A Translator for ASN.1 into LOTOS

Muffy Thomas[ab]

[a] Dept. of Computing Science, University of Glasgow, Glasgow, Scotland.
[b] Under Sponsorship from British Telecom Group Technology and Development and the Esprit LOTOSphere project .

Abstract

A translator from ASN.1 to LOTOS is described: an ASN.1 module is translated into a LOTOS specification which consists of a collection of ACT ONE data types and the single constant process `stop`. The translator provides other functions and has been formally specified. The results have been checked using the *topo* LOTOS compiler. The translator may be used as part of a combined specification technique: ASN.1 for the data type descriptions and basic LOTOS for process descriptions.

Keyword Codes: D.2.1, D.2.2
Keywords: Software Engineering, Requirements/Specifications; Tools and Techniques.

0. Introduction

ASN.1 (Abstract Notation One) [ISO 8824] is a notation for defining the data types used in OSI application layer protocols. The syntax of the language is specified as an International Standard, but the semantics is only informally specified using natural language.

LOTOS (Language of Temporal Ordering Specification) [ISO 8807] is a notation for defining processes *and* data types; although it was designed with OSI application layer standards in mind, it is also a general purpose specification language for distributed and concurrent systems. The data type sublanguage of LOTOS is the algebraic specification language ACT ONE, the remainder of the language is called basic LOTOS. Both the syntax and the semantics of the language are formally specified as an International Standard.

In [Thomas 89] a first, formal, denotational semantics for ASN.1 was presented. As a result of feedback from the ASN.1 community, the semantics has evolved quite significantly [Thomas 89, Thomas 90, Thomas 91r, Thomas 92]. This paper describes a translator tool from ASN.1 to LOTOS, based on the formal semantics in [Thomas 92]. The results of translator have been checked using the *topo* LOTOS compiler. Essentially, an ASN.1 module is translated into a LOTOS specification which consists of a collection of ACT ONE data types and the single constant process `stop`. The translator provides other functions as well.

The main aim of the translation is to enable the integration of protocols written in both notations and the integration of tools and associated encodings. A translation between ASN.1 and ACT ONE allows a protocol specifier to use a mixture of notations: ASN.1 for the data type descriptions and basic LOTOS for process descriptions. In particular, the translator may be used by a protocol specifier who later instantiates the trivial (`stop` process by a more appropriate behaviour. Moreover, by defining a relationship between the two notations, we are better able to discuss the relative merits of each language and to suggest improvements.

An overall objective has been to give a *framework* for the translation from ASN.1 to ACT ONE. Since the standard description of ASN.1 semantics has been informal, we expected that several aspects of our formal semantics would have to be changed because a different interpretation of the standard was intended. Thus, we were more concerned with a good framework in which to define and experiment with the semantics, rather than the exact definitions, in some cases.

In this paper we do not discuss the semantics in detail, but we concentrate on the practical implications of the semantics. A prototype translator, based on the denotational semantics, has been implemented in Miranda (Miranda is a trademark of Research Software Ltd.); see [Turner 86] for a discussion of the language. The translator has been evolving since 1989 when the first formal semantics was defined. Functional language prototypes have been essential to the evolution: each successive semantics has been easily implemented and demonstrated to the ASN.1 community for feedback.

The structure of the paper is as follows. Section 1 contains an overview and comparison of the ASN.1 and LOTOS languages. In Section 2 we discuss related work and Section 3 contains an overview of the design decisions involved in making the semantics. An overview of the semantics is given in Section 4 and in Section 5 we describe the implementation of the prototype translator and give some examples of its use. Section 6 contains an example translation and in Section 7 some further features of ASN.1 are discussed. In Section 8 we review the evolution of the translation and discuss the results; in the final section we present our conclusions and directions for future work.

1. ASN.1 and LOTOS

Before presenting an overview of the syntax of ASN.1 and ACT ONE, the important differences and similarities between ASN.1 and LOTOS are highlighted below.

ASN.1	LOTOS
• formal syntax	• formal syntax
• informal semantics	• formal semantics
• data types	• data types
• no processes	• processes
• ISO/CCITT standard	• ISO standard
• encoding rules	• no encoding rules
• concise, brief notation	• verbose notation for data types
• used extensively, for some time	• newer technology
• modules	• no modules

1.1 ASN.1

ASN.1 is a language for defining types and values. Here, we concentrate on the basic type definitions; other aspects of types such as subtypes, tables, etc. are discussed later.

