Translation of CORBA IDL to Z

Chris Taylor, Eerke Boiten and John Derrick
Computing Laboratory, University of Kent,
Canterbury, CT2 7NF, UK
C.N.Taylor-1,E.A.Boiten,J.Derrick@ukc.ac.uk

Abstract

We provide a translation from CORBA IDL to the Z specification notation. Our main motivation is to integrate CORBA IDL into an existing multi-language framework for viewpoint specification and consistency checking. However, the translation could also serve as the basis for a reverse translation from a subset of Z into IDL, and for a translation into Z from IDL specifications augmented with Z annotations that express behavioural constraints not expressible in IDL itself.

Keywords. CORBA IDL, Z, translation, specification languages, signatures, viewpoints, consistency.

1 Introduction

Formal specification methods should be used in areas that require extra precision, and where practitioners have accepted some degree of formalisation and standardisation. One such area is distributed systems, for which the ISO's Open Distributed Processing (ODP) framework [5] has a developing international standard. ODP is a particular example of a framework for describing complex systems from several different *viewpoints* [4], each a partial specification from a different perspective.

In the viewpoint method, the problem arises of ensuring consistency between different viewpoints. The work described in this paper relates to our earlier work [2, 3], concerning formal approaches to viewpoints and viewpoint consistency, particularly in relation to ODP. The general approach adopted in this work is to say that partial specifications are mutually consistent if and only if they have a *common refinement*. The problem of establishing consistency is then reduced to one of defining appropriate refinement rules that allow such a common refinement to be found, a process sometimes referred to as *unification*.

In terms of current practice regarding distributed systems, the Common Object Request Broker Architecture (CORBA) [6] has become a de facto standard. Supported by a large consortium of computing companies, CORBA provides a platform-neutral object-oriented framework for the specification and construction of distributed systems. Such systems typically consist of heterogenous components, written in various programming languages, and running on a variety of hardware and software platforms.

By using a neutral Interface Description Language (IDL) [6]), CORBA allows diverse components to be integrated smoothly into a single distributed system. IDL is not a programming language, but a specification language, with a formal syntax, but no formal semantics. It allows the public interfaces of various objects in distributed components to be defined — in terms of signatures showing visible attributes and operations — without specifying how, or in what programming language, those objects are implemented. The types of attributes and of operation parameters and return values can be specified, but nothing else, e.g. it is not possible to specify the preconditions or postconditions of operations, or to specify constraints on their temporal ordering.

The CORBA architecture provides the necessary supporting infrastructure for the actual software of a distributed system — involving remote procedure calls, etc. — to be developed from an IDL specification of the distributed components. Since the components are specified only via their interfaces, the implementations of individual components can be changed without affecting the behaviour of the overall distributed system, as long as the interfaces do not change. Although CORBA is an object-oriented architecture, individual system components need not be written in object-oriented programming languages — all that is required is that external interactions with such non-object-oriented components (for example, relational databases) can be made to appear like interactions with virtual objects, in the object-oriented sense.

Because of the limited expressiveness of IDL, the Z specifications produced by the translation described in this paper are 'skeletal', with almost no behavioural constraints on operations, apart from minimal assumptions about the effect of implicit 'get' and 'set' operations on the state attributes to which they apply. All other operations are just defined by their signatures, in terms of inputs and outputs of certain types. However, translation into such a sparse form of Z is still useful, because:

- The specifications that it produces can be developed into more detailed and more behaviourally constrained specifications using Z refinement techniques [11].
- It can be applied to viewpoint unification by finding a common refinement of the translated specification and Z specifications expressing other viewpoints.
- It could serve as the basis for an enhanced translation process in which Z constraints are expressed as comments in an IDL specification, and are converted into actual behavioural constraints in the Z output file, so that the Z annotations become part of the formal semantic content of the specification, in addition to the 'skeletal' translation resulting from the IDL syntax alone.

A general reverse translation from Z to IDL is not possible, because of the much greater expressive power of Z, and the uncertainty of how to interpret arbitrarily named Z schemas in terms of IDL. In the translation from IDL to Z, schema naming conventions and textual markers are used to encode the structure of the IDL, so that one Z construct — the schema — can be used variously to represent IDL state attribute sets, operations, and structured datatypes. A reverse translation — from a certain subset of Z specifications only — would have to indicate the intended structure of the IDL output, either by using the same text markers and schema naming conventions in reverse, or by using a semi-automated translation process that is given structural information by the user in some other form. The latter approach would be more practical, because in the context of viewpoint specification, it would be inconvenient for Z specifications representing other viewpoints to be forced to use the long schema names required by the naming conventions of the IDL to Z translation. Such a reverse translation might be useful in developing a CORBA implementation of a system from a Z specification.

