SOFTWARE VERIFICATION RESEARCH CENTRE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 01-14

Representing ASN.1 in Z

Antonio Cerone

June 2001
(Revised: November 2002)

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

In: C. Johnson, P. Montague and C. Steketee, editors, *Proceedings of the Australasian Information Security Workshop (AISW2003)*, Adelaide, Australia, Conferences in Research and Practice in Information Technology, Vol. 21, pages 9–16 Australian Computer Society, 2003.

Representing ASN.1 in Z

Antonio Cerone

Abstract

ASN.1 (Abstract Syntax Notation One) has been increasingly used in defining the data structures used in internet security protocols. In this paper we present a framework for translating ASN.1 specification into Z. We use a restricted version of ASN.1, which is however sufficiently powerful to specify important network communication protocols. Finally, we present an example of translation based on the Cryptographic Message Syntax of S/MIME.

1 Introduction

In the last two decades ASN.1 (Abstract Syntax Notation One) [6, 7, 10] has been increasingly used in defining the data structures used in internet security protocols [2, 3, 4, 5, 11]. ASN.1 is a standard notation to give an abstract representation to the syntactical structure of datatypes used in network communication protocols. It is independent of the encoding algorithm used to represent the actual instantiations of the datatypes, but contains enough details to provide a model suitable for analysis.

Our overall research aims to verify security protocols using a refinement approach. Since we are interested in security protocols, whose data structure is specified using ASN.1, we need to represent ASN.1 in some formal language commonly used in verification environments. Z [12] is a highly abstract model-oriented specification notation. It has already been used for modelling security protocols [1, 9, 8] and has a well-established notion of refinement [13]. It therefore provides a sound basis for our research. However, to get started, we must first translate security protocol datatype structures expressed in ASN.1 into equivalent Z declarations. In this paper we define just such a mapping.

ASN.1 is presented in a descriptive fashion in cryptic reports without any attempts to give any formal characterization to static and dynamic semantics [2, 6, 10]. Static semantics is implicitly introduced in such reports by informally describing the restrictions to impose on the possible specifications allowed by the syntax of ASN.1. Our translation into Z implicitly includes all aspects of static semantics: only the ASN.1 definitions that meet the restriction required by the static semantics have a representation in Z.

The subset of ASN.1 [6] we are going to formalise has been used to define the data structures used by the S/MIME internet protocol [4, 5]. Moreover, we are going to work with ASN.1 without going into the details of the Basic Encoding Rules (BER) or the Distinguished Encoding Rules (DER) used to represent values of each ASN.1 type as a string of bits [6]. The semantics given to our translation will be, however, consistent with the Encoding Rules.

Section 2 briefly introduces the ASN.1 notation. Section 3 defines the framework for representing the syntax and static semantics of ASN.1 in Z. Section 4 shows how ASN.1's module structure can be represented in Z. Finally, Section 5 presents a brief example of translation based on the Cryptographic Message Syntax of S/MIME [4].

2 The Notation

Historically ASN.1 [2, 6, 7, 10] was originated in the 1980s [2] by the need of representing in a machine-independent way the complex data structures that are used in internet protocols [7]. The notation has then been revised and extended during the last two decades [10].

Internet protocols utilise repetitive and optional structures built-up from primitive data types. There are three main construction mechanisms:

repetition an ordered (`SEQUENCE OF`) or unordered (`SET OF`) finite collection of components having the same type;

record an ordered (`SEQUENCE`) or unordered (`SET`) finite collection of fields, each having a distinct name and a type;

alternative a choice (`CHOICE`) among a finite set of alternative types.

Mutual recursion is permitted and, combined with the alternative construction mechanism, allows the recursive definition of data structures which can still have a finite representation of some of their values.

3 Abstract Syntax of Types

Only a subset of ASN.1 is actually used in defining the data structures of internet protocols [6, 7]. In this paper we just consider this subset of ASN.1 and, in the following, we will use ASN.1 to denote such a subset. As a consequence, some of the definitions given in the Standard [2, 10] will appear in our paper in a simplified form.

In this section we present the ASN.1 data types, and, for each of them, define a Z type to represent the structure of that ASN.1 data type. The disjoint union *TYPE* of such Z types represents all possible data types that can be defined using the ASN.1 abstract syntax.

The structure of an ASN.1 type is given by a Z free type definition [12, p.81] as shown in Figure 1. This definition is used to translate ASN.1 datatypes into a Z type as shown in Section 5; the constructors of the *TYPE* free type, which are injections, will be used to define the complex datatype used in security protocols [4].

