INTERNATIONAL STANDARD

ISO/IEC 10165-4

First edition
1992-09-01

AMENDMENT 3
1998-12-15

Information technology — Open Systems Interconnection — Structure of management information: Guidelines for the definition of managed objects

AMENDMENT 3: Guidelines for the use of Z in formalizing the behaviour of managed objects

Technologies de l'information — Interconnexion de systèmes ouverts — Structures des informations de gestion: Partie 4: Principes directeurs pour la définition des objets gérés

AMENDEMENT 3: Principes directeurs pour l'utilisation de Z dans la formalisation du comportement de l'objet géré

Reference number
ISO/IEC 10165-4:1992/Amd.3:1998(E)

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Amendment 3 to ISO/IEC 10165-4:1992 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 33, *Distributed application services*, in collaboration with ITU-T. The identical text is published as ITU-T Rec. X.722/Amd.3.

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

INFORMATION TECHNOLOGY – OPEN SYSTEMS INTERCONNECTION – STRUCTURE OF MANAGEMENT INFORMATION: GUIDELINES FOR THE DEFINITION OF MANAGED OBJECTS

AMENDMENT 3
Guidelines for the use of Z in formalizing the behaviour of managed objects

1) Table of contents

Add the following reference to the table of contents:

Annex B – Guidelines for the use of Z in formalizing the behaviour of managed objects

2) Subclause 2.1

Add the following reference to 2.1:

– CCITT Recommendation X.731 (1992) | ISO/IEC 10164-2:1992, *Information technology – Open Systems Interconnection – Systems Management: State management function.*

3) New subclause 2.3

Add a new subclause as follows:

2.3 Additional references

– ISO/IEC 13568:[1)], *Information technology – Z specification language*.

[1)] Presently at the stage of draft.

4) New Annex B

Add a new Annex B, as follows:

Annex B

Guidelines for the use of Z in formalizing the behaviour of Managed Objects

(This annex does not form an integral part of this Recommendation | International Standard)

B.1 Introduction

This annex contains a technical guide on the use of the Z language for defining the behaviour of managed objects which support OSI management interworking. It is informative and not normative. It does not require Formal Definition Techniques (FDTs) to be used to specify MO behaviour. If FDTs are to be used, it does not require Z to be used; other languages such as SDL are also suitable. Even if Z is to be used, other ways of specifying MO behaviour are possible.

Formal specifications of MO behaviour can be directly valuable because they are clear and unambiguous. The act of producing a formal specification forces the details of the behaviour to be analysed closely. Thus, it can also be used as a tool to identify and correct ambiguities which might go undetected in a specification relying solely on natural language. For these reasons formal specification can be useful to improve behaviour specification.

This annex contains an illustrative example that demonstrates current best practice. It aims to establish a common basis and understanding of this particular formal approach which will help achieve consistency in similar developments. It should provide a useful starting point for GDMO users wishing to use Z to improve their behaviour specifications.

It is aimed at an audience familiar with the basic concepts of managed object specification using the GDMO templates, and the Z language.

For the remainder of this annex, the terms “managed object” and “MO” will be used to refer to a managed object class definition given using the GDMO templates.

B.2 Language issues

The Z notation is a formal specification notation based on set theory and predicate calculus. It possesses sufficient expressive power to be able to describe single classes of managed objects.

However, there exists no notion of encapsulation in Z. A Z specification typically consists of a model of some state and a collection of operations to modify the state. There is no method built into Z to parcel the state and its operations up into a single module and re-use it in another specification. The consequence of this becomes apparent when it becomes necessary to describe managed objects which inherit variables and behaviour from other managed object class definitions.

The effect of inheritance can be achieved by the technique of schema inclusion at the expense of some clarity. In all other respects Z is suitable for expressing single classes of managed objects.

B.3 What needs to be translated

The behaviour definitions, or parts there of, need to be translated from the informal description into Z. The extent to which the remaining parts of the GDMO templates need to be formalized depends largely on the needs of the specifier.

The GDMO templates already include a semi-formal definition of data types in ASN.1. It is possible to write a Z specification using these ASN.1 definitions as a basis for types used in the Z specification, and this saves a significant amount of work.

However, if a specification is written in this way, then it makes it a greater task for the specifier to ensure that it is syntactically correct. Without Z specifications of the ASN.1 definitions, it is not possible to use existing Z tools which provide support for checking the syntax and static semantics of a Z specification.

In summary, it is possible to improve the behaviour definitions by using Z without re-writing the ASN.1 data types, but there is a significant benefit to be gained by a full translation of the ASN.1 data types into Z. Examples of how to convert ASN.1 Basic Types into Z are provided in B.7.1.

