Formal Specification and Testing of a Management Architecture

G. P. A. Fernandes[*] *and J. Derrick*
University of Kent
Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(Telephone: +44-1227-764000, Email: {gpaf,jd1}@ukc.ac.uk.)

Abstract
The importance of network and distributed systems management to supply and maintain services required by users has led to a demand for management facilities. Open network management is assisted by representing the system resources to be managed as objects, and providing standard services and protocols for interrogating and manipulating these objects. This paper examines the application of formal description techniques to the specification of managed objects by presenting a case study in the specification and testing of a management architecture. We describe a formal specification of a management architecture suitable for scheduling and distributing services across nodes in a distributed system. In addition, we show how formal specifications can be used to generate conformance tests for the management architecture.

Keywords
Managed objects behaviour; Management architecture; Formal description techniques; Open Distributed Processing; Conformance.

1 INTRODUCTION

The importance of network and distributed systems management to supply and maintain services required by users has led to a demand for management facilities. However, fully integrated management systems which will cope with management of large-scale distributed applications and their underlying communication services are still not available. Such applications require open management to integrate their components, which may have been obtained from a number of sources; the cost of system administration will depend to a large extent on how easy it is to perform this management integration. The creation of open distributed management depends upon there being a common representation for the resources being managed. This can be achieved by the creation of a suitable family of managed object definitions.

At present the nature of the resources to be managed and the behaviour they are expected to exhibit are expressed in natural language, structured and organized using a simple specification technique set out in the Guidelines for the Definition of Managed Objects (GDMO) [9]. The informal nature of this technique makes the implementation and testing of managed objects expensive, because much skilled

[*]Work supported by JNICT Program *PRAXIS XXI* (Portugal) under grant No. BD/2804/93

effort is needed to interpret the specifications and construct suitable tests. Formal description techniques (FDTs) offer the promise of improved quality and cost reduction by removing errors and ambiguities from the specification and automating aspects of both implementation and testing. Indeed, interworking will depend on specification and testing and product cost will depend on the efficiency of these processes. The aim of our work is to test the applicability of FDTs to managed object specification by formally specifying a realistic and large application using an object-oriented variant of the formal technique Z.

In this paper we show how formal techniques can be used to specify a management architecture suitable for scheduling and distributing services across nodes in a distributed system. The aim of the architecture is to optimise the use of resources by distributing the load and managing the resources available in order to fulfil the requirements of application services [6]. In section 2 of the paper we describe the management model we use. Section 3 discusses the management infrastructure we have been developing. Section 4 shows how we can apply an object-oriented variant of the formal language Z to the specification of interacting managed objects by formally specifying the architecture, and in section 5 we show how test generation methods can be applied to Z specifications of managed objects.

2 MANAGEMENT MODEL

The role of *management* is to monitor and control the system to be managed, so it fulfils the requirements both of the owners and the users of the system. The management model presented in this paper is a distributed object-oriented model based on the Open Distributed Processing (ODP) Reference Model [11] and the OSI management model [8]. The Reference Model of ODP provides a framework for the standardisation of Open Distributed Processing. The OSI Systems Management provides mechanisms for the monitoring, control and coordination of those resources which allow communication to take place in the OSI environment (OSIE).

The existing approaches to management address mainly network management. Many of the ideas included in the OSI management model, a standard for network management, can be used for distributed systems. However, it must be taken into account that while network management concentrates on largely autonomous network devices, distributed systems management addresses components which are much more dependent on each other.

One of the most important ideas in OSI Systems Management is the use of object-oriented principles to define management information and interfaces. The devices in the network that are subject to management are viewed as *managed objects*. Organisational requirements require partition of OSI management into functional areas, such as security, account and fault management, or for other management purposes, such as by geographical, technological or organisational structure. To reflect this, managed objects are organised into management *domains*. Managed objects in a particular domain are subject to a common management policy, which consists of a set of rules constraining the behaviour of those objects. The ability to specify precisely management policies, independent of the implementation is an important benefit of formal specification.