We assume that the sets of identifiers (*IDENTIFIER*), the sets of references (*REFERENCE*), and the sets of values (*VALUE*) are given Z types.

$$[IDENTIFIER, REFERENCE, VALUE]$$

In ASN.1 *identifiers*, *value references*, *type references* and *module references* all consist of one or more letters, digits and hyphens, with a hyphen neither occurring as the last character nor immediately following another hyphen. Identifier and value references always start with a lower-case letter, whereas type references and module references always start with an upper-case letter [10]. In our translation we use

- *IDENTIFIER* to represent identifiers and value references;
- *REFERENCE* to represent type references and module references.

This disjunction of the two sets is ensured in the Z representation by the definition of two separate types *IDENTIFIER* and *REFERENCE*.

In the rest of this section, for each of the alternatives in the free type above, we give the ASN.1 abstract syntax and we introduce the Z representation. Every alterative type is represented in Z by a schema definition. A schema definition consists of a *schema name*, a *declaration part* and a *predicate* [12, p.51]. The declarations in a schema definition have a local scope. We will also make use of axiomatic descriptions [12, p.50], which are unnamed schema definitions with a global scope.

The following meta-syntax is used in describing ASN.1 notation [6].

`CHOICE`
reserved words in capitalized typewriter style;

Var,**Var**$_{\mathbf{i}}$
bold words (possibly indexed) denote variables;

[ ]
bold square brackets denote an optional term;

()
bold parentheses group related terms;

. . .
bold dots indicate repeated occurrences;

|
bold vertical bar delimits alternatives within a group;

$$
\begin{array}{lll}
TYPE ::= & simple\langle\langle SimpleType\rangle\rangle & \text{– simple type} \\
\mid & tagged\langle\langle TaggedType\rangle\rangle & \text{– tagged type} \\
\mid & any\langle\langle AnyType\rangle\rangle & \text{– type } \texttt{ANY} \\
\mid & choice\langle\langle Alternative\rangle\rangle & \text{– type } \texttt{CHOICE} \\
\mid & sequence\langle\langle Sequence\rangle\rangle & \text{– type } \texttt{SEQUENCE} \\
\mid & sequenceof\langle\langle SequenceOf\rangle\rangle & \text{– type } \texttt{SEQUENCE OF} \\
\mid & set\langle\langle Set\rangle\rangle & \text{– type } \texttt{SET} \\
\mid & setof\langle\langle SetOf\rangle\rangle & \text{– type } \texttt{SET OF}
\end{array}
$$

Figure 1: Structure of an ASN.1 type

=
: bold equal sign expresses terms as subterms.

3.1 Simple Types

Simple Types are *atomic* types; they do not consist of components. The following simple types are relevant to the PKCS standards [11]: `BIT STRING`, `IA5String`, `INTEGER`, `NULL`, `OBJECT IDENTIFIER`, `OCTET STRING`, `PrintableString`, `T61String`, `UTCTime`. Since we are not interested in the encoding, we will not distinguish between string types and non-string types.

We represent Simple Types in Z by a free type definition without constructors as shown in Figure 2.

3.2 Tagged Types

All ASN.1 types apart from `ANY` and `CHOICE` can be given a tag. Tagging is commonly used to distinguish types which have the same structure but play different roles within an application. At a lower level tagging can be also used to remove ambiguities in the definition of a structured type. For example, optional components of the same data structure, constructed as a record, need to be given distinct tags to avoid ambiguity. Therefore tagged types are abstractly the same if and only if they have the same tag, independently of their name.

There are four classes of tags:

universal are defined in CCITT Recommendation X.208 [2] and have the same meaning in all applications;

application are specific to a given application;

private are specific to a given enterprise;

context-specific are specific to a given structured type.

Classes *universal*, *application* and *private* have explicit names in the syntax and are used to distinguish between types with the same structure whereas class *context-specific* has a null name in the syntax and is used to tag optional components of record-like data structures.

Tagging can be *implicit*, when derived from another type by changing its tag, or *explicit*, when derived from another type by adding a tag.

The structure of an ASN.1 tagged type

[[**Class**] **Number**] (`IMPLICIT` | `EXPLICIT`) **Type**

with

Class = `UNIVERSAL` | `APPLICATION` | `PRIVATE`

is represented in Z, in the context of the *TYPE* definition above, by a schema

```
─ TaggedType ──────────────────────
│ class : Class
│ number : OptNat
│ tagmethod : TagMeth
│ type : TYPE
├──────────────────────
│ type ∉ (ran any) ∪ (ran choice)
└──────────────────────
```

$$
\begin{array}{llll}
SimpleType & ::= & bitstring & \text{– type } \texttt{BIT STRING} \\
 & | & iastring & \text{– type } \texttt{IA5String} \\
 & | & integer & \text{– type } \texttt{INTEGER} \\
 & | & null & \text{– type } \texttt{NULL} \\
 & | & objectide & \text{– type } \texttt{OBJECT IDENTIFIER} \\
 & | & octet & \text{– type } \texttt{OCTET STRING} \\
 & | & printable & \text{– type } \texttt{PrintableString} \\
 & | & tstring & \text{– type } \texttt{T61String} \\
 & | & utctime & \text{– type } \texttt{UTCTime}
\end{array}
$$