We assume some familiarity with ASN.1, the reader is referred to [ISO 8824] for a complete description of the language. ASN.1 types are either primitive or compound. The former include the usual basic types such as integers, bit strings etc.; the latter include homogeneous sequences and sets, heterogeneous sequences and sets, choice types, etc. The notation for the compound types is very concise and allows for possibilities such as optional and default component types. An example ASN.1 module is given below.

```
example1 DEFINITIONS ::=
BEGIN
     T1 ::=  SET OF T3
     T2 ::=  SEQUENCE { one INTEGER,two BOOLEAN OPTIONAL,
                      three INTEGER DEFAULT 3}
     T3 ::=  INTEGER  {zero(0), one(1)}
     T4 ::=  SET OF ENUMERATED {one(1), two(2), three(3)}
     T5 ::=  SEQUENCE {four INTEGER, COMPONENTS OF T2}
     T6 ::=  CHOICE {on BOOLEAN, num INTEGER}
     T7 ::=  CHOICE {num INTEGER, on BOOLEAN}
END
```

1.2 ACT ONE

ACT ONE is an algebraic specification language for specifying abstract data types. The philosophy underpinning the language is that data types are algebras: sets *and* operations, and that an abstract data type is a representation independent specification of (possibly) many data types. Specifications are equational and the semantics of specifications is initial algebra. We assume some familiarity with ACT ONE and algebraic specification, the reader is referred to [ISO 8807] and [Ehrig Mahr 85] for full descriptions of the language and concepts. We use three of the ACT ONE structuring concepts: parameterisation, actualisation, and combination. We do not describe the language constructs here; instead, we illustrate them with some examples. The basic form of a specification body is a "flat" specification. A "flat" specification consists of three parts: the sorts, the operation names and arities, and the equations. For example, a specification of simple truth values is:

```
type    Simple_Truth_Values is
sorts   bool
opns    true : -> bool
        false : -> bool
        not : bool -> bool
eqns
forall  b:bool
ofsort  bool
        not(not(b)) = b;
        not(true) = false;
        not(false) = true;
endtype
```

Specifications can be reused in another specification. For example, a specification of simple numbers can use the specification of simple truth values as follows:

```
type    Simple_Numbers is Simple_Truth_Values
sorts   num
opns    0 : -> num
        succ : num -> num
        iszero : num -> bool
eqns
forall  n:num
ofsort  bool
        iszero(0) = true;
        iszero(succ(n)) = false;
endtype
```

Specifications can be generic, or parameterised. The parameter requirements are specified as formal sorts, operations and equations. A parameterised specification can be instantiated, or actualised by another specification which meets the formal parameter requirements. This means that there must be an appropriate correspondence between formal and actual sorts and operations such that the operation arities match and the (formal) equational theory is included in the actual equational theory (after renaming).

2. Related Work

The early work of [Bochmann Deslauriers 89] on translating ASN.1 into ACT ONE differs from our work in several ways. Their approach does not consider optional or default types, nor does it distinguish between heterogeneous sequences and sets. Our treatment of heterogeneous sequences and sets is much more complex; reflecting, we believe, the very powerful nature of the notation. Moreover, they translate all the primitive types directly into standard library types, losing some of the ASN.1 features such as conversions between hexadecimal and bit strings and the ability to introduce local identifiers. Although we also translate some primitive types into standard library types, our approach still allows for the introduction of local identifiers.

The work of Segala [Segala 89a, Segala 89b] defines a translation from ASN.1 to ACT ONE by enhancing ACT ONE with a set of macro constructors. The macro constructors correspond very closely to the ASN.1 type constructors, thus the actual translation is less complex than the one contained herein. Since our aim was to produce standard ACT ONE, we did not consider such a "two-level" approach.

The paper by [Burmeister et al 90] outlines an approach which is similar to ours. The essential difference is the degree of formality: the results are presented much less rigorously in order to make the approach "user friendly". The LOTOS library types are used extensively for the primitive types, but our approach agrees in principle on most compound data types; excepting tagging, and homogeneous sets and sequences where the differences are more fundamental. The author is not aware of an implementation.

The One2One translator [Brady et al 90] also translates ASN.1 into ACT ONE. The approach taken differs from ours, and the others mentioned above, in that the entire ASN.1 language has been modelled, rather than specific modules. Thus, in their approach, the translation of an ASN.1 type T is not an ACT ONE type T, but an operation of a (pre-defined) sort ASN1_TYPE.

An important feature of our translation is that it was formally specified before implementation. None of the related work, to our knowledge, is based on a formal specification of the translation.

3. Design Decisions

In this section we outline some of the important features of the semantics and the design decisions taken. A full discussion of the design decisions, and the evolution of the design, is given in [Thomas Rudkin MacLeod 90, Thomas 91r, Thomas 92]. Our basic design principle was to put as much as possible into the model. Therefore tags, classes, and identifiers are defined within the ACT ONE 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.

It is important to note that the ASN.1 language standard only defines the various types of *data,* but not the associated *operations* on the data. Therefore, in most cases, we have inferred the associated operations when translating into ACT ONE.

3.1 Primitive Types

The primitive types, or at least parts thereof, are taken from the LOTOS standard library (Annex A) wherever possible. These library types do not always exactly reflect the type descriptions in the ASN.1 standard, but the LOTOS library types are well known and so we have tried to compromise between staying faithful to the type descriptions and the convenience of using the standard library. For example, we use the standard library specification of natural numbers in our specification of the integers.