The rest of the paper is organized as follows. ODP viewpoints are briefly described in the remaining subsection of the introduction. Section 2 discusses the usefulness of the form of 'skeletal' Z produced by the translation. Section 3 outlines the actual translation process, and describes the main syntactic constructs of IDL, and how they are translated into Z. Section 4 summarises the paper, and is followed by an Appendix that gives an extended example of an IDL to Z translation.

1.1 ODP Viewpoints

The ODP architecture [5] specifies five viewpoints, as follows:

- *Enterprise Viewpoint* — focuses on the overall scope, purpose, and policies of the system.
- *Information Viewpoint* — specifies at a high level the information involved in the system, and how it is processed.
- *Computational Viewpoint* — a functional decomposition of the system into objects that interact via specific interfaces.
- *Engineering Viewpoint* — a specification of the mechanisms and functions needed to support interaction between the distributed objects of the system.
- *Technology Viewpoint* — concerns the concrete technological infrastructure of a system, i.e. the particular hardware and software components involved, and how they are interrelated.

The viewpoints are informally defined in natural language — albeit at much greater length than given above — and so are inevitably somewhat vague. ODP does not prescribe particular software or hardware, or particular specification notations for each viewpoint. However, some notations are more appropriate than others for the various viewpoints, and in practice, different viewpoints may often be expressed in different notations. For example, CORBA IDL is well suited to the Computational Viewpoint; and the formal specification language Z (see [8]) is well suited to the Information Viewpoint. Thus in relation to ODP, an IDL to Z translation is helpful as regards the unification of the Computational and Information viewpoints, since it allows a Computational Viewpoint specification in IDL to be converted into a form suitable for unification with an Information Viewpoint specification in Z.

Z is not the only language suitable for the Information Viewpoint: for example, Object-Z [7], an object-oriented variant of Z, would also be suitable, and might at first sight seem more appropriate, given that IDL is object-oriented. However, standard Z is preferred here, because the refinement techniques for Z are much better developed than those for Object-Z.

2 Skeletal Z

The target language for our translation is a minimal, "skeletal" form of Z. Because of its refinement (and hence unification) properties, this is still a very useful subset of Z.

The traditional Z "states and operations" specification style defines abstract data types (ADTs) of the form $(State, Init, \{Op_i\}_{i \in I})$ (where $STATE$ is a set of states, $Init$ is an initialisation condition, and the operations Op_i are indexed by a set I) and relations of data refinement between them. Such ADTs contain both a "signature" and behavioural information. To remove the latter, we can construct the *signature* of an ADT defined by the following schemas

$$\begin{array}{|l} \hline State_\Sigma \\ \hline \end{array} \qquad \begin{array}{|l} \hline Op_{i,\Sigma} \\ \Delta State_\Sigma \\ ?Op_i;\ !Op_i \\ \hline \text{false} \\ \hline \end{array}$$

$$\begin{array}{|l} \hline Init_\Sigma \\ State'_\Sigma \\ \hline \end{array}$$

where $?Op_i$ and $!Op_i$ represent the declarations of inputs and outputs of Op_i, respectively. This signature, allowing any initial state, and any outcome of an operation, has the property that any ADT of the given form is a refinement of it. (If we were assuming the guarded interpretation of operations, as in Object-Z, where preconditions may *not* be weakened, we would additionally need to include one boolean variable for each operation in the state, representing that operation being enabled.) For that reason, it is automatically consistent with any ADT of that same form. In a viewpoints approach, it represents exactly the structure of the ADT, without behavioural restrictions.

A slight modification of the standard Z ADT, called the *grey box datatype* in [1] provides, in addition to the standard ADT components, collections of state variables which are observable and modifiable. An interpretation of grey box datatypes is given, which adds "get" operations for all observable components, and "set" operations for all writable components. Signatures of such grey boxes could be given as grey box data types, or in the interpretations as standard Z ADTs. In the latter case, the relevant state components should be included in the signature's state as well. Such a signature would indeed be refined by any "grey box" of the same structure according to the refinement rules for grey box datatypes as given in [1].

3 IDL Syntax and Its Translation to Z

A typical IDL specification describes the public interfaces — i.e. the visible attributes and operations, and the type restrictions on them — of a collection of object classes. It may also define structured record types, variables, and constants. Some classes may be related to others by inheritance. Related classes and other type definitions etc. may be grouped into named modules.