Figure 2: Simple Types

where

$$
\begin{array}{lll}
Class & ::= & noclass \\
 & | & universal \\
 & | & application \\
 & | & private \\
OptNat & ::= & nonat \\
 & | & optnat\langle\langle \mathbb{N} \rangle\rangle \\
TagMeth & ::= & explicit \\
 & | & implicit
\end{array}
$$

The predicate part of schema *TaggedType* enforces the ASN.1 constraint that `ANY` and `CHOICE` types cannot be given a tag by asserting that the *type* component of the schema belongs to neither of the ranges of the constructors *any* and *choice*. Special values *noclass* and *nonat* respectively denote the absence of class name and number

3.3 CHOICE Type

The `CHOICE` type denotes a union of one or more alternatives. It is represented in ASN.1 as follows.

`CHOICE` {
  [$\mathbf{Identifier_1}$] $\mathbf{Type_1}$
  ...
  [$\mathbf{Identifier_n}$] $\mathbf{Type_n}$ }

where $\mathbf{Identifier_1}, \ldots, \mathbf{Identifier_n}$ are optional, distinct identifiers for the alternatives. $\mathbf{Type_1}, \ldots, \mathbf{Type_n}$ are the types of the alternatives. The identifiers are mainly for documentation, and they do not affect the values of the types or their encoding in any way. Types must have distinct tags.

We define a function which returns the tag associated with a *TYPE* type using an axiomatic description.

$$
\begin{array}{|l}
tagging : TYPE \nrightarrow (Class \times OptNat) \\
\hline
\mathrm{dom}\, tagging = \mathrm{ran}\ tagged \\
\forall\, t: TYPE \mid t \in \mathrm{dom}\, tagging \bullet \\
\quad \mathrm{first}(\, tagging\ t) = (tagged^{\sim}\ t).class \wedge \\
\quad \mathrm{second}\ (tagging\ t) = (tagged^{\sim}\ t).number
\end{array}
$$

The *tagging* function is partial because not every ASN.1 type is a tagged one. The $^{\sim}$ operator denotes the relational inversion [12, p.100]. Notice that the *class* and *number* attributes are visible within the above axiomatic description due to the recursive definition of *TYPE*, which includes the *TaggedType* schema, where the two attributes are defined.

Since **Identifier** may not be present, we introduce a special value *noide*, which is not an element of *IDENTIFIER*, to denote the absence of **Identifier**.

$$
\begin{array}{lll}
OptIde & ::= & noide \\
 & | & optidentifier\langle\langle IDENTIFIER \rangle\rangle
\end{array}
$$

The structure of an ASN.1 `CHOICE` type is then represented in Z by a schema

Alternative

$$alternative : \mathbb{F}_1(OptIde \times TYPE)$$

$$\#((\text{first} \mathrel{(\!|} alternative \mathrel{|\!)}) \setminus \{noide\}) = \#(alternative \cap ((\text{ran}\, optidentifier) \times TYPE)) \land$$
$$\#tagging \mathrel{(\!|} \text{second} \mathrel{(\!|} alternative \mathrel{|\!)} \mathrel{|\!)} = \#alternative$$

Every alternative of a choice is represented by a finite non-empty set [12, p.114] of pairs. The first conjunct of the predicate ensures that the identifiers are all distinct while the second conjunct ensures that the types of different alternatives have different tags.

3.4 ANY Type

The ASN.1 `ANY` type denotes an arbitrary value of an arbitrary type. The arbitrary type may be defined in the registration of an object identifier or associated with an integer index. The structure of an ASN.1 `ANY` type

`ANY` [`DEFINED BY` **Identifier**]

where **Identifier** is an identifier, is represented in Z by the following trivial schema

AnyType

$$definedby : OptIde$$

3.5 Sequence

The `SEQUENCE` type denotes an ordered finite collection of one or more types. It is represented in ASN.1 as follows,

`SEQUENCE` {
  [$\mathbf{Identifier_1}$] $\mathbf{Type_1}$
    [`OPTIONAL` | `DEFAULT` $\mathbf{Value_1}$],
  ...
  [$\mathbf{Identifier_n}$] $\mathbf{Type_n}$
    [`OPTIONAL` | `DEFAULT` $\mathbf{Value_n}$] }

where $\mathbf{Identifier_1}$, ..., $\mathbf{Identifier_n}$ are optional, distinct identifiers for the components, $\mathbf{Type_1}$, ..., $\mathbf{Type_n}$ are the types of the components, and $\mathbf{Value_1}$, ..., $\mathbf{Value_n}$ are optional default values for the components.