3.2 Classes and Tags

We have defined ACT ONE types `TAGS` and `CLASS`, corresponding to ASN.1 tags and classes, and they are included in the translation of every (ASN.1) type. A `Tag`-sorted operation named `tag` is also included in the translation of each ASN.1 type and the appropriate equation is given for that type. Since (re-)tagging may be implicit or explicit, we may also prefix tags.

3.3 Type Equivalence

ASN.1 types and values may be renamed within in a module; we consider renamed types, and types with renamed values, to be equivalent. For example a module may contain the following declarations:

```
T1  ::=  INTEGER
T2  ::=  INTEGER
T3  ::=  INTEGER {largest(42)}
T4  ::=  SEQUENCE {on BOOLEAN, num T1}
T5  ::=  SEQUENCE {on BOOLEAN, num T2}
T6  ::=  SEQUENCE {on BOOLEAN, int T2}
```

We consider `T1, T2`, and `T3` to be equivalent (the renaming of `42` introduced by `T3` affects both `T1` and `T2`). The equivalence is extended to a congruence (equivalent types may be substituted in any context). For example, `T4` and `T5` are congruent because `T1` and `T2` are equivalent, but `T4` and `T6` are not congruent because the component names (i.e. `num` and `int`) differ.

Equivalent (congruent) ASN.1 types denote identical ACT ONE types. In ACT ONE the use of different names for identical types is achieved by renaming with the `is` construct. For example, if `T` and `T'` are names for identical types, then `T'` can be defined by `T' is T`. When translating an ASN.1 module containing several equivalent types, one of those types is translated to an ACT ONE type; all others just enumerate different names for that type.

The formal rules for type equivalence are given in [Thomas 91r]. The type equivalence used here is not the only possibility; a discussion of the requirements for and the specification of ASN.1 type equivalences is contained in [Thomas Rudkin MacLeod 91].

3.4 Type Names and Sort Names

The names of types and sorts within a translated module are unique. Every named ASN.1 type with name `T` is translated into an ACT ONE type with sort of interest `T_sort`. The operation names within a translated module may be overloaded but only if every ground term within the LOTOS specification is unique. For example, we may have a type T1 with an operation `f:T1_sort -> T1_sort` and a type T2 with an operation `f:T2_sort -> T2_sort`. Then, every term of form `f(x)` has a unique sort: if `x` has sort `T1_sort` then `f(x)` has sort `T1_sort`; if `x` has sort `T2_sort` then `f(x)` has sort `T2_sort`. However, we may not have two constants `f:-> T1_sort` and `f:-> T2_sort`, as then the sort of `f` would be ambiguous.

3.5 Scope of Identifiers

Two ASN.1 types, `INTEGER` and `BIT STRING`, may include the use of identifiers, but there is no explicit mention of the scope of these identifiers in the original standard for the language. 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 translation of the type, the identifiers are defined as (constant) operators of the relevant sort and their bindings are given by the equations. This is illustrated in the example in §6. The issues of scope are discussed further in [Thomas Rudkin MacLeod 90] and [Thomas Rudkin MacLeod 91].

3.6 Omissions

With respect to the standard [ISO 8824], we have omitted the ANY type and External types. Some further features of ASN.1, such as tables, functions and subtypes, are discussed in §7.

4. Overview of the Semantics

A *denotational* semantics is essentially a mapping from a source language (the *syntax*) to a target language (the *semantics*). The main features of a denotational semantics for ASN.1 are the syntactic domains, (in this case, essentially the syntactic categories of ASN.1) the semantic domains, (in this case, essentially the syntactic categories of ACT ONE and lists of ACT ONE specifications) and the semantic evaluation functions. The functions are defined such that the evaluation of a compound ASN.1 construct depends on the evaluation of the components.

4.1 New Type Names

The denotation, or translation, of an ASN.1 data type is essentially an ACT ONE specification of the same name, but it may include other ACT ONE data types. The new ACT ONE type names may be introduced in two ways.

First, ASN.1 allows the use of unnamed component types whereas ACT ONE does not. For example, when translating the ASN.1 type

```
T ::= SET OF SEQUENCE OF INTEGER
```

a name for the intermediate type `SEQUENCE OF INTEGER` is required in ACT ONE. Assuming that we have (parameterised) ACT ONE types named `SEQUENCE` (with formal sort `Data` and sort of interest `Seq`) and `SET` (with formal sort `Data` and sort of interest `Set`), we would translate the type `T` into

```
type T is SET actualizedby Sequence_1
using sortnames  T_sort for Set,
                 Sequence_1_sort for Data endtype
```

where the intermediate type `SEQUENCE_1` is defined by :

```
type Sequence_1 is SEQUENCE actualizedby ASN1INTEGER
using sortnames  Sequence_1_sort for Seq,
                 IntString for Data endtype
```

and the type `ASN1INTEGER` is the translation of `INTEGER`.

Second, new type names may be introduced when types are (re-)tagged. Since the only difference between a tagged type and an untagged type is one equation, we give the common specification a new name and include this specification in the other two specifications, inserting the additional equations as appropriate.

