ODP Computational to Information Viewpoint Mappings: A 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

The reference model of Open Distributed Processing (ODP) prescribes the use of a number of viewpoints (i.e. partial specifications). Specifications written in these viewpoints are likely to use different notations, e.g. the Computational Viewpoint is likely to include descriptions given in CORBA IDL, whilst the Information Viewpoint might well use a schema-based notation such as Z. To support such a specification scenario in this paper we describe a translation from a subset of CORBA IDL to the Z specification notation, which has been implemented in a prototype translator based on the IDL parsing tool *HaskellDirect*. Although our main motivation is to integrate CORBA IDL into an existing multi-language framework for viewpoint specification and consistency checking, the translation could also serve as the basis for a reverse translation from a subset of Z into IDL. In addition, it will help support a translation into Z from IDL specifications augmented with Z annotations that express behavioural constraints not expressible in IDL itself.

Keywords. ODP, viewpoints, CORBA IDL, Z, translation, specification languages, signatures, 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 [9] is a developing international standard. ODP is a example of an approach in which an overall system is described from several *viewpoints* [8], each a partial specification of a different aspect or perspective. In such approaches, the problem arises of ensuring that different viewpoints — sometimes in different formalisms — are "consistent", in some sense. The work described in this paper relates to our earlier work [1, 2, 13, 14, 15, 16] on viewpoint consistency (particularly in relation to ODP), which assumes that partial specifications are mutually consistent if and only if they have a *common refinement*. Finding such a refinement is sometimes referred to as *unification*. The problem then becomes one of defining appropriate

refinement rules which preserve certain properties, while allowing omission of details concerned with implementation or irrelevant to the viewpoint concerned.

Our work on using the specification language Z [12] for partial specification has led to new insights into refinement. A variety of refinement relations have been defined [4, 6] to account for specifications being "partial" in a number of different senses. For example, a specification can be partial in describing only a certain aspect of the behaviour (e.g. a state-space specification without timing constraints), only a subset of the possible operations, only a certain perspective (e.g. an external user's), only a certain level of implementation, or only a certain subsystem of the whole.

In terms of current practice regarding distributed systems, the Common Object Request Broker Architecture (CORBA) [10], supported by a large consortium of computing companies, has become a de facto standard. It provides a platform-neutral object-oriented framework for the specification and construction of such systems, which typically consist of heterogenous components, written in various languages, and running on a variety of hardware and software platforms. By using a neutral specification language called the Interface Description Language (IDL), CORBA allows such diverse components to be integrated smoothly into a single distributed system.

The ODP architecture [9] specifies five named 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.

These viewpoints are informally defined in natural language — albeit at greater length than given here — and so are inevitably somewhat vague. For the sake of generality and flexibility, the prescription of particular specification formalisms, software, or hardware is deliberately avoided in ODP. In particular, ODP and CORBA were formulated entirely independently of one another. However, since it is an objected-oriented formalism for describing external interfaces, CORBA IDL is a natural choice for the Computational Viewpoint. Likewise the formal specification language Z [12], a powerful and general formal specification language — although by no means the only possible choice — seems suitable for the Information Viewpoint. Thus in relation to ODP, an IDL to Z translation would allow a Computational Viewpoint specification in IDL to be converted into a form suitable for common refinement with an Information Viewpoint specification in Z.

This work fits into a larger programme of work[1] that considers cross-viewpoint mappings alongside specific enhancements to the ODP Enterprise and Computational Viewpoints. For example, we have considered how to specify (semi-formally) the Enterprise Viewpoint with an eye towards relating it to an Information Viewpoint specification written in Object-Z [11]. To this end, [13, 14, 15] discuss a case study where an Information Viewpoint specification written in Object-Z was compared with another Object-Z specification representing the Enterprise Viewpoint, generated via a translation from a formalism for expressing enterprise policies. The use of Z and Object-Z for the Information Viewpoint is also discussed in [3]. Not much work has been done on relating IDL to formal specification languages, but there is an example involving RAISE [19], and the RoZ tool [7] relates UML to Z.