B.3.1 From GDMO templates to Z

This subclause contains general guidelines of how to go about translating a managed object from its informal description as given in this Recommendation | International Standard into Z. It should be stressed at the outset that such a translation can only be carried out informally since a formal translation would require, as a minimum, that both the source and target languages be formal.

Moreover, as with any mapping between two distinct languages, there is bound to be some mismatch between their constructs. The problem multiplies when one of the languages happens to be informal or to include informal components.

In this subclause some of the main features of the templates defined in this Recommendation | International Standard are listed together with the ways in which they differ from or correspond to constructs in Z. In the process, general ways of resolving the mismatch or advice on how they may be tackled individually on an ad-hoc basis is offered.

This annex will concentrate on what is necessary to describe the behaviour of a managed object. Additional information on how to convert ASN.1 types is provided in B.6.

B.3.2 Datatypes

The first step is to rewrite the datatypes from this Recommendation | International Standard as Z types. ASN.1 provides the usual facilities of datatyping but its constructors are biased towards the description of datastreams communicated between systems.

In ASN.1, the type constructors are defined as forms of list. In Z, types are sets. Although it is possible to model the ASN.1 type constructors as sequences in Z, it is sometimes more natural to consider the operations available on the ASN.1 types and to map them to Z types which more clearly describe their structure. The ASN.1 sequence and set types can be mapped to Z tuples. The ASN.1 sequence-of type can be mapped to a Z sequence. The ASN.1 set-of type can be mapped to a Z set.

ASN.1 includes special support for encoding, such as type labels and default values. This does not need to be represented in Z since it doesn't affect the behaviour definition.

Subclause B.6.2 provides additional information on how to convert ASN.1 types.

B.3.3 MO Attributes

Managed objects are defined to have certain management attributes. These attributes have a datatype defined in ASN.1. They are assigned object identifiers. They also may have a matches-for property. Two ways to model such attributes have been proposed:

- simple attribute types; and
- attribute types as schemas.

The simplest is to represent the MO attribute within the MO as a Z variable with the appropriate datatype. Then separately we will need a constant definition which represents the object identifier of that attribute. This constant will be related to the actual attribute by convention only. We can use the actual fixed matches-for property when matching operations are defined for that attribute. An example of this is given in B.6.3.

It is also possible to encapsulate all these properties of an attribute in a single schema type which will then be the type of the Z variable modelling the MO attribute. Thus, the schema will include the value of the attribute as well as the object identifier and the matches-for property if any. An example of this is given in B.6.4. Where matching rules other than equality are required, it is possible to define the matches-for parameter as a Z relation over the type of the value of the attribute. This allows the formal representation of arbitrary ad-hoc matching rules, which may be important for scoping, filtering and object selection.

It is difficult to model ASN.1 type ANY in Z. One case where this is common is to give lists of attribute values. Thus, a fully formal model will probably require a Z free type combining the attribute types already defined. An example of this can be found in B.6.1 and B.6.5.

Object identifiers are formally modelled by a given set.

[OBJECTID]

B.3.4 Other Object Identifiers

Many things besides attributes also have an Object Identifier. It is convenient to introduce them all as constants in axiomatic definitions. The convention of suffixing them with “Oid” will be used. Typically such constants will be needed for classes, packages and notifications.

An example is:

packagesPackageOid : OBJECTID
allomorphsPackageOid : OBJECTID
topClassOid : OBJECTID

B.3.5 Inheritance and Compatibility

Z can be used to build inheritance hierarchies of MOs by using schema inclusion to model inheritance and specialization. This does correctly model the behaviour of an MO class and its sub-types but it fails to make explicit the strong sub-typing relationship that is really present. For that, a language that models inheritance explicitly is needed.

Thus Z can be used to define individual MOs satisfactorily, but to be able to talk about inheritance and compatibility, the additional power of a language that models inheritance explicitly is needed.

Inheritance is not supported by Z. It can be modelled by simple schema inclusion of state schemas.

The definition of MO inheritance requires sub-classes to be compatible. Unfortunately this does not require the sub-classes to be sub-types in Z. Thus, typically an MO can report its actual class. Since the actual class attribute always reports an object’s actual class, a sub-class cannot report the class of a super-class. Therefore a sub-class cannot exhibit the same behaviour as its super-class in returning the value of its actual class attribute (i.e. it is not substitutable), even if it is behaving allomorphically. Therefore managed object class sub-classing is not equivalent to Z sub-types, where a sub-type would exhibit the same behaviour as its super-type. However, a sub-class exhibits very little “unsubstitutable” behaviour.

