object Model

For each type definition that is not an alias definition found in an ASN.1 module one Java class with the same name will be generated which 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, references to them will be recursively resolved to the base type definition. This is also true 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 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 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 1.

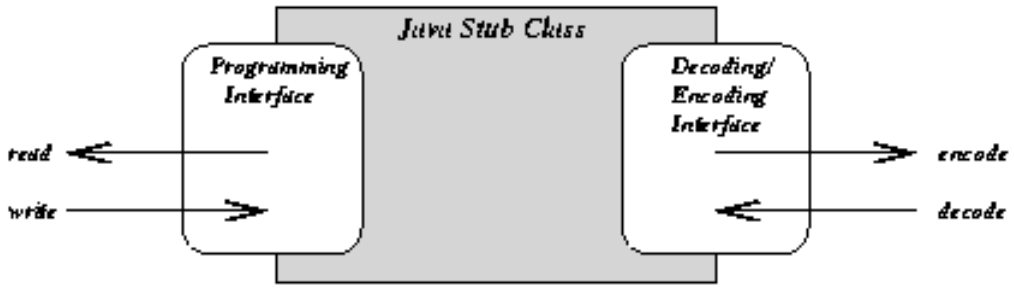

Figure 1: Interfaces provided by a Java stub class.

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 defines the two methods `encode` and `decode` which resemble the encoding/decoding interface.

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

- simple types,
- `SEQUENCE` types,
- `SET` types,
- `CHOICE` types,
- `SET OF` types, and
- `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 available package-wide.

3 The Decoding/Encoding Interface

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` is the abstract base class from which all decoder classes for a particular set of encoding rules are derived. Decoding is currently implemented for DER, BER [ITU88] and Indefinite BER. The class `com.ibm.asn1.BERDecoder` can be used to decode all data that was encoded using any of these encoding rules.

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 ASN.1 encoding rules: `com.ibm.asn1.DEREncoder`, `com.ibm.asn1.BEREncoder`, and `com.ibm.asn1.BERIndefiniteEncoder`. The set of encoding rules according to which the encoding is performed depends on the concrete object which the user supplies to the encoding functions.

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.

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] INTEGER
```

contained in a module X509 the following Java class is generated [1].

```
package X509;

public class SerialNumber implements X509 {

  public BigInteger value;

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

  public SerialNumber(BigInteger arg) {
```

[1]The Java files also contain import clauses for classes provided by JDK 1.1 and the ASN.1 run-time system. These clauses are omitted in all Java code shown in this paper.

```
  value = arg;
}

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

/** encoding method.
 * @param enc
 *        encoder object derived from com.ibm.asn1.ASN1Encoder
 * @exception com.ibm.asn1.ASN1Exception
 *            encoding error
 */
public void encode (ASN1Encoder enc) throws ASN1Exception {
  enc.nextIsImplicit(enc.makeTag(enc.APPLICATION_TAG_CLASS,44));
  enc.encodeInteger(value);
}

/** decoding method.
 * @param dec
 *        decoder object derived from com.ibm.asn1.ASN1Decoder
 * @exception com.ibm.asn1.ASN1Exception
 *            decoding error
 */
public void decode (ASN1Decoder dec) throws ASN1Exception {
  dec.nextIsImplicit(dec.makeTag(dec.APPLICATION_TAG_CLASS,44));
  value = dec.decodeInteger();
}

/** print method (variable indentation)
 * @param os
 *        PrintStream representing the print destination (file, etc)
 * @param indent
 *        number of blanks that precede each output line.
 */
public void print (PrintStream os, int indent) {
  os.print(value.toString());
}

/** default print method (variable indentation)
 * @param os
 *        PrintStream representing the print destination (file, etc)
 */
public void print (PrintStream os) {
```

```
    print(os,0);
  }
}
```

4.2 SEQUENCE Types

The Java stub classes generated for 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[2]. 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
      certificationPath of type ForwardCertificationPath */
  public ForwardCertificationPath certificationPath = null;

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

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

[2]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 = arg.certificationPath;
}

/** encoding method.
 * @param enc
 *        encoder object derived from com.ibm.asn1.ASN1Encoder
 * @exception com.ibm.asn1.ASN1Exception
 *            encoding error
 */
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.
 * @param dec
 *        decoder object derived from com.ibm.asn1.ASN1Decoder
 * @exception com.ibm.asn1.ASN1Exception
 *            decoding error
 */
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)
 * @param os
 *        PrintStream representing the print destination (file, etc)
 * @param indent
 *        number of blanks that precede each output line.
 */
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('}');
  }

  /** default print method (variable indentation)
   * @param os
   *        PrintStream representing the print destination (file, etc)
   */
  public void print (PrintStream os) {
    print(os,0);
  }
}
```

