From 1 Notation to Another One: An ACT-ONE Semantics for ASN.1

Muffy Thomas
Dept. of Computing Science, Glasgow University, Scotland.
Under sponsorship from British Telecom Research.

1 Introduction

ASN.1 (Abstract Syntax Notation One) [ISO 8824], is a notation for defining the data values used in OSI Application layer protocol standards. Although it is a language with an International Standard - [ISO 8824] - the standard is written in natural language (English) and is therefore open to the usual problems and ambiguities of natural language specifications that can lead to different interpretations of the same standard.

We have defined a formal semantics for ASN.1 using the algebraic abstract data type language ACT-ONE. ACT-ONE is a sublanguage of the ISO language LOTOS [ISO 8807] and has a formal mathematical semantics. The aims of giving an ACT-ONE semantics to ASN.1 are twofold. The first is to provide a formal semantics for the language ASN.1 [ISO 8824]; clearly, a formal semantics is desirable for any language with a large community of users. The second is to provide a means of relating (i.e. comparing and translating) ASN.1 and ACT-ONE specifications. This will allow us to integrate protocol specifications written in the two languages and exploit support tools for both environments. The formalisation of this relationship will allow the designer to use existing ASN.1 data descriptions (with their associated encoding rules) in LOTOS process descriptions and to identify a subset of LOTOS (ACT-ONE) data types that can be compiled into ASN.1 types. Thus, a formal mathematical semantics for ASN.1 will allow a rigorous discussion of the features and ambiguities within the two languages and this may lead to suggestions for improvements.

The purpose of this paper is to motivate the need for such a semantics and to demonstrate some parts of the semantics. The complete semantics of the language is given in [Thomas 89].

In section 2, we give an overview of the approach, which is denotational in style, and we discuss some of the problems of defining a formal semantics for ASN.1. In section 3 we give an example using a primitive type of ASN.1 and in the following section discuss some of the constructed types of ASN.1. In section 5 we present our conclusions and suggestions for improvements and compare our approach with the related work. Directions for future work are presented in the final section.

We assume some familiarity with both ASN.1 and ACT-ONE; the reader may consult Annex E of [ISO 8824] for a tutorial introduction to ASN.1 and [Ehrig Mahr 85] for an introduction to algebraic specification and the language ACT-ONE.

2 Outline of Semantics

Each ASN.1 data type *denotes* an ACT-ONE data type and the style of the semantics is *denotational*: the denotation (or meaning) of an ASN.1 type may depend on the denotations of its components. The approach is "syntactic" in the sense that we map ASN.1 types into

ACT-ONE type *specifications*; these latter specifications denote algebras (as defined by the initial semantics of ACT-ONE). ACT-ONE is a high-level specification language with a two-level semantics: we use some of the high-level ACT-ONE structuring constructs in our semantic descriptions, and introduce some additional meta-level constructs in our descriptions.

The ASN.1 language allows the description of data at the level of *values*, *types* and *modules* (collections of types and values); thus three semantic evaluation functions are defined: $\mathcal{Eval}$ (for values), $\mathcal{Tval}$ (for types), and $\mathcal{Mval}$ (for modules). The most important of these is the function $\mathcal{Tval}$ which gives meanings to the primitive and constructed ASN.1 types: it maps an ASN.1 type into an ACT-ONE type. In this paper we shall restrict our attention to the function $\mathcal{Tval}$, which is only defined for well-formed ASN.1 types.

We have not considered the encoding and decoding rules of ASN.1 as we consider these to be "implementation" aspects of the language rather than an integral part of the language which must be translated. If encoding/decoding rules are required for the ACT-ONE equivalents of ASN.1 data types, then perhaps encoding/decoding rules for ACT-ONE types in general should be defined.

