An Object-Oriented Approach to the Formal Specification of ODP Trader*

Jin Song Dong and Roger Duke

Software Verification Research Centre, Department of Computer Science,
University of Queensland (email: jason@cs.uq.oz.au, rduke@cs.uq.oz.au)

An Object-Z formal specification of the ODP Trader is presented which concisely captures the hierarchical context structure, the distribution of information and the communication aspects of the Trader. Particular use is made of generic classes, class inheritance and object aggregation (i.e. the instantiation of collections of similar objects) to give a structured specification that can be readily extended.

Keyword Codes: D.2.1; D.2.10; D.3.1
Keywords: Software Engineering, Specification; Design; Formal Languages

1. INTRODUCTION

An informal view of Trader is given in ISO Document N743[9]. Issues involved in applying the formal description techniques Z[11], LOTOS[6] and SDL'92[1] to the specification of Trader have been addressed by Fisher et. al.[4]. A detailed Z specification of the Trader is given in Annex G[8] of ISO Document N743. An Object-Z[2][2] version of the specification is also given in Annex G, although that version is similar to the Z specification and does not make particular use of the object-oriented features of Object-Z.

In this paper, Object-Z is used to give an object-oriented specification which concisely captures the hierarchical context structure, the distribution of information and the communication aspects of the Trader. In particular,

- generic classes are combined with class inheritance to structure the specification, leading to a specification that can be readily extended;
- the hierarchical context structure of Trader is captured by imposing a suitable graph structure on an aggregate of context objects;
- objects are identified directly within the model without the need to explicitly name objects, as underlying the semantics of an Object-Z specification is the understanding that each object has a unique persistent identity that distinguishes it from every other object, regardless of class[3, 10].

*In Proceedings of the IFIP TC6/WG6.1 International Conference on Open Distributed Processing, Berlin, 13-16 September, 1993, pp. 341-352, North-Holland.

[2]Object-Z is an extension of the Z specification language to accommodate object-oriented concepts.

The specification presented in this paper covers some aspects of Trader not covered in [8], namely, multiple-context searching, connection and type management (attach and detach an importer, add and delete a type)[7] and federation. However, as the specification concentrates on high level functionality, implementation aspects of Trader are not covered.

In Section 2 a special Object-Z construct is introduced to collect the union of output from a distributed set of objects that all respond to a given message. This construct will be needed when collecting information from several contexts that all respond to a search request from an importer (multi-context-search). In Section 3 generic classes are introduced that can be inherited directly to define the basic classes of Trader in Sections 4 and 5. In Section 5 the specification of Trader is extended to consider a federation of Traders.

Abbreviations

The following abbreviations are adopted throughout this specification (an 's' is added if needed to form plurals.)

cont	context	mc	matching constraint
cm	context manager	req	require
crit	criteria	sel	select
del	delete	serv	service
elmt	element	sp	service property
exp	exporter	st	service type
imp	importer	sup	super
info	information	tm	type manager

This paper assumes the reader has some familiarity with the Object-Z specification language and with the informal description of Trader.

2. OUTPUT COLLECTION IN OBJECT-Z

Within Trader, when an importer makes a search request various contexts may respond with information. This information needs to be collected together and passed back to the importer. In this section an Object-Z construct is introduced to facilitate the collection of information from various objects.

The general format of the construct is

$$\bigcup a : sa \bullet a.Op[p!]$$

where sa is a set of objects of some class A, say, and $p!$ is an output parameter of the operation Op defined in A. Furthermore, the output parameter $p!$ must be of type $\mathbb{P}\, T$ for some set T, say. This construct has the effect that the operation Op is applied to each object a in the set sa and the individual outputs $p!$ are collected (by union) and output to the environment as the value of $p!$. That is, the output $p!$ is not the individual subset $p!$ of T resulting from any one application of operation Op to some a in sa, but rather the subset of T obtained from the union of all such outputs $p!$ when Op is applied to all a in sa.

As an illustration of this construct, consider the following example. An object of class A stores a set of numbers (ns). If it performs the Out operation, this set of numbers is output to the environment as the value of parameter $ns!$. The class B contains an aggregate (aa) of A objects. The Out operation of this class has the effect that some subset (sa) of the aggregate of A objects is selected and the operation Out as defined in class A is applied to each object a in sa, with the union of every individual output being output to the environment as the value of parameter $ns!$.

