20

TDL: an ASN.1-based tests description language

Paul KABORE, André SCHAFF
CRIN & INRIA Lorraine, Bâtiment Loria, B.P. 239,
54506 Vandoeuvre-les-Nancy Cedex France, Tel: 83.59.20.48,
Fax: 83.41.30.79, e-mail: {kabore,schaff} @loria.fr

Abstract

We present in this paper a language to describe tests for ASN.1 data structures. The main problem in test data generation is to derive from abstract data structures consistently inputs which are effective to find faults in the system under test. The cause is twofold. First, semantic information is not included with the ASN.1 declarations. Second, since the test input space is generally infinite, we must relay on external statements of the properties of the tests we want to use. The intention of this paper is to formalize semantic informations description in ASN.1 and to define a formalism which allow the description of test data. A test description language is then proposed to bridge the large gap between ASN.1 specifications and actual test inputs.

Keywords

ASN.1, tests description, test data generation.

1 INTRODUCTION

ASN.1 [ISO-8824 90, ISO-8824.1 92] (Abstract Syntax Notation One) has been standardized to define the data structures of Protocol Data Units. Techniques have been developed that generate test sequences for the behaviour part of communicating protocols. These test sequences attempt to reveal the presence of specific faults during testing [Bochmann 91]. One of the most difficult parts of testing is the actual generation of test data, which has traditionally been done by hand. The work described in this paper is a first attempt to automatically generate test data following the fault based testing techniques. Basically test data are created by setting parameters in the PDU structure defined in ASN.1. As the complexity of protocols increases, we would like to create these test data automatically or semi-automatically using a test data generator tool. However, parameters included in the PDUs have special meanings and informations about these parameters are described in natural language which is not suitable for computer processing. Thus the problem is how to generate adequate test sets of finite size with respect to a system under test and effective in identifying faults, given the following problems:

1. since formal description of parameters setting conditions is missing, how to automatically generate test data which fulfill these conditions;
2. the input space is generally too large and often infinite, so which range of data must be generated to allow a decision on the reliability of a system under test in a reasonable limit of time.

B. Baumgarten et al. (eds.), *Testing of Communicating Systems*
© Springer Science+Business Media Dordrecht 1996

In most testing environments, problem 1 is solved by getting the test executor to set the parameters interactively. Problem 2 is caused by the nondeterminism in the choice of ASN.1 values similar to nondeterminism in specification languages: the possible choices of a value which obey some ASN.1 type may be too large and it would be very hard to decide which instances can be selected. Some methods have been proposed in software engineering to generate test data. These methods include random tests data generation ([Myers 79],[Duran 84]), partition testing [Ammann 93] ... Random testing should be adequate when a fault-prone region is uniformly distributed across the complete input space. Performance of partition-testing depends on the partition criteria. We propose a formalism which allows test designers to describe test data and then their expertise to an automatic test data generator. Hence, we present a formalism to specify constraints on ASN.1 data structures and a Test Description Language, TDL, which uses this formalism to specify test data. A test description consists in partitioning the input space using constraints.

The remainder of the paper is organised as follows: in the next section we describe the constraints specification formalism. Then we use this formalism to define in section 3 the Tests Description Language. Finally a conclusion is provided in section 4.

2 CONSTRAINTS

This section deals with an extension to the ASN.1 description language with constraints. The objective of this extension is to formalize the informal description of the PDUs parameter settings conditions. Some similar works have been made in [Varvitsiotis 93] and [Van de Burgt 89]. Our constraints definition formalism can be viewed as an extension of these works and we follow the constraints definition syntax of ASN.1 92 [ISO-8824.3 92]. In particular, we introduce operators into the ASN.1 language. These operators have been introduced for the following reasons:

1. specification languages generally use operators to describe data types. When we would like to use ASN.1 as a specification language, it would be necessary to use operators over ASN.1 types (e.g in GDMO [ISO-10165.4 92, Bapat 93]);
2. to describe tests for a given ASN.1 data structure, it is useful to be able to specify some test data by combining specific components of this structure using operators.

ASN.1 92 allows sub-type constraints and general constraints definition [ISO-8824.3 92]. However, the sub-type constraints definition formalism is not general enough (e.g we cannot specify constraints which combine several ASN.1 structure components) and general constraints are still specified informally. Our aim is to formally specify the dependencies between components of an ASN.1 data structure. The constraints description formalism is used to specify dependencies describing either conformance of PDUs instances to their ASN.1 abstract structure or test data.