2.1 Syntactic Categories

In this section, we review the BNF description of the abstract syntax for ASN.1 types from [ISO 8824], excepting the non-terminals NamedType and Ntype, which we have modified. Also, two constructions are excluded: the tagged type and the any type. Comments are enclosed in "/*" and "*/" and the only BNF operators used are "::=" and "|".

```
Typereference        ::= ASCII_String

Type                 ::= primitive | constructed | Typereference

primitive            ::= INTEGER {NamedNumberList} | INTEGER | BOOLEAN |
                         BIT STRING {NamedBitList} | BIT STRING | OCTET STRING | NULL

constructed          ::= SEQUENCE OF Type | SET OF Type |
                         SEQUENCE {ElementTypeList} | SET {ElementTypeList} |
                         CHOICE {AlternativeTypeList}

NamedNumberList      ::= Identifier(SignedNumber) |
                         Identifier(SignedNumber):NamedNumberList

NamedBitList         ::=Identifier(Number) | Identifier(Number): NamedBitList

ElementTypeList      ::= NamedType | NamedType : ElementTypeList

AlternativeTypeList  ::= Ntype | Ntype : AlternativeTypeList

/*Ntype and NamedType differ from the ISO description */
/* N is normal, O is optional, D is default type with default value V*/

NamedType            ::=<Ntype, N> | <Ntype,O> | <Ntype,D,V> | Components of Type

Ntype                ::= <Identifier,NType> | Type | Identifier < NType
```

2.2 Semantic Domains

The semantic domains consist essentially of *abstract* ACT-ONE specifications as defined in [ISO 8807]. However, for readability we have in many cases used the concrete syntax of ACT-ONE rather than the abstract syntax. Any specification references not defined in this paper are references to specifications from the standard library of data types given in Annex A, [ISO 8807].

We briefly review the structure of ACT-ONE specifications below. We use only two of the structuring concepts in the following: combination and parameterisation; the full definition of the abstract syntax (using all structuring concepts) can be found in [ISO 8807].

ACT-ONE_Spec ::= type Name is Pexpr endtype

Name ::= ASCII_string

Pexpr ::= Namexpr Pspec | Name actualized by Namexpr [using repl]

Namexpr ::= Name | Namexpr, Name

Pspec ::= [formal sorts Sortlist] [formal opns Opnlist] [formal eqns Eqlist]
[sorts Sortlist] [opns Opnlist] [eqns Eqlist]

The full definitions of types Opnlist, Sortlist, and Eqlist are standard and have been omitted.

Some additional operations on specifications have been defined for convenience: operations to add and remove sorts, operations, and equations. The full definitions of these operations: *plussorts*, *pluseqns*, *plusopns*, *minuseqns*, and *minusopns* respectively, are given in [Thomas 89]. In addition to these operations, we shall refer to the (bodies) of specifications with the operation

body : Name -> Pexpr,

which binds the specification names to their respective bodies.

2.3 Type Evaluations

The denotation of an ASN.1 type is *essentially* an ACT-ONE specification. However, because new specification names may need to be introduced, the evaluation of a type may affect the environment. Thus, the type evaluation function is:

Tval: (Type x Environment) -> (Pexpr x Environment)

where x is the usual product type constructor and -> is the usual (continuous) function space constructor. We assume that each ASN.1 type name is mapped to the identical ACT-ONE type name and so only one simple environment for referencing and dereferencing names is required. The environment is defined by:

Environment = Typereference -> Type

and the associated projection functions are

proj1: (Pexpr x Environment) -> Pexpr

proj2: (Pexpr x Environment) -> Environment.

As an example of the need for environments, consider the ASN.1 declaration

T ::= SEQUENCE OF SEQUENCE OF INTEGER.

SOL [|Typereference|] ρ = SOL[|$lookup$[|Typereference, ρ |]|] ρ

2.4 Interpretation of ASN.1 Standard

The ASN.1 language standard explicitly defines various types of *data*, but not the associated *operations* on the data. Many of the relevant selector operations on data are implicitly present in the standard, for example, the operations to select the first and trailing bits in a bit string. However, many operations are not made explicit and must be inferred. For example, selectors for the SEQUENCE type are not defined in the standard, so we have assumed a selector for each Element Type (named either by its position in the Element Type List, or by its identifier). Selectors for the SET type have not been introduced because of the commutative property of the type.