The rest of the paper is organized as follows. Section 2 explains the main syntactic constructs of IDL, and how they are translated into Z. Section 3 describes a prototype implementation of the translation, based on an adaptation of the *HaskellDirect* IDL parsing tool. Section 4 discusses the nature and utility of the "skeletal Z" specifications produced by the IDL to Z translation, and section 5 is a summary and conclusion.

2 IDL Syntax and Its Translation to Z

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

- *Type definition* — a datatype definition (*structured*, *enumeration*, or *abbreviation* type).
- *Variable declaration* — declares a variable of a specific type.
- *Constant declaration* — declaration of 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, for example 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 produced by the tool, rather than in the LaTeX mark-up form produced by the translation from IDL. In other words, the target language representation is the standard LaTeX markup for Z, as used to produce the examples of IDL to Z translation shown in this paper.

Z is a state-based formal specification language [12] based on set theory and predicate logic, which ties in naturally with the ODP Information Viewpoint.

[1] http://www.cs.ukc.ac.uk/research/tcs/openviews/index.html

A typical Z specification consists of a general state schema (a labelled box containing typed state attributes, with an invariant constraining their values), an initialisation schema (defining the permissible initial states), and operation schemas (possibly with input and output variables), which define the ways in which the system state can change.

Because IDL concentrates on signature information rather than behavioural constraints, most of the translation from IDL produces declarative state signature information in Z. However, some Z operation schemas are produced as well, from the translation of explicit IDL interface methods, and of the implicit "get" and "set" methods associated with IDL interface attributes.

2.1 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 *HaskellDirect* parsing tool ignores IDL comments, so in this prototype, 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.

2.2 IDL Types, Variables, and Constants

IDL has a few standard types, e.g. `int`, `char`, `string`, `long`, and `float`. The identifiers for these types are left unchanged in the translation to Z, and a "boilerplate" is assumed that defines them in terms of standard or given Z types (e.g. *int* is defined as an abbreviation for the standard Z type $\mathbb{Z}$).

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, which are like the record types used in many programming languages. 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

Date

$$year : int$$
$$month : string$$
$$day : int$$

** 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 the "boilerplate" completely covers the translation of IDL's predefined types) simply as:

$$ch == char$$

2.3 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. 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 used in other modules, and which are not. It is, however, possible to see which imported identifiers are used, 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 special 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

2.4 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 possible approach might be to translate IDL interfaces to "grey box data types" [5], which extend the normal states-and-operations style in Z with explicit lists of observable and modifiable state components.) An interface may inherit the attributes and operations of another. In the version of IDL used in the prototype implemented, multiple inheritance is not 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) in this case is:

** BEGIN interface: Property

Property

$price : int$
$taxCategory : char$
$floorArea : int$

Property%Get%price

$\Delta Property$
$Result!! : int$

$Result!! = price = price'$

Property%Set%price

$\Delta Property$
$newval?? : int$
$Result!! : void$

$price' = newval??$

(... + *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.
- Special reserved variable names *newval*?? and *Result*!! are used for the input and output variables of "get" and "set" operations. The double question mark and double exclamation mark suffixes distinguish such variables clearly from other variables. *Result*!! has the trivial 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.)

2.5 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 *void*. Methods can have parameters, which may have any of the modes *in*, *out*, or *inout* — indicating 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.

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). Both an input and

output variable (of the same name, but with "?" and "!" suffixes respectively) are produced for each *inout* parameter. 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. Note that in this case, in which the IDL interface has no public attributes, an empty state schema is produced in Z.

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

Z translation:

*** BEGIN interface: Thing

Thing

$$\Delta Thing$$

Thing%*op*1

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

Thing%*op*2

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

** END interface: Thing

2.6 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

Manager

Employee
..+ any new attrs.

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

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

Manager may also have additional, non-inherited operation schemas.