A

$$ns : \mathbb{P}\,\mathbb{N}$$

Out

$$\begin{array}{l} ns! : \mathbb{P}\,\mathbb{N} \\ \hline ns! = ns \end{array}$$

B

$$aa : \mathbb{P}\,A$$

$Select$

$$\begin{array}{l} sa : \mathbb{P}\,A \\ \hline sa \subseteq aa \end{array}$$

$$\begin{array}{l} Out \mathrel{\widehat=} Select \bullet \\ \qquad \bigcup a : sa \bullet a.Out[ns!] \end{array}$$

3. GENERIC CLASSES: A SPECIFICATION LIBRARY

In this section we introduce three generic classes forming a small class library from which Trader classes are defined by inheritance in Sections 4 and 5.

3.1. A Generic Set Class

We begin by defining the generic class $S[X]$ denoting a set of elements of X. The class contains operations to add elements to, and delete elements from, the set. As well as being inherited by the other two library classes defined in this section, $S[X]$ will be used in Section 4 to define by inheritance the type manager ($TypeManager$), context ($Cont$) and importers ($Imps$) classes.

$S[X]$

$$elmts : \mathbb{P}\,X$$

$\textsc{Init}$

$$elmts = \varnothing$$

Add

$$\begin{array}{l} \Delta(elmts) \\ e? : X \\ \hline e? \notin elmts \\ elmts' = elmts \cup \{e?\} \end{array}$$

Del

$$\begin{array}{l} \Delta(elmts) \\ e? : X \\ \hline e? \in elmts \\ elmts' = elmts - \{e?\} \end{array}$$

3.2. A Generic DAG Class

The generic class $DAG[X]$ denoting a directed acyclic graph is defined by inheriting[3] $S[X]$. The relation between elements is called *super* because this class will be applied to

[3]Operation schemas of the inherited class and those of the same name declared explicitly in the derived class are merged[2].

define the Trader context manager (*ContManager*) class in Section 4 where in that case the relation captures the concept of 'is a super context of'. The operation *Add* inherited from $S[X]$ is extended to incorporate the linking of the input element to its immediate super (parent) elements.

$DAG[X]$

$S[X]$

$$
\begin{array}{|l}
super : X \leftrightarrow X \\
\hline
\operatorname{dom} super \subseteq elmts \\
\operatorname{ran} super \subseteq elmts \\
\nexists e : elmts \bullet (e, e) \in super^{+}
\end{array}
$$

Add

$$
\begin{array}{|l}
\Delta(super) \\
sups? : \mathbb{P}\, X \\
\hline
sups? \subseteq elmts \\
super' = super \cup \\
\quad \{e : sups? \bullet (e, e?)\}
\end{array}
$$

Del

$$
\begin{array}{|l}
\Delta(super) \\
\hline
super' = \{e?\} \lhd super \rhd \{e?\}
\end{array}
$$

3.3. A Generic UG Class

The generic class $UG[X]$ denoting an undirected graph is defined by inheriting $S[X]$. This class will be used in Section 5 to define by inheritance the Trader federation system (*FederatedTraders*) class representing the links between federated traders.

$UG[X]$

$S[X]$

$$
\begin{array}{|l}
links : X \leftrightarrow X \\
\hline
\operatorname{dom} links \subseteq elmts \\
\forall\, x, y : X \bullet \\
\quad x \underline{links}\ y \Leftrightarrow y \underline{links}\ x
\end{array}
$$

Del

$$
\begin{array}{|l}
\Delta(links) \\
\hline
links' = \{e?\} \lhd links \rhd \{e?\}
\end{array}
$$

$Connect$

$$
\begin{array}{|l}
\Delta(links) \\
x?, y? : elmts \\
\hline
x? \neq y? \\
\neg\ x? \underline{links}\ y? \\
links' = links \cup \\
\quad \{x? \mapsto y?, y? \mapsto x?\}
\end{array}
$$

Cut

$$
\begin{array}{|l}
\Delta(links) \\
x?, y? : elmts \\
\hline
x? \underline{links}\ y? \\
links' = links - \\
\quad \{x? \mapsto y?, y? \mapsto x?\}
\end{array}
$$

4. THE TRADER SPECIFICATION

At this level of abstraction, the low-level detail of information matching required to pair importer requests to exporter services is not described; rather, information matching is captured by defining abstract data types whose internal structure is not specified.