The ACT-ONE denotation of each ASN.1 type defines a (semantic) congruence on the various syntactic forms of data. It is not clear from the standard whether or not a *syntactic* test for equality is also required. We have assumed that because equality predicates are used (informally) by users of the ASN.1 notation, syntactic equality predicates should be included in each of the ACT-ONE denotations. These operators (called "eq" and "neq") could be omitted if not required; here, they are omitted in the constructed types.

Several ASN.1 type declarations include the use of identifiers, but there is no explicit mention in the standard of the scope of those identifiers. We have assumed that the scope of an identifier which is used in a type declaration is exactly the scope of that type. This is achieved by incorporating the identifiers into the model: in the denotation of the type, the identifiers are defined as (constant) operators of the relevant sort and their bindings are given by the equations.

In order to give meaning to *tagged* types, the following ACT-ONE type of TAGS, which is an enrichment of the type CLASS, is included in every denotation of an ASN.1 type:

```
type CLASS is
BOOL
sorts Class
opns        universal          : Class
            application        : Class
            private            : Class
            context_specific: Class
endtype

type TAGS is
DecNatRepr, CLASS
sorts Tag
opns        <_,_>              : Class, Decstring -> Tag
endtype
```

3 Primitive Types

The primitive ASN.1 types are integers, booleans, bit strings, octet strings, and the null type. The types of bit strings and octet strings allow several representations of values: binary, hexadecimal, and lists of non-zero bit positions, thus the respective ACT-ONE specifications for these types are rather complex. Therefore, to illustrate the semantics, we choose a simple

In ACT-ONE, a name for the intermediate type SEQUENCE OF INTEGER is required. Assuming that we have a (parameterised) ACT-ONE type SEQUENCE, we would translate the declaration of T into

T is SEQUENCE actualized by Dummy
using sortnames Seq for Data

where

Dummy is SEQUENCE actualized by INTEGER
using sortnames Int for Data.

There are several functions associated with environments; their types are given below using the additional type constructor + for disjoint union.

the empty environment

new: Environment

the update operation

_[_/_]: (Environment x Type x Typereference) -> Environment

the name lookup operation

name: ((Type + Typereference) x Environment) -> (Typereference + {undef})

the type lookup operation

lookup: ((Type + Typereference) x Environment) -> (Type + {undef})

an operation to generate a new name

newname: Environment -> Typereference

and two predicates

isname: (Type + Typereference) -> Bool
istype: (Type +Typereference) -> Bool

We note that ***name*** and ***lookup*** are the identity functions for Typereference and Type respectively. For example, for all t:Type, for all ρ:Environment, ***lookup***(t,ρ) = t.

Finally, in order to define the denotation of some constructed types, the "sorts of interest" of the corresponding (ACT-ONE) denotations of the ASN.1 types must be defined. These are given by the following function ***SOI*** which maps an ASN.1 type into an ACT-ONE sort.

SOI : ((Type + Typereference) x Environment) -> Sort

SOI [|BIT STRING|]ρ = BString
SOI [|BIT STRING {NamedBitList}|] ρ = BString
SOI [|INTEGER|] ρ = DecString
SOI [|BOOLEAN|] ρ = Boole
SOI [|OCTETSTRING|] ρ = Octetstring
SOI [|SET OF Type|] ρ = Set
SOI [|SEQUENCE OF Type |] ρ = Seq
SOI [|SET {ElementTypeList}|] ρ = Set
SOI [|SEQUENCE {ElementTypeList}|] ρ = Seq
SOI [|CHOICE {AlternativeTypeList}|] ρ = Choice

example: the type of integers both with and without identifiers. The denotation of ASN.1 integers, with and without identifiers, uses the following ACT-ONE specification of integers (which refers to the standard library type DecNatRepr):

```
type INTEGER is
DecNatRepr,TAGS
opns        - : DecString -> Decstring
            - : Nat -> Nat
            tag : DecString -> Tag
eqns
forall ds,x,y:Decstring
ofsort Nat
      NatNum(-ds) = -(NatNum(ds))
ofsort Tag
      tag(ds) = <universal,2>
ofsort Bool
      x eq y   =>     -(x) eq     -(y)    = true
      x neq y  =>     -(x) eq     - ( y ) = false
      x eq y   =>     -(x) neq    -(y)    = false
      x neq y  =>     -(x) neq    - ( y ) = true
endtype
```

Assuming that ρ is an environment, the ACT-ONE denotation of integers without identifiers is given by:

$\mathcal{T}val$[|INTEGER|]ρ = *body*(INTEGER) x ρ.