In this way it can be seen that MO inheritance as defined in this Recommendation | International Standard allows specific behaviour in a parent which is inconsistent with the behaviour of its children. Since there is a very limited amount of this non-substitutable behaviour, an MO class can be represented by two class specifications. One captures the behaviour that any instance and also any extended MO must exhibit. The other is a specialization and captures that behaviour exhibited only by instances of the compatible class and not by any extensions. It is this latter specification that is instantiated to give the complete behaviour of an actual MO instance.

B.3.6 Packages

Many parts of the functionality of a class may be present in some individual MOs and not in others. This Recommendation | International Standard describes this process by grouping functionality into conditional packages. Then, each MO instantiates appropriate packages. In Z functionality cannot be provided in this conditional way but it is possible to make the behaviour of the MO depend on which packages are instantiated. This is straightforward because the MO must contain a management attribute called packages which lists the object identifiers of the packages actually instantiated. Thus, to model behaviour in a conditional package, the behaviour itself becomes conditional on the presence of the package identifier in the packages attribute.

B.3.7 The Class

To define an MO class it is necessary to represent its attributes and its operations. Attributes become part of the Z state schema and operations become Z operation schemas.

B.3.7.1 Attributes

The attributes of the managed object are declared in a state schema. Each attribute is given a type, which may be of a type declared in the ASN.1 part of the GDMO template, or which may use types declared in Z in a fully formal model.

B.3.7.2 The Get operation

The manager may request a Get operation to be performed on an MO. The CMISE definition of M-Get has many parameters but most of these are concerned with access control and object selection and so on. In this instance Get may be modelled at the Managed Object Boundary ignoring these issues and replacing the single Get operation by a number of Get<name> operations, where <name> is a single attribute.

B.3.7.3 The GetAll operation

A GetAll operation, which has no input, is also modelled. It returns a non-empty set of Attribute Values.

B.3.7.4 Replace operations

Set on an individual MO is requested by the CMISE M-Set operation. This specification models the Replace Operations as seen at the MO Boundary instead. In this specification, Replace Operations refers to the attribute operations set, set to default, add and remove.

The consequence of this is that a Z schema to represent each modification is specified.

B.3.7.5 Notifications

Notifications are unrequested messages sent by the MO to report events within it. However they are not modelled as operations. Instead they are modelled as outputs from operations that happen on the MO. Thus, any operation (whether invoked by the manager or internally by the resource) can generate output and if it causes a notification that notification should be part of that operation's output.

This means that the output of a Z operation schema that can cause notifications should be a *set* of notifications. Then those occasions on which it does not emit a notification can be represented by giving an empty set as output.

The data in a notification consists of an EventType which is the object identifier of its standard definition. This is followed by various information relevant to that particular notification. The object identifier can typically be defined as a constant and the particular data as a schema-type. The behaviour of the notification is included in any object which can generate the notification.

B.3.7.6 Actions

Actions are operations performed by the manager on the MO. They are very naturally represented by Z operations.

B.3.8 Specification of the system of objects

The rest of the annex describes how to represent the behaviour of a single object. When considering object creation/deletion, name bindings, containment and naming, it is necessary to describe the state of the system where the objects reside. Object creation and deletion can be represented by a change of state of this system. Name binding and containment can be represented by a relation over the set of objects. Naming can then be defined in terms of this relation.

B.4 An example

In this subclause, example definitions for the MO class top and State Management attributes are given. Since the main concern of this guide is the modelling of behaviour, the creation of Z types from ASN.1 types is not presented in this subclause. A full formal definition is given in B.7.

B.4.1 top

The first class to be defined is *top*, which is the ultimate parent (in the inheritance hierarchy) of all MOs.

top has four management attributes, *objectClass*, *packages*, *allomorphs* and *nameBinding*. *objectClass* holds the object identifier of the class, while *packages* holds the object identifiers of the packages it instantiates. *nameBinding* holds the object identifier of the name binding used to instantiate the object and *allomorphs* holds the object identifiers of the classes to which the object can be allomorphic. Since management attributes can be in packages, the attributes present in MOs of a given class can vary. This is modelled by including an additional modelling attribute called *attributes*, which holds the object identifiers of the attributes that are actually instantiated in the individual MO. Note that all the attributes present in *top* are fixed for the lifetime of any individual MO.

Z does not explicitly model interfaces, and so it is not possible to formally define which operations are invoked internally or externally by the manager.