4.1. Primary Data Types

We begin by defining the basic data types needed to specify Trader. The given type

$$[\mathit{ServProp}]$$

of service properties will denote the collection of specific data values particular to an exporter (e.g. specific details of the service offered such as the cost, location, etc.). The type

$$\mathit{ServType} == \mathbb{P}\,\mathit{ServProp}$$

of service types will denote the set of all possible service properties, i.e. all possible collections of data values that an exporter may possess. The type

$$\mathit{MatchingConstraints} == \mathbb{P}\,\mathit{ServProp}$$

of matching conditions will denote a set of service properties that can form the basis of a search request by an importer (e.g. a range of possibilities for data values such as cost, location, etc.). The given type

$$[\mathit{CritInfo}]$$

of criteria selecting information will denote the criteria to be used for selecting between the options when several exporters satisfy a search request. To be specific, we shall suppose there is a function *Optimize* that returns the most appropriate service property when given a suitable selection criterion and a compatible set of service properties.

$$\mathit{Optimize} : \mathit{CritInfo} \times \mathbb{P}\,\mathit{ServProp} \nrightarrow \mathit{ServProp}$$

4.2. The Type Manager

The class *TypeManager* denotes the collection of all the service types in Trader. This collection can be updated by adding a new service type or by deleting an existing service type.

$$\begin{array}{|l}
\hline
\textit{TypeManager} \\
S[\textit{ServType}][\textit{types}/\textit{elmts}, \textit{Add}[st?/e?], \textit{Del}[st?/e?]] \\
\hline
\end{array}$$

The symbol '/' denotes renaming of identifiers. In this case, the attribute *elmts* is replaced by *types*, while for both of the operations *Add* and *Del* the input $e?$ is replaced by $st?$.

4.3. Exporter and Context

The class *Exp* denotes an exporter of Trader.

$$\begin{array}{|l}
\hline
\textit{Exp} \\
\begin{array}{|l}
\hline
st : \textit{ServType} \\
sp : \textit{ServProp} \\
\hline
sp \in st \\
\hline
\end{array}
\quad
\begin{array}{|l}
\hline
\textit{Modify} \\
\Delta(sp) \\
new_sp? : \textit{ServProp} \\
\hline
sp' = new_sp? \\
\hline
\end{array} \\
\hline
\end{array}$$

The context class *Cont* denotes an aggregate of exporter objects. A context also contains a set of permitted service types: for any exporter of the context, the service type of the exporter has to be included in the set of permitted service types of the context.

$$\begin{array}{|l}
\hline
\textit{Cont} \\
S[\textit{Exp}][\textit{exps}/\textit{elmts}, \textit{AddExp}/\textit{Add}, \textit{DelExp}/\textit{Del}] \\
\begin{array}{|l}
\hline
\textit{permitypes} : \mathbb{P}\, \textit{ServType} \\
\hline
\forall\, e : \textit{exps} \bullet e.st \in \textit{permitypes} \\
\hline
\end{array} \\
\begin{array}{|l}
\hline
\textit{Search} \\
st? : \textit{ServType} \\
mc? : \textit{MatchingConstraints} \\
list! : \mathbb{P}(\textit{Exp} \times \textit{ServProp}) \\
\hline
list! = \{e : \textit{exps} \mid e.st = st? \wedge e.sp \in mc? \bullet (e, e.sp)\} \\
\hline
\end{array} \\
\hline
\end{array}$$

The *Search* operation finds all exporters in the context whose service type and service property meet the supplied service type and matching conditions.

4.4. The Context Manager

The context manager class *ContManager* denotes an aggregate of contexts. The relation *super* inherited from $DAG[Cont]$ denotes the relation 'is a super-context of'.