Definition 21 *Given an ASN.1 type T, the domain of T $\mathcal{D}(T)$, is the set of all ASN.1 values that obey T.*

We specify constraints by defining elements of $\mathcal{D}(T)$, i.e. sets of ASN.1 values. Sets are defined symbolically using predicates, one or more predicates per set. A predicate is a function over $\mathcal{D}(T)$ that returns TRUE or FALSE and denotes the set of values that satisfy the predicate (i.e. values for which the predicate returns TRUE). So, operators over constraints are the usual operators over sets such as the UNION, the INTERSECTION and the COMPLEMENT operators. Relational operators ($=, <, <=, >, >=, <>$) define the set {TRUE} or {FALSE} and are used over predicates. A constraint specification is an expression with enumerated sets or predicates as terms. Predicates are constructed on attributes as in [Varvitsiotis 93].

2.1 Attributes, predicates and actions

Given an ASN.1 value V of type T, V carries a set of attributes on which we can apply predicates and actions. Attributes carried by V can either satisfy the predicates or not. For a given attribute, we define a set of predefined predicates and actions shown in figure 1. Furthermore, complex predicates and actions can be constructed combining existing predicates (respectively actions) using logical operators (respectively mathematical operators). On figure 1, `X` denotes a value of some ASN.1 type, `encoded(X)` is an action that computes the encoded value of `X` according to some encoding rules (e.g BER [ISO-8825 87]). `#X` is the length of `X` if `X` is a STRING, SEQUENCE OF or SET OF value. `odd()` (respectively `even()`) is a predicate which returns TRUE if its argument is odd (respectively even). `pres(X)` and `abs(X)` indicates if `X` is present or absent for optional types. `X[i]` is the element at position `i` of `X`, for STRING or SEQUENCE OF values. The `sorted()` predicate supposes the existence of a total order over the domain of the ASN.1 type: lexicographic order over characters strings and natural order over integers and reals. `alphabet()` is an action that returns the set of characters or elements of its argument, if this argument is of STRING, SEQUENCE OF or SET OF type. The remainder of this section deals with complex predicates and actions definition. The section also presents the operators used to combine defined predicates and actions into complex ones. For convenience, predicates and actions are defined on types, which means that they have to be applied to any instance of that types.

Attributes	Predicates	Actions
Value	odd(x)[2], even(x)[2], uppercase(x)[3], lowercase(x)[3] sorted(x)[3] [5]	x [1] encoded(x)[1] x[i][3] [5]
Length		#x[3] [4] [5]
Alphabet		alphabet(x)[3] [4] [5]
Option	pres(x)[6], abs(x)[6]	

(1) Applicable to all types
(2) Applicable to Integer type
(3) Applicable to string types
(4) Applicable to Set Of type
(5) Applicable to Sequence Of type
(6) Applicable to optional types

Figure 1 Attributes for ASN.1 values

We will use the pointing notation defined in [ISO-8824.1 92] to refer to a specific element of an ASN.1 structure:

- the notation `@`*component* refers to the element *component* of the outermost structure of an ASN.1 definition;
- the notation `@.`*component* refers to the element *component* of the innermost structure of an ASN.1 definition.

This notation has been extended as follows:

- the notation `@` refers to the current element;
- the notation `@`$component_1.component_2...component_n$ refers to the $component_n$ assuming that $component_1, \ldots, component_{n-1}$ are structured components.

For example, consider the following ASN.1 definition:

```
T ::= SET {b REAL,                          -- 1
           c SEQUENCE { b BOOLEAN,          -- 2
                        e INTEGER           -- 3
                      }
          }
```

If we write @ or @b on line 1, that points out the component b of this line. On the other hand, to refer to the component b of line 2, we can use the notations @, @.b, or @c.b, written on this line. Writing @b on line 2 or 3 refers to the component b of line 1. The notations @.b and @c.b used on line 3 points out the component b of line 2.