TopState

allomorphs: $\mathbb{F}$ *OBJECTID* *objectClass: OBJECTID* *nameBinding: OBJECTID* *packages:* $\mathbb{F}$ *OBJECTID* *attributes:* $\mathbb{F}$ *OBJECTID*
$\{objectClassOid, nameBindingOid\} \subseteq attributes$ $allomorphsPackageOid \in packages \Rightarrow allomorphsOid \in attributes$ $packagesPackageOid \notin packages$ $packages \neq \varnothing \Rightarrow packagesOid \in attributes$

attributes is not an MO attribute but a new state component defined for convenience. It lists the MO attributes an MO includes. Thus, the invariant enforces that it must contain the object identifiers of the appropriate attributes as described in B.3.3 (and defined in B.7.4). *objectClass* and *nameBinding* are mandatory. *packages* is present if any registered package is instantiated apart from *packagesPackage*. In this case this means *allomorphsPackage*.

The operation *TopGetNameBinding* interrogates the MO and returns the value of the *nameBinding* attribute, without changing *TopState*. *TopGetNameBinding* is invoked by the manager.

TopGetNameBinding

Ξ *TopState* *result!: OBJECTID*
result! = nameBinding

The operations *TopGetAllomorphs*, *TopGetObjectClass* and *TopGetPackages* have not been defined here. Note that there is no operation to get *attributes*, since *attributes* is not a real MO attribute as specified in the GDMO template.

TopGetAll gets all the attribute values of an object. It always returns values for *objectClass* and *nameBinding*. If conditional packages or allomorphs are present, then it gets those too. *TopGetAll* is invoked by the manager.

TopGetAll

Ξ *TopState* *result!:* $\mathbb{P}$ *AttributeValues*
attributes = # result! *ObjectClassValue objectClass* $\in$ *result!* *NameBindingValue nameBinding* $\in$ *result!* *PackagesOid* $\in$ *attributes* $\Rightarrow$ *packagesValue packages* $\in$ *result!* *AllomorphsOid* $\in$ *attributes* $\Rightarrow$ *allomorphsValue allomorphs* $\in$ *result!*

TopEventReport is a way to model notifications. *TopEventReport* occurs spontaneously and represents the way event reports are not controlled by the manager.

TopEventReport

Ξ *TopState* *notification!: EventInfo*

B.4.2 StateManagement class

This class does not reflect any specific MO class. Instead it reflects the behaviour of any object which includes any of certain standard attributes: *administrativeState*, *operationalState*, and *usageState*. It is more convenient within this framework to understand this inclusion as inheritance and it does serve as a useful example.

The state schema includes the *TopState* definitions and predicates, and defines some additional variables and predicate conjunctions.

StateManagementState

TopState
administrativeState:
AdministrativeState
operationalState: OperationalState
usageState: UsageState

$operationalState = disabled \Rightarrow usageState = idle$
$administrativeState = locked \Rightarrow usageState = idle$
$usageState = idle \Rightarrow administrativeState \neq shuttingDown$

State Management inherits the operations from Top. Although there is no mechanism built into Z to inherit operations, it is straightforward to redefine the operations in terms of the new state. The predicate part of *StateManagementState* follows from the definition of the State Management function in CCITT Rec. X.721 (1992) | ISO/IEC 10165-2:1992 and CCITT Rec. X.731 (1992) | ISO/IEC 10164-2:1992.

The operation *SMGetNameBinding* can be easily defined, since it has no effect upon the new state variables declared in *StateManagementState*. The definition of *TopGetNameBinding* can be re-used:

SMGetNameBinding

TopGetNameBinding
Ξ *StateManagementState*

Definitions for operations to get the other attributes of *StateManagementState* have also been omitted from this example. The operations *SMGetAllomorphs*, *SMGetObjectClass* and *SMGetPackages* can re-use the definitions from Top as for *SMGetNameBinding*. New operations will need to be defined for *GetSMAdministrativeState*, *GetSMOperationalState* and *SMGetUsageState*. *SMEventReport* may also be re-used.

The *SMGetAll* schema also makes use of an operation defined on *TopState*. It includes the definition of *TopGetAll* and strengthens the postcondition.

SMGetAll

Ξ *StateManagementState*
TopGetAll

$administrativeStateOid \in attributes$
$\quad \Rightarrow administrativeStateValue\ administrativeState \in result!$
$OperationalStateOid \in attributes$
$\quad \Rightarrow operationalStateValue\ operationalState \in result!$
$UsageStateOid \in attributes$
$\quad \Rightarrow usageStateValue\ usageState \in result!$

The *SMReplaceAdministrativeState* operation describes behaviour specific to the State Management class whereby the administrative state is replaced by another value supplied as an input. Depending on the state of the object when the operation is carried out, the usage state may also be changed. The operational state is not altered by the operation.