ContManager

$DAG[Cont][conts/elmts, Add[c?/e?], Del[c?/e?]]$

SelConts

$cs? : \mathbb{P}\, Cont$

$cs? \subseteq conts$

$$AddExp \mathrel{\widehat=} SelConts \bullet \bigwedge c : cs? \bullet c.AddExp$$

$$DelExp \mathrel{\widehat=} SelConts \bullet \bigwedge c : cs? \bullet c.DelExp$$

$$Search \mathrel{\widehat=} SelConts \bullet \bigcup c : super^*(\!|\ cs?\ |\!) \bullet c.Search[list!]$$

GetExp

$list? : \mathbb{P}(Exp \times ServProp)$
$selCrit? : CritInfo$
$theOne! : Exp \times ServProp$

$list? \neq \varnothing$
$theOne! \in list?$
$(selCrit?, \operatorname{ran} list?) \in \operatorname{dom} Optimize$
$\exists\, e : \operatorname{dom} list? \bullet theOne! = (e, Optimize(selCrit?, \operatorname{ran} list?))$

$$Select \mathrel{\widehat=} Search \parallel GetExp$$

By placing the *AddExp* operation in this class, the context manager has control as to which contexts are available to an exporter.

The *Search* operation applies the construction of Section 2 to search all contexts and sub-contexts of the input set $cs?$ of contexts, i.e. $super^*(\!|\ cs?\ |\!)$, and collect all the responses into the output $list!$.

4.5. Importers

The class *Imp* denotes an importer of Trader. Two request operations, *ReqSearch* and *ReqSelect*, are provided.

Imp

ReqSearch

$st! : ServType$
$mc! : MatchingConstraints$
$list? : \mathbb{P}(Exp \times ServProp)$

ReqSelect

$st! : ServType$
$mc! : MatchingConstraints$
$selCrit! : CritInfo$
$theOne? : Exp \times ServProp$

The class *Imps* denotes an aggregate of importers.

$$
\begin{array}{|l}
\hline
\textit{Imps} \\
S[Imp][imps/elmts, Add[i?/e?], Del[i?/e?]] \\
\quad \begin{array}{|l}
\hline
\textit{SelImp} \\
i? : Imp \\
\hline
i? \in imps \\
\hline
\end{array} \\
ReqSearch \mathrel{\widehat=} SelImp \bullet i?.ReqSearch \\
ReqSelect \mathrel{\widehat=} SelImp \bullet i?.ReqSelect \\
\hline
\end{array}
$$

4.6. The Trader System

The class *TraderSystem* consists of a type manager object, a context manager object and an importers object, and the communication between the exporters and importers is specified.

$$
\begin{array}{|l}
\hline
\textit{TraderSystem} \\
\begin{array}{|l}
\hline
tm : TypeManager \\
cm : ContManager \\
imps : Imps \\
\hline
\forall\, c : cm.conts \bullet \\
\quad c.permitypes \subseteq tm.types \\
\hline
\end{array}
\quad
\begin{array}{|l}
\hline
\textsc{Init} \\
tm.\textsc{Init} \\
cm.\textsc{Init} \\
imps.\textsc{Init} \\
\hline
\end{array} \\
AddType \mathrel{\widehat=} tm.Add \\
DelType \mathrel{\widehat=} tm.Del \\
Modify \mathrel{\widehat=} [e? : Exp \mid e? \in \bigcup\{c : cm.conts \bullet c.exps\}] \bullet e?.Modify \\
Export \mathrel{\widehat=} cm.AddExp \\
Withdraw \mathrel{\widehat=} cm.DelExp \\
AddCont \mathrel{\widehat=} cm.Add \\
DelCont \mathrel{\widehat=} cm.Del \\
Attach \mathrel{\widehat=} imps.Add \\
Detach \mathrel{\widehat=} imps.Del \\
Search \mathrel{\widehat=} imps.ReqSearch \parallel cm.Search \\
Select \mathrel{\widehat=} imps.ReqSelect \parallel cm.Select \\
\hline
\end{array}
$$

The invariant of this class insures that all permitted types of each context in the system are types known to the type manager. The (implicit) input $e? : Exp$ to the *Modify* operation identifies an exporter regardless of its context, if service properties of an exporter inside a context is modified then any other context containing the exporter $e?$ will also be similarly modified, without the need to explicitly perform the *Modify* operation. The operation names used here correspond to the names used in ISO Document N743[9]. The input/output protocol diagram of *TraderSystem* is given in Figure 1.