3 MANAGEMENT ARCHITECTURE

In this section we present an architecture to support management of distributed systems, addressing in particular the issue of distributing the workload submitted to a distributed system by its users. Distributed scheduling is used in order to locate a new service on the most appropriate node, taking into account the current state of the system and the quality of service requirements of the service.

A centrally-located allocator – *Distributed System Manager (*DSM*)* – is responsible for taking decisions in order to determine to which node in the system each service will be allocated. To determine if a node is suitable to instantiate a service, the DSM has to compare the quality of service (QoS) requirements of the service with the resources provided by the node. Placement is based on the last known state of the system, which is stored by the DSM, and updated by the monitoring information it receives from node managers.

The foundation of any management system is a database containing information about the components being managed. This type of database is often referred to as Management Information Base (MIB). The MIB is a structured collection of managed objects which represent the components that are monitored and controlled by a management system. Each node in the system maintains a MIB that reflects the status of the managed objects at that node.

The *Node Manager* is an entity local to a node, responsible for managing the objects within that node and reporting monitoring information to the DSM. It is capable of performing management operations on managed objects on behalf of a DSM and of emitting management notifications on behalf of a managed object to inform the DSM about the occurrence of an event. The node manager monitors and controls the services instantiated on the node and collects information about the resources available. This information is stored in the local MIB.

The DSM also maintains a MIB where information about the nodes under the DSM control is stored. After initialisation, the DSM issues requests for monitoring information only when it is trying to find a suitable location for a service. The set of polled nodes will send monitoring information which will be used to update the DSM MIB.

All newly created services are instantiated by the DSM upon request by the *trader*. The (ODP) trader is an object that provides a service which accepts and stores service offers from potential providers (servers) and hands out this information on request to potential clients. The DSM selects a suitable location for the service requested and asks the local node manager to instantiate that service.

4 SPECIFYING MANAGED OBJECTS FORMALLY

This section illustrates how we have used an object-oriented variant of Z to specify our management architecture.

Z [16] is a state based formal description technique (FDT), and Z specifications consist of informal English text interspersed with formal mathematical text. The formal part describes the abstract state of the system (including a description of the initial state of the system), together with the collection of available operations, which manipulate the state. The descriptions are given in terms of set theory and first-order predicate calculus. The *schema calculus* provides a useful (and visual) way to structure specifications, and to provide for a degree of modularity in the definition of operations. Z has proved to be one of the most enduring formal description techniques, partly because of its simplicity and readability. It has gained significant industrial usage and support over the years. Z has been shown to be a suitable vehicle for the specification of information related activities, and because of this has been considered a suitable language for use in the information viewpoint within ODP.

However, modern distributed systems are object-based, and for this reason there has been interest in extending Z to facilitate an object-oriented specification style. This allows for a proper definition of inheritance, and for encapsulation to be used to structure the specification in terms of classes and objects.

Object-Z [5] and ZEST [13, 2] are similar object-oriented extensions of Z. They both use the concept of a class to encapsulate the descriptions of an object's state with its related operations. In addition, they provide support for inheritance, instantiation and the description of communication between objects. In this paper we use ZEST to specify our managed objects, although a description in Object-Z would be very similar. Using an object-oriented variant of Z allows a hierarchy of classes to be developed as the Guidelines for the Definition of Managed Objects [9] indicate.