An IDL specification is used to specify the abstract structure of a distributed system, in terms of the external interfaces of the object classes involved, without specifying how the objects are implemented. An example of an IDL specification, consisting of two modules, with some interface and structured datatype definitions, is shown in the Appendix.

3.1 Outline of the Translation Process

This paper is mainly concerned with an abstract specification of the translation from IDL to Z, rather than with a detailed account of the implementation. However, it is helpful to give a brief outline of the actual translation process, as follows:

- The input is an IDL specification, contained in one or more plain-text files.
- In the initial stage of the translation, the IDL input is parsed into data structures in the functional programming language Haskell (see [10]), using Glasgow University's *HDirect* IDL parsing tool.
- The Haskell data structures from the initial stage are used to generate a single plain-text file containing Latex mark-up code.
- Latex is run on the Latex mark-up code to produce a Postscript file for printing.

The aim is a fairly simple translation that preserves the linear order of syntactic units in the input file, making it easy to correlate the output with the input, and facilitating reverse translation back into IDL. If such a reverse translation is required to reproduce the original file structure, then the single Z file produced initially will need to contain text markers indicating the boundaries of the IDL files from which it was derived.

3.2 IDL Syntactic Constructs

Formally, an IDL *specification* is a sequence of one or more IDL *definitions*, each of which is one of the following:

- *Type definition* — a datatype definition (*structured*, *enumeration*, or *abbreviation* type).
- *Variable declaration* — declares a variable of a specific type.
- *Constant declaration* — declares an expression of fixed value.
- *Module definition* — a named group of related definitions.
- *Interface definition* — like the external view of an OO-class, i.e. its public attributes and methods.
- *Exception definition* — user-defined if declared explicitly in a specification (CORBA also has standard exceptions, which are not declared explicitly).

The current translation is of a subset of IDL, omitting exception definitions (the treatment of which is the subject of further work). The remaining constructs are now described in the following subsections, following a subsection on the handling of identifiers and comments. The Z translations of various IDL constructs are shown as they would appear in the final Postscript file, rather than in the Latex mark-up form produced by the translation from IDL.

3.3 Identifiers and Comments

Neither IDL nor Z impose very strong formal syntactic restrictions on identifiers, i.e. the strings of characters used as the names of types, variables, and so on. However, certain identifiers are reserved for common types — e.g. `char`, `int`, etc. in IDL, and $\mathbb{Z}$, $\mathbb{N}$, etc. in Z. Also, it is common in Z to follow certain conventions for distinguishing different kinds of identifier, so that for example, given type names are upper-case, state attribute names are lower-case, operation and schema names are upper-case initial, and so on.

Two possible options for the translation of IDL identifiers to Z are:

1. Translate mostly 'as found' — i.e. make no case modifications or changes to the names of standard types. Just add prefixes corresponding to module names, where necessary; add ? and ! to input and output variable names; and include abbreviations for standard types (e.g. $int == \mathbb{Z}$) in a general 'boilerplate' (see section 2.2).

2. Respect all Z conventions — modify case if necessary, and change standard names in specification, e.g. change `int` to $\mathbb{Z}$.

Of these two, option 1 is used, because it is far simpler, prevents potential name clashes arising from case changes, and makes it much easier to compare the Z output produced with the IDL source, and to reverse translate it back into IDL.

In IDL, `//` indicates the start of a single-line comment, i.e. any text to the right of it is treated as a comment; and the strings `/*` and `*/` can be used to enclose a single- or multiple-line comment, as shown below.

```
/* IDL comment over one
or more lines. */

// IDL comment on one line.
```

The *HDirect* parsing tool ignores IDL comments, so they cannot be passed to the Z output file. However, 'text wrappers', marking the start and end of the translations of various IDL constructs, will be included in the output, helping to compensate for the lack of comments. Such wrappers will also greatly assist reverse translation back into IDL, since they will identify explicitly the IDL syntactic constructs from which each 'chunk' of the Z output has been derived. This is important because several different IDL constructs will be translated into the same Z construct, i.e. the Z schema.

3.4 IDL Types, Variables, and Constants

IDL has a small number of standard given typenames, including: `int`, `char`, `string`, `long`, `float`. As explained in the previous section, the identifiers for these types are not changed in the translation to Z. However, the translation of any particular IDL file assumes a 'boilerplate' that defines these types as Z types. (The boilerplate could be included as a header in any Z output file produced by the translation, or could just be assumed to be in a separate file.) Thus the boilerplate includes the definitions:

$$
\begin{array}{l}
int == \mathbb{Z} \\
[char, long, float] \\
string = \mathrm{seq}\ char
\end{array}
$$

Although arithmetic operators for the numeric types `float` and `long`, analogous to those in the standard Z toolkit for the integer type $\mathbb{Z}$, are not needed by the translation process, they might be required for the purposes of refining Z specifications produced by the translation process.