There is an additional requirement that an ASN.1 sequence must satisfy: the types of any consecutive series of components with the `OPTIONAL` and `DEFAULT` qualifier, as well as of any component immediately following that series, must have distinct tags. The need for such a restriction is due to the fact that types are distinguished depending on their tagging rather than on their name.

We represent the `OPTIONAL` and `DEFAULT` qualifiers as follows.

$$OptionalOrDefault ::= noqual \mid optional \mid default\langle\!\langle VALUE \rangle\!\rangle$$

We can now define a component of a sequence as a triple.

$$Comp ::= comp\langle\!\langle OptIde \times TYPE \times OptionalOrDefault \rangle\!\rangle$$

In order to meet the restriction above, we have also to define an auxiliary function on components of sequences, *isOorD*, which returns true if the component is optional or default and false otherwise.

$$isOorD : Comp \twoheadrightarrow \mathbb{B}$$

$$(comp \mathbin{⨾} isOorD) \mathrel{(\!|} OptIde \times TYPE \times \{optional\} \mathrel{|\!)} = \{\text{true}\}$$
$$(comp \mathbin{⨾} isOorD) \mathrel{(\!|} OptIde \times TYPE \times default \mathrel{(\!|} VALUE \mathrel{|\!)} \mathrel{|\!)} = \{\text{true}\}$$
$$(comp \mathbin{⨾} isOorD) \mathrel{(\!|} OptIde \times TYPE \times \{noqual\} \mathrel{|\!)} = \{\text{false}\}$$

Constructor *comp* and function *isOorD* are composed using the ⨾ relational composition operator [12, p.97] and the resultant function gives value true if and only if it is applied to a triple in *Comp* which has the optional or default qualifier.

We define a function that characterizes the set of the consecutive series of components that have the `OPTIONAL` and `DEFAULT` qualifier, with the exception of the last component of the series, which might not have `OPTIONAL` or `DEFAULT` qualifier.

$$SeqComp == \mathrm{seq}_1\, Comp$$

$$\begin{array}{l} optdefseq : SeqComp \rightarrow \mathbb{F}\, SeqComp \\ \hline \forall\, s : SeqComp \mid optdefseq\ s = \\ \quad \{u : SeqComp \mid u \text{ in } sequence^{\sim} s \wedge \\ \quad\quad \forall\, c : Comp \mid c \text{ in } u \bullet \\ \quad\quad\quad c = last\, u \vee \\ \quad\quad\quad (c \text{ in } front\, u \wedge isOorD\ \ c)\} \end{array}$$

We also define functions that project an element of *Comp* onto its identifier, if present, and onto its type.

$$\begin{array}{l} ideof : Comp \nrightarrow OptIde \\ typeof : Comp \rightarrow TYPE \\ \hline \mathrm{dom}\ ideof = \\ \quad \mathrm{ran}(comp\ (\!|\ (\mathrm{ran}\, optidentifier) \times \\ \quad\quad\quad TYPE \times \\ \quad\quad\quad OptionalOrDefault\ |\!))\ \wedge \\ \forall\, i : OptIde;\ t : TYPE;\ o : \\ \quad OptionalOrDefault \bullet \\ \quad ideof(comp(i, t, a)) = i \wedge \\ \quad typeof(comp(i, t, a)) = t \end{array}$$

Finally, the structure of the `SEQUENCE` type is represented by a schema as follows.

$$\begin{array}{l} \textbf{Sequence} \\ seqcomplist : SeqComp \\ \hline (seqcomplist \mathbin{;} ideof) \in iseq\ IDENTIFIER \wedge \\ \forall\, s : SeqComp;\ c_1, c_2 : Comp \mid \\ \quad s \in optdefseq\ seqcomplist \bullet \\ \quad c_1 \text{ in } s\ \wedge\ c_2 \text{ in } s\ \wedge\ c_1 \neq c_2\ \Rightarrow \\ \quad\quad tagging(typeof\ c_1) \neq \\ \quad\quad tagging(typeof\ c_2) \end{array}$$

The first conjunct ensures that the identifiers are all distinct by asserting that the projections of the elements of *seqcomplist* on the identifier gives an injective sequence of identifiers [12, p.118]. The second conjunct guarantees the restrictions on the tags of components that we have described at the beginning of this section.