Assuming that $\mathcal{L}val$ is a function used in the definition of $\mathcal{E}val$ and maps ASN.1 integer values into ACT-ONE integers values, the denotation of integers with identifiers is given by:

$\mathcal{T}val$[|INTEGER NamedNumberList|] ρ =
$proj1$:($\mathcal{T}val$[|INTEGER|] ρ)
    plusopns $addid$(NamedNumberList)
    pluseqns $addeq$(NamedNumberList)
    x ρ

where
$addeq$:List(Id x SignedNumber) -> Eqns
$addid$: List(Id x SignedNumber) -> Opns

$addid$(id(n):(id'(n'):l)) = id:DecString, $addid$(id'(n):l)
$addid$(id(n):[]) = id:DecString

$addeq$(id(n):(id'(n'):l)) = (ofsort DecString, id = $\mathcal{L}val$(n)), $addeq$(id'(n'):l)
$addeq$(id(n):[]) = (ofsort DecString, id = $\mathcal{L}val$(n)).

As an example, consider the following ASN.1 type declaration.

```
DAYSOFTHEMONTH ::= INTEGER {first(1),last(31)}
```

Given an appropriate environment ρ, we translate this declaration into:

type DAYSOFTHEMONTH is $\mathcal{T}val$[| INTEGER {first(1),last(31)} |]ρ.

$proj1(\mathcal{T}val$[| INTEGER {first(1),last(31)} |]ρ)

=
body(INTEGER)
plusopns first:DecString
last:DecString
pluseqns
ofsort DecString first = 1;
ofsort DecString last = 31;
endtype

=
DecNatRepr,TAGS
opns - : DecString -> Decstring
- : Nat -> Nat
tag : DecString -> Tag
first:DecString
last:DecString
eqns
for all ds,x,y:Decstring
ofsort Nat
NatNum(-ds) = -(NatNum(ds))
ofsort Tag
tag(ds) = <universal,2>
ofsort Bool
x eq y => -(x) eq -(y) = true
x neq y => -(x) eq -(y) = false
x eq y => -(x) neq -(y) = false
x neq y => -(x) neq -(y) = true
ofsort DecString
first = 1
last = 31
endtype

4 Constructed Types

The constructed ASN.1 types are (homogeneous) sequences and sets (SEQUENCE OF Type and SET OF Type resp.), the two heterogeneous *record* types (SEQUENCE {ElementTypeList} and SET {ElementTypeList} resp.), and the choice type (CHOICE {AlternativeTypeList}). The choice type is quite straightforward and so we do not discuss it here.

4.1 Set and Sequence Types

The denotations of (homogeneous) sequences and sets use the parameterised types SEQUENCE and SET; new type names may be required in order to actualise the parameters. The only

difference between the two types is that the SET type has a commutative constructor operation; we consider only sequences in our example. The ACT-ONE parameterised type SEQUENCE and the ACT-ONE denotation of a sequence type (SEQUENCE OF Type) are given below. We should note that new type names are introduced only when the argument Type is not a *named* type (i.e. it is neither a type name nor a type already associated with a name in the current environment.)

```
type SEQUENCE is
TAGS
formalsorts     Data
sorts           Seq
opns      { }       : Seq
          _++_      : Data, Seq -> Seq
          tag       : Seq -> Tag
eqns
forall s:Seq
ofsort Tag
     tag(s) = <universal, 16>
endtype
```

$\mathcal{T}val$[|SEQUENCE OF Type]ρ =
if (***name***[|Type|]ρ) <> undef
then
(SEQUENCE actualized by ***name***[|Type|]ρ
using $\mathcal{SOL}$[|Type|]ρ for Data) x ρ
else
(SEQUENCE actualized by ***newname***(ρ)
using $\mathcal{SOL}$[|Type|]ρ for Data) x ρ[***newname***(ρ)/Type]

As an example, consider the ASN.1 type declaration

HOLIDAYS ::= SEQUENCE OF DAYSOFTHEMONTH

where DAYSOFTHEMONTH is defined in the previous section. Given an appropriate environment ρ,***isname***[|DAYSOFTHEMONTH|] is true and so the first branch of the conditional applies. Thus,

$\mathcal{T}val$[|SEQUENCE OF DAYSOFTHEMONTH|]ρ =
(SEQUENCE actualized by DAYSOFTHEMONTH using Decstring for Data) x ρ.