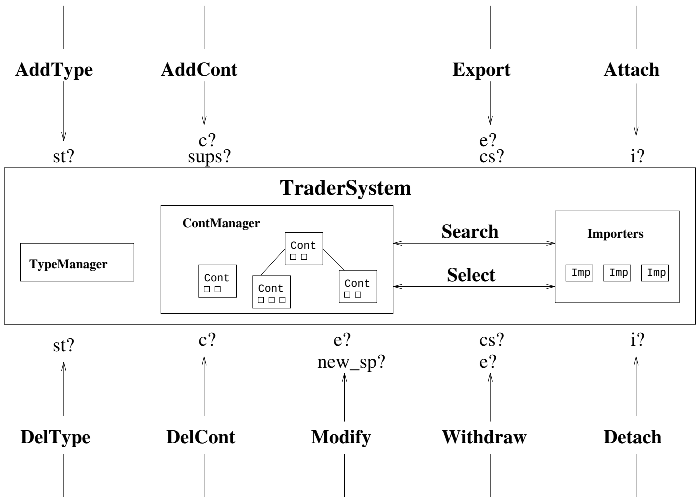

Figure 1. Trader System Protocol.

This completes the specification of a single trader; in the next section the specification is extended to specify a system of federated traders.

5. A FEDERATION OF TRADERS

Since a trader is itself a server, the set of traders can be organized into a federation where a trader can advertise its interfaces to any other related trader in the federation. As a result, if an importer is unable to have its request satisfied by a trader, all related traders can be sent the request. To simplify matters, we shall suppose the relation between traders is symmetric[4].

5.1. Towards a Federated Trader System

The *TraderSystem* class needs to be modified to meet the requirements of a federation: some importer and context manager operations are promoted so that a trader can communicate with other traders in the federation.

$FTrader$

$TraderSystem$

$ReqSearch \mathrel{\widehat{=}} imps.ReqSearch$

$ReqSelect \mathrel{\widehat{=}} imps.ReqSelect$

$GetExp \mathrel{\widehat{=}} cm.GetExp$

$CmSearch \mathrel{\widehat{=}} cm.Search$

$CmSelect \mathrel{\widehat{=}} cm.Select$

[4]In practice, the actual relation between traders is more complex[9].

5.2. The Trader Federation

When an importer requests some information from a trader in the federation, the trader will attempt to provide the information. If, however, the trader cannot provide the information, the request is passed to each related trader in the federation. In effect, a priority is given to the trader first contacted by the importer, with related traders contacted only if the first trader is unable to supply the information requested. This priority can be captured by the schema-override operator $\oplus$[5].

$$
\begin{array}{|l}
\hline
\textit{FederatedTraders} \\
\hline
UG[FTrader][traders/elmts, Add[t?/e?], Del[t?/e?]] \\
\\
\begin{array}{|l}
\hline
\textit{SelTrader} \\
\hline
t? : FTrader \\
\hline
t? \in traders \\
\hline
\end{array} \\
\\
AddType \mathrel{\widehat=} SelTrader \bullet t?.AddType \\
DelType \mathrel{\widehat=} SelTrader \bullet t?.DelType \\
Modify \mathrel{\widehat=} SelTrader \bullet t?.Modify \\
Export \mathrel{\widehat=} SelTrader \bullet t?.Export \\
Withdraw \mathrel{\widehat=} SelTrader \bullet t?.Withdraw \\
AddCont \mathrel{\widehat=} SelTrader \bullet t?.AddCont \\
DelCont \mathrel{\widehat=} SelTrader \bullet t?.DelCont \\
Attach \mathrel{\widehat=} SelTrader \bullet t?.Attach \\
Detach \mathrel{\widehat=} SelTrader \bullet t?.Detach \\
Search \mathrel{\widehat=} SelTrader \bullet \\
\qquad t?.ReqSearch \\
\qquad \| \\
\qquad (\bigcup t : links(\!|\ \{t?\}\ |\!) \bullet t.CmSearch[list!]) \oplus (t?.CmSearch \wedge [list! \neq \varnothing]) \\
Select \mathrel{\widehat=} SelTrader \bullet \\
\qquad t?.ReqSelect \\
\qquad \| \\
\qquad ((\bigcup t : links(\!|\ \{t?\}\ |\!) \bullet t.CmSearch[list!]) \parallel t?.GetExp) \oplus t?.CmSelect \\
\hline
\end{array}
$$

In the *Search* operation above, $links(\!|\ \{t?\}\ |\!)$ represents the set of all traders related to $t?$ in the federation. The override operator $\oplus$ gives the priority to $t?$ to perform the search first. If the search fails, each related trader is searched and the outputs collected using the construct of Section 2. The operation *Select* is defined similarly.