In both cases, the new names depend on the type constructor and the type concerned. For example, sequence types have the names `Sequence_0`, `Sequence_1`, ... , integer types have the names `Integer_0`, `Integer_1`, ... , choice types have the names `Choice_0`, `Choice_1`, ... , etc. The names for tagged types include the tag number and class.

4.2 Environments

Because both source and target languages have *named* types, *environments* are necessary to allow for the referencing and dereferencing of types. There are two types of environment: source environments and target environments. They are defined as mappings between ASN.1 type names (the ASN.1 category Typereference) and ASN.1 types (the ASN.1 category Type), and queues of ACT ONE type names (the ACT ONE category Name)and ACT ONE types (the ACT ONE category Pexpr), respectively. The environments are defined by:

Envs = Typereference $\rightarrow$ Type and Envt = Queue(Name) $\rightarrow$ Pexpr

where $\rightarrow$ is the function space constructor.

4.3 The Main Semantic Evaluation Functions

The ASN.1 language allows for the description of data at the level of *values*, *types* and *modules* (collections of types and values); thus five main semantic evaluation functions are defined: **Eval** (for value evaluations), **Vdecl** (for value declarations), **Tval** (for type evaluations), **Tdecl** (for type declarations) and **Mval** (for module evaluations).

The definitions of these functions are too complex to be given here; instead, to give a flavour of the semantics, we describe the types of these functions. The functions have higher order types, although the result types are tuples. This mixture of curried/uncurried types is purely a matter of personal taste and does not reflect any particular feature of the semantics.

Informally, the function **Mval** gives meaning to an ASN.1 module: it maps a list of ASN.1 type and value declarations into a list, or environment, (there are no module constructs in ACT ONE) of ACT ONE data types. The function **Tval** gives meanings to the primitive and compound ASN.1 types: it maps an ASN.1 type into an ACT ONE type. The function **Tdecl** maps an ASN.1 type declaration - a type reference and body - into an ACT ONE type declaration with the same type reference. For example, an ASN.1 type declaration of the form

```
    T ::= ASN.1_type
```

is mapped into the ACT ONE declaration

```
    type T is ACT_ONE_type
```

where `ACT_ONE_type` is the image of `ASN.1_type` under **Tval**. The function **Eval** maps an ASN.1 value into the appropriate ACT ONE value and **Vdecl** maps a value declaration into an ACT ONE type declaration.

More formally, the function **Eval** maps an ASN.1 value, given an ASN.1 type and target environment, into an appropriate ACT ONE value. The type of the function **Eval** is given by:

Eval : Value $\rightarrow$ Type $\rightarrow$ Envt $\rightarrow$ ACT ONE_value.

The result of evaluating an ASN.1 value declaration is a essentially an ACT ONE type declaration. For example, an ASN.1 value declaration of the form

```
    V  T ::= ASN.1_value
```

is mapped into the ACT ONE type declaration

```
    type T is ACT_ONE_type'
```

where `ACT_ONE_type'` is `ACT_ONE_type` with an additional constant `v` and equation `v = v'`, where `v'` is the evaluation of `ASN.1_value`. The given target environment is then updated with the appropriate enrichment to the type `T`. Thus, the function **Vdecl** maps a value declaration, given an ASN.1 type and source and target environments, into a target ACT ONE environment; the type of the function **Vdecl** is given by:

Vdecl: Valuereference $\rightarrow$ Value $\rightarrow$ Type $\rightarrow$ Envs $\rightarrow$ Envt $\rightarrow$ Envt.

The denotation of an ASN.1 type is essentially an ACT ONE specification. However, since new (ACT ONE) specifications may be introduced during the evaluation of a type, the target environment may be updated. Moreover, although the corresponding new ASN.1 names and types are not of interest as such, the source environment is updated because type equivalence is only defined on ASN.1 types. Therefore, the denotation is actually a triple. The type of the function **Tval** is given by:

Tval: Type $\rightarrow$ Envs $\rightarrow$ Envt $\rightarrow$ (Pexpr, Envs, Envt).

The result of evaluating an ASN.1 type declaration is essentially a target ACT ONE environment. For example, when a declaration has the form `Tname ::= Type`, then the resulting environment is the given target environment updated with the binding of `Tname` to the evaluation of `Type`. `Type` may or may not already exist in the given target environment. Since the evaluation of `Type` may involve the evaluation and naming of its component types, the source environment may also be updated with any new types. Therefore, the denotation is actually a pair of environments. The type of the function **Tdecl** is given by:

Tdecl: Typereference $\rightarrow$ Type $\rightarrow$ Envs $\rightarrow$ Envt $\rightarrow$ (Envs,Envt).

The denotation of an ASN.1 module, given a given source and target environment, is a (target) ACT ONE environment. The type of the function **Mval** is given by:

Mval: AssignmentList $\rightarrow$ Envs $\rightarrow$ Envt $\rightarrow$ Envt.