IDL interface inheritance can cross module boundaries, in which case inherited names are prefixed by module names. Some versions of IDL allow multiple inheritance, others do not. 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

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

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

(Plus also, *Get* and *Set* op. schemas for inherited attrs., and schemas for any inherited ops. other than *Get* and *Set* ops.)

** END interface X#A

3 Implementation

A prototype tool has been implemented to perform the translation from IDL to Z for a subset of the syntax of IDL. The tool incorporates the freely available *HaskellDirect* IDL parsing tool, originally developed at Glasgow University, supplemented with some additional code, also written in the functional programming language Haskell [17].

Work is in progress to develop another tool to perform translations from Z to IDL, at least for a restricted subset of Z, augmented with some information from the user to indicate the intended interpretation of each Z schema (in the opposite IDL to Z translation, several different IDL constructs are translated into Z schemas). The aim is to incorporate such translation tools into a toolset for viewpoint specification.

In outline, the actual IDL to Z translation process is as follows:

- The input is an IDL specification, in one or more plain-text files.
- The IDL input is parsed into data structures in the functional programming language Haskell [17], using Glasgow University's *HaskellDirect* IDL parsing tool.
- The Haskell data structures are translated into strings, which are outputted to a single plain-text file containing LaTeX mark-up code.
- The LaTeX tool is run on the LaTeX mark-up code to produce a Postscript file for printing or viewing in a screen window, which can be shown alongside another window displaying the IDL input.

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.

The translation prototype takes only a few seconds to translate and display several pages of IDL input and the corresponding Z output, most of this time being accounted for by the running of LaTeX to produce the Postscript output, and the generation of the windows to display the output — the parsing and translation itself takes very little time.

The initial work on the tool has provided a "proof of concept" for the IDL to Z translation. Future work is envisaged which would extend the translation to the full syntax of IDL, add a convenient user-interface, and ultimately incorporate the tool into a wider toolset for viewpoint specification and refinement, with particular application to the ODP framework.

4 Refinement and "Skeletal Z" specifications

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 [4, 18].
- 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 content of the specification, in addition to the "skeletal" translation resulting from the IDL syntax alone. (Analogously to what the RoZ tool [7] does for UML.)

In addition, since the translation is injective, a reverse translation is possible. This could be used to generate IDL implementation templates from a restricted subclass of Z specifications, supplemented with information about the intended interpretation in IDL of Z schemas (which are used variously in the IDL to Z translation to represent states, operations, and structured datatypes). A general Z to IDL translation would lose most of the information contained in the Z predicate.

In conventional *data refinement* in Z [4, 18], an abstract ADT is refined to a concrete one by changing the state space and operations. Let the abstract and concrete ADT's be $(AStates, AInit, \{AOp_i \mid i \in I\})$ and $(CStates, CInit, \{COp_i \mid i \in I\})$, respectively (where $AInit$ and $CInit$ are initialisation predicates). The concrete ADT refines the abstract one, with respect to a *retrieval relation* $Retr$ between their state spaces, if and only if the following conditions hold:

- *Initialisation.* Every concrete initial state in $CInit$ is related by $Retr$ to some abstract initial state in $AInit$.
- *Inputs and outputs.* The operations AOp_i and COp_i have the same input and output variables (if any).
- *Applicability.* If a given abstract pre-state $AState$ and set of inputs satisfies the pre-condition of AOp_i, then any concrete pre-state $CState$ related to $AState$ by $Retr$ satisfies the pre-condition of COp_i, given the same inputs.
- *Correctness.* For any pre-states $AState$ and $CState$ related by $Retr$, if $AState$ satisfies the pre-condition of AOp_i, and $CState'$ is a concrete post-state reachable from $CState$ via COp_i, then there is some abstract post-state $AState'$ related by $Retr$ to $CState'$.

If the conditions above are satisfied, the concrete ADT "simulates" the abstract ADT, insofar as every one of its possible behaviours corresponds, via the retrieval relation, to a behaviour of the abstract ADT.