An IDL specification can include definitions of *variables* and *constants* of various types, e.g. the definition:

```
char c;
```

declares a variable `c` of type `char`. This translates into Z syntax as:

$$c : char$$

(In addition, a module name may need to be prefixed to the variable name, if the variable is declared inside a module. This issue is discussed further in the section on IDL modules.)

Apart from the built-in types, types can be declared in various other ways, which are now described.

Structured types — like the record types used in many programming languages — can be defined. The fields of such a type may be of another structured type. An example of a structured type is:

```
struct Date {
    int     year
    string  month;
    int     day;
}
```

This is translated into Z as a schema, with an appropriate text wrapper, i.e. as

** BEGIN struct: Date

$$
\begin{array}{|l}
\hline
\textit{Date} \\
year : int \\
month : string \\
day : int \\
\hline
\end{array}
$$

** END struct: Date

Types can also be *enumerated types*, as in for example

```
enum Season { SPRING, SUMMER, AUTUMN, WINTER };
```

which is translated into Z as

$$Season == SPRING \mid SUMMER \mid AUTUMN \mid WINTER$$

The following is an example of an *abbreviated type* definition, which declares `ch` as an alias or abbreviation for the standard type `char`.

```
typedef char ch;
```

This is translated into Z (assuming that *char* is declared as a given type in the 'boilerplate' header of the Z output file) as:

$$ch == char$$

3.5 IDL Modules

An IDL *module* is a named collection of *definitions*. Unlike an IDL *interface*, a module does not represent a class of objects, in the object-oriented sense — it is simply a name-space for a group of related definitions. A module must be contained in one IDL source file. A file may contain several top-level modules, but a common convention is for each file to contain only one top-level module. There seems to be no requirement to have a single top-level 'system' module, although this might be a useful convention to adopt.

The general form of a module is:

```
/* ....<comment>....... */
module <module-name>
{...
<definitions>
...
}; // end of module <module-name>
```

Since a module is itself a kind of definition, modules can be nested. Modules can refer to definitions from other modules, in which case the names of those other modules are prefixed to the identifiers being imported, using "`::`"as a separator. There is no IDL syntax for specifying explicitly at the top of a module which identifiers it exports or imports. Thus it is not possible to tell by looking at an individual module alone, which of its definitions are exported to other modules, and which are not. It is, however, possible to see which identifiers are imported, simply by looking for identifiers which have a prefix of the form "`<moduleName>::`".

IDL modules can import identifiers from each other, as in for example:

```
module A {
    enum XType {X1, X2};
    typedef YType B::XType; }

module B {
    enum XType {X1, X2};
    typedef YType A::XType; }
```

Such mutual importing means that a translation of IDL modules using the Z *section* construct is impossible, because the sections of a Z specification must be ordered into a DAG-hierarchy by specifying the parents (if any) of each section, in such a way that the identifiers of sibling sections are not visible to each other. (An implicit 'root' section is assumed, from which all other sections are ultimately descended. The identifiers visible in a given section are its own, plus those of any of its ancestors, which must be distinct from its own, e.g. a variable x can't be defined locally in one section, and in one of its ancestor sections.)

IDL modules can be represented instead by adding module name prefixes (terminating with a reserved character, e.g. #) to all identifiers within a module (except those already prefixed with the name of another module) during the translation of that module into Z. When modules are nested inside modules, this will require two or more prefixes to be added, with the outermost module name being prefixed last. Applied to the IDL example just shown, this approach produces the following Z output:

** BEGIN module: A

$$A\#XType == A\#X1 \mid A\#X2$$
$$A\#Ytype == B\#XType$$

** END module: A

** BEGIN module: B

$$B\#XType == B\#X1 \mid B\#X2$$
$$B\#Ytype == A\#XType$$

** END module: B

3.6 IDL Interface Attributes

An IDL *interface*, which can have both *attributes* and *methods*, corresponds roughly to a class in object-oriented terms, but it is the *external* or *public* view of a class, i.e. it defines only those attributes and methods visible to an external user of objects of that class. One interface may inherit the attributes and operations of another. Multiple inheritance is allowed.

IDL interface attributes can be both readable and writable (with implicit 'get' and 'set' operations), or read-only (with an implicit 'get' operation only). Attributes are readable and writable unless explicitly declared as read-only.