3.6 Sequence of

The `SEQUENCE OF` type denotes an ordered finite collection of zero or more occurrences of a given type. It is represented in ASN.1 as follows,

`SEQUENCE OF` **Type**

where **Type** is a type.

The structure of the `SEQUENCE OF` type is represented as follows.

$$\begin{array}{l} \textbf{SequenceOf} \\ type : TYPE \end{array}$$

3.7 Set

The `SET` type denotes an unordered finite collection of one or more types. It is represented in ASN.1 as follows,

`SET` {
  [$\mathbf{Identifier_1}$] $\mathbf{Type_1}$
    [`OPTIONAL` | `DEFAULT` $\mathbf{Value_1}$],
  ...
  [$\mathbf{Identifier_n}$] $\mathbf{Type_n}$
    [`OPTIONAL` | `DEFAULT` $\mathbf{Value_n}$] }

where $\mathbf{Identifier_1}$, ..., $\mathbf{Identifier_n}$ are optional, distinct identifiers for the components, $\mathbf{Type_1}$, ..., $\mathbf{Type_n}$ are the types of the components, and $\mathbf{Value_1}$, ..., $\mathbf{Value_n}$ are optional default values for the components. The types must have distinct tags.

We represent components and optional and default qualifiers in the same way as for sequences. The structure of the `SET` type is represented as follows.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ Set \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
setelemlist : \mathbb{F}_1\, Comp \\
\hline
\#ideof ⦇ setelemlist ⦈ = \\
\quad \#(setelemlist \cap \\
\qquad (comp ⦇ (\operatorname{ran} optidentifier) \times \\
\qquad\qquad TYPE \times \\
\qquad\qquad OptionalOrDefault ⦈)) \land \\
\forall c_1, c_2 : \mathbb{F}_1\, Comp \mid c_1, c_2 \in setelemlist \bullet \\
\qquad c_1 \neq c_2 \Rightarrow \\
\qquad\quad (tagging(typeof\ c_1)) \neq \\
\qquad\quad (tagging(typeof\ c_2))
\end{array}
$$

The first assertion ensures that the identifiers are all distinct while the second assertion guarantees that all components have distinct tags.

3.8 Set of

The `SET OF` type denotes an unordered finite collection of zero or more occurrences of a given type. It is represented in ASN.1 as follows.

`SET OF` **Type**

where **Type** is a type.

It is represented as follows.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ SetOf \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
type : TYPE \\
\hline
\end{array}
$$

4 Abstract Syntax of Modules

In this section we use a simplified syntax for an ASN.1 module. It assumes implicit tagging as a default and it allows the definition of most of the important internet protocols such as S/MIME [4]. Our restricted version of an ASN.1 module is represented in concrete syntax as follows.

ModuleDef =
  ModuleIdentifier
  `DEFINITIONS IMPLICIT TAGS` ::=
  `BEGIN`
  ModuleBody
  `END`

The **ModuleIdentifier** syntax category consists of a module reference and a possibly empty list of component object identifiers.

ModuleIdentifier =
  Reference [**ObjectIdentifier**]

The **Reference** syntax category denotes type references and module references. The **ObjectIdentifier** syntax category is defined in Section 4.2.

The **ModuleBody** syntax category is defined as follows.

ModuleBody =
  [**Exports**] [**Imports**] **AssignmentList**

The **Exports** and **Imports** syntax categories are sequences of types, which are respectively exported to other modules and imported from other modules or from the external enviroment.

Exports =
  `EXPORT` **ReferenceList**;
ReferenceList =
  $\mathbf{Reference_1}, \ldots \mathbf{Reference_n}$
Imports =
  `IMPORT` **ImportList**;
ImportList =
  ReferenceList `FROM` **ModuleIdentifier**
AssignmentList =
  $\mathbf{Assignment_1}, \ldots \mathbf{Assignment_n}$

The Z type needed to represent an ASN.1 module is defined as follows.

$$
MODULE ::= module \langle\!\langle ModuleIde \times Imports \times Exports \times Body \rangle\!\rangle
$$

where

$$
\begin{array}{l}
ModuleIde == \\
\quad REFERENCE \times OBJIDE \\
Imports == \\
\quad \operatorname{seq}_1((\mathbb{F}_1\, REFERENCE) \times ModuleIde) \\
Exports == \mathbb{F}_1\, REFERENCE \\
Body == \operatorname{seq}_1 ASSIGNMENT
\end{array}
$$

Types *ASSIGNMENT* and *OBJIDE* will be defined in the next two sections.

4.1 Assignments

The ASN.1 syntax of an assignment of a type expression to an identifier is given as follows.