In the context of partial or viewpoint specification, conventional data refinement seems too restrictive. Some of our previous work [15] suggests other forms of refinement that should be allowed — for example adding new operations, or replacing several operations by one, which has more inputs.

5 Conclusion

This paper has described a translation from CORBA IDL to Z, as performed by a prototype translation tool, which uses Glasgow University's IDL parser, *HaskellDirect*. The translation takes one or more IDL text files as input, and produces a single text file containing LaTeX mark-up code for Z.

The IDL to Z translation is intended for use in the context of *viewpoint* specification, an approach in which systems are specified from several different perspectives or "viewpoints". The ISO's ODP architecture for open distributed systems is a particular example of a viewpoint approach, based on five named and informally described viewpoints. These include the *Computational Viewpoint* (a specification of the interfaces of the distributed objects of a system) and the *Information Viewpoint* (a high-level specification of the information processed by a system). Although ODP prescribes no particular formalisms for its viewpoints, IDL and Z are well suited to the Computational and Information Viewpoints respectively. Thus an IDL to Z translation facilitates viewpoint unification in a multiple-viewpoint, multiple-formalism context, because it allows an IDL specification to be converted into a form in which it can be unified with Z specifications via mutual refinement.

Since the IDL to Z translation is injective, it can provide the basis for a reverse translation into IDL, from a subset of Z that follows certain naming conventions, and is supplemented with certain textual markers indicating structure. The existing translation could also be adapted into one 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 *HaskellDirect* IDL to Haskell parsing tool (see `http://www.dcs.gla.ac.uk/fp/software/hdirect/`) for permission to use it.

References

[1] 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.

[2] 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.

[3] E. Boiten, H. Bowman, J. Derrick, P. Linington, and M. Steen, "Viewpoint consistency in ODP", *Computer Networks*, 34(3), August 2000, pp. 503–537.

[4] E. Boiten and J. Derrick, *Refinement in Z and Object-Z: Foundations and Advanced Applications*, Springer-Verlag FACIT series, Spring 2001 (forthcoming).

[5] E. Boiten and J. Derrick, "Grey Box Data Refinement", *International Refinement Workshop & Formal Methods Pacific '98*, eds. J. Grundy, M. Schwenke, and T. Vickers, Springer-Verlag, 1998, pp. 45–49.

[6] E. Boiten and J. Derrick, "Liberating Data Refinement", *Mathematics of Program Construction*, 5th International Conference, Ponte de Lima, volume 1837, Lecture Notes in Computer Science, eds. R.C. Backhouse and J.N. Oliveira, Springer-Verlag, July 2000, pp. 144–166.

[7] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud, "An Overview of RoZ: A Tool for Integrating UML and Z Specifications", *12th Conference on Advanced information Systems Engineering (CAISE'2000)*, Lecture Notes in Computer Science, Springer-Verlag, vol. 1789, 2000.

[8] 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.

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

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

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

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

[13] M. Steen and J. Derrick, "Formalising ODP Enterprise Policies", *EDOC*, 1999, University of Mannheim, Germany, IEEE Publishing.

[14] M. Steen and J. Derrick, "ODP Enterprise Viewpoint Specification", *Computer Standards and Interfaces*, 22:165-189, September 2000.

[15] C. Taylor, "Comparison of ODP Viewpoint Specifications in Object-Z: A Case Study", University of Kent, Tech. Rep. No. 7-00, March 2000.

[16] C. Taylor, J. Derrick, and E. Boiten, "A Case Study in Partial Specification: Consistency and Refinement for Object-Z", *Proc. of ICFEM 2000*, pp. 177–185, IEEE, September 2000.

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

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

[19] V. Zadorozhny, "Towards an integrated CORBA/RAISE Semantic Interoperable Environment", Tech. Rep. 117, UNU/IIST, P.O.Box 3058, Macau, July 1997.