4.2 Records

In this section we discuss the semantics of the two heterogeneous *record* types: sequences of types and sets of types. These types are more complicated than the other constructed types for three reasons. First, they are parameterised by several (possibly) distinct types; second, the parameters may be *optional* or *default*, thus affecting the way in which values of these types are constructed; and third, the parameters may be *component* or *selection* types.

If a type is *optional* then a value of that type may be omitted from values of the constructed type; if a type is *default* then when a value of that type is omitted in a value of the constructed type, the default value is assumed to be present. A *component* type is used as an abbreviation for

the components of a type; for example, if T is SEQUENCE {INTEGER, BIT STRING}, then a reference to COMPONENTS OF T is a reference to its component types INTEGER and BIT STRING. A *selection* type is used to refer to a component of a constructed type with the same identifier; for example, if T is CHOICE {id1 INTEGER, id2 BIT STRING}, then id1< T refers to the type id1 INTEGER.

As before, the only difference between the sets and sequences is the commutativity of constructors. We choose the simpler type of sequences as an example.

The denotation of the sequence type involves several auxiliary functions which we give after the equation for the SEQUENCE {ElementTypeList} type.

```
Tval [|SEQUENCE {ElementTypeList}|]ρ =
mapname(flatten(ElementTypeList)ρ)ρ, TAGS
sorts Seq
opns { }    :Seq
     plusopns(seqarities(flatten(ElementTypeList)ρ,Seq)ρ)
eqns
forall s:Seq
ofsort Tag
     tag(s) = <universal, 16>
pluseqns(defaults(flatten(ElementTypeList)ρ,Seq)ρ)
pluseqns(selectors(flatten(ElementTypeList)ρ,Seq)ρ)
endtype  x ρ
```

The auxiliary functions are defined below. First, we give informal descriptions:

mapname	converts a list of ASN.1 types and type references into a list of ACT-ONE type references.
mapsort	converts a list of ASN.1 types and type references into a list of ACT-ONE sorts and identifier names.
flatten:	expands component and selection types in type list.
expandcomp	expands components in type list.
expandsel	expands selection types in type list.
findid	finds a field name in a list of types.
seqarities	adds arities of operators for sequence construction, and selection, allowing for default and optional types to be omitted. The selector names are the resp. identifiers if they are defined and the argument position number (eg. 1,2,3...) otherwise.
selectors	defines equations for selectors.
defaults	defines equations for default values.