An example of an IDL interface is:

```
interface Property {
    attribute int price
    attribute char taxCategory
    readonly attribute int floorArea }
```

This is translated to a Z schema interpreted as a state schema, plus associated 'get' and/or 'set' operation schemas for each attribute (prefixed with the state schema name, to distinguish them from the get and set operations of other interfaces). Thus the corresponding Z output (including a text wrapper) is:

** BEGIN interface: Property

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Property}} \\
\hline
price : int \\
taxCategory : char \\
floorArea : int \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Property\%Get\%price}} \\
\hline
\Delta Property \\
Result!! : int \\
\hline
Result!! = price = price' \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Property\%Set\%price}} \\
\hline
\Delta Property \\
newval?? : int \\
Result!! : void \\
\hline
price' = newval?? \\
\hline
\end{array}
$$

(... + *Get* and *Set* schemas for other attributes)

** END interface: Property

Points to note:

- The names of 'get' and 'set' operation schemas are constructed in a standard way, by inserting the strings '%*Get*%' and '%*Set*%' between the interface name and the attribute name.
- Reserved variable names *newval*?? and *Result*!! are used for the input and output variables of 'get' and 'set' operations. The double question and exclamation mark suffixes distinguish such variables clearly from other variables. *Result*!! has the vacuous type *void* for 'set' operations.
- When a 'get' operation takes place, the value of the attribute concerned is the same before and after the operation, but other attributes are unconstrained.
- When a 'set' operation takes place, the attribute concerned has the new value (possibly the same as its previous value) after the operation, but other attributes are unconstrained.

Given the assumptions above, 'get' and 'set' operations impose only very weak constraints on the prestate and poststate. This is a reasonable approach, because in an implementation of an interface, public attributes may be mutually constrained by invariants, so that for example, setting one attribute may involve changing another. (For example, an interface for stacks might have a public but read-only attribute representing the number of items on the stack, which increases every time an item is pushed onto the stack.)

3.7 IDL Interface Methods

In addition to the 'get' and 'set' methods implied by attribute declarations, interfaces can have explicitly declared methods. Every method has a result type value, which in some cases is the vacuous type *void*. Methods can have parameters, which may have any of the modes *in*, *out*, or *inout* — representing respectively, locations from which input data to the method are read, locations to which output data are written, and locations used in both the foregoing ways, i.e. from which data are read initially, and to which data (of the same type) are written subsequently.

For translation into Z, the result value and any `out` parameters are converted into operation schema output variables (with a '!' suffix added to each IDL parameter name). Any `in` parameters are translated into operation input variables (with a '?' suffix added to each IDL parameter name). Any `inout` parameters are translated into an input and an output variable of the same name, but with '?' and '!' suffixes respectively. The output variable representing the method's result value is clearly distinguished from those associated with `out` or `inout` parameters, by using the reserved name *Result*!!.

An example is shown below.

```
interface Thing {
    void op1 (in  char c, out  char d);
    string op2 (inout    int i); }
```

Z translation:

*** BEGIN interface: Thing

Thing

Thing%op1

$\Delta Thing$
$c? : char$
$d! : char$
$Result!! : void$

Thing%op2

$\Delta Thing$
$i? : int$
$i! : int$
$Result!! : string$

** END interface: Thing

3.8 IDL Interface Inheritance

One IDL interface can inherit from another. In the following example, the interface `Manager` inherits from `Employee`.

```
interface Manager : Employee
{ ....
 <+ any new attrs. and ops.>
}
```

Such inheritance can be represented in Z by schema inclusion of the state schema for the class concerned, e.g. in this case by

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Manager}} \\
\hline
\textit{Employee} \\
\ldots+\text{ any new attrs.} \\
\hline
\end{array}$$

The Z translation of the subclass will also include operation schemas that use schema inclusion to represent inheritance — e.g. if the Z translation of the *Employee* interface has an operation schema *Employee%promote*, the Z translation of the *Manager* interface will include the operation schema

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Manager\%promote}} \\
\hline
\textit{Employee\%promote} \\
\hline
\end{array}$$

Manager may also have additional, non-inherited operation schemas. For further examples of the use of schema inclusion to represent interface inheritance, see the Appendix.