4.3 SET Types

The elements of an ASN.1 `SET` type can be unordered 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 [ITU93] 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 a integer constants `<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. These 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.
    * @param enc
    *        encoder object derived from com.ibm.asn1.ASN1Encoder
    * @exception com.ibm.asn1.ASN1Exception
    *            encoding error
    */
  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.
  * @param dec
  *        decoder object derived from com.ibm.asn1.ASN1Decoder
  * @exception com.ibm.asn1.ASN1Exception
  *            decoding error
  */
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)
  * @param os
  *        PrintStream representing the print destination (file, etc)
  * @param indent
  *        number of blanks that preceed each output line.
  */
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("}");
  }

  /** default print method (variable indentation)
    * @param os
    *        PrintStream representing the print destination (file, etc)
    */
  public void print (PrintStream os) {
    print(os,0);
  }

}
```

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 [Jav95]. 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.

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.
  * @param enc
  *        encoder object derived from com.ibm.asn1.ASN1Encoder
  * @exception com.ibm.asn1.ASN1Exception
  *            encoding error
  */
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.
  * @param dec
  *        decoder object derived from com.ibm.asn1.ASN1Decoder
  * @exception com.ibm.asn1.ASN1Exception
  *            decoding error
  */
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)
  * @param os
  *        PrintStream representing the print destination (file, etc)
  * @param indent
  *        number of blanks that preceed each output line.
  */
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('}');
  }

  /** default print method (variable indentation)
    * @param os
    *        PrintStream representing the print destination (file, etc)
    */
  public void print (PrintStream os) {
    print(os,0);
  }

}
```

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, that is `Boolean` and `Double` respectively.

5 ASN.1 Data Type Models

Snacc for Java supports two different data type models

- the *basic model*, and
- the *advanced 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 [ITU93]. If the basic model is applied, only these leaf nodes are mapped directly to Java types[3]. The mapping rules used are shown in table 1.

Although there is a type mapping provided for ASN.1 REAL types, the runtime system does not provide the necessary coding routine currenlty. The generated code will therefore not compile.

[3]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

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. If the advanced model (default) is used, these useful types are also considered as leaf nodes and are mapped directly to Java types as shown in table 2.

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

6 Compiler Directives

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 scheme therefore provides greater flexibility than the advanced data type model.

6.1 Syntax

Snacc for Java compiler directives have the form of a special comment

```
--pragma {attribute:"value"}+
```

The attribute value pairs can either be listed in one pragma comment or specified by list of consecutive comments.

6.2 Position in an ASN.1 Specification

Compiler directives can be placed at three locations in the ASN.1 source specification. Compiler directives specified after the `:=` of an ASN.1 type definition affect all occurrences of this type in the specification. In this case the ASN.1 type is *always* mapped to the Java type specified.

Compiler directives that are to affect a particular element must be placed after the comma that follows the element. In case of the lase element, where there is no comma, the comment must immediately follow the element specification.

Additionally, *Snacc for Java* recognizes module-wide compiler directives. These compiler directives must be placed after the `:=` of an ASN.1 module definition.

6.3 Semantics

The only module-wide compiler directive is currently indicated by the attribute `packagePrefix` whose value represents the package prefix (in Java dot notation) for the Java package to be generated. If present, this compiler directive overrides the value of the `-P` command line option (see 8) for the particular module.

Snacc for Java compiler directives for type definitions must contain the following attributes:

- `javaTypeName` – this is the name of the existing Java type to which the ASN.1 type is to be mapped.
- `constructor` – The value given is treated as a mask which is used to generate constructor calls for the Java class specified by `javaTypeName`. The value should represent a valid Java statement and must therefore start with `%s =` followed by an initializer expression. If the constructor of the Java class requires a decoder object as an argument, it must be called at a certain position relative to the input stream. In this case specify `%s = null;\\n` for the constructor mask (deferred construction) and use the `decodeRoutine` compiler directive instead.
- `encodeRoutine` – the mask for an encoding routine. It can contain one `%s` which is the placeholder for the name of the Java stub class member to be encoded. The name of the encoder object in an encode routine of a Java stub class is `enc`.