2.2 Constraints specification

The ASN.1 sub-typing mechanism allows the refinement of a parent type. For example, a sub-type can be defined for an OCTET STRING type by either constraining its alphabet or its size:

```
MyType ::= OCTET STRING (FROM ('1'|'2'|'3'))(SIZE(1..50))
```

Constraints are defined using intervals or enumerated sets of literal (integer values or characters). It is possible to combine set and interval expressions in a same constraint definition like this:

```
MyType ::= INTEGER ((5|10|15)|(20..100))
```

However, we cannot define set intersections, set exclusions and constraints using symbolic or computed values. For components of an ASN.1 structure, we would like to define dependencies between these components using suitable operators. For this purpose we have introduced operators in the ASN.1 language:

```
T ::= SET { a BIT STRING,
            b INTEGER (#@.a+10),
            c REAL (10..(@.b+#@.a)/2)
          }
```

This specification means that for an instance of type T, the component b must have a value equal to the length of component a plus 10. And the value of component c must be between 10 and the value computed from the formula (@.b+#@.a)/2. Dependencies between numeric attributes (integer values, string length, etc.) can be expressed using the usual mathematical operators: +, -, *, /, div, mod. Set intersections and set exclusions are expressed as follows:

```
T1 ::= INTEGER ((10..500) AND ODD())
T2 ::= INTEGER ((10..500) AND NOT (5|15|75|115))
```

For values of a STRING type, constraints can be expressed on their length as on numeric values shown above. Constraints can also be expressed using the alphabet() action to define dependencies between the alphabet of the values and alphabets of other components using the from and + (string concatenation) operators. For BIT STRING values, we can

use the following additionnal operators: && (bitwise logical *and*), || (bitwise logical *or*), ! (bitwise logical *not*). For string values, we can also do some comparisons as well as define intervals assuming the lexicographic order:

```
T1 ::= SET { b OCTET STRING,
             c OCTET STRING (FROM (('A'..'Z') AND NOT ALPHABET(@.b))),
             d OCTET STRING (@.b+@.c)    -- string concatenation
           }

T2 ::= Set { b BIT STRING,
             c BIT STRING,
             d BIT STRING (!@.b && @.c)
           }

T3 ::= SET {
             a SET OF INTEGER (0..100),
             b SET (10+@.a) OF INTEGER,
             c SEQUENCE OF BIT STRING,
             d SEQUENCE (!@.c) OF BIT STRING
          }
```

Operators used on STRING, SEQUENCE OF or SET OF values are grouping operators and model iterations: these operators have to be interpreted as follows:

```
10+{1,2,3} = {11,12,13}
!{"1001001","1100"} = {"0110110","0011"}
{1,2,3} < {11,12,13}
```

Constraints on SET OF and SEQUENCE OF types are as in ASN.1 92 extended with operators.

```
T1 ::= SET {
        a INTEGER,
        b SEQUENCE,
        c SET (SIZE (@.a+#@.b) OF INTEGER (0..@.a+#@.b)
        }
```

General constraints are constraints which are defined and specified by designers. The syntax used in [ISO-8824.3 92] for general constraints is:

```
ASN.1 type (CONSTRAINED BY
                 { -- enbglish text describing the constraints}
              !exceptionSpec
             )
```

So the general constraints specification is a special form of ASN.1 comments. No means are given to describe them formally. It is only stated that one can use a programming language for this purpose. We attempt here to formalize the informal description of general constraints. This formalization is based on constraints specifications as above. The difference from it is the use of conditional constraints, variables and a few more operators that allow the description of more complex constraints. Indeed, every constraint can be

defined using the general constraint definition syntax. In sub-typing, the attribute constrained is generally implicit and in the general constraints syntax attributes are explicitly pointed out as we'll see later. Each constraint definition is introduced by the key word `constrained by` as in [ISO-8824.3 92]. Conditional constraints are defined as follows:

```
IF conditions THEN assertions FI
IF conditions THEN assertions ELSE assertions FI
```

where *conditions* and *assertions* have the same syntax. Examples of general constraints are shown below:

```
T1 ::= OCTET STRING (CONSTRAINED BY { variables:
                                          b,c;
                                        constraints:
                                          b FROM('a'),
                                          c FROM('b'),
                                          @ = b+c}
                     !exceptionSpec)


 T2  ::= SEQUENCE {
        a INTEGER {un(10), two(20), trois(30)},
        b BIT STRING OPTIONAL
                 (CONSTRAINED BY { IF @.a >= 10 THEN PRES(@) FI;
                                   IF @.a = 20 THEN #@.b <= 8 FI
                                 }
                  !exceptionSpec),
        c BOOLEAN,
        d SEQUENCE OF INTEGER
             (CONSTRAINED BY { IF @.c THEN #@.d < #@.b FI;
                               IF NOT(@.c) AND @.a > 10 THEN @[2] = @.a*2 FI
                             }
              !exceptionSpec)
        }
```