The formal definitions are given, using the one element list as the base case (because empty argument lists are not allowed) and ε as the "absent" identifier, or field name, by:

```
mapname(<Ntype,X>:t':elist)ρ
=    name(Ntype',ρ): mapname(t':elist)      if Ntype =  <id,Ntype'>
=    name(Ntype,ρ): mapname(t':elist)       otherwise
```

```
mapname(<Ntype,X>t:[])ρ
=    name(Ntype',ρ)                                  if Ntype = <id,Ntype'>
=    name(Ntype,ρ)                                   otherwise


mapsort(t:t':elist)ρ
=    < SOI(lookup[|T|]ρ)ρ,id>: mapsort(t':elist)ρ    if t=<<id,T>,X>
=    < SOI(lookup[|T|]ρ)ρ,ε>: mapsort(t':elist)ρ     otherwise

mapsort(t:[])ρ
=    <SOI(lookup[|T|]ρ)ρ,id>                         if t=<<id,T>,X>
=    <SOI(lookup[|T|]ρ)ρ,ε>                          otherwise

flatten(t:t':eList)ρ
=    expandsel(expandcomp(t:t':elist)ρ)ρ

expandcomp(t:t':elist)ρ
=    expandcomp(elist')ρ ++ expandcomp(t':elist)ρ
                                                     if t = COMPONENTS OF T
=    t: expandcomp(t':elist)ρ                        otherwise

     where lookup[|T|]ρ = SEQUENCE {elist'}

expandcomp(t:[])ρ
=    expandcomp(elist')ρ                             if t = COMPONENTS OF T
=    t                                               otherwise

     where lookup[|T|]ρ = SEQUENCE {elist'}

expandsel(t:t':elist)ρ
=    <id,expand(id,Ntyp)ρ >:expandsel(t':elist)ρ     if t = <id < Ntyp, X>
=    t:expandsel(t':elist)ρ                          otherwise

expandsel(t:[])ρ
=    <id,expand(id,Ntyp)ρ >                          if t = <id < Ntyp, X>
=    t                                               otherwise

expand(id,Ntyp)ρ
= findid(id,expandsel(elist)ρ)
     if   (lookup(Ntyp)ρ=CHOICE {elist}
          or lookup(Ntyp)ρ=SEQUENCE {elist} or lookup(Ntyp)ρ=SET {elist})

findid(id  t:t':elist)
=    <id,T>                                          if t = <<id,T>,X>
=    findid(id,t':elist)                             otherwise
```

findid(id,t:[])
= <id,T> if t = <<id,T>,X>
= undef otherwise

The remaining three operations are *specified* as follows:

seqarities(S_1,...,S_n,s)ρ
defines a declaration of a set of operators containing:

let ***mapsort***(S_1,...,S_n)ρ = <s_1,i_1>,...,<s_n,i_n>, in

i) the n-ary operator { i_1 _, i_2 _,..., i_n _ } : s_1,...,s_n -> s

ii) n unary operators defined by:
(for all j:1≤j≤n if i_j = ε then j: s -> s_j else i_j : s -> s_j)

iii) the m-ary operators which allow optional and default types to be absent:
(for all sequences <t_1,j_1>,...,<t_m,j_m>, s.t.
m<n and <t_1,j_1>,...,<t_m,j_m> is a proper subsequence of <s_1,i_1>,...,<s_n,i_n>
(for all i, 1≤i≤n s.t. s_i ∉ {t_1,...,t_m} => (S_i =<T,O> or S_i =<T,D>)
the m-ary operator { j_1 _, j_2 _,..., j_m _ } : t_1,...,t_m -> s))

selectors(S_1,...,S_n,s)ρ
defines a set of equations containing:

let ***mapsort***(S_1,...,S_n)ρ = <s_1,i_1>,...,<s_n,i_n>, in

(for all j:1≤j≤n
if i_j = ε *then*
forall x_1:s_1,...,x_n:s_n
ofsort s_j
$j(x_1,...,x_n) = x_j$
else
forall x_1:s_1,...,x_n:s_n
ofsort s_j
$i_j(x_1,...,x_n) = x_j$)

defaults(s_1,...,s_n,s)ρ
defines a set of equations containing:

let ***mapsort***(S_1,...,S_n)ρ = <s_1,i_1>,...,<s_n,i_n> in

(for all j ∈ {1,...,n}
(for all s_j s.t. S_j = <T,D,V_j>
forall x_1: s_1,..., x_{j-1}: s_{j-1}, x_{j+1}: s_{j+1}, ..., x_n: t_n
ofsort s
{i_1 x_1,..., i_{j-1} x_{j-1}, i_{j+1} x_{j+1},..., i_n x_n}
= { i_1 x_1,..., i_{j-1} x_{j-1}, i_j V_j , i_{j+1} x_{j+1},..., i_n x_n}))

As an example, consider the following ASN.1 type declaration involving a default type:

Ex1 ::= SEQUENCE {index INTEGER, hash BIT STRING DEFAULT 0000}.

Using our abstract syntax, the right hand side is:

SEQUENCE {<index, <INTEGER,N>>,<hash,<BIT STRING,D,0000>>}.