Assignment =
  TypeAssignment | **ValueAssignment**
TypeAssignment =
  Reference ::= **Type**
ValueAssignment =
  Identifier ::= **Value**

We represent the domain of the assignment in Z by a free type as follows.

$$ASSIGNMENT ::= typeass\langle\langle REFERENCE \times TYPE\rangle\rangle \mid valass\langle\langle IDENTIFIER \times VALUE\rangle\rangle$$

We also define two partial functions *typeofref* and *valueofide*, which return the type and the value respectively assigned to the reference and identifier given as arguments.

$$\begin{array}{l} typeofref : REFERENCE \rightarrow\!\!\!\!\!\!+ \; TYPE \\ valueofide : IDENTIFIER \rightarrow\!\!\!\!\!\!+ \; VALUE \\ \hline (\forall r : REFERENCE \mid r \in \mathrm{dom}\, typeofref \bullet \\ \quad (r, typeofref\, r) \in \mathrm{dom}\, typeass) \land \\ (\forall i : IDENTIFIER \mid i \in \mathrm{dom}\, valueofide \bullet \\ \quad (i, valueofide r) \in \mathrm{dom}\, valass) \end{array}$$

4.2 Object Identifier Values

Object identifiers are used to give an unambiguous identification of entities which provide external services, that is services not provided by the modelled protocol. Such an entity can be an algorithm, an attribute type or a registration authority that defines other object identifiers.

Object identifier is the only type for which ASN.1 provides not only a notation for the type itself, but also for values of that type. A value of type object identifier is represented in ASN.1 as follows.

ObjectIdentifier =
  { $\mathbf{Component_1 \ldots Component_n}$ }
$\mathbf{Component_i}$ =
  $\mathbf{Identifier_i}$ |
  $\mathbf{Identifier_i}$ ($\mathbf{Value_i}$) |
  $\mathbf{Value_i}$

where $\mathbf{Identifier_1}$, ..., $\mathbf{Identifier_n}$ are identifiers and $\mathbf{Value_1}$, ..., $\mathbf{Value_n}$ are optional integer values. Only the identifiers that are defined in X.208 [2] can appear without associated integer values.

In the original definition of ASN.1 [2] there is an optional **Identifier** in front of the sequence of components to abbreviate a part of the sequence of components. In revised versions [10] of the definition of ASN.1 abbreviations can occur anywhere in the sequence of components. For simplicity, in our definition we do not allow abbreviations.

The *CompDef* set defines the possible structures of components.

$$CompDef ::= ide\langle\langle IDENTIFIER\rangle\rangle \mid both\langle\langle IDENTIFIER \times VALUE\rangle\rangle \mid val\langle\langle \mathbb{N}\rangle\rangle$$

Partial function *objide* is the constructor of values of type *OBJIDE*.

$$\begin{array}{l} objide : (\mathrm{seq}_1\, CompDef) \rightarrow\!\!\!\!\!\!+ \; VALUE \\ \hline \{i : IDENTIFIER \mid \\ \quad (ide\ i)\ \mathrm{in}\ (\mathrm{dom}\ objide)\} \subseteq IdeX_{208} \end{array}$$

where $IdeX_{208}$ is the set of the identifiers that are defined in X.208. Now we can define the *objide* Z type as a subtype of *VALUE*.

$$OBJIDE == \{v : VALUE \mid v \in \mathrm{ran}\, objide\}$$

5 Example

The initial fragment of the

`CryptographicMessageSyntax`

```
CryptographicMessageSyntax
    { iso(1) member-body(2) us(840) rsadsi(113549)
      pkcs(1) pkcs-9(9) smime(16) module(0) scm(1) }

DEFINITIONS IMPLICIT TAGS ::=
BEGIN
IMPORTS
  Name
    FROM InformationFramework
           { joint-iso-itu-t ds(5) module(1)
             informationFramework(1) 3 }
  AlgorithmIdentifier, AttributeCertificate, Certificate,
  CertificateList, CertificateSerialNumber
    FROM AuthenticationFramework
           { joint-iso-itu-t ds(5) module(1)
             authenticationFramework(7) 3 } ;

ContentInfo ::= SEQUENCE {
  contentType ContentType,
  content [0] EXPLICIT ANY DEFINED BY contentType }

ContentType ::= OBJECT IDENTIFIER
  .
  .
  .
END
```

Figure 3: Initial fragment of the `CryptographicMessageSyntax` module of S/MIME

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ Assignment_1\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
ContentType : REFERENCE \\
a_1 \ : \ ASSIGNMENT \\
\hline
a_1 = typeass(ContentType, \\
\qquad\qquad (simple\ objectide)) \\
\hline
\end{array}
$$

Figure 4: Z representation of the first assignment in `CryptographicMessageSyntax`

module of S/MIME is defined in ASN.1 as shown in Figure 3 [4]. Let us translate the `ContentInfo` type into Z. Notice that a similar definition of the `ContentInfo` type is given in the Cryptographic Message Syntax Standard (PKCS #7) [11], where, however, the `content` component is optional.