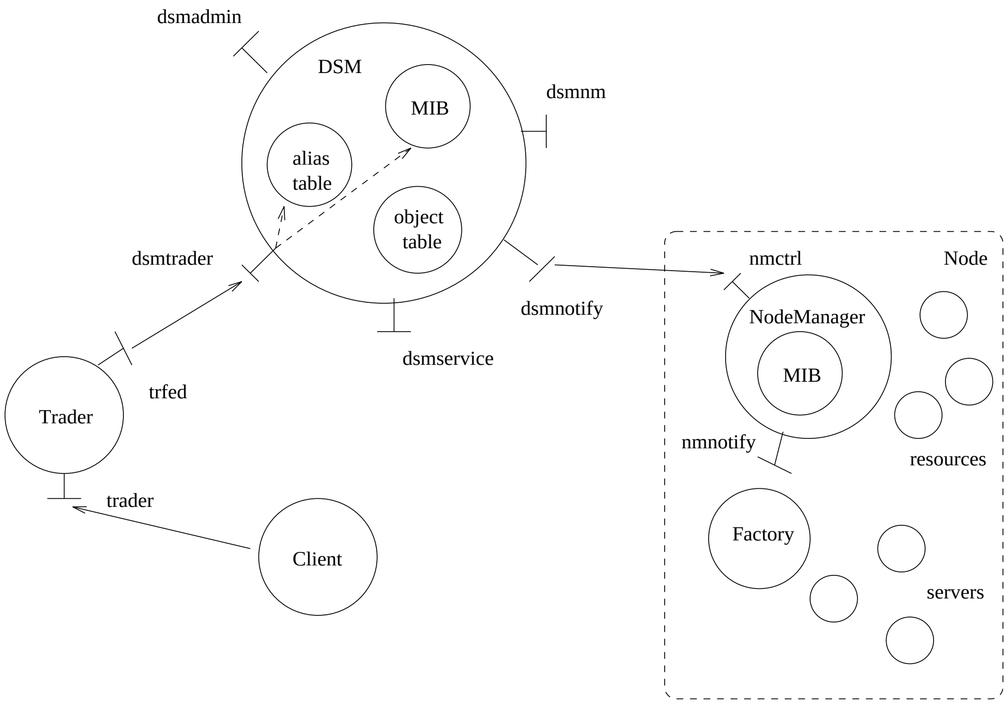

Figure 1 The management architecture.

We specify the collection of objects shown in Figure 1. The complete specification, called *MgtSystem*, defines a number of objects (*DSM*, *Nodes*, *Trader*) with a description of how they interact. To illustrate the specification of a single object, we will consider the *DSM* object. Some familiarity with the Z language is assumed.

4.1 Specifying a single object - the DSM

We model a managed object class by a ZEST class specification which encapsulates a number of fixed attributes, a state schema declaring the variable attributes, and a collection of operation schemas. In a fashion similar to Z, these are enclosed by lines with the class name at the top (here *DSM*).

Inside the class are the variables and operations used in the class, some of these are for internal use only, while others form part of the interface of the class. We document this by beginning each class with a description of the interfaces. A ZEST class may have several interfaces, and each interface defines what is visible at that particular interface of the object. A name appearing in an interface corresponds to either an operation or attribute, for example, the *dsmnm* interface consists of the operations *UpdateNode* and

NoResources (the **add** tells us these operations are included in the interface). Attributes and operations not appearing in any of the named interfaces are then internal to the class. In the formal specification, the names of the interfaces document the interface partition, and they are not used when invoking the operations.

Variable attributes are declared in a state schema, and their initial values are given by the schema *INIT* - here we specify that all variable attributes are in their own initial states (i.e. the ones given by *their INIT* schemas). The variable attributes in the DSM are in fact instances of appropriate classes. They represent the data concerning aliases (i.e. service descriptions), objects created and the results of node monitoring (the MIB). The declaration $dsm\_mib : DSM\_MIB$ declares dsm_mib to be an instance of the class DSM_MIB, which is specified as a ZEST class consisting of the data stored in the MIB together with operations to access and update that data. The *dsm_mib* contains information about the resources available on the nodes managed by the DSM. The DSM can manipulate the *dsm_mib* via its operations, for example, the information stored in the *dsm_mib* for a node can be updated by the node calling the DSM *UpdateNode* operation, which calls *Update_Node* in the *dsm_mib*.

Re-use is supported by the definition of *generic* classes and operations. For example, *DSMTable* is a generic class defined in terms of two generic parameters which can be instantiated with particular types (*Alias*, *Handle*, etc) in differing contexts.