5. Overview of the Translator

The prototype translator, based on the denotational semantics, is implemented in Miranda (Miranda is a trademark of Research Software Ltd.). Miranda was chosen because it is a polymorphic, lazy, functional programming language with user defined algebraic types. The syntactic and semantic domains are implemented as Miranda types and each semantic function is a Miranda function of the appropriate type.

As an example, consider the implementation of the function **Mval**, as described above. **Mval** is defined by two cases: type declarations and value declarations. Consider the case of type declarations. Each type declaration is evaluated with a given source and target environment; both the source environment and target environment are updated with the appropriate names and types as the declarations are processed, plus any additional names and types required when evaluating a compound type. We use the notation env[`name/type`] to denote the function env with the additional mapping of `name` to `type`, (assuming that `name` is not already bound in env). The ASN.1 syntax appears in `Courier` font, whereas the semantics is given in Geneva font. We use the notation `x:tl` for an ASN.1 list of `Typeassignment` with `x` as the head of the list. The semantic evaluation function, for this case, is given by:

Mval ([|`(Tname ::= T):tl`|]) envs envt = **Mval** `tl` envs'[`Tname/T`] envt'
    where (envs',envt')= **Tdecl** [| `Tname ::= T` |] envs envt.

In Miranda, this function is implemented by the code:

```
typeassignment ::= TA typereference asn1type
sourceenv == [typeassignment]
targetenv == [([typereference],pexpr)]
mval :: ([typeassignment],targetenv, sourceenv) -> targetenv
mval(hd:tl,envt,envs) = mval(tl,envt',envs'++[hd])
        where (envs',envt')= tdecl (hd,envs,envt)
```

The type `typeassignment` is a (user) defined composite type with type constructor `TA` and component types `typereference` and `asn1type` (these are implementations of the respective ASN.1 syntactic types and are not given here). The type `sourceenv` is the implementation of Envs and is a synonym for the type of lists of `typeassignment`; the type `targetenv` is the implementation of Envt and is a synonym for the type of lists of pairs of lists of `typereference` and `pexpr` (not given here). We implement the environment functions by using the implementation of the ASN.1 syntactic type `typeassignment` for the source environment and lists of pairs for the target environment. In the implementation of target environment, we use lists to implement queues. The type and definition of the function `mval` should be obvious, and it is easy to verify this implementation against the denotational semantics.

5.1 Using the Translator

The translator has been extensively tested and the results processed by the *topo* LOTOS compiler: one of LITE tools (from the LOTOSphere Esprit project).

An ASN.1 module is given as an ASCII string, according to the syntax defined in [ISO 8824]. Type names must begin with a character in the range 'A' ... 'Z'; all other identifiers must begin with a character in the range 'a' .. 'z'. A module may contain comments and tabulation characters; the order in which types are defined is irrelevant.

The translator consists of a collection of Miranda functions. There are five functions of interest to the user: `translate`, `sortofinterest`, `names`, `translatevalue`, and `equivalent`. In all cases the functions require a file containing an ASN.1 module; the types (all higher order) and informal specifications of the functions, as comments, are given below.

```
type  NULL is TAGS
sorts Null
opns
      null:  -> Null
      tag: Null -> Tag
eqns
forall x:Null
ofsort Tag
      tag(x) = mktag(UNIVERSAL,5);
endtype
behaviour stop
endspec
```

Function: `sortofinterest`

```
Miranda sortofinterest "T3" "example2"
Null
Miranda sortofinterest "T4" "example2"
Bool
```

Function: `names`

```
Miranda names "T3" "example2"
NULL
T1
T2
T3
```

Function: `equivalent`

```
Miranda equivalent "T1" "T2" "example2"
True
Miranda equivalent "T1" "T4" "example2"
False
```

Function: `translatevalue`

```
Miranda translatevalue "NULL" "T1" "example2"
null
Miranda translatevalue "TRUE" "T4" "example2"
true
```

6. An Example Translation

This section contains the translation of another example ASN.1 module. It is not exhaustive; indeed, many features of ASN.1 are not used, for brevity. We are able, though, to illustrate the three main ways in which an ASN.1 type is translated: either as an actualisation of a parameterised type (c.f. `T1`), or as a more complex, combined type constructed "on the fly" (c.f. `T2`), or as a primitive type with some modifications (c.f. `T3`).

```
example3 DEFINITIONS ::=
BEGIN
    T1 ::=   SET OF T3
    T2 ::=   SEQUENCE { one INTEGER, two BOOLEAN OPTIONAL,
                        three INTEGER DEFAULT 3}
    T3 ::=   INTEGER  {zero(0), one(1)}
END
```