Assuming an environment ρ, the first component of the ACT-ONE denotation of this type is given by:

```
proj1(Tval [|SEQUENCE {   <index, <INTEGER,N>>,
                          <hash,<BIT STRING,D,0000>>} |]ρ) =
mapname(flatten(<index,<INTEGER,N>>,<hash,<BITSTRING,D,0000>>)ρ)ρ, TAGS
sorts       Seq
opns { }    :Seq
plusopns
(seqarities(mapsort(flatten
(<index,<INTEGER,N>>,<hash,<BITSTRING,D,0000>>)ρ,Seq)ρ)ρ)
eqns
forall s:Seq
ofsort Tag
      tag(s) = <universal, 16>
pluseqns(defaults(mapsort(flatten
      (<index,<INTEGER,N>>,<hash,<BITSTRING,D,0000>>)ρ,Seq)ρ)ρ)
pluseqns(selectors(mapsort(flatten
      (<index,<INTEGER,N>>,<hash,<BITSTRING,D,0000>>)ρ,Seq)ρ)ρ)
end type
```

```
=
INTEGER, BITSTRING, TAGS
sorts       Seq
opns { }    :Seq
plusopns(seqarities(<DecString,index>,<BString,hash>,Seq)ρ)
eqns
forall s:Seq
ofsort Tag
      tag(s) = <universal, 16>
pluseqns(defaults(<DecString,index>,<BString,hash>,Seq)ρ)
pluseqns(selectors(<DecString,index>,<BString,hash>,Seq)ρ)
endtype
```

```
=
INTEGER, BITSTRING, TAGS
sorts       Seq
opns        { }                 :Seq
            {index _, hash _}   : DecString, BString -> Seq
            {index _ }          : DecString -> Seq
            index               : Seq -> Decstring
            hash                : Seq -> BString
```

```
eqns
forall s:Seq, i: DecString, h: BString
ofsort Tag
      tag(s) = <universal, 16>
ofsort Seq
      {index i} = {index i, hash 0000}
ofsort DecString
      index({index i, hash h }) = i
ofsort BString
      hash({index i, hash h }) = h
endtype
```

As a second example, consider the following ASN.1 type declaration involving a components type:

Ex2 ::= SEQUENCE { INTEGER, COMPONENTS OF Ex1}.

Assuming that ρ is suitable environment containing the declaration of Ex1 as given above, we have:

```
proj1(Tval [|SEQUENCE { INTEGER, COMPONENTS OF Ex1 } |]ρ) =
mapname(flatten(INTEGER,COMPONENTS OF Ex1)ρ)ρ, TAGS
sorts       Seq
opns { }    :Seq
plusopns(seqarities(mapsort
      (flatten(INTEGER,COMPONENTS OF Ex1)ρ,Seq)ρ)ρ)
eqns
forall s:Seq
ofsort Tag
      tag(s) = <universal, 16>
pluseqns(defaults(mapsort
      (flatten(INTEGER,COMPONENTS OF Ex1)ρ,Seq)ρ)ρ)
endtype

=
mapname(expandsel(expandcomp(INTEGER,COMPONENTS OF Ex1)ρ)ρ)ρ, TAGS
sorts       Seq
opns { }    :Seq
plusopns(seqarities(mapsort(expandsel
      (expandcomp(INTEGER,COMPONENTS OF Ex1)ρ,Seq)ρ)ρ)ρ)
eqns
forall s:Seq
ofsort Tag
      tag(s) = <universal, 16>
pluseqns(defaults(mapsort(expandsel
      (expandcomp(INTEGER,COMPONENTS OF Ex1)ρ,Seq)ρ)ρ)ρ)
endtype

=
mapname
(<ε,INTEGER>,<index,<INTEGER,N>>,<hash,<BITSTRING,D,0000>>)ρ, TAGS
sorts       Seq
opns { }    :Seq
```

```
plusopns
(seqarities(mapsort
(< ε, INTEGER>,<index,<INTEGER,N>>,<hash,<BITSTRING,D,0000>>)ρ,Seq)ρ)
eqns
forall s:Seq
ofsort Tag
      tag(s) = <universal, 16>
pluseqns(defaults(mapsort
(< ε, INTEGER>,<index,<INTEGER,N>>,<hash,<BITSTRING,D,0000>>)ρ,Seq)ρ)
end type

=
INTEGER, BITSTRING, TAGS
sorts                  Seq
opns
{ }                    :Seq
{_, index _, hash _}   : DecString, Decstring, BString -> Seq
{ _, index _ }         : DecString, DecString-> Seq
1                      : Seq -> DecString
index                  : Seq -> DecString
hash                   : Seq -> BString
eqns
forall s:Seq, i,j: DecString, h: BString
ofsort Tag
      tag(s) = <universal, 16>
ofsort Seq
      { i, index j } = {index i, index j, hash 0000 }
ofsort DecString
      1({i, index j,hash h }) = i
      index({i, index j,hash h }) = j
ofsort BString
      hash({i, index j,hash h }) = h
endtype
```