SMReplaceAdministrativeState

$$
\begin{array}{|l}
\Delta StateManagementState \\
\Xi TopState \\
input?: AdministrativeState \\
\hline
administrativeState' \in \\
\quad \textbf{IF}\ usageState \neq idle \\
\quad \textbf{THEN}\ \{\ unlocked \mapsto unlocked, locked \mapsto locked, \\
\qquad\qquad shuttingDown \mapsto locked, locked \mapsto shuttingDown, \\
\qquad\qquad shuttingDown \mapsto shuttingDown\} (\!|\{\ input?\}|\!) \\
\quad \textbf{ELSE}\ \{unlocked \mapsto unlocked,\ \ locked \mapsto locked, \\
\qquad\qquad shuttingDown \mapsto locked\}\ \ (\!|\{\ input?\}|\!) \\
administrativeState' = locked \Rightarrow\ usageState' = idle \\
administrativeState' \neq locked \Rightarrow\ usageState' = usageState \\
operationalState' = operationalState \\
\hline
\end{array}
$$

The behaviour specified in the predicate part of the schema is a formalization of the informal description in CCITT Rec. X.731 | ISO/IEC 10164-2. For completeness, operations to replace the operational state and the usage state should also be defined.

Finally there are a number of other operations which describe behaviour specific to the State Management class. These operations are not listed here, although they may be found in B.7. These operations include *SMCapacityDecrease*, *SMCapacityIncrease*, *SMDisable*, *SMEnable*, *SMNewUser* and *SMUserQuit*.

B.4.3 Instantiable classes

Neither of the classes described above can be instantiated. The procedure that has been followed can be continued. *StateManagement* can be re-used to define a class called *CIRCUIT*, which in turn can be used to define *ECIRCUIT* and hence the instantiable class *ActualECircuit*.

This has been omitted from the guide, since the procedures are exactly the same as those that have outlined and repetition adds nothing.

B.5 Outstanding Issues

In this subclause, the main issues encountered in the course of the translation from GDMO-based managed objects specifications to Z are listed. Where an issue relates to Z not having a corresponding construct for a particular feature of managed objects specification, the proposed informal treatment used in this annex will also be included.

B.5.1 Behaviour Definition in Managed Objects

In the templates, the term 'behaviour definition' is used for almost all entities whether they are data or processes. In the latter case, it may include information about the actual behaviour (in the strict sense), or just static information about the entity such as its intended use, or both. When translating, one needs to analyse the text which comes under this heading, and extract the relevant behavioural information for the entity concerned. This behavioural information will be used in the formal translation, while the actual text may be included as a comment inside the Z specification.

B.5.2 Internal operations in Z

An internal operation in a managed object represents the case where a notification is emitted spontaneously (with no management invocation involved). Internal operations are also a desired feature of many other systems. Currently, in Z, this feature is represented informally by a comment in the natural language text which is an important feature of any well-written Z specification.

B.5.3 Abstract classes in Z

Sometimes it is useful to identify abstract classes: i.e. classes with no instantiations of their own. Some MO classes (like *top*) cannot be instantiated. It would be helpful to be able to show which parts of the corresponding Z specifications represent classes that can be instantiated. This is taken care of by informal annotation at present.

B.5.4 PARAMETER semantics

The incorporation of PARAMETER semantics into objects is not considered in this Recommendation | International Standard.

B.6 Converting ASN.1 Datatypes to Z

Issues for translation will be described for each ASN.1 constructor in turn.

B.6.1 Simple types

ASN.1 includes some simple types which are built-in. These do have a standardized structure but it is usually not interesting in the context of these specifications and so they can mostly be represented as given sets. There are a wide variety of character string types:

[NUMERICSTRING, PRINTABLESTRING, TELETEXSTRING,
VIDEOTEXSTRING,VISIBLESTRING, IA5STRING,
GRAPHICSTRING, GENERALSTRING]

Two of these have synonyms:

T61STRING == TELETEXSTRING
ISO64STRING == VISIBLESTRING

Of the other simple types, Integer can be represented by $\mathbb{Z}$, Boolean and Null by free types:
Boolean ::= btrue | bfalse
Null ::= null

Note that these free types also define the value notation for these types.

Real, Bit String, and Octet String, can usually be taken to be given sets (though it may sometimes be necessary to structure the Bit and Octet String types).

[REAL,BITSTRING,OCTETSTRING]

This Recommendation | International Standard also describes another special type which will be provided as a given set.

[OBJECTID]

Here *OBJECTID* represents an ASN.1 Object Identifier.

Object Identifiers are in fact non-empty sequences of $\mathbb{N}$ and it may be convenient to model them as such, instead of as a given set. In this case some thought must be given to an appropriate value notation.

There are also some “useful” types which are defined in ASN.1 within the ASN.1 standard. Thus, although they could be defined in terms of the other ASN.1 constructs, it is again convenient to provide them as given sets.