In general, IDL interface inheritance can be multiple, and can cross module boundaries, in which case inherited names are prefixed by module names. An example involving both multiple and cross-module inheritance is shown below. (N.B. `A` inherits from `B`, and from `C` in module `Y`. Interfaces `A` and `B` are assumed here to be declared inside a module `X`.)

```
interface A : B, Y::C {
    readonly attribute int ANewAttr;
}
```

This is translated into Z as:

** BEGIN interface: X#A

$$\begin{array}{|l}
\multicolumn{1}{l}{X\#A} \\
\hline
X\#B \\
Y\#C \\
\textit{ANewAttr} : \textit{int} \\
\hline
\end{array}$$

(... *Get* and *Set* op. schemas for inherited attrs.)

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{A\%Get\%ANewAttr}} \\
\hline
\Delta A \\
\textit{Result}!! : \textit{int} \\
\hline
\textit{Result}!! = \textit{ANewAttr} = \textit{ANewAttr}' \\
\hline
\end{array}$$

(... Schemas for any inherited ops. other than *Get* and *Set* ops.)

** END interface X#A

4 Conclusion

This paper has described a translation from CORBA IDL into Z, as performed by a prototype translation which uses Glasgow University's IDL parsing tool, *HDirect*. The 'skeletal' form of Z produced is still useful because of its refinement properties. The translation takes one or more IDL text files as input, and produces a single text file containing Latex mark-up code for generating Z in Postscript form. It is intended for use in relation to *viewpoint* specification, particularly in the context of ODP, where IDL and Z are well suited to the *Computational* and *Information Viewpoints* respectively. The translation facilitates unification between these two viewpoints, because it allows an IDL specification to be converted into a form in which it can be unified with a Z specification via mutual refinement.

The translation could provide the basis for a reverse translation into IDL, from a limited subset of Z, augmented with structural information in some form. It could also be adapted into a translation that maps IDL specifications containing Z expressions as comments into full Z specifications. Such a hybrid notation would allow the 'skeletal' interface information encoded in IDL to be augmented with behavioural constraints.

Acknowledgements

We would like to thank the providers of the *HDirect* IDL to Haskell parsing tool (see `http://www.dcs.gla.ac.uk/fp/software/hdirect/`) for permission to use it.

References

[1] E.A. Boiten and J. Derrick. Grey box data refinement. In J. Grundy, M. Schwenke, and T. Vickers, editors, *International Refinement Workshop & Formal Methods Pacific '98*, Discrete Mathematics and Theoretical Computer Science series, pages 45–59, Canberra, September 1998. Springer-Verlag.

[2] E. Boiten, J. Derrick, H. Bowman, and M. Steen, "Constructive Consistency Checking for Partial Specification in Z", *Science of Computer Programming*, 35, 1999, pp. 29–75.

[3] H. Bowman, E. Boiten, J. Derrick, and M. Steen, "Viewpoint Consistency in ODP, a General Interpretation", *Formal Methods for Open Object-Based Distributed Systems*, eds. E. Najm and J.-B. Stefani, Chapman & Hall, March 1996, pp. 189–204.

[4] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development", *Int. Jour. on Software Engineering and Knowledge Engineering*, 1992, 2(1), pp. 31–58.

[5] P.F. Linington, "RM-ODP: The Architecture", *ICODP*, eds. K. Raymond and L. Armstrong, Chapman and Hall, February 1995, Brisbane, Australia, pp. 15–33

[6] R. Otte, P. Patrick, and M. Roy, *Understanding CORBA: The Common Request Broker Architecture*, Prentice Hall, 1996.

[7] G. Smith, *The Object-Z Specification Language*, Kluwer Academic Publishers, 2000.

[8] J.M. Spivey, *The Z Notation: A Reference Manual*, 2nd edition, Prentice Hall 1992.

[9] S. Thompson, *Haskell: The Craft of Functional Programming*, 2nd edition, Addison-Wesley, 1999.

[10] J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof*, Prentice Hall, 1996.

Appendix: Example of an IDL to Z Translation

This appendix shows a small IDL specification (a slight variant of an example given in [6]), followed by its translation into Z. Note that the IDL contains a forward declaration of an interface name, because interfaces have to be declared before their first use in another syntactic construct. Such forward declarations are not needed in Z, but are translated into comment text in the Z output file, so as to be available for reverse translation. The 'boilerplate' — containing declarations and definitions of standard types (such as `char`, `string`, `float`, etc.) — is not shown in the Z output file. It could either be included as a header, or assumed to be in another file.