Note that components are always referred using the pointing notation, while variables are not. Constraints are defined as disjunctions of conjunctions of constraints: if `a`, `b`, `c` and `d` are constraints, the definition

```
(CONSTRAINED BY {a,b,c;
                 a,d}
)
```

is equivalent to:

```
(CONSTRAINED BY {(a AND b AND c) OR
                 (a AND d)}
)
```

3 THE TESTS DESCRIPTION LANGUAGE

Since exhaustive testing is infeasible, the problem of selecting input data is the problem of guessing which tests will provide the most information about the tested system. The iterative aspect of tests production is a mechanical task. However, deciding which test inputs have the right properties requires ingenuity, experience and sometimes intuition. These informations cannot be provided by the ASN.1 structures. The tester's knowledge of potential pitfalls and boundary conditions within the implementation is a guide in that search. The Test Description Language allows test designers to provide a symbolic description of properties of the test inputs over which a given system should be tested. This section describes the TDL language, a test data description language for ASN.1 structures. This language uses the constraints description formalism introduced in the previous section.

3.1 Properties

TDL allows the description of test inputs in a structured and coherent way according to ASN.1 data structures. Test inputs are described symbolically with a collection of properties. A property describes a set of alternatives for some ASN.1 leaf type (e.g INTEGER, STRING, ...). The alternatives of a given property are exclusive choices. The idea behind the notion of properties is the mathematical equivalence classes. A property is a set of equivalence classes, each alternative of the property corresponding to an equivalence class. For some ASN.1 structure, a test template can be designed by selecting exactly one equivalence class from the properties associated to each leaf type of this structure. And a test input is obtained by choosing a representative of each equivalence class selected. So we can create one or many test inputs for a given test template.

A property describes some characteristics a test designer want a test input to have. These characteristics are naturally described using constraints. These constraints define a set of ASN.1 values and a test input must satisfy these constraints. The stronger are the constraints, the more precise are the test input described. We then define a notion of tests refinement: the most precise properties are defined using constants (i.e. ASN.1 values) and the less precise are the empty properties (no more restriction is made on the input domain than the one indicated in the standard document). Suppose the following ASN.1 structure:

```
MyType ::= Sequence {
            a INTEGER,
            b BOOLEAN
           }
```

If no properties are associated to components `a` and `b`, we have the possible input sets
`a` $\in$ `]-`∞`, +`∞`[` and `b` $\in$ `{TRUE, FALSE}`
Since this input space is infinite, the test inputs which can be chosen from this set may not be precise. Meanwhile, if we define the properties

```
PROP a {prop1 (100)},
PROP b {prop1 (TRUE)}
```

we would have one possible choice of input

```
{a 100,b TRUE}
```

We can weaken the constraints and then extend the properties as follows

```
(1)  PROP a {prop1 (100)}                              : 2 possible choices
(2)  PROP a {prop1 (1..100)}, PROP b {prop1 (TRUE)}    : 100 possible choices
(3)  PROP a {prop1 (1..100)}                           : 200 possible choices
```

Figure 2 shows how tests are refined.

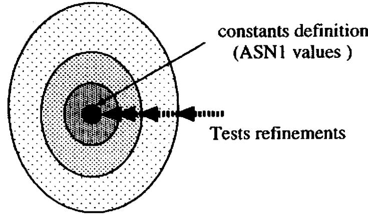

Figure 2 Tests refinement

Properties definition begins with PROP followed by the set of alternatives. Each alternative is given a name and its characteristics are described using constraints. The example properties shown above are defined using one alternative per property. An example with more alternatives is

```
PROP a {small(1..100)|middle(500..1000)|big(1000 ..<5000)}
PROP b {small((1..20) OR (60..100))|big(500..1000)}
```

A test template is constructed by choosing one alternative from each property definition, for example {(a small), (b small)} for the above definition. If for a given property, the number of test inputs that have to be created is equal to the number of its alternatives, each alternative can be chosen exactly once. However, if the number of the test inputs is less than the number of alternatives, we must choose an alternative in the set of alternatives. The key question is how to make a choice from a set of alternatives. In the previous definition, an alternative may be chosen at random with an equal probability of choosing any particular alternative. This may not be the best way of choosing alternatives. For many systems, we would like to choose some alternatives more often than others, because generally errors are not uniformly distributed among the alternatives (see figure 3).