- `decodeRoutine` – the mask for the decode function. It can contain one `%s` which is the placeholder for the name of the Java stub class member that is to be assigned to the decoded value. The name of the decoder object is `dec`.
- `printRoutine` – the mask for the print function. The name of the print stream is `os`. The name of the Java stub class member to be printed is denoted by `%s`.

Whereas the specification of the `javaTypeName` attribute is mandatory, the function mask attributes can be omitted. In this case, their values are derived from the Java type name given using the rule

```
constructor:  "%s = null;\n"
encodeRoutine:"%s.encode(enc);\n"
decodeRoutine:"%s = new <javaTypeName>(dec)\n"
printRoutine: "os.print(%s.toString());\n"
```

In general, the Java class types used to represent ASN.1 types can be divided in two categories. The first category comprises those Java classes that are subtyped from `com.ibm.asn1.ASN1Type`. These classes provide own encoding and decoding routines as well as a print routine.

```
void encode (com.ibm.asn1.ASN1Encoder enc);
void decode (com.ibm.asn1.ASN1Decoder dec);
void print  (java.io.PrintStream os);
```

If the decoding routine does not exist, there must be a constructor which directly populates the class object by decoding. The optional compiler directive attributes default to this case.

The second category consists of the Java utility classes for ASN.1. For these class types the abstract classes `com.ibm.asn1.ASN1Decoder` and `com.ibm.asn1.ASN1Encoder` provide routines

```
<javaTypeName> decode<javaTypeName> ();
void           encode<javaTypeName> (<javaTypeName> arg);
```

Java utility classes usually provide a method `toString`.

6.4 Example

1. *Modification of all occurrence of a certain ASN.1 type.* In the following example

```
Validity ::= --pragma javaTypeName='Validity' -- SEQUENCE {
    notBefore UTCTime,
    notAfter  UTCTime
}
```

Snacc for Java would assume that there is a Java class `Validity` which provides a `toString` method. The `Validity` class must be supported by the decoder and encoder classes which provide methods `decodeValidity()` and `encodeValidity(Validity arg)` respectively.

2. *Modification of a single element type.* If the above example is slightly modified:

```
Validity ::= SEQUENCE {
    notBefore UTCTime, --pragma javaTypeName='MyStartTime'
                       --pragma printRoutine='"$%$s".print()'
    notAfter  UTCTime
}
```

only the `notBefore` element of the `Validity` type would be affected. This element would be mapped to a Java type `MyStartTime` which also provides its own print method.

The mapping of the `notAfter` element would remain unchanged. Depending on the data type model in use, it would be mapped either to `java.util.Calendar` or to `byte[]`.

7 Installation

The *Snacc for Java* installation contains the following directory structure. Assuming that the tool has been downloaded and untarred to a directory <S4J>, the *Snacc for Java* directory structure looks as follows:

- <S4J>/bin contains the executable snacc4j.
- <S4J>/asn1 contains the ASN.1 useful type specification.
- <S4J>/classes contains the class files of ASN.1 run-time system and the ASN.1 useful types. This directory must be included in the CLASSPATH when compiling code which has been generated by *Snacc for Java*.
- <S4J>/classdoc contains the documentation of the ASN.1 run-time system.
- <S4J>/doc contains this documentation in postscript and PDF-format as well as a subdirectory <S4J>/doc/SnaccForJava with the html version.

8 Usage

Snacc for Java can be invoked with the following options

```
snacc4j
[-h] [-D]
[-d] [-classpath] [-P]
[-mak] [-val]
[-B] ([-u <useful types ASN.1 file>] | -nu)
[-mm] [-mf <max file name length>]
<ASN.1 file list>
```