The ACT ONE has been generated using the translator but the script has been edited and some types removed for brevity. Comments have been added to the script: they appear, in a this font, within (* and *).

```
(* The LOTOS translation of the ASN.1 file example3 *)
specification Dummy:noexit
library
Boolean, NaturalNumber, NatRepresentations, HexNatRepr, HexString,
HexDigit, DecNatRepr, DecString, DecDigit, OctNatRepr, OctString,
OctDigit, BitNatRepr, BitString, Bit
endlib
type  TAGS is  ... endtype
type  ASN1BOOLEAN is ... endtype
type  BASICINTEGER is .. endtype
type  DIGIT is ... endtype
type  DIGITSTRING is ... endtype

(*two constants, named zero and one, are added to SIGNEDINTEGER, as a*)
(* consequence of the declaration in T3 *)
type  SIGNEDINTEGER is DIGITSTRING, BASICINTEGER
sorts Sign, IntString
opns
      zero:  -> IntString
      one:  -> IntString
     ...
(* a SIGNEDINTEGER has the form: int(+,s) or int(-,s), where s is a *)
(* DIGITSTRING *)
endtype

type  ASN1INTEGER is SIGNEDINTEGER, TAGS
opns
      tag: IntString -> Tag
eqns
forall x:IntString
ofsort Tag
      tag(x) = mktag(UNIVERSAL,2);
endtype

(* DATA is the formal parameter specification for sets*)
type DATA is
formalsorts Data
endtype

(* ASN1SET is the parameterised specification for sets*)
type  ASN1SET is Data, TAGS, Boolean
sorts Set
opns
      empty:  -> Set
      mksetof: Data, Set -> Set
      tag: Set -> Tag
eqns
forall s:Set,
       x, y:Data
```

```
ofsort Tag
      tag(s) = mktag(UNIVERSAL,17);
ofsort Set
      mksetof(x,mksetof(y,s)) = mksetof(y,mksetof(x,s));
endtype

(* Recall: T1 ::= SET OF T3 *)
type T1 is ASN1SET actualizedby T3 using
sortnames T1_sort for Set,  IntString for Data
opnnames  emptyT1 for empty
endtype
(* Recall: T2 ::= SEQUENCE {one INTEGER,two BOOLEAN OPTIONAL, *)
(*                         three INTEGER DEFAULT 3}*)
type  T2 is TAGS, ASN1BOOLEAN, ASN1INTEGER
sorts T2_sort
opns
(* The optional and default values may be absent: there are 3 ways to construct a *)
(* T2 sequence*)
      mk_T2: IntString, Bool, IntString -> T2_sort
      mk_T2: IntString, Bool -> T2_sort
      mk_T2: IntString, IntString -> T2_sort
      mk_T2: IntString -> T2_sort
(* The component names are selector operations*)
      one: T2_sort -> IntString
      two: T2_sort -> Bool
      three: T2_sort -> IntString

      tag: T2_sort -> Tag
eqns
forall s:T2_sort, x1:IntString,x2:Bool,x3:IntString
ofsort Tag
      tag(s) = mktag(UNIVERSAL,16);
ofsort Bool
      two(mk_T2(x1,x2,x3)) = x2;
ofsort IntString
      one(mk_T2(x1)) = x1;
      three(mk_T2(x1,x3)) = x3;
      one(mk_T2(x1,x3)) = x1;
      three(mk_T2(x1,x2,x3)) = x3;
      one(mk_T2(x1,x2,x3)) = x1;
ofsort T2_sort
(* If the integer  value is missing, then it is 3 by default*)
      mk_T2(x1,int(+,Dig(3))) = mk_T2(x1);
      mk_T2(x1,x2,int(+,Dig(3))) = mk_T2(x1,x2);
endtype

(* Recall: T3::= INTEGER  {zero(0), one(1)} *)
type  T3 is ASN1INTEGER endtype

behaviour stop
endspec
```

```
translate :: string -> string
|| translate f = the LOTOS translation of the ASN.1 module in file  f.

sortofinterest :: string -> string -> string
|| sortofinterest t f = the sort of interest for the translation
|| of type t, given the ASN.1 module in file f.

names :: string -> string -> string
|| names t f = the list of type names for the translation of type
|| t, given the ASN.1 module in file f.

translatevalue :: string -> string -> string -> string
|| translatevalue v t f = the ACT ONE translation of value v of
|| type t, given the ASN.1 module in file f.

equivalent :: string -> string -> string -> bool
|| equivalent t1 t2 f = the equivalence between the types t1 and
|| t2, given the ASN.1 module in file f.
```

These functions are illustrated by example below. In each case, assume that the file "example2" contains the following simple ASN.1 module:

```
example2 DEFINITIONS ::=
BEGIN
  T1 ::= NULL
  T2 ::= NULL
  T3 ::= T2
  T4 ::= BOOLEAN
END
```

In each example, the expression to be evaluated follows the Miranda prompt; the result of the evaluation begins on the following line. We note that the library specification `Boolean` is used to model truth values within ACT ONE types; the specification `ASN1BOOLEAN` is used as the denotation of the ASN.1 type `BOOLEAN`. We have removed most of the type `TAGS`, for brevity.

