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: Eval (for values), Tval (for types), and Mval (for modules). The most important of these is the function 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 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] | 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].

\[
\text{ACT-ONE Spec} ::= \text{type Name is Pexpr endtype}
\]

\[
\text{Name} ::= \text{ASCII_string}
\]

\[
\text{Pexpr} ::= \text{Nameexpr Pspec | Name actualized by Nameexpr [using repl]}
\]

\[
\text{Nameexpr} ::= \text{Name | Nameexpr, Name}
\]

\[
\text{Pspec} ::= \begin{array}{c}
\text{[formal sorts Sortlist]} \quad \text{[formal egns Opnlist]} \\
\text{[formal egns Eqlist]}
\end{array}
\]

\[
\text{[sorts Sortlist]} \quad \text{[egns Opnlist]} \quad \text{[egns Eqlist]}
\]

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, plusegns, plusopns, minusegns, and minusopns respectively, are given in [Thomas 89]. In addition to these operations, we shall refer to the (bodies) of specifications with the operation

\[
\text{body : Name -> Pexpr,}
\]

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:

\[
\text{eval : (Type x Environment) -> (Pexpr x Environment)}
\]

where \( x \) is the usual product type constructor and \( -\rightarrow \) 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:

\[
\text{Environment = Typereference -> Type}
\]

and the associated projection functions are

\[
\text{proj1 : (Pexpr x Environment) -> Pexpr}
\]

\[
\text{proj2 : (Pexpr x Environment) -> Environment.}
\]

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

\[
T ::= \text{SEQUENCE OF SEQUENCE OF INTEGER.}
\]
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 notation 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 \textsc{sequence of integer} is required. Assuming that we have a (parameterised) ACT-ONE type \textsc{sequence}, we would translate the declaration of \texttt{T} into

\[
\begin{align*}
T & \text{ is sequence actualized by Dummy} \\
    & \text{ using sortnames Seq for Data} \\
\text{where} \\
    & \text{ Dummy is sequence actualized by integer} \\
    & \text{ using sortnames Int for Data.}
\end{align*}
\]

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

the empty environment

\[
\begin{align*}
\text{new}: & \text{ Environment} \\
\end{align*}
\]

the update operation

\[
\begin{align*}
(\text{Environment} \times \text{Type} \times \text{Typereference}) \rightarrow \text{Environment}
\end{align*}
\]

the name lookup operation

\[
\begin{align*}
\text{name}: & \text{ ((Type + Typereference) x Environment) \rightarrow (Typereference + \{\text{undef}\})}
\end{align*}
\]

the type lookup operation

\[
\begin{align*}
\text{lookup}: & \text{ ((Type + Typereference) x Environment) \rightarrow (Type + \{\text{undef}\})}
\end{align*}
\]

an operation to generate a new name

\[
\begin{align*}
\text{newname}: & \text{ Environment \rightarrow Typereference}
\end{align*}
\]

and two predicates

\[
\begin{align*}
\text{isname}: & \text{ (Type + Typereference) \rightarrow Bool} \\
\text{istype}: & \text{ (Type + Typereference) \rightarrow Bool}
\end{align*}
\]

We note that \texttt{name} and \texttt{lookup} are the identity functions for \texttt{Typereference} and \texttt{Type} respectively. For example, for all \texttt{t:Type}, for all \texttt{p:Environment}, \texttt{lookup}(t,p) = 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 \texttt{Sol} which maps an ASN.1 type into an ACT-ONE sort.

