

Representing ASN.1 in Z

Antonio Cerone

Software Verification Research Centre
The University of Queensland,
Brisbane QLD 4072
Email: antonio@svrc.uq.edu.au

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.

Keywords: Abstract Syntax Notation One, Z specification language, security protocols.

1 Introduction

In the last two decades ASN.1 (Abstract Syntax Notation One) (Kalinski 1993, Lamouth 1996, AS/NZS 8824.1:1998 1998) has been increasingly used in defining the data structures used in internet security protocols (CCITT 1988*a*, CCITT 1988*b*, Housley 2002*a*, Housley 2002*b*, RSA-Security 2002). 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 (Spivey 1989) is a highly abstract model-oriented specification notation. It has already been used for modelling security protocols (Boyd 1993, Long, Fidge & Cerone 2002, Long 2003) and has a well-established notion of refinement (Wordsworth 1992). 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 (CCITT 1988*a*, Kalinski 1993, AS/NZS 8824.1:1998 1998). 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 (Kalinski 1993) we are going to formalise has been used to define the data structures used by the S/MIME internet protocol (Housley 2002*a*, Housley 2002*b*). 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 (Kalinski 1993). 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 (Housley 2002*a*).

2 The Notation

Historically ASN.1 (CCITT 1988*a*, Kalinski 1993, Lamouth 1996, AS/NZS 8824.1:1998 1998) was originated in the 1980s (CCITT 1988*a*) by the need of representing in a machine-independent way the complex data structures that are used in internet protocols (Lamouth 1996). The notation has then been revised and extended during the last two decades (AS/NZS 8824.1:1998 1998).

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 (Kalinski 1993, Lamouth 1996). 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 (CCITT 1988*a*, AS/NZS 8824.1:1998 1998) 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 (Spivey 1989, 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 (Housley 2002*a*).

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 (AS/NZS 8824.1:1998 1998). 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 alternative type is represented in Z by a schema definition. A schema definition consists of a *schema name*, a *declaration part* and a *predicate* (Spivey 1989, p.51). The declarations in a schema definition have a local scope. We will also make use of axiomatic descriptions (Spivey 1989, p.50), which are unnamed schema definitions with a global scope.

The following meta-syntax is used in describing ASN.1 notation (Kalinski 1993).

CHOICE

reserved words in capitalized typewriter style;

Var, Var_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;

=

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 (RSA-Security 2002): BIT STRING, IA5String, INTEGER, NULL, OBJECT IDENTIFIER, OCTET STRING, Printable-String, 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 (CCITT 1988*a*) 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

<i>TaggedType</i>	
<i>class</i> :	<i>Class</i>
<i>number</i> :	<i>OptNat</i>
<i>tagmethod</i> :	<i>TagMeth</i>
<i>type</i> :	<i>TYPE</i>
<hr/>	
<i>type</i> ∉ (ran any) ∪ (ran choice)	

where

<i>Class</i>	::=	<i>noclass</i>
		<i>universal</i>
		<i>application</i>
		<i>private</i>
<i>OptNat</i>	::=	<i>nonat</i>
		<i>optnat</i> ⟨⟨N⟩⟩
<i>TagMeth</i>	::=	<i>explicit</i>
		<i>implicit</i>

$TYPE ::=$	$simple\langle\langle SimpleType \rangle\rangle$	- simple type
	$tagged\langle\langle TaggedType \rangle\rangle$	- tagged type
	$any\langle\langle AnyType \rangle\rangle$	- type ANY
	$choice\langle\langle Alternative \rangle\rangle$	- type CHOICE
	$sequence\langle\langle Sequence \rangle\rangle$	- type SEQUENCE
	$sequenceof\langle\langle SequenceOf \rangle\rangle$	- type SEQUENCE OF
	$set\langle\langle Set \rangle\rangle$	- type SET
	$setof\langle\langle SetOf \rangle\rangle$	- type SET OF

Figure 1: Structure of an ASN.1 type

$SimpleType ::=$	$bitstring$	- type BIT STRING
	$iastring$	- type IA5String
	$integer$	- type INTEGER
	$null$	- type NULL
	$objectide$	- type OBJECT IDENTIFIER
	$octet$	- type OCTET STRING
	$printable$	- type PrintableString
	$tstring$	- type T61String
	$utctime$	- type UTCTime

Figure 2: Simple Types

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 {
 [Identifier₁] Type₁
 ⋮
 [Identifier_n] Type_n }

where **Identifier**₁, ..., **Identifier**_n are optional, distinct identifiers for the alternatives. **Type**₁, ..., **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.