Function: `translate`

```
Miranda translate "example2"

(* The LOTOS translation of the ASN.1 file example2 *)
specification Dummy:noexit
library Boolean endlib
type  TAGS is ...
type  T1 is NULL endtype
type  T2 is NULL endtype
type  T3 is NULL endtype
type  T4 is ASN1BOOLEAN  endtype

type  ASN1BOOLEAN is Boolean, TAGS
opns
      tag: Bool -> Tag
eqns
forall b:Bool
ofsort Tag
      tag(b) = mktag(UNIVERSAL,1);
endtype
```

7. Further Features of ASN.1

The aim of this paper is primarily to describe the translator tool and so we only briefly discuss the features of ASN.1 which are not implemented. Several further features of ASN.1 have been considered and formalised: subtypes, tables, functions, identifier scope and type compatibility relations.

In [Thomas 91], subtypes are formalised as a set of inference rules describing the subtype relation. This formalisation highlights several inconsistencies and omissions in the natural language description. The typing system of ACT ONE does not allow a faithful translation of the ASN.1 subtype relation: although each subtype may be translated into an ACT ONE type, the relation between types cannot, in all cases, be preserved using the current ACT ONE structuring operations. A natural solution to this problem is to extend ACT ONE from many-sorted to order-sorted algebra (we note that such an extension might also be desirable for other reasons, for example, for error handling).

The table type is an enhancement which replaces macros as a construct for defining new, arbitrary types in ASN.1. In [Thomas 90t], we consider a semantics for tables. Since table types and table values allow for the declarations, in effect, of higher order types, they cannot have ACT ONE denotations. However, we do give some formal rules for evaluating table cells (given a set of table types and values) as (basic) ASN.1 values and types; the result is encodable values and types, in most cases.

In [Thomas 90f], we consider a semantics for the function enhancement. Since function (definitions) are neither values nor types, they cannot be translated into ACT ONE. However, since function applications may denote values and types, they can be translated, in the usual way, given a suitable environment containing the function definitions.

The issues of type equivalence, compatibility and scoping are discussed, with a view to formalisation, in [Thomas Rudkin MacLeod 91]. We believe that a formalisation of these issues should have been part of the original standard; moreover, had the semantics of the entire language been formalised earlier, these aspects would not have been overlooked.

8. Discussion

8.1 Comparison of ASN.1 and ACT ONE

The LOTOS sublanguage ACT ONE is a general purpose algebraic specification language. Although it is very expressive (we can define any computable algebra), the basic constructs are very primitive indeed and the specification of even trivial types is laborious. The static type system is very limiting; dynamic types, or at least order-sorted typing, would be a great enhancement.

On the other hand, ASN.1 is a very simple, concise, but powerful notation for certain kinds of data types. For example, the semantics of the heterogeneous sequence and set constructions, with implicit, default and component types, illustrates the conciseness of the ASN.1 notation. ASN.1 is very convenient to use when defining record-like types with only a variety of injection and projection operations. Of course, types with an "algorithmic" component cannot be defined within the language; in this case, a language like ACT ONE is required. However, within the context of communications and specifying OSI protocols, such "algorithmic" types would be a rare occurrence.

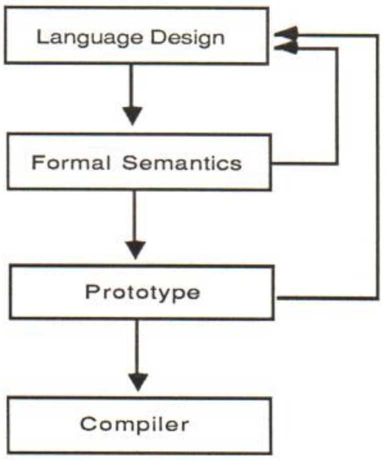
An important application of this work is the development of a *combined specification technique*. The user of this combined language can effectively use ASN.1 as the data typing part of LOTOS: the result is a technique with the compactness of ASN.1 and the expressibility of ACT ONE. A preliminary report on experience with such an approach at British Telecom is contained in [Rudkin MacLeod 91].

8.2 Role of the Prototype Translator

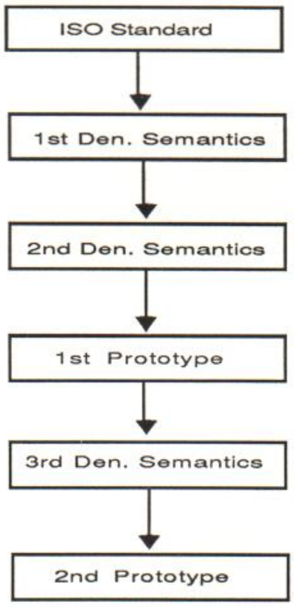
As stated earlier, one of the aims of formalising the ASN.1 standard is to allow a more rigorous discussion of the features of the languages and interpretations of the standard. The prototype translator has been an important feature in this discussion as experimentation has led to new ambiguities and problems. The fact that the prototype has been formally specified has improved both confidence in its reliability and ease of modification. Viewed as an experiment in the application of formal methods, the results have been very encouraging.