DSM

interface *dsmnm* **add** *UpdateNode*, *NoResources*
interface *dsmservice* **add** *InstallAlias*, *RemoveAlias*, ...
interface *dsmtrader* **add** *LookupConstraintsFailed*, *LookupConstraintsSuccess*, ...
interface *dsmnotify* **add** *CapsuleTerminated*
$id : DSMId$

$alias\_table : DSMTable[Alias, AliasData]$
$object\_table : DSMTable[Handle, NotificationData]$
$dsm\_mib : DSM\_MIB$

INIT

$alias\_table.INIT \wedge object\_table.INIT \wedge dsm\_mib.INIT$

$UpdateNode \mathrel{\widehat=} dsm\_mib.Update\_Node[nmId?/nodeId?, nmData?/nodeData?, TRUE/result!]$

```
The other ZEST operations come here
```

The behaviour of a class is described by specifying ZEST operations. Each ZEST operation describes how the output is related to the input and how the state changes as a result of invoking the operation. For example, *UpdateNode* is an operation defined in terms of an operation in the *dsm_mib* object, but with the names of the inputs and outputs changed (e.g. *nmId?* is used instead of *nodeId?* etc).

4.2 The Node class

The *Node* class encapsulates the node entities and operations involved in management. It includes an instance of *NM*, the node manager class, and an instance of the *Factory* class. The factory is the entity responsible for the instantiation of services.

$Node$

interface $external$ **add** $CheckRequirements, SendInfo, Instantiate, Kill$

$nm : NM$
$factory : Factory$

$id : NMId$

$id = nm.id$

$CheckRequirements \mathrel{\widehat=} nm.CheckRequirements, SendInfo \mathrel{\widehat=} nm.SendInfo$
$Instantiate \mathrel{\widehat=} nm.Instantiate$
$\ldots$

An *axiomatic declaration* specifies that the identity ($id : NMId$) of the $Node$ class is equal to the identity of the node manager in the class ($id = nm.id$). The definition $SendInfo \mathrel{\widehat=} nm.SendInfo$ promotes the operation $SendInfo$ defined in the object nm to be an operation of the class $Node$.

The *Node Manager* monitors and controls the services instantiated on the node and collects information about the resources available. The node manager class is represented by NM below.

NM

interface $nmctrl$ **add** $CheckRequirements, SendInfo, Instantiate, Kill$
interface $nmnotify$ **add** $CapsuleTerminated$
$UpdateEntry == DSMId \times SentTime$
$UpdateList == \mathrm{seq}\ UpdateEntry$
$UpdateRecord == MIBUpdateTime \times UpdateList$
$id : NMId$

$nm\_mib : NM\_MIB$
$updateRecord : UpdateRecord$

$CheckRequirements$

$dsmId? : DSMId$
$updated! : Bool$
$dsmId! : DSMId$

$dsmId! = dsmId?$
let $mibTime == updateRecord.1 \land updateList == updateRecord.2$
in $(\exists_1\ sentTime : SentTime \bullet (dsmId?, sentTime) \in \mathrm{ran}\ updateList \land$
$(sentTime = mibTime \Rightarrow updated! = TRUE) \land$
$(sentTime \neq mibTime \Rightarrow updated! = FALSE))$

$SendInfo$

$dsmId? : DSMId$
$nmId! : NMId$
$nmData! : NM\_MIB$
$dsmId! : DSMId$

$nmId! = id \land nmData! = nm\_mib \land dsmId! = dsmId?$

$Instantiate \mathrel{\widehat=} \ldots$

The information concerning the resources provided by the node is stored in the *nm_mib*. This information is reported by the node manager to the *DSM* to keep the *dsm_mib* updated.

Different management domains, with different responsibilities and purposes may coexist. A *DSM* is the agent responsible for the management of the nodes which are members of one domain. The same node can be a member of different domains, being therefore under the control of more than one DSM. The node manager stores, for each DSM it is associated with, its identifier and the last time monitoring information was sent to update that DSM's *dsm_mib*. For this purpose, the *NM* state includes *updateRecord* of type *UpdateRecord* which is declared as follows (with appropriate definitions for the components):

$$UpdateEntry == DSMId \times SentTime$$
$$UpdateList == \mathrm{seq}\ UpdateEntry$$
$$UpdateRecord == MIBUpdateTime \times UpdateList$$

updateRecord contains information about the last time the *nm_mib* was updated (*MIBUpdateTime*) and the last time information from the *nm_mib* was sent to each DSM.