$tagging : TYPE \Rightarrow (Class \times OptNat)$
$\text{dom } tagging = \text{ran } tagged$
$\forall t: TYPE \mid t \in \text{dom } tagging \bullet$
$\text{first}(tagging\ t) = (tagged \sim t).class \wedge$
$\text{second}(tagging\ t) = (tagged \sim t).number$

The $tagging$ function is partial because not every ASN.1 type is a tagged one. The \sim operator denotes the relational inversion (Spivey 1989, 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**.

$OptIde ::= noide$
$\mid optidentifier\langle\langle IDENTIFIER \rangle\rangle$

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

$Alternative$
$alternative : \mathbb{F}_1(OptIde \times TYPE)$
$\#((\text{first } (\mid alternative \mid)) \setminus \{noide\}) =$
$\#(alternative \cap$
$((\text{ran } optidentifier) \times TYPE)) \wedge$
$\#tagging(\mid \text{second } (\mid alternative \mid) \mid) =$
$\#alternative$

Every alternative of a choice is represented by a finite non-empty set (Spivey 1989, 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,

$$\text{SEQUENCE } \{ \begin{array}{l} [\text{Identifier}_1] \text{Type}_1 \\ \quad [\text{OPTIONAL} \mid \text{DEFAULT Value}_1], \\ \dots \\ [\text{Identifier}_n] \text{Type}_n \\ \quad [\text{OPTIONAL} \mid \text{DEFAULT Value}_n] \end{array} \}$$

where $\text{Identifier}_1, \dots, \text{Identifier}_n$ are optional, distinct identifiers for the components, $\text{Type}_1, \dots, \text{Type}_n$ are the types of the components, and $\text{Value}_1, \dots, \text{Value}_n$ 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.

$$\text{OptionalOrDefault} ::= \text{noqual} \quad \left| \begin{array}{l} \text{optional} \\ \text{default}\langle\langle \text{VALUE} \rangle\rangle \end{array} \right.$$

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

$$\text{Comp} ::= \text{comp}\langle\langle \text{OptIde} \times \text{TYPE} \times \text{OptionalOrDefault} \rangle\rangle$$

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.

$$\begin{array}{|l} \text{isOorD} : \text{Comp} \rightarrow \mathbb{B} \\ \hline (\text{comp} \circledast \text{isOorD}) \langle \text{OptIde} \times \text{TYPE} \times \{ \text{optional} \} \rangle = \{ \text{true} \} \\ (\text{comp} \circledast \text{isOorD}) \langle \text{OptIde} \times \text{TYPE} \times \text{default}\langle \text{VALUE} \rangle \rangle = \{ \text{true} \} \\ (\text{comp} \circledast \text{isOorD}) \langle \text{OptIde} \times \text{TYPE} \times \{ \text{noqual} \} \rangle = \{ \text{false} \} \end{array}$$

Constructor comp and function isOorD are composed using the \circledast relational composition operator (Spivey 1989, 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.

$$\text{SeqComp} == \text{seq}_1 \text{Comp}$$

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

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

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

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

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

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 (Spivey 1989, 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.

$$\begin{array}{|l} \text{SequenceOf} \\ \hline \text{type} : \text{TYPE} \end{array}$$

3.7 Set

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

$$\text{SET } \{ \begin{array}{l} [\text{Identifier}_1] \text{Type}_1 \\ \quad [\text{OPTIONAL} \mid \text{DEFAULT Value}_1], \\ \dots \\ [\text{Identifier}_n] \text{Type}_n \\ \quad [\text{OPTIONAL} \mid \text{DEFAULT Value}_n] \end{array} \}$$

where $\text{Identifier}_1, \dots, \text{Identifier}_n$ are optional, distinct identifiers for the components, $\text{Type}_1, \dots, \text{Type}_n$ are the types of the components, and $\text{Value}_1, \dots, \text{Value}_n$ 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.

<i>Set</i>
<i>setelemlist</i> : \mathbb{F}_1 <i>Comp</i>
$\#ideof(\setelemlist) = \#(\setelemlist \cap (comp(\text{ran } optidentifier) \times TYPE \times OptionalOrDefault)) \wedge$
$\forall c_1, c_2 : \mathbb{F}_1 \text{Comp} \mid c_1, c_2 \in \setelemlist \bullet$
$c_1 \neq c_2 \Rightarrow (tagging(typeof\ c_1)) \neq (tagging(typeof\ c_2))$

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.