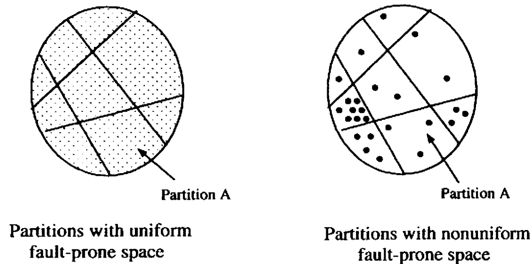

Figure 3 Partitions in a test space

To handle this problem, a solution is to allow weights to be associated to alternatives to force some to be selected more often than others. Indeed, much of the research that will go in the TDL description language concerns the identification of selection rules. Probably the

most natural selection rule is specifying the probability of selecting a particular alternative. Probabilities model the repartition of potential faults among the alternatives. Suppose that for four faults detected, the probable repartition according to component a is two inputs with property *small*, one input with property *middle* and one input with property *big*. The test description must be as follows:

```
PROP a {petit%2(1..100)|moyen%1(500..1000)|grand%1(>5000)}
```

Properties for string components are described using the constraints definition formalism on string types, e.g constraints specified using the length and alphabet attributes. For example, we may define the following properties for a component s of OCTET STRING type:

```
PROP s { normal%2 (UPPERCASE(@[1]) AND #@ IN (1..100))|
         error%1 (LOWERCASE(@[1]) AND #@ IN (1..100))
       }
```

Moreover, we would like to describe tests with a particular distribution of the characters in a given string. This is particularly useful for generating tests for arithmetic circuits and network managed objects which model network resources. Some arithmetic circuits and then the managed objects which model these circuits are sensitive to the number of ones and zeros of a bit string. We then extend the manner we associate weights to alternatives to characters of strings. The following definition allows the generation of bit strings with a lots of ones and a few zeros:

```
PROP s {normal%2 (FROM(%25 '1'|%5 '0'))|...
       }
```

The most obvious use of weights is to guide the test generator to favour those test data that the designer feels will detect the most errors.

3.2 Tests description

A test description for an ASN.1 structure is a collection of properties of the components of that structure. A test description is assigned a name which may be the same as the ASN.1 type name. For example, for the ASN.1 type

```
MyType ::= SET { a INTEGER,
                 b BITSTRING
               }
```

we can define the following test

```
MyTest ::= { PROP a { small(1..10) | big(80..100)},
             PROP b { one(FROM ('1')) | zero(FROM('0'))}
           }
```

When defining tests, there are many cases that require the use of a value that must be computed before. Such a value must be generated by the generator in an intermediate stage. Variables allow test designers to do this. Suppose one wants to add another alternative to the above definition in order to create a test input for which the value of

component b is an arbitrary number of ones followed by an arbitrary number of zeros. One must use variables as the following example illustrates:

```
MyTest ::= { Variables:
                   x,y;
                Properties:
                   PROP x { one(FROM('1'))},
                   PROP y { zero(FROM('0'))},
                   PROP a { small(1..10) | big(80..100)}
                   PROP b { one(x:one) | zero(y:zero) |
                               one_zero(x+y)}  -- concatenation of x and y
             }
```

The operation ":" models reusability of alternatives already defined in the same ASN.1 structure. To import alternatives, you indicate the name of the property followed by the set of alternatives. Suppose you have defined a property for component a as follows:

```
PROP a { one(1..10) | two(...) | three(...)}
```

You can import the alternatives defined for a to define alternatives for some component b belonging to the same structure, if a and b have compatible types:

```
PROP b { one (@.a:one) | two_three(@.a:one|@.a:three)}
```

So you can group many imported alternatives into one alternative that is the union of these alternatives. It is possible to refine the imported alternatives using the grouping operators (since an alternatives is "set of" values) as follows:

```
PROP b { one (@.a:one+10) | ...}
```

which is equivalent to:

```
PROP b { one(11..20) | ... }
```

Refinement operators are operators over alternatives. To import alternatives defined in another test description you use the "::" operator:

```
MyTest::Myprop
```

which imports all the alternatives defined for property MyProp defined in the test MyTest. You can select the alternatives of Myprop, group some of them into one alternative using the operator ":" as shown above. Operators ":" and "::" are syntactic operators and allow reusability, extention and refinement of tests descriptions.