The assignment to the `ContentType` reference is easily represented in Z by a constant a_1 defined as shown in Figure 4. The assignment to the `ContentInfo` reference is represented by a constant a_2 defined as shown in Figure 5. The definition above fully captures both the syntax and the static semantics of the *ContentInfo* type.

The **Exports** syntactic category is empty and it is therefore represented in Z by $\varnothing$. The Z schema in Figure 6 represents the content of the **Imports** syntactic category. The `CryptographicMessageSyntax` module can finally be represented in Z as shown in Figure 7. The $ide\ joint\!-\!iso\!-\!itu\!-\!t$ component of the two object identifiers in the

$$Imports_{CryptographicMessageSyntax}$$

schema above is the representation of the `joint-iso-itu-t` identifier, which is defined in X.208 and associated with value 2.

6 Conclusions

In this paper we have presented a translation of ASN.1 into Z. Such a translation shows that ASN.1 can be concisely represented in Z. In this way we also formalised previously informal work and merged different draft documents [2, 6, 7, 10].

Our Z model captures both ASN.1 syntax and static semantics in an unambiguous way. It thus provides a sound starting point for formalising security protocols in Z [9, 8].

Acknowledgements

I would like to thank Colin Fidge, Peter Kearney, Andrea Maggiolo-Schettini and John Yesberg for helpful discussions. Colin also reviewed an initial draft of this paper providing many useful comments.

The final version of this paper was prepared during my visit at the Dipartimento di Informatica, University of Pisa. I would like to thank Andrea Maggiolo-Schettini and the University of Pisa for support and hospitality. My trip to Pisa was funded by a Travel Grant from The University of Queensland.

Presentation of this paper was assisted by Australian Research Council Linkage Grant LP0347620, Formally-Based Security Evaluation Procedures.

References

[1] C. Boyd. Security architectures using formal methods. *IEEE Journal on Selected Areas in Communication*, 11(5):694–701, June 1993.

[2] CCITT. Reccomendation X.208: Specification of abstract syntax notation one (ASN.1), 1988.

[3] CCITT. Reccomendation X.509: The director—authentication framework, 1988.
URL: http://sunsite.org.uk/public/computing/ccitt/ccitt-standards/1992/X/x509_1.asc.gz.

[4] R. Housley. Cryptographic message syntax (RFC 3369). Technical report, Network Working Group, RSA Laboratories, August 2002.
URL: http://www.ietf.org /rfc /rfc3369.txt.

[5] R. Housley. Use of the RSAES-OAEP key transport algorithm in CMS — internet draft. Technical report, S/MIME Working Group, RSA Laboratories, August 2002.
URL: http://www.itef.org /internet-drafts / draft-ietf-smime-cms-rsaes-oaep-06.txt.

$$
\begin{array}{l}
\textit{Assignment}_2 \\
\hline
\quad \textit{ContentInfo} : \textit{REFERENCE} \\
\quad a_2 : \textit{ASSIGNMENT} \\
\hline
\quad \exists\, t_1 : \textit{TYPE};\ s_1 : \textit{AnyType} \bullet t_1 = \textit{any}\ s_1\ \wedge (s_1 = \textit{definedby}(\textit{optidentifier contentType}))\ \wedge \\
\quad (\exists\, t_2 : \textit{TYPE};\ s_2 : \textit{TaggedType} \bullet (t_2 = \textit{tagged}\ s_2)\ \wedge \\
\qquad\qquad (s_2.\textit{Class} = \textit{noclass})\ \wedge \\
\qquad\qquad (s_2.\textit{number} = \textit{optnat}0)\ \wedge \\
\qquad\qquad (s_2.\textit{tagmethod} = \textit{explicit})\ \wedge \\
\qquad\qquad (s_2.\textit{type} = t_1)\ \wedge \\
\quad (\exists\, t_3 : \textit{TYPE};\ s_3 : \textit{Sequence};\ c_1, c_2 : \textit{Comp} \bullet \\
\qquad (t_3 = \textit{sequence}\ s_3)\ \wedge \\
\qquad (s_3.\textit{seqcomplist} = \langle c_1, c_2 \rangle)\ \wedge \\
\qquad (c_1 = \textit{comp}(\textit{optidentifier contentType}, \textit{typeofref ContentType}, \textit{noqual}))\ \wedge \\
\quad (c_2 = \textit{comp}(\textit{optidentifier content}, t_2, \textit{noqual}))))\ \wedge \\
\quad (a_2 = \textit{typeass}(\textit{ContentInfo}, (\textit{simple}, t_3)))
\end{array}
$$