This example very clearly illustrates the advantages of ASN.1: it is a very brief, but powerful, notation for certain kinds of data types. Namely, ASN.1 is very convenient to use when defining types with only a variety of injection and projection operations; of course it is not suitable for defining a type with an "algorithmic" component. In this case, a more powerful language such as ACT-ONE is required.

5 Conclusions

Several formal description techniques (FDTs) and notations are now used in the specification of OSI protocols, so it is very important to understand each of the techniques and the relationships between them. In this paper we have given an overview of an ACT-ONE semantics for ASN.1 using several examples for illustration. We have briefly mentioned some of the problems of giving such a formal semantics to a language with a natural language standard: clearly, different interpretations are possible.

One of the aims of formalising the standard is to allow a more rigorous discussion of the features of the languages and interpretations of the standard. The work of [Bochmann

Deslauriers 89] on translating ASN.1 into ACT-ONE differs from our work in several ways. For example, their approach does not consider optional or default types, nor does it distinguish between sequences and sets. Moreover, they translate the primitive types into standard library types, thus losing some of the ASN.1 features such as conversions between hexadecimal and bit strings and the ability to introduce local identifiers. In summary, the emphasis of their work appears to be on the construction of a compiler rather than the faithful preservation of the description of ASN.1 given in the standard.

On the other hand, our aim has been to stay faithful to the standard as much as possible. This approach has revealed problems both with ASN.1 and ACT-ONE. For example, selection types are allowed (in the syntax) to appear in any context whereas the standard description *seems* to imply that they may only refer to CHOICE types (our semantics gives meaning to any occurrence in a constructed type). Also, the question of equality on types remains unresolved in the standard. With respect to the target language ACT-ONE, we have found that many of the specifications would have been more easier and more elegant to write if we had a partial ordering on sorts. For example, the idea of many different representations of bit strings is best described using a partial ordering on the respective sorts; in ACT-ONE (without subsorting) we are forced to have several disjoint sorts and explicit coercion functions between them. These and other points concerning the two languages are discussed more fully in [Thomas 89].

6 Further Work

The semantics will be implemented as a compiler from ASN.1 to ACT-ONE. Also, we will identify a subset of ACT-ONE types which will allow us to define the inverse of the semantics and thus compilation from ACT-ONE into ASN.1. Finally, the work must be extended to include the entire language, including macros.

References

[Bochmann Deslauriers 89]
G. v. Bochmann, M. Deslauriers, Combining ASN1 support with the LOTOS language, in H. Brinksma, G. Scoll, C. Vissers (Eds.), Protocol Specification, Testing and Verification IX, North-Holland (in print).

[Thomas 89]
M. Thomas, A Semantics for ASN.1, Draft Report, 1989.

[Ehrig Mahr 85]
H. Ehrig, B. Mahr, Fundamentals of Algebraic Specification, Springer-Verlag, 1985.

[ISO 8824]
Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One, International Organisation for Standarisation, ISO 8824:1987 (E).

[ISO 8807]
Information processing systems - Open Systems Interconnection - LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour, ISO 8807.

Acknowledgements

Acknowledgement is made to the Research and Technology Board of British Telecom for permission to publish this paper. We thank David Freestone, Keith Rayner, Steve Rudkin, and

Doug Steedman for providing many stimulating comments, and Kei Davis for careful proof-reading.