Dependencies are defined using mathematical or other operators as we have seen in section 2. If you want to define a test data which is the combination of values of other components, in order to specify dependencies of some PDUs parameters or to test some particular combinations, for the ASN.1 structure

```
MyType ::= SEQUENCE { a [0] INTEGER,
                      b [1] INTEGER OPTIONAL,
                      c SET {d BIT STRING,
```

```
                                    e BOOLEAN
                                  }
                          }
```

you can define it as follows:

```
MyTest ::= { PROP a {small(1..100) | big(500..1000)},
             PROP b {first(@.a+100) | second(ABS(@))},
             PROP c { PROP d { first(IF PRES(@.b) THEN #@ IN (1..50)) |
                                second(IF ABS(@.b) THEN #@ IN (50..100))
                              },
                      PROP e { ... }
                    }
           }
```

4 CONCLUSION

Test descriptions are an important intermediate representation between ASN.1 abstract data structures and actual test data. In general, it is desirable to know the extent to which a test description can be derived from an abstract data specification and the extent to which a tester must rely on information external to the specification. In this paper we have helped to answer this question by presenting a mechanism that will help the tester when creating test data from ASN.1 abstract specifications.

It is unreasonable to expect the derivation of test data from abstract structures without explicit statements of the properties of the tests we want to create. Thus, the role of tests description as identified here is to relieve the burden of routine tasks and free the test engineer to concentrate on other areas, e.g properties he wants to test. Tests description can be useful in early phases of the design process to identify interesting properties of the ASN.1 specification. And we hope to provide test creators with as much automated support as possible to derive consistently test data. In future work, we will try to provide selection procedures that designate which combinations of alternatives should be tested and representatives test inputs for each combination.

REFERENCES

[Ammann 93] Paul Ammann et Jeff Offut. *Using formal methods to mechanize category-partition testing.* Techreport no. ISSE-TR-93-105, George Mason University, Virginia, USA, 1993.

[Bapat 93] Subodh Bapat. *Richer semantics for management information.* Integrated Network Management III, pages 15–28. *Elsevier Science Publishers B.V.*, 1993.

[Bochmann 91] G. v. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi et G. Luo. *Fault Models in Testing.* J. Kroon, R.J. Heijink et E. Brinksma, éditeurs, *Protocol Test Systems* IV, pages 17–30, October 1991.

[Duran 84] J.W. Duran et S.C. Ntafos. *An evaluation of random testing.* IEEE Trans. on SE, SE-10(4):438–444, 1984.

[ISO-10165.4 92] ISO, *Structure of Management Information - Part 4: Guidelines for the Definition of Managed Objects*, IS, ISO-10165.4, January 1992.

[ISO-8807 87] ISO, *LOTOS: A Formal Description Technique based on the Temporal Ordering of Observable Behaviour*, Draft International Standard, ISO-8807, June 1987.

[ISO-8824 90] ISO, *Specification of Abstract Syntax Notation Number One (ASN.1)*, International Standard, ISO-8824, December 1990.

[ISO-8824.1 92] ISO, *"Abstract Syntax Notation Number One (ASN.1) - Part 1: Specification of Basic Notation"*, Draft International Standard, ISO-8824.1, december 1992.

[ISO-8824.3 92] ISO, *"Abstract Syntax Notation Number One (ASN.1) - Part 3: Constraint specification"*, Draft International Standard, ISO-8824.3, december 1992.

[ISO-8825 87] ISO, *Specification of Basic Encoding Rule for Abstract Syntax Notation Number One (ASN.1)*, International Standard, ISO-8825, december 1987.

[Myers 79] G. L. Myers. *The Art of Software Testing.* John Wiley & Sons Inc., 1979.

[Van de Burgt 89] S.P. Van de Burgt et P.A.J. Tilanus. *Attributed ASN.1. FORTE'89*, pages 298–310, December 1989.

[Varvitsiotis 93] A.P. Varvitsiotis et G.I. Stassinopoulos. *Extending asn.1 into a full-fledged constraint language in the context of osi protocol conformance testing.* Computer Networks and ISDN Systems, 25(11):1243–1263, June 1993.