<i>SetOf</i>
<i>type</i> : TYPE

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 (Housley 2002a). 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 environment.

```

Exports =
  EXPORT ReferenceList;
ReferenceList =
  Reference1, ... Referencen
Imports =
  IMPORT ImportList;
ImportList =
  ReferenceList FROM ModuleIdentifier
AssignmentList =
  Assignment1, ... Assignmentn

```

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

MODULE ::= *module*⟨⟨*ModuleIde* ×
Imports ×
Exports ×
Body⟩⟩

where

ModuleIde ==
REFERENCE × OBJIDE
Imports ==
seq₁((\mathbb{F}_1 REFERENCE) × *ModuleIde*)
Exports == \mathbb{F}_1 REFERENCE
Body == seq₁ ASSIGNMENT

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*⟨⟨*REFERENCE* ×
TYPE⟩⟩
|
valass⟨⟨*IDENTIFIER* ×
VALUE⟩⟩

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.

<i>typeofref</i> : REFERENCE → TYPE <i>valueofide</i> : IDENTIFIER → VALUE
(∀ <i>r</i> : REFERENCE <i>r</i> ∈ dom <i>typeofref</i> • (<i>r</i> , <i>typeofref</i> <i>r</i>) ∈ dom <i>typeass</i>) ∧ (∀ <i>i</i> : IDENTIFIER <i>i</i> ∈ dom <i>valueofide</i> • (<i>i</i> , <i>valueofide</i> <i>r</i>) ∈ dom <i>valass</i>)

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 =
{ **Component**₁...**Component**_n }
Component_{*i*} =
Identifier_{*i*} |
Identifier_{*i*} (Value_{*i*}) |
Value_{*i*}