The operation *CheckRequirements* can be called on a node by any *DSM* that controls that node. This operation checks if the information in the *nm_mib* has not been changed since the last time it was sent to the DSM identified by *dsmId?*, in which case returns $updated! = TRUE$. If the *nm_mib* has been changed then the node manager will have to retrieve the new information from the *nm_mib*, as specified in *SendInfo*, and send it to the DSM. *Instantiate* is the operation provided by the node manager that allows a new instance of a service to be created via the factory, its definition is omitted here.

4.3 Specifying the interaction between objects

The complete specification contains definitions of the trader class (*Trader*) and a *Nodes* class. The interactions between objects of these classes is given by *MgtSystem*. This class contains an object of type Trader, a distributed systems manager (i.e. an object of type *DSM*) together with a set of nodes being managed (*Nodes*) on which services can be scheduled. The class *Nodes* will contain a set of objects of type *Node* together with operations to add and delete nodes etc. Operations are defined in *MgtSystem* which describe how objects in the class interact and communicate. We have omitted some of the operations and the type definitions.

MgtSystem

$trader : Trader$
$dsm : DSM$
$nodes : Nodes$
$suitableNodes : \mathbb{P}\, NMId$

INIT

$trader.INIT \land dsm.INIT \land nodes.INIT$

$$
\begin{array}{l}
\mathit{PollNodes} \mathrel{\widehat{=}} [\mathit{suitableNodes}? : \mathbb{P}\,\mathit{NMId}, \Delta(\mathit{suitableNodes})] \bullet \\
\quad \bigwedge \mathit{node} : \mathit{suitableNodes}? \bullet \\
\quad ((\mathit{node}.\mathit{CheckRequirements} \bullet [\mathit{updated}! = \mathit{TRUE} \wedge \mathit{suitableNodes}' = \mathit{suitableNodes} \cup \{\mathit{node}\}]) \\
\quad \vee((\mathit{node}.\mathit{CheckRequirements} \bullet [\mathit{updated}! = \mathit{FALSE}]) \mathbin{;} \mathit{node}.\mathit{SendInfo} \mathbin{;} \mathit{dsm}.\mathit{UpdateNode})) \\
\mathit{NewActivation} \mathrel{\widehat{=}} \\
\quad (\mathit{DSMCreateNewActSuccess} \mathbin{;} \\
\qquad (\mathit{PollNodes} \bullet [\mathit{suitableNodes} = \varnothing] \wedge [\mathit{result}! : \mathit{DSMServiceStatus} \mid \mathit{result}! = \mathit{NoSuitableNode}]) \\
\quad \vee(\mathit{PollNodes} \bullet [\mathit{suitableNodes} \neq \varnothing] \mathbin{;} \\
\qquad (\mathit{InstantiateService} \bullet [\mathit{instantiateResult}! = \mathit{FALSE}] \wedge \\
\qquad\quad [\mathit{result}! : \mathit{DSMServiceStatus} \mid \mathit{result}! = \mathit{FailedToCreateServer}]) \\
\quad \vee(\mathit{InstantiateService} \bullet [\mathit{instantiateResult}! = \mathit{TRUE}] \mathbin{;} \\
\qquad\quad \mathit{DSMLookupUpdate} \wedge [\mathit{result}! : \mathit{DSMServiceStatus} \mid \mathit{result}! = \mathit{DSMServiceSuccess}]))) \\
\ldots
\end{array}
$$

NewActivation specifies the behaviour corresponding to the creation of a new service instance (called an activation). When the DSM decides to create a new instance of a service, a suitable node will have to be found to allocate that instance. The DSM looks up in the *dsm_mib* for nodes that can provide the service requirements (this is specified in *DSMCreateNewActSuccess*) and polls them to check if they can still provide the same requirements.

The operation *PollNodes* specifies the sequence of operations that are performed when the suitable nodes are polled. The DSM invokes *CheckRequirements* on each node. This operation will return *updated!*, which is *TRUE* if the information in the *nm_mib* has not been changed since the last time it was sent to the DSM, and *FALSE* otherwise. In the last case the node manager will retrieve the updated information from the *nm_mib* (as specified in *SendInfo*) and send it to the *DSM* by calling its *UpdateNode* operation. *UpdateNode* will update the information in the *dsm_mib* for that node.