The example relates to information that might typically be held about employees in a distributed corporate database system. At the top level, the IDL specification consists of a large module called `CORP` and a much smaller one called `ENGINEERING`, containing respectively general classes relating to employees, and classes relating to the corporation's engineering department. The module `CORP` declares some structured record types and types, and defines five interfaces — `Department`, `Employee`, `Manager`, `Personnel` (corresponding to employees working in the personnel department), and `PersonnelManager`. `PersonnelManager` inherits from both `Manager` and `Personnel`, which both inherit from `Employee`. The module `ENGINEERING` defines the interface `EmployeeLocator` (representing a class of systems that provide information about engineering employees), and the interface `PersonnelManager`, which inherits from the interface of the same name in the module `CORP`.

```
module CORP {
// Module representing info. about employees of a corporation
interface Employee; // forward declaration

// Type definitions used throughout module
typedef long Badgenum;
typedef long DeptNum;
enum DismissalCode { DISMISS_FIRED, DISMISS_QUIT };
enum DenyApprovalReasons { REASON, CODES };

// Structured data types
```

```
struct PersonalData {
    string    last_name;
    string    first_name;
    string    middle_name;
    string    phone;
    string    site;
};

typedef PersonalData EmpPersonalData;

struct EmpData {
    BadgeNum           id;
    EmpPersonalData    personal_info;
    char               job_class;
    float              hourly_rate;
};

struct DeptInfo {
    DeptNum    id;
    string     name;
};

// Interface definitions
interface Department {
    attribute DeptInfo             DeptID;
    readonly attribute Employee    manager_obj;
}; // end of interface Department

interface Employee {
    attribute EmpData                  personal_data;
    readonly attribute Department      department_obj;
    void promote    ( in char           new_job_class );
    void dismiss    ( in DismissalCode  reason,
                      in string         description );
    void transfer   ( in Department     new_dept_obj );
}; // end of interface Employee

interface Manager : Employee {
    void approve_transfer ( in Employee    employee_obj,
                            in Department  current_department,
                            in Department  new_department );
}; // end of interface Manager

interface Personnel : Employee {
    Employee hire ( in  EmpData     employee_data,
                    in  Department  department_obj,
                    out BadgeNum    new_employee_id );
}; // end of interface Personnel

interface PersonnelManager : Personnel, Manager {
```

```
    void arbitrate (  );
}; // end of interface PersonnelManager

}; //end of module CORP

module ENGINEERING {

// Interface definitions
interface EmployeeLocator {
    void FindEngineer ( in  CORP::BadgeNum     id,
                        out CORP::PersonalData info);
};

interface PersonnelManager : CORP::PersonnelManager {
};

}; // end of module ENGINEERING
```

The Z translation now follows. To save space, not all schemas are shown.

** BEGIN module: CORP
** forward declaration: interface Employee

$$CORP\#BadgeNum == long$$

$$CORP\#DeptNum == long$$

$$CORP\#DismissalCode ::= CORP\#DISMISS\_FIRED \mid CORP\#DISMISS\_QUIT$$

$$CORP\#DenyApprovalReasons ::= CORP\#REASON \mid CORP\#CODES$$

** BEGIN struct: CORP#PersonalData

$$\begin{array}{l} \hline CORP\#PersonalData \\ \hline last\_name : string \\ first\_name : string \\ middle\_name : string \\ phone : string \\ site : string \\ \hline \end{array}$$

** END struct: CORP#PersonalData

$$CORP\#EmpPersonalData == CORP\#PersonalData$$

** BEGIN struct: CORP#EmpData

$$\begin{array}{l} \hline CORP\#EmpData \\ \hline id : CORP\#BadgeNum \\ personal\_info : CORP\#EmpPersonalData \\ job\_class : char \\ hourly\_rate : float \\ \hline \end{array}$$

** END struct: CORP#EmpData

** BEGIN struct: CORP#DeptInfo

$$\begin{array}{|l}
\hline
\textit{CORP\#DeptInfo} \\
\hline
id : CORP\#DeptNum \\
name : string \\
\hline
\end{array}$$

** END struct: CORP#DeptInfo

** BEGIN interface: CORP#Department

$$\begin{array}{|l}
\hline
\textit{CORP\#Department} \\
\hline
DeptID : CORP\#DeptInfo \\
manager_obj : CORP\#Employee \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{CORP\#Department\%Get\%DeptId} \\
\hline
\Delta CORP\#Department \\
Result!! : CORP\#DeptInfo \\
\hline
Result!! = DeptID = DeptID' \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{CORP\#Department\%Set\%DeptID} \\
\hline
\Delta CORP\#Department \\
newval?? : CORP\#DeptInfo \\
Result!! : void \\
\hline
DeptID' = newval?? \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{CORP\#Department\%Get\%manager_obj} \\
\hline
\Delta CORP\#Department \\
Result!! : CORP\#Employee \\
\hline
Result!! = manager_obj = manager_obj' \\
\hline
\end{array}$$