This completes the specification of the Trader federation.

[5]The schema-override operator is defined by $A \oplus B \mathrel{\widehat=} B \vee (A \wedge \neg\ pre\ B)$[5].

6. DISCUSSION AND CONCLUSION

Several important features of Trader not included in the Z specification in Annex G[8] of ISO Document N743 are captured by the Object-Z specification presented in this paper.

- The hierarchical structure of Trader is captured by the class structure of the specification. For example, a federated system object contains an aggregate of trader objects, each of which contains a type manager object, an aggregate of context objects and an aggregate of importer objects. Each context object in turn contains an aggregate of exporter objects. This embedded structuring is not easily captured in Z.

- The information exported by Trader is distributively stored in the exporter objects which may be shared between many different contexts. This information distribution is captured directly in the Object-Z specification. In the Z specification[8] however, this information is not distributed but is captured by centralized global state variables, and consequently when the service property of an exporter is modified, the global variables have to be modified.

- Aspects of Trader security control are implicitly captured in the Object-Z specification. For example, when an exporter wishes to advertise its service, the appropriate contexts are not decided by the exporter but by the context manager. This is captured in the specification by having the *AddExp* operation, which determines the contexts to which an exporter is added, defined in the context manager class rather than in the exporter class. Similarly, the contexts to be searched in response to a request from an importer is decided by the context manager. Such security aspects are possible in Object-Z because of the encapsulation afforded by the class structure.

- The concurrency and communication properties of Trader are captured in the specification by the Object-Z synchronous message passing operator, $\|$. The distribution of information and its collection is captured by the distributed conjunction operator (e.g. $\bigwedge x : xs \bullet x.Op$) and the distributed output collection construct (e.g. $\bigcup x : xs \bullet x.Op[p!]$) defined in Section 2.

An incremental development style has been adopted throughout this paper, leading to a specification which can be readily extended and modified as required. This is important, not only because it facilitates the inclusion of low-level detail deliberately omitted, but also because features of Trader are likely to change as the system develops. Object-Z is particularly suitable for specifying in such a style.

Work has been done to refine within Object-Z the abstract data types of the specification to more concrete structures, e.g. refining the service property abstractly captured by the given type *ServProp* to a mapping from a set of service attribute names to a set of service attribute values, and refining the *Optimize* function to specific maximize or minimize functions on a particular service attribute. Space prevents further discussion of this here.

REFERENCES

1. CCITT. *CCITT Specification and description language SDL.* Geneva, 1992. Recommendation Z.100 (SDL'92).
2. R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language. In T. Korson, V. Vaishnavi, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice-Hall, 1991.
3. R. Duke and G. Rose. Modelling object identity. In *Proc. 16th Australian Comput. Sci. Conf. (ACSC-16)*, pages 93–100, February 1993.
4. J. Fischer, A. Prinz, and A. Vogel. Different FDT's confronted with different ODP-viewpoints of the trader. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial Strength Formal Methods*, volume 670 of *Lect. Notes in Comput. Sci.*, pages 332–350. Springer-Verlag, April 1993.
5. I. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice-Hall, 1987.
6. ISO TC97/SC21. *LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988. Intl. Standard 8807.
7. ISO/IEC JTC1/SC21/WG7 N743. *ANNEX E Relationship to Candidate ODP CommonFunctions(Temporary)*, November 1992.
8. ISO/IEC JTC1/SC21/WG7 N743. *ANNEX G Z specification of the Trader*, November 1992.
9. ISO/IEC JTC1/SC21/WG7 N743. *Working Document on Topic 9.1 - ODP Trader*, November 1992.
10. G. Rose. Object-Z. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 59–77. Springer-Verlag, 1992.
11. J.M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, 1989.