[GENERALIZEDTIME, UTCTIME, OBJECTDESCRIPTOR, EXTERNAL]

Any

ASN.1 allows a special type ANY which can contain any other ASN.1 type at all. Such a type is not allowed within Z and it would be difficult to extend it to include one. However given any known set of types, it is possible to define a Z free type which can include any of those other types. An alternative strategy is to define ANY as a given set for typechecking purposes. This is satisfactory as long as nothing else is done with it. The type *AttributeValues* usually replaces *ANY*. This is defined below.

B.6.2 Structured types

Other types in ASN.1 are built up by constructors.

Set

ASN.1 Sets can be represented as either tuples or schemas in Z. Z tuples do not allow components to be named and so schemas may be more appropriate. However the Z value notation for schemas is less convenient. Tagging is neither needed nor possible in Z since the components of the “set” can always be discriminated either by their position in a tuple or their component name in a schema.

Components in this and other structured types can be *OPTIONAL*. This can be represented in Z by augmenting the type of the optional component with a special "absent" value. *DEFAULT* values cannot be conveniently represented as a feature of a datatype. It is possible to represent behaviour implied by the default within any operations on that data.

Sequence

ASN.1 Sequences can be modelled in exactly the same way as ASN.1 Sets since the only difference is that there is an explicit order. Because this is the case, it could be argued that a tuple is more appropriate but schemas can also be used.

Set-of

ASN.1 Set-of types are actually bags and can be defined in Z as such. It should be noted that the MIM explicitly requires all such bags to be treated as sets and so it is in fact more appropriate to model the type as a Z set.

When subtyping of an ASN.1 type is required, it is usually necessary to add a predicate constraint to the type. In some cases, for example integer or schema sub-types, this can be done in the type definition itself. Otherwise (for example bag sub-types) the constraint must be applied to variables defined to be of that type.

Sequence-of

ASN.1 Sequence-of types can be conveniently modelled as Z sequences.

Choice

ASN.1 Choice types are straightforward enumerations and can be modelled by Z free types.

This type introduces a serious scoping problem. Within ASN.1 the constructors within a Choice are local to that type. Thus, a single constructor name can be used in more than one enumeration. In Z these names are global and cannot be re-used. This problem must usually be resolved by changing the names of the constructors, typically by prefixing them with the name of their type.

A similar problem arises when ASN.1 Types are generated that are synonyms for Integer (say) but with named values. These named values are local to the synonym type in ASN.1 but global synonyms for integers in Z. Again this must be fixed by changing the names of the constructors.

B.6.3 Simple attribute types

The simplest is to represent the MO attribute within the MO as a Z variable with the appropriate datatype. A constant definition which represents the *OBJECTID* of that attribute will be needed. When matching operations are defined for that attribute, the actual standardized value of the matches-for parameter can be used.

Thus consider the MO attribute administrativeState. We will have defined a type:

AdministrativeState ::= unlocked | locked | shuttingDown

A constant to represent the attribute's Object Identifier can be defined:

administrativeStateOid : OBJECTID

The actual value of the identifier can be presented a constraint on that axiomatic definition.

Then within the MO a state variable will be defined:

MOState

administrativeState: AdministrativeState

This solution is straightforward and convenient but does require lists of *OBJECTID* axiomatic definitions. It also makes the link between the name of the attribute and its *OBJECTID* purely syntactic. The convention of suffixing with *Oid* has been adopted.

B.6.4 Attribute types as schemas

It is also possible to encapsulate these features of an attribute in a single schema type which will be the type of the Z variable modelling the MO attribute:

AdministrativeStateType

value:AdministrativeState *Oid:OBJECTID*
Oid = ⟨4, 3, 19, 27, 1, 3⟩

It is important to provide a value for the *OBJECTID* here since it is necessary to imply that it cannot be changed even though the value can.

A structure for *OBJECTID* has not been defined, but writing:

$OBJECTID == \mathbf{seq_1}\ \mathbb{N}$

is one possibility that would make the previous schema make sense. This schema could also hold the matches-for parameter if it was thought important to represent this within the specification.

Then the MO would contain an attribute with this type:

MOState

administrativeState : AdministrativeStateType

Reference to its value or its Oid would be made via component selection as in *administrativeState.value*.

This technique conveniently gives semantics to the connection between an attribute and its *OBJECTID*. However, it may seem strange to specification readers that the Oid is present in the MO state even though it cannot change (and is in fact a global constant known at specification time).