where **Identifier**₁, ..., **Identifier**_n are identifiers and **Value**₁, ..., **Value**_n are optional integer values. Only the identifiers that are defined in X.208

```

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

(CCITT 1988a) can appear without associated integer values.

In the original definition of ASN.1 (CCITT 1988a) there is an optional **Identifier** in front of the sequence of components to abbreviate a part of the sequence of components. In revised versions (AS/NZS 8824.1:1998 1998) 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.

$$\begin{array}{l}
 \text{CompDef} ::= \text{ide}\langle\langle\text{IDENTIFIER}\rangle\rangle \\
 \quad \quad \quad \left| \text{both}\langle\langle\text{IDENTIFIER} \times \text{VALUE}\rangle\rangle \right. \\
 \quad \quad \quad \left. \text{val}\langle\langle\mathbb{N}\rangle\rangle \right.
 \end{array}$$

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

$$\left| \begin{array}{l}
 \text{objide} : (\text{seq}_1 \text{CompDef}) \rightarrow \text{VALUE} \\
 \hline
 \{i : \text{IDENTIFIER} \mid \\
 \quad (\text{ide } i) \text{ in } (\text{dom } \text{objide})\} \subseteq \text{IdeX}_{208}
 \end{array} \right.$$

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*.

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

5 Example

The initial fragment of the

CryptographicMessageSyntax

module of S/MIME is defined in ASN.1 as shown in Figure 3 (Housley 2002a). 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) (RSA-Security 2002), where, however, the *content* component is optional.

The assignment to the *ContentType* reference is easily represented in Z by a constant a_1 defined

$ \begin{array}{l} \text{Assignment}_1 \text{---} \\ \text{ContentType} : \text{REFERENCE} \\ a_1 : \text{ASSIGNMENT} \\ \hline a_1 = \text{typeass}(\text{ContentType}, \\ \quad \quad \quad (\text{simple objectide})) \end{array} $

Figure 4: Z representation of the first assignment in CryptographicMessageSyntax

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 \emptyset . 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 (CCITT 1988a, Kalinski 1993, Lamouth 1996, AS/NZS 8824.1:1998 1998).

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 (Long et al. 2002, Long 2003).

<i>Assignment₂</i>
$ContentInfo : REFERENCE$ $a_2 : ASSIGNMENT$
$\begin{aligned} & \exists t_1 : TYPE; s_1 : AnyType \bullet t_1 = any\ s_1 \wedge (s_1 = definedby(optidentifier\ contentType)) \wedge \\ & (\exists t_2 : TYPE; s_2 : TaggedType \bullet (t_2 = tagged\ s_2) \wedge \\ & \quad (s_2.Class = noclass) \wedge \\ & \quad (s_2.number = optnato) \wedge \\ & \quad (s_2.tagmethod = explicit) \wedge \\ & \quad (s_2.type = t_1) \wedge \\ & (\exists t_3 : TYPE; s_3 : Sequence; c_1, c_2 : Comp \bullet \\ & \quad (t_3 = sequence\ s_3) \wedge \\ & \quad (s_3.seqcomplist = \langle c_1, c_2 \rangle) \wedge \\ & \quad (c_1 = comp(optidentifier\ contentType, typeofref\ ContentType, noqual)) \wedge \\ & \quad (c_2 = comp(optidentifier\ content, t_2, noqual))) \wedge \\ & (a_2 = typeass(ContentInfo, (simple, t_3))) \end{aligned}$

Figure 5: Z representation of the second assignment in CryptographicMessageSyntax

<i>ImportsCryptographicMessageSyntax</i>
$i : Exports$
$\begin{aligned} i = \langle \langle \{ Name \}, (InformationFrameWork, \\ \quad objide\langle ide\ joint\ iso\ itu\ t, both(ds, 5), both(modules, 1), \\ \quad both(informationFrameWork, 1), val3 \rangle \rangle) \\ \langle \{ AlgorithmIdentifier, AttributeCertificate, Certificate, \\ \quad CertificateList, CertificateSerialNumber \}, \\ \quad (AuthenticationFrameWork, \\ \quad \quad objide\langle ide\ joint\ iso\ itu\ t, both(ds, 5), both(modules, 1), \\ \quad \quad both(authenticationFrameWork, 7), val3 \rangle \rangle) \rangle \end{aligned}$

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

<i>CryptographicMessageSyntax</i>
$m : MODULE$
$\begin{aligned} & (\exists b : seq_1\ Body \bullet \\ & \quad b = \langle Assignment_1.a_1, Assignment_2.a_2, \dots \rangle \wedge \\ & (\exists o : OBJIDE \bullet \\ & \quad o = objide(both(iso, 1), both(member\ body, 2), both(us, 840), \\ & \quad \quad both(rsadsi, 113549), both(pkcs, 1), both(pkcs-9, 9), \\ & \quad \quad both(smime, 16), both(modules, 0), both(cms, 1)) \wedge \\ & (\exists n : ModuleIde \bullet n = (CryptographicMessageSyntax, o) \wedge \\ & \quad m = module(n, \emptyset, ImportsCryptographicMessageSyntax.i, b))) \end{aligned}$

Figure 7: Z representation of the CryptographicMessageSyntax module

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

- Boyd, C. (1993), 'Security architectures using formal methods', *IEEE Journal on Selected Areas in Communication* 11(5), 694–701.
- CCITT (1988*a*), 'Reccomendation X.208: Specification of abstract syntax notation one (ASN.1)'.
- CCITT (1988*b*), 'Reccomendation X.509: The director—authentication framework'.
URL: http://sunsite.org.uk/public/computing/ccitt/ccitt-standards/1992/X/x509_1.asc.gz.
- Housley, R. (2002*a*), Cryptographic message syntax (RFC 3369), Technical report, Network Working Group, RSA Laboratories.
URL: <http://www.ietf.org/rfc/rfc3369.txt>.
- Housley, R. (2002*b*), Use of the RSAES-OAEP key transport algorithm in CMS — internet draft, Technical report, S/MIME Working Group, RSA Laboratories.
URL: <http://www.ietf.org/internet-drafts/draft-ietf-smime-cms-rsaes-06.txt>.
- Kalinski, B. S. J. (1993), A layman's guide to a subset of ASN.1, BER and DER, Technical report, RSA Laboratories.
- Lamouth, J. (1996), *Understanding OSI*, International Thomson Computer Press.
- Long, B. W. (2003), Formalising key distribution in the presence of trust using Object-Z, in C. Johnson, P. Montague & C. Steketee, eds, 'Proceedings of the Australasian Information Security Workshop and the Workshop on Wearable, Invisible, Context-Aware, Ambient, Pervasive and Ubiquitous Computing', Vol. 21 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society.
- Long, B. W., Fidge, C. & Cerone, A. (2002), A Z based approach to verifying security protocols, Technical Report 02-02, Software Verification Research Centre, The University of Queensland.
URL: <http://www.svrc.uq.edu.au/Publications/2002/svrc2002-002.html>.
- AS/NZS 824.1:1998 (1998), Information technology—Abstract Syntax Notation One. Part 1: specification of basic notation, Technical report, Australian/New Zealand Standard.
- RSA-Security (2002), 'Public-key cryptography standards', RSA Laboratories, Web Page.
URL: <http://www.rsasecurity.com/rsalabs/pkcs/>.
- Spivey, J. M. (1989), *The Z Notation*, International Series in Computer Science, Prentice-Hall.
- Wordsworth, J. B. (1992), *Software Development with Z — A Practical Approach to Formal Methods in Software Engineering*, International Series in Computer Science, Prentice-Hall.