Figure 5: Z representation of the second assignment in `CryptographicMessageSyntax`

$$
\begin{array}{l}
\textit{Imports}_{\textit{CryptographicMessageSyntax}} \\
\hline
\quad i : \textit{Exports} \\
\hline
\quad i = \langle (\{\textit{Name}\}, (\textit{InformationFrameWork}, \\
\qquad \textit{objide}\langle \textit{ide joint–iso–itu–t}, \textit{both}(\textit{ds}, 5), \textit{both}(\textit{modules}, 1), \\
\qquad\qquad \textit{both}(\textit{informationFrameWork}, 1), \textit{val}3 \rangle)) \\
\qquad (\{\textit{AlgorithmIdentifier}, \textit{AttributeCertificate}, \textit{Certificate}, \\
\qquad \textit{CertificateList}, \textit{CertificateSerialNumber}\}, \\
\qquad (\textit{AuthenticationFrameWork}, \\
\qquad \textit{objide}\langle \textit{ide joint–iso–itu–t}, \textit{both}(\textit{ds}, 5), \textit{both}(\textit{modules}, 1), \\
\qquad\qquad \textit{both}(\textit{authenticationFrameWork}, 7), \textit{val}3 \rangle)))\rangle
\end{array}
$$

Figure 6: Z representation of the `IMPORT` part of the `CryptographicMessageSyntax`

$$
\begin{array}{l}
\textit{CryptographicMessageSyntax} \\
\hline
\quad m : \textit{MODULE} \\
\hline
\quad (\exists\, b : \mathrm{seq}_1\, \textit{Body} \bullet \\
\qquad b = \langle \textit{Assignment}_1.a_1, \textit{Assignment}_2.a_2,\ \ldots \rangle\ \wedge \\
\quad (\exists\, o : \textit{OBJIDE} \bullet \\
\qquad o = \textit{objide}\langle \textit{both}(\textit{iso}, 1), \textit{both}(\textit{member–body}, 2), \textit{both}(\textit{us}, 840), \\
\qquad\qquad \textit{both}(\textit{rsadsi}, 113549), \textit{both}(\textit{pkcs}, 1), \textit{both}(\textit{pkcs–}9, 9), \\
\qquad\qquad \textit{both}(\textit{smime}, 16), \textit{both}(\textit{modules}, 0), \textit{both}(\textit{cms}, 1)\rangle\ \wedge \\
\quad (\exists\, n : \textit{ModuleIde} \bullet n = (\textit{CryptographicMessageSyntax}, o)\ \wedge \\
\quad m = \textit{module}(n, \varnothing, \textit{Imports}_{\textit{CryptographicMessageSyntax}}.i, b))))
\end{array}
$$

Figure 7: Z representation of the `CryptographicMessageSyntax` module

[6] B. S. J. Kalinski. A layman's guide to a subset of ASN.1, BER and DER. Technical report, RSA Laboratories, 1993.

[7] J. Lamouth. *Understanding OSI*. International Thomson Computer Press, 1996.

[8] B. W. Long. Formalising key distribution in the presence of trust using Object-Z. In C. Johnson, P. Montague, and C. Steketee, editors, *Proceedings of the Australasian Information Security Workshop and the Workshop on Wearable, Invisible, Context-Aware, Ambient, Pervasive and Ubiquitous Computing*, volume 21 of *Conferences in Research and Practice in Information Technology*. Australian Computer Society, 2003.

[9] B. W. Long, C. Fidge, and A. Cerone. A Z based approach to verifying security protocols. Technical Report 02-02, Software Verification Research Centre, The University of Queensland, 2002. URL: http://www.svrc.uq.edu.au /Publications /2002 /svrc2002-002.html.

[10] AS/NZS 8824.1:1998. Information technology—Abstract Syntax Notation One. Part 1: specification of basic notation. Technical report, Australian/New Zealand Standard, 1998.

[11] RSA-Security. Public-key cryptography standards. RSA Laboratories, Web Page, 2002. URL: http://www.rsasecurity.com /rsalabs / pkcs /.

[12] J. M. Spivey. *The Z Notation*. International Series in Computer Science. Prentice-Hall, 1989.

[13] J. B. Wordsworth. *Software Development with Z — A Practical Approach to Formal Methods in Software Engineering*. International Series in Computer Science. Prentice-Hall, 1992.