B.6.5 AttributeValues type

As mentioned above, it is difficult to model ASN.1 type ANY in Z. One case where this is common is to give lists of attribute values. A Z free type definition combining the attribute types already defined will be required. This approach works as long as the set of attributes in use is fixed at specification time. Then, typically, it will look something like:

AttributeValues ::= administrativeStateValue ⟨⟨AdministrativeState ⟩⟩|
objectClassValue ⟨⟨OBJECTID ⟩⟩|
nameBindingValue ⟨⟨OBJECTID ⟩⟩|
packagesValue ⟨⟨ℙ OBJECTID⟩⟩|
allomorphsValue ⟨⟨ℙ OBJECTID ⟩⟩|
operationalStateValue ⟨⟨OperationalState ⟩⟩|
usageStateValue ⟨⟨UsageState ⟩⟩

B.7 A full example

This subclause presents the full formal model on which the example in B.4 is based. It is presented in the traditional Z style of declaration before use: that is, the type definitions converted from ASN.1 appear at the start and the behaviour definitions appear at the end.

One point of specification style is worth commenting on. The definitions *AttributeValues* and *OBJECTINSTANCE* are mutually recursive. This is technically illegal in Z, and so the following has been done to permit the definition. *AttributeValues* has been introduced as a given set. *OBJECTINSTANCE* is then defined using the given set *AttributeValues*. This definition of *OBJECTINSTANCE* is then used to introduce the restrictions that *AttributeValues* is allowed to take. The proof obligation to show that such sets actually exist has been discharged, but is not presented.

B.7.1 ASN.1 basic types

[NUMERICSTRING, PRINTABLESTRING, TELETEXSTRING, VIDEOTEXSTRING]
[VISIBLESTRING, IA5STRING, GRAPHICSTRING, GENERALSTRING]

T61STRING == TELETEXSTRING

ISO64STRING == VISIBLESTRING

Boolean ::= btrue | bfalse

Null ::= null

[REAL,BITSTRING,OCTETSTRING]

[OBJECTID]

[ANY]

[GENERALIZEDTIME,UTCTIME,OBJECTDESCRIPTOR,EXTERNAL]

B.7.2 MO Attributes

The following given set is a placeholder for a more complex and complete free type definition given incrementally as each new class is defined.

[AttributeValues]

AttributeValuesOptional ::= present ⟨⟨AttributeValues ⟩⟩ | absent

RelativeDistinguishedName == AttributeValues

*RDNSequence == **seq** RelativeDistinguishedName*

DistinguishedName == RDNSequence

OBJECTINSTANCE ::= distinguishedName⟨⟨DistinguishedName⟩⟩ | nonSpecificForm⟨⟨ℕ⟩⟩
| localDistinguishedName⟨⟨RDNSequence⟩⟩

B.7.3 Notifications

ProbableCause == OBJECTID

SpecificIdentifier ::= globalvalue⟨⟨OBJECTID⟩⟩ | localValue⟨⟨ℕ⟩⟩

SpecificProblems == ℙ SpecificIdentifier

SpecificProblemsOptional ::= sPPresent⟨⟨SpecificProblems⟩⟩ | sPAbsent

PerceivedSeverity ::= indeterminate | critical | major | minor | warning | cleared

BackedUpStatus == Boolean

BackedUpStatusOptional ::= bUSPresent⟨⟨BackedUpStatus⟩⟩ | bUSAbsent

ObjectInstanceOptional ::= oIPresent⟨⟨OBJECTINSTANCE⟩⟩ | oIAbsent

TrendIndication ::= lessSevere | noChange | moreSevere

TrendIndicationOptional ::= tIPresent⟨⟨TrendIndication⟩⟩ | tIAbsent

ObservedValue ::= int ⟨⟨ℕ⟩⟩ | real⟨⟨REAL⟩⟩

ObservedValueOptional ::= oVPresent ⟨⟨ObservedValue⟩⟩ | oVAbsent

ThresholdLevelInd ::=
up ⟨⟨ObservedValue ×ObservedValueOptional ⟩⟩
| down ⟨⟨ObservedValue × ObservedValueOptional ⟩⟩

ThresholdLevelIndOptional ::= tLIPresent ⟨⟨ThresholdLevelInd ⟩⟩ | tLIAbsent

ArmTimeOptional ::= aTPresent⟨⟨GENERALIZEDTIME⟩⟩ | aTAbsent

ThresholdInfo ==
OBJECTID × ObservedValue × ThresholdLevelIndOptional × ArmTimeOptional

ThresholdInfoOptional ::= thIPresent⟨⟨ThresholdInfo⟩⟩ | thIAbsent

NotificationIdentifier == ℕ

NotificationIdentifierOptional ::= nIPresent ⟨⟨NotificationIdentifier⟩⟩ | nIAbsent

CorrelatedNotifications == ℙ((ℙNotificationIdentifier) × ObjectInstanceOptional)

CorrelatedNotificationsOptional ::= cNPresent ⟨⟨CorrelatedNotifications ⟩⟩ | cNAbsent