- `-h` prints a help message
- `-D` dumps the tree of the parsed ASN.1 modules to stdout (helpful for debugging)
- `-d` <dir> target directory, denotes the location of the Java source and class file tree to be generated
- `-classpath` additional classpath string to be used when compiling the Java files. If not present the CLASSPATH environment variable is used instead.
- `-P` <pkg> package prefix to be used for all given ASN.1 modules. If -p a.b is given then the Java classes representing the ASN.1 type definitions will be put in package a.b.¡pkg¿, where ¡pkg¿ is the (adjusted) name of the ASN.1 module
- `-mak` do not generate Makefile
- `-val` generate value definitions (limited) note: if none of -mak or -val are given, all are generated.
- `-B` generate Java classes that use the basic data type model (The useful types must be known, but their corresponding Java classes are not used in the generated code.)
- `-u <file>` ¡filename¿ specifies the ASN.1 file with definition of the useful types (i.e. PrintableString). See the asn-useful.asn1 file (in the Snacc for Java/asn1 directory). This file can also be specified via the environment variable `ASN_USEFULS`.
- `-nu` do not use any useful types
- `-mm` mangle output file name into module name (by default, the output file inherits the input file's name, with only the suffix replaced)
- `-mf` <num> num is maximum file name length for the generated source files

9 Programming Support

- The generated Java stub classes offer a method

  ```
  void print(java.io.PrintStream os);
  ```

  for debugging purposes. This method prints the ASN.1 data indented according to their hierarchical structure to the destination associated with the stream parameter.

- The generated Java stub classes contain `javadoc`-compatible comments.

- *Snacc for Java* generates a makefile which allows to compile the generated Java stub classes. The makefile can also be used to generate .html documentation files about the Java stub classes.

9.1 Directory tree

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 ommited 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 commad 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 package qualifier can be specified either through the `-P` command line option or using the `packagePrefix` compiler directive as explained in section 6. The package name is the name is derived from the ASN.1 module name by converting all upper-case characters to lower case.

The package qualifier can the specified using the `-P` command line option in form of a Java package name (dot notation). The tree structure is depicted in figure 2.

9.2 Using the makefile

The makefile provides three goals `all` (default goal), `docs` and `clean`.

- Running make all or simply `make` will invoke the Java compiler on the Java source files. The compiled class files will be located in `TARGET_DIR/classes/`<`package qualifier`>`/`<`package name`>.

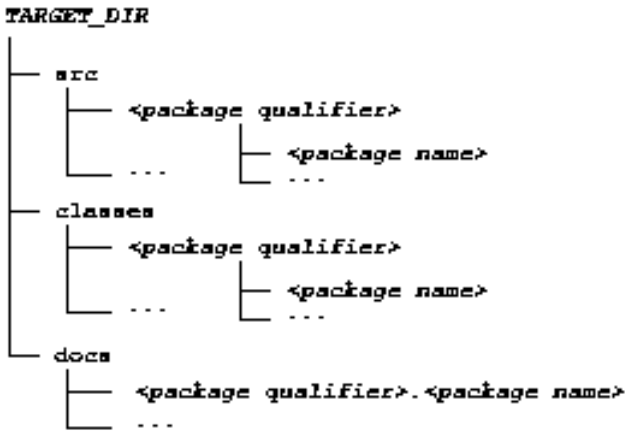

Figure 2: The standard directory tree.

- Running `make docs` will invoke `javadoc` to generate the HTML-documentation for the Java files. The documentation files can be found in `TARGET_DIR/docs/<package qualifier>.<package name>`.
- Running `make clean` will remove all the generated source, class and documentation files

10 Limitations

- The ASN.1 `REAL` type is currently not supported.
- *Snacc for Java* can only parse ASN.1 specifications that adhere to the X.208 standard [ITU93]. Syntactical add-ons such as special macros are not supported.
- Default values other than `OBJECT IDENTIFIER`, `INTEGER`, and `BOOLEAN` are currently not supported.

References

[Eir96a] Thomas Eirich: ASN.1 Coding Module.
`http://assam.zurich.ibm.com/JavaCafe/docs/com.ibm.asn1-index.html`

[ITU88] International Telecommunication Union: Open Systems Interconnection – Model and Notation – Specification of basic encoding rules for Abstract Syntax Notation One (ASN.1). ITU-T, 1993.

[ITU93] International Telecommunication Union: Open Systems Interconnection – Model and Notation – Specification of Abstract Syntax Notation One (ASN.1). ITU-T, 1993.

[Jav95] JavaSoft, Inc.: The Java Language Specification. JavaSoft Inc., 1995. `http://java.sun.com/doc/language_specification.html`