The translator has been applied to several real applications with success. However, it is only a prototype and does have its limitations. For example, the use of optional and default values in (heterogeneous) sequences and sets causes a combinatorial explosion of operations and equations (see the example in §6 for a hint!). An attempt to translate the ACSE protocol [Wilson 92] which includes an AARQ-apdu type with 8 optional values and an AARE-apdu with 4 optional values caused the implementation to exhaust the heap. To overcome this problem, the translator must be optimised and/or reimplemented, or the protocol transformed to eliminate some of the optional values.

There were many problems with formalising the ASN.1 standard. We cannot review the problems here, but note that the current semantics and translator are the result of progress through the following language design life cycle:



More specifically, the progress has been the following:



9. Future Work

First, we must incorporate any further feedback from the ASN.1 community. We need to know whether or not the semantics accurately describes ASN.1 as commonly used, other ways in which the translator might be used, and what further tools/environments might be useful.

Second, the ASN.1 language, the formal semantics and the translator must be extended to include any feedback and extensions to the language such as subtypes and tables; a more robust translator should be produced when all the language features are fixed.

9.1 Conclusions

This paper describes a translator tool from ASN.1 to LOTOS: an ASN.1 module is translated into a LOTOS specification which consists of a collection of ACT ONE data types and a single constant process. The translator is based on a formal semantics for ASN.1 and provides other functions as well.

A prototype translator has been implemented and the results of translator have been checked using the *topo* LOTOS compiler.

An interesting application of this work is the development of a *combined specification technique*. A translation between ASN.1 and ACT ONE allows a protocol specifier to use a mixture of notations: ASN.1 for the data type descriptions and basic LOTOS for process descriptions. In particular, such the translator may be used by a protocol specifier who later instantiates the (stop) process by a more appropriate behaviour. The result is a technique with the compactness of ASN.1 and the expressibility of ACT ONE.

Acknowledgements

This work has been sponsored by British Telecom Research. The author is indebted to Steve Rudkin, Ed MacLeod and Doug Steadman who have provided valuable technical expertise, insight, and encouragement; and to Jeremy Wilson for applying the translator to some real problems.

References

[Bochman Deslauriers 89] G. V. Bochmann, M. Deslauriers, Combining ASN.1 support with the LOTOS language, *Proc. IFIP Symp. on Protocol Specification, Testing and Verification IX*, North Holland Publ., 1989.

[Brady et al 91] F. Brady, A.G. Boshier, D. Pitt, B.M. Szczygiel, One2One - A Tool for Translating ASN.1 to ACT ONE, in *Proc. FORTE '90*, J. Quemeada, J. Manas, E . Vasquez (Ed.), Elsevier Science Publishers B.V. Amsterdam, 1991.

[Burmeister et al 90] J. Burmeister, J de Meer, R. Ahooja, K-H. Weiss, From ASN.1 to ACT ONE, Contribution to the CCITT Q19/VII - DAF Meeting Geneva, February 1990.

[Ehrig Mahr 85] H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification 1*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1985.

[ISO 8824] Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One, International Organisation for Standardisation, ISO 8824:1987 (E) plus Recommendation X.208.

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

[Segala 89a] R. Segala, A Formal Definition of Constructors for ACT ONE, Technical Memo HPL-ISC-TM-89-164, Hewlett-Packard Laboratories, Bristol, England, 1989.

[Segala 89b] R. Segala, ASN.1 into ACT ONE using macro constructors, Draft Technical Memo, Hewlett-Packard Laboratories, Bristol, England, 1989.

[Rudkin MacLeod 91] S. Rudkin, E. MacLeod, Combined Specification Using ASN.1 and LOTOS, report to BritishTelecom, 1991.

[Thomas 89] M. Thomas, From 1 Notation to Another One: An ACT-ONE Semantics for ASN.1, in *Proc. FORTE '89*, S.T.Vuong (Ed.), Elsevier Science Publishers B.V. Amsterdam, 1990.

[Thomas 90] M. Thomas, A Semantics for ASN.1, Report No. 2, report to British Telecom, March 1990.

[Thomas 90t] M. Thomas, An ACT ONE Semantics for ASN.1 Tables, report to British Telecom, October 1990.

[Thomas 90f] M. Thomas, A Review of the Proposal for ASN.1 Functions, report to British Telecom, October 1990.

[Thomas 91] M. Thomas, ASN.1 Subtypes, report to British Telecom, March 1991.

[Thomas 91r] M. Thomas, A Semantics for ASN.1, Report No. 3, report to British Telecom, March 1990.

[Thomas 92] M. Thomas, A Semantics for ASN.1, Report No. 4, report to British Telecom, January 1992.

[Thomas Rudkin MacLeod 90] M. Thomas, S. Rudkin, E. MacLeod, A Summary of Issues: Translating ASN.1 into ACT ONE, report to British Telecom, November 1990.

[Thomas Rudkin MacLeod 91] M. Thomas, S. Rudkin, E. MacLeod, ASN.1 Type Equivalence, Value References and Identifiers, report to British Telecom, March 1991.

[Turner 86] D. Turner, An Overview of Miranda, *SIGPLAN Notices*, December 1986.

[Wilson 92] J. Wilson, private communication, 1992.