AttributeValueChangeDefinition == ℙ(OBJECTID × AttributeValuesOptional × AttributeValues)

AttributeValueChangeDefinitionOptional ::=
aVCDPresent ⟨⟨ AttributeValueChangeDefinition ⟩⟩ | aVCDAbsent

MonitoredAttributes == ℙOBJECTID

MonitoredAttributesOptional ::= mAPresent ⟨⟨ MonitoredAttributes⟩⟩ | mAAbsent

ProposedRepairActions == ℙSpecificIdentifier

ProposedRepairActionsOptional ::= pRAPresent ⟨⟨ProposedRepairActions⟩⟩ | pRAAbsent

AdditionalTextOptional ::= adTPresent ⟪GRAPHICSTRING⟫ | aDTAbsent

ManagementExtension == OBJECTID × Boolean × ANY

AdditionalInformation == ℙ ManagementExtension

AdditionalInformationOptional ::= aIPresent ⟪AdditionalInformation ⟫ | aIAbsent

SourceIndicator ::= resourceOperation | managementOperation | sIUnknown

SourceIndicatorOptional ::= sIPresen ⟪SourceIndicator ⟫ | sIAbsent

AttributeIdentifierList == ℙ OBJECTID

AttributeIdentifierListOptional ::= atIPresent ⟪AttributeIdentifierList ⟫ | atIAbsent

Attribute == OBJECTID × AttributeValues

AttributeList == ℙ Attribute

AttributeListOptional ::= aLPresent ⟪AttributeList ⟫ | aLAbsent

AlarmInfo

probableCause: ProbableCause
specificProblems: SpecificProblemsOptional
perceivedSeverity: PerceivedSeverity
backedUpStatus: BackedUpStatusOptional
backUpObject: ObjectInstanceOptional
trendIndication: TrendIndicationOptional
thresholdInfo: ThresholdInfoOptional
notificationIdentifier: NotificationIdentifierOptional
correlatedNotifications: CorrelatedNotificationsOptional
stateChangeDefinition: AttributeValueChangeDefinitionOptional
monitoredAttributes: MonitoredAttributesOptional
proposedRepairActions: ProposedRepairActionsOptional
additionalText: AdditionalTextOptional
additionalInformation: AdditionalInformationOptional

AttributeValueChangeInfo

sourceIndicator: SourceIndicatorOptional
attributeIdentifierList: AttributeIdentifierListOptional
attributeValueChangeDefinition: AttributeValueChangeDefinitionOptional
notificationIdentifier: NotificationIdentifierOptional
correlatedNotifications: CorrelatedNotificationsOptional
additionalText: AdditionalTextOptional
additionalInformation: AdditionalInformationOptional

ObjectInfo

sourceIndicator: SourceIndicatorOptional
attributeList: AttributeListOptional
notificationIdentifier: NotificationIdentifierOptional
correlatedNotifications: CorrelatedNotificationsOptional
additionalText: AdditionalTextOptional
additionalInformation: AdditionalInformationOptional

RelationshipChangeInfo

sourceIndicator: SourceIndicatorOptional
attributeIdentifierList: AttributeIdentifierListOptional
relationshipChangeDefinition: AttributeValueChangeDefinitionOptional
notificationIdentifier: NotificationIdentifierOptional
correlatedNotifications: CorrelatedNotificationsOptional
additionalText: AdditionalTextOptional
additionalInformation: AdditionalInformationOptional

SecurityAlarmInfo

notificationIdentifier: NotificationIdentifierOptional
correlatedNotifications: CorrelatedNotificationsOptional
additionalText: AdditionalTextOptional
additionalInformation: AdditionalInformationOptional

StateChangeInfo

sourceIndicator: SourceIndicatorOptional
attributeIdentifierList: AttributeIdentifierListOptional
stateChangeDefinition: AttributeValueChangeDefinitionOptional
notificationIdentifier: NotificationIdentifierOptional
correlatedNotifications: CorrelatedNotificationsOptional
additionalText: AdditionalTextOptional
additionalInformation: AdditionalInformationOptional

EventInfo ::= attributeValueChange «AttributeValueChangeInfo»
| *communicationsAlarm «AlarmInfo»*
| *environmentalAlarm «AlarmInfo»*
| *equipmentAlarm «AlarmInfo»*
| *integrityViolation «SecurityAlarmInfo»*
| *objectCreation «ObjectInfo»*
| *objectDeletion «ObjectInfo»*
| *operationalViolation «SecurityAlarmInfo»*
| *physicalViolation «SecurityAlarmInfo»*
| *processingError «AlarmInfo»*
| *qualityOfServiceAlarm «AlarmInfo»*