** END of interface: CORP#Department

** BEGIN interface: CORP#Employee

$$\begin{array}{|l}
\hline
\textit{CORP\#Employee} \\
\hline
personal_data : CORP\#EmpData \\
department_obj : CORP\#Department \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{CORP\#Employee\%Get\%personal_data} \\
\Delta CORP\#Employee \\
Result!! : CORP\#EmpData \\
\hline
Result!! = personal_data = personal_data' \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{CORP\#Employee\%Set\%personal_data} \\
\Delta CORP\#Employee \\
newval?? : CORP\#EmpData \\
Result!! : void \\
\hline
personal_data' = newval?? \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{CORP\#Employee\%Get\%department_obj} \\
\Delta CORP\#Employee \\
Result!! : CORP\#Department \\
\hline
Result!! = department_obj = department_obj' \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{CORP\#Employee\%promote} \\
new_job_class? : char \\
Result!! : void \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{CORP\#Employee\%dismiss} \\
reason? : CORP\#DismissalCode \\
description? : string \\
Result!! : void \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{CORP\#Employee\%transfer} \\
new_dept_obj? : CORP\#Department \\
Result!! : void \\
\hline
\end{array}$$

** END interface: CORP#Employee

** BEGIN interface: CORP#Manager

$$\begin{array}{|l}
\multicolumn{1}{l}{CORP\#Manager} \\
CORP\#Employee \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{CORP\#Manager\%Get\%personal_data} \\
CORP\#Employee\%Get\%personal_data \\
\hline
\end{array}$$

(... + other ops. inherited from CORP#Employee)

$CORP\#Manager\%approve\_transfer$

$\Delta CORP\#Manager$
$employee\_obj? : CORP\#Employee$
$current\_department? : CORP\#Department$
$new\_department? : CORP\#Department$
$Result!! : void$

** END interface: CORP#Manager

** BEGIN interface: CORP#Personnel

$CORP\#Personnel$

$CORP\#Employee$

$CORP\#Personnel\%Get\%personal\_data$

$CORP\#Employee\%Get\%personal\_data$

(... + other ops. inherited from CORP#Employee)

$CORP\#Personnel\%hire$

$\Delta CORP\#Personnel$
$employee\_data? : EmpData$
$department\_obj? : CORP\#Department$
$new\_employee\_id : CORP\#BadgeNum$
$Result!! : CORP\#Employee$

** END interface: CORP#Personnel

** BEGIN interface: CORP#PersonnelManager

$CORP\#PersonnelManager$

$CORP\#Personnel$
$CORP\#Manager$

$CORP\#PersonnelManager\%Get\%personal\_data$

$CORP\#Employee\%Get\%personal\_data$

(... + other ops. inherited from CORP#Employee — common ancestor of CORP#Personnel and CORP#Manager)

$CORP\#PersonnelManager\%hire$

$CORP\#Personnel\%hire$

$CORP\#PersonnelManager\%arbitrate$
$\Delta CORP\#PersonnelManager$
$Result!! : void$

** END interface: CORP#PersonnelManager

** END module: CORP

** BEGIN module: ENGINEERING

** BEGIN interface: ENGINEERING#EmployeeLocator

$ENGINEERING\#EmployeeLocator$
$dummyAttr : void$

$ENGINEERING\#EmployeeLocator\%FindEngineer$
$\Delta ENGINEERING\#EmployeeLocator$
$id? : CORP\#BadgeNum$
$info! : CORP\#PersonalData$
$Result!! : void$

** END interface: ENGINEERING#EmployeeLocator

** BEGIN interface: ENGINEERING#PersonnelManager

$ENGINEERING\#PersonnelManager$
$CORP\#PersonnelManager$

$ENGINEERING\#PersonnelManager\%Get\%personal\_data$
$CORP\#PersonnelManager\%Get\%personal\_data$

$ENGINEERING\#PersonnelManager\%Set\%personal\_data$
$CORP\#PersonnelManager\%Set\%personal\_data$

$ENGINEERING\#PersonnelManager\%Get\%department\_obj$
$CORP\#PersonnelManager\%Get\%department\_obj$

$ENGINEERING\#PersonnelManager\%promote$
$CORP\#PersonnelManager\%promote$

(... + other ops. inherited from the $CORP\#PersonnelManager$ interface)

** END interface: ENGINEERING#PersonnelManager

** END module: ENGINEERING