\[
\begin{align*}
\text{Sol}: & \text{ ((Type + Typereference) x Environment) \rightarrow Sort} \\
\text{Sol \{BIT STRING\}p} & = \text{BString} \\
\text{Sol \{BIT STRING \{NamedBitList\}\}p} & = \text{BString} \\
\text{Sol \{INTEGER\}p} & = \text{DecString} \\
\text{Sol \{BOOLEAN\}p} & = \text{Boole} \\
\text{Sol \{OCTETSTRING\}p} & = \text{Octetstring} \\
\text{Sol \{SET OF Type\}p} & = \text{Set} \\
\text{Sol \{SEQUENCE OF Type \}p} & = \text{Seq} \\
\text{Sol \{SET \{ElementTypeList\}\}p} & = \text{Set} \\
\text{Sol \{SEQUENCE \{ElementTypeList\}\}p} & = \text{Seq} \\
\text{Sol \{CHOICE \{ElementTypeList\}\}p} & = \text{Choice}
\end{align*}
\]
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
    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
endtype
```

Assuming that \( p \) is an environment, the ACT-ONE denotation of integers without identifiers is given by:

\[
Tval[\lfloor\text{INTEGER}\rfloor\lfloor p = body(\text{INTEGER}) \times p.
\]

Assuming that \( lval \) is a function used in the definition of \( Eval \) and maps ASN.1 integer values into ACT-ONE integers values, the denotation of integers with identifiers is given by:

\[
Tval[\lfloor\text{INTEGER NamedNumberList}\rfloor\lfloor p =
  \text{proj}_1((Tval[\lfloor\text{INTEGER}\rfloor\lfloor p)
    \text{plusopns addid}(\text{NamedNumberList})
    \text{plusopns adeq}(\text{NamedNumberList})
  \times p
\]

where

\[
\text{adeq} : \text{List}(\text{Id x SignedNumber}) \rightarrow \text{Eqns}
\]

\[
\text{addid} : \text{List}(\text{Id x SignedNumber}) \rightarrow \text{Opns}
\]

\[
\text{addid}(\text{id}(n):(\text{id}'(n):!)) = \text{id:DecString}, \text{addid}(\text{id}'(n):!) = \text{id:DecString}
\]

\[
\text{adeq}(\text{id}(n):(\text{id}'(n):!)) = (\text{ofsort DecString}, \text{id = lval(n)}), \text{adeq}(\text{id}'(n):!) = (\text{ofsort DecString}, \text{id = lval(n)}).
\]

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

\[
\text{DAYSOFTHEMONTH ::= INTEGER \{first(1),last(31)}
\]
Given an appropriate environment $\rho$, we translate this declaration into:

\begin{verbatim}

type DAYSOFTHEMONTH is Trval[[' INTEGER {first(1),last(31)} ||$\rho$.

proj1(Trval[] INTEGER {first(1),last(31)} ||$\rho$)

= 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
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
  formal sorts Data
  sorts Seq
  opns
    { } : Seq
    _+_ : Data, Seq -> Seq
    tag : Seq -> Tag
  end

eops
  forall s:Seq
  ofsort Tag
    tag(s) = <universal, 16>
endtype
```

\[ Travl[[SEQUENCE OF Type]]p = \]
\[ if (name[[Type]]p) <> undef \]
\[ then \]
\[ (SEQUENCE actualized by name[[Type]]p using sort[[Type]]p for Data) \times p \]
\[ else \]
\[ (SEQUENCE actualized by newname(p) using sort[[Type]]p for Data) \times p[newname(p)/Type] \]
\[ As an example, consider the ASN.1 type declaration \]

\[ \text{HOLIDAYS ::= SEQUENCE OF DAYSOFTHEMONTH} \]

where DAYSOFTHEMONTH is defined in the previous section. Given an appropriate environment \( p, isname[[DAYSOFTHEMONTH]] \) is true and so the first branch of the conditional applies. Thus,

\[ Travl[[SEQUENCE OF DAYSOFTHEMONTH]]p = \]
\[ (SEQUENCE actualized by DAYSOFTHEMONTH using Decstring for Data) \times p. \]

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

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 \( \text{SEQUENCE} \{ \text{ElementTypeList} \} \) type.

\[
\text{IVAL} [\text{SEQUENCE} \{ \text{ElementTypeList} \}] p = \\
\text{mapname}(\text{flatten}(\text{ElementTypeList}) p, \text{TAGS}) \\
\text{SORTS} \text{Seq} \\
\text{OPS} \{ \} : \text{Seq} \\
\;
\text{plusopns}(\text{securities}(\text{flatten}(\text{ElementTypeList}) p, \text{Seq}) p) \\
\text{CONS} \\
\text{forall} \ s : \text{Seq} \\
\text{OFSORT} \text{Tag} \\
\text{tag}(s) = \langle \text{universal, 16} \rangle \\
\text{pluseqns}(\text{defaults}(\text{flatten}(\text{ElementTypeList}) p, \text{Seq}) p) \\
\text{pluseqns}(\text{selectors}(\text{flatten}(\text{ElementTypeList}) p, \text{Seq}) p) \\
\text{endtype} x p
\]

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

- \( \text{mapname} \) converts a list of ASN.1 types and type references into a list of ACT-ONE type references.
- \( \text{mapsort} \) converts a list of ASN.1 types and type references into a list of ACT-ONE sorts and identifier names.
- \( \text{flatten} \) expands component and selection types in type list.
- \( \text{expandcomp} \) expands components in type list.
- \( \text{expandsel} \) expands selection types in type list.
- \( \text{findid} \) finds a field name in a list of types.
- \( \text{securities} \) 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.
- \( \text{selectors} \) defines equations for selectors.
- \( \text{defaults} \) 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 \( e \) as the "absent" identifier, or field name, by:

\[
\text{mapname}(\langle Ntype, X : t: \text{elist} \rangle p) = \\
\begin{align*}
\text{name}(Ntype', p) : \text{mapname}(t': \text{elist}) & \quad \text{if } Ntype = \langle \text{id}, Ntype' \rangle \\
\text{name}(Ntype, p) : \text{mapname}(t': \text{elist}) & \quad \text{otherwise}
\end{align*}
\]
\[ \text{mapname}(\text{<Ntype}, X \rightarrow t[])) \]
\[ = \text{name}(\text{<Ntype}, \rho) \text{ if Ntype} = \text{<id, Ntype}> \]
\[ = \text{name}(\text{Ntype}, \rho) \text{ otherwise} \]

\[ \text{mapsort}(t::\text{elist}) \]
\[ = \text{<S01(lookup([T])\rho), id>: mapsort}(t::\text{elist}) \text{ if } t = \text{<id, T>, X}> \]
\[ = \text{<S01(lookup([T])\rho, e>: mapsort}(t::\text{elist}) \text{ otherwise} \]

\[ \text{mapsort}(t[]) \]
\[ = \text{<S01(lookup([T])\rho, id> if } t = \text{<id, T>, X}> \]
\[ = \text{<S01(lookup([T])\rho, e> otherwise} \]

\[ \text{flatten}(t::\text{eList}) \]
\[ = \text{expandsel}(\text{expandcomp}(t::\text{elist})\rho) \]

\[ \text{expandcomp}(t::\text{elist}) \]
\[ = \text{expandcomp}(\text{elist'}\rho) +\text{ expandcomp}(t::\text{elist})\rho \text{ if } t = \text{COMPONENTS OF T } \]
\[ = t::\text{expandcomp}(t::\text{elist})\rho \text{ otherwise} \]

where \text{lookup([T])\rho} = \text{SEQUENCE} \{\text{elist'}\}

\[ \text{expandcomp}(t[]) \]
\[ = \text{expandcomp}(\text{elist'}\rho) \text{ if } t = \text{COMPONENTS OF T } \]
\[ = t \text{ otherwise} \]

where \text{lookup([T])\rho} = \text{SEQUENCE} \{\text{elist'}\}

\[ \text{expandsel}(t::\text{elist}) \]
\[ = \text{<id, expand(id, Ntyp)\rho>: expandsel}(t::\text{elist})\rho \text{ if } t = \text{id < Ntyp, X}> \]
\[ = t::\text{expandsel}(t::\text{elist})\rho \text{ otherwise} \]

\[ \text{expandsel}(t[]) \]
\[ = \text{<id, expand(id, Ntyp)\rho>: if } t = \text{id < Ntyp, X}> \]
\[ = t \text{ otherwise} \]

\[ \text{expand}(\text{id, Ntyp}) \]
\[ = \text{findid}(\text{id, expandsel(\text{elist})\rho}) \]
\[ = \text{<id, T> if } t = \text{id < Ntyp, X}> \]
\[ = \text{findid}(\text{id, t::\text{elist}) otherwise} \]

\[ \text{findid}(t::\text{elist}) \]
\[ = \text{<id, T> if } t = \text{<id, T>, X}> \]
\[ = \text{findid}(\text{id, t::\text{elist}) otherwise} \]
\[ \text{findid}(\text{id}, t; []) \]
\[ = \begin{cases} \text{id}, T & \text{if } t = \langle \text{id}, T, X \rangle \\ \text{undefined} & \text{otherwise} \end{cases} \]

The remaining three operations are specified as follows:

**securities**\((S_1, ..., S_n, s)_p\)

defines a declaration of a set of operators containing:

let \(\text{mapsort}(S_1, ..., S_n)_p = \langle s_1, i_1 \rangle, ..., \langle s_n, i_n \rangle\), in

i) the \(n\)-ary operator \(\{ i_1, ..., i_2, ..., i_n \} : s_1, ..., s_n \rightarrow s\)

ii) \(n\) unary operators defined by:
(for all \(j: 1 \leq j \leq n\) if \(i_j = \varepsilon\) then \(s \rightarrow s_j\) else \(i_j : s \rightarrow s_j\))

iii) the \(m\)-ary operators which allow optional and default types to be absent:
(for all sequences \(\langle t_1, i_1 \rangle, ..., \langle t_m, i_m \rangle\), s.t.
\(m < n\) and \(\langle t_1, i_1 \rangle, ..., \langle t_m, i_m \rangle\) is a proper subsequence of \(\langle s_1, i_1 \rangle, ..., \langle s_n, i_n \rangle\)
(for all \(i, 1 \leq i \leq n\) s.t. \(S_i \in \{ t_1, ..., t_m \} \Rightarrow (S_i = \langle T, O \rangle \text{ or } S_i = \langle T, D \rangle)\)
the \(m\)-ary operator \(\{ j_1, ..., j_m \} : t_1, ..., t_m \rightarrow s\)\)

**selectors**\((S_1, ..., S_n, s)_p\)

defines a set of equations containing:

let \(\text{mapsort}(S_1, ..., S_n)_p = \langle s_1, i_1 \rangle, ..., \langle s_n, i_n \rangle\), in

(for all \(j: 1 \leq j \leq n\)
if \(i_j = \varepsilon\) then
\begin{align*}
&\text{forall } x_1 : s_1, ..., x_n : s_n \\
&\text{ofsort } s_j \\
&\quad j(x_1, ..., x_n) = x_j
\end{align*}
else
\begin{align*}
&\text{forall } x_1 : s_1, ..., x_n : s_n \\
&\text{ofsort } s_j \\
&\quad i_j(x_1, ..., x_n) = x_j
\end{align*}

**defaults**\((s_1, ..., s_n, s)_p\)

defines a set of equations containing:

let \(\text{mapsort}(S_1, ..., S_n)_p = \langle s_1, i_1 \rangle, ..., \langle s_n, i_n \rangle\) in

(for all \(j \in \{ 1, ..., n \}\)
\begin{align*}
&\text{forall } x_1 : s_1, ..., x_j : s_j, ..., x_n : s_n \\
&\text{ofsort } s \\
&\quad \{ 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 \} \}
\end{align*}
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 $p$, the first component of the ACT-ONE denotation of this type is given by:

```
profile Tval [SEQUENCE {<index, <INTEGER,N>>, <hash, <BIT STRING,D,0000>>>} ||p] =
mapname(flatten(<index, <INTEGER,N>>, <hash, <BIT STRING,D,0000>>))p, TAGS
sorts Seq
ops {} : Seq
plusops
(separities(mapsort(flatten
(<index, <INTEGER,N>>, <hash, <BIT STRING,D,0000>>))p, Seq)p)p
eqns
toall.s:Seq
tofsort Tag
tag(s) = <universal, 16>
pluseqns(defaults(mapsort(flatten
(<index, <INTEGER,N>>, <hash, <BIT STRING,D,0000>>))p, Seq)p)p
pluseqns(selectors(mapsort(flatten
(<index, <INTEGER,N>>, <hash, <BIT STRING,D,0000>>))p, Seq)p)p
end type
```

```
INTEGER, BITSTRING, TAGS
sorts Seq
ops {} : Seq
plusops(separities(<DecString,index>,<BString,hash>,Seq)p)
eqns
toall.s:Seq
tofsort Tag
tag(s) = <universal, 16>
pluseqns(defaults(<DecString,index>,<BString,hash>,Seq)p)
pluseqns(selectors(<DecString,index>,<BString,hash>,Seq)p)
endtype
```

```
INTEGER, BITSTRING, TAGS
sorts Seq
ops {}
{<index, hash>} : Seq
{<index, index>} : DecString, BString -> Seq
{<index>} : DecString -> Seq
{<hash>} : Seq -> Decstring
hash : Seq -> BString
```
eons
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:

\[ \text{Ex2} ::= \text{SEQUENCE} \{ \text{INTEGER, COMPONENTS OF Ex1} \}. \]

Assuming that \( p \) is suitable environment containing the declaration of \( \text{Ex1} \) as given above, we have:

\[
\text{proj1(Tval } [\text{SEQUENCE} \{ \text{INTEGER, COMPONENTS OF Ex1} \}] \p p =
\text{mapname(flatten(INTEGER,COMPONENTS OF Ex1)p}, \text{TAGS}
\text{sorts } \quad \text{Seq}
\text{opns } \quad \{ \} : \text{Seq}
\text{plusopns(sequities(mapsort)
  \quad (flatten(INTEGER,COMPONENTS OF Ex1)p,Seq)p)p)
}\]
\[
\text{eons}
forall s:Seq
ofsort Tag
  tag(s) = <universal, 16>
\text{plusseqns(defaults(mapsort(extendset}
  \quad (extendcomp(INTEGER,COMPONENTS OF Ex1)p,Seq)p)p)p)
endtype
=
\text{mapname}
(<\text{<INTEGER>,<index,}<\text{INTEGER},\text{N}>,<\text{hash,}<\text{BITSTRING,D,0000}>>})p, \text{TAGS}
\text{sorts } \quad \text{Seq}
\text{opns } \quad \{ \} : \text{Seq}
plusonps
  (seqarities(mapsort
  (< e, INTEGER>,<index,<INTEGER,N>>,<hash,<BITSTRING,D,0000>>)p,Seq)p)
opns
forall s:Seq
ofsort Tag
  tag(s) = <universal, 16>
pluseonps(defaults(mapsort
  (< e, INTEGER>,<index,<INTEGER,N>>,<hash,<BITSTRING,D,0000>>)p,Seq)p)
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
opns
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]

[Thomas 89]

[Ehrig Mahr 85]

[ISO 8824]

[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.