The $\bullet$ notation in $(\mathit{node}.\mathit{CheckRequirements} \bullet [\mathit{updated}! = \mathit{FALSE}])$ signifies enrichment in that $[\mathit{updated}! = \mathit{FALSE}]$ enriches the environment in which $\mathit{node}.\mathit{CheckRequirements}$ is interpreted. Distributed conjunction, as in $\bigwedge \mathit{node} : \mathit{suitableNodes}? \bullet \ldots$ is a convenient mechanism to take the conjunction for each *node* of type $\mathit{suitableNodes}?$ of the expression following the $\bullet$.

The operation *PollNodes* also illustrates communication in Object-Z/ZEST using the operator $\mathbin{;}$. The operator composes the two operations in the given order (therefore it is not commutative) with the following communication mechanism. Communication is left to right and hidden, outputs of the left operand equate to inputs of the right operand with the same basename (i.e. ignoring ! and ?) and both are hidden [5]. Thus in the communication $\mathit{node}.\mathit{SendInfo} \mathbin{;} \mathit{dsm}.\mathit{UpdateNode}$ the outputs of the first operation are used as inputs to the second operation. Sequential chains (as in $(\mathit{node}.\mathit{CheckRequirements} \bullet [\mathit{updated}! = \mathit{FALSE}]) \mathbin{;} \mathit{node}.\mathit{SendInfo} \mathbin{;} \mathit{dsm}.\mathit{UpdateNode}$) are interpreted left-associatively.

After *PollNodes* has been performed, the global variable *suitableNodes* will contain the set of nodes that can still provide the service requirements. *InstantiateService* specifies the behaviour corresponding to the allocation of a new service instance to a node selected from *suitableNodes*.

5 DERIVING TESTS FROM FORMAL SPECIFICATIONS

One potential application of formal techniques is in the automation of some or all of the process of testing. In order to support conformance testing of distributed systems, ODP defines conformance within the reference model. In addition, Part 4 of the ODP reference model defines an *architectural semantics* which provides an interpretation of the ODP modelling and specification concepts in Z and a number of other formal languages. This interpretation includes the definition of conformance in each language. Thus conformance assessment of an ODP system written using a formal technique begins with the architectural semantics, since it provides a means to interpret the specification, and hence to provide for meaningful test generation, and also to define the location of conformance and reference points, i.e. at which locations the testing will take place.

Because behaviour is represented by the Z operation schemas, a conformance statement in a Z specification corresponds to one or more operation schemas. Behaviour is said to conform if the post-condition and invariant predicates of this information manipulation are satisfied in the associated Z schemas.

A reference point will occur at an interface where tests can be applied to check for conformance. In Z, interfaces are associated with collections of operation schemas, so reference points can be considered to reside at the pre-conditions of the operation schemas. However, a Z specification will not in itself identify which reference points are programmatic, perceptual, interworking or interchange. Such identification would accompany the specification as informal commentary.

Appropriate work on test generation from Z specifications includes [18, 3, 1, 17, 7], however, little of this work is specifically targeted towards distributed systems or ODP in particular. Exceptions to this include [3, 17]. [17] describes important work done under the Prost project in the UK on the testability of managed object specifications. ZEST is used to specify managed objects, and an inheritance hierarchy is constructed which facilitates the construction of a sound and complete test suite. Importantly, though, the test generation aims to supply heuristics and is not automatic. The heuristics provide a collection of tests together with a residual component which makes explicit the functionality not covered by the test suite. The tests generated form an independent and orthogonal collection of tests.

Because of the inheritance hierarchy, the reuse of tests between related specifications is maximised. A prototype tool-set developed by Logica provides organisational support for the collection of test specifications as they are generated.

We illustrate the use of the method defined in [17] by deriving tests for operations specified in our management architecture. The method derives a formal specification (also written in Z) of conformance tests from each managed object, by producing a collection of tests for each operation in the managed object.

The method describes three actions: partitioning, weakening and simplification, to construct a set of tests for each operation. The method is based on the idea of testing only some of the interesting inputs and only some of the consequences of the operation, a test is therefore an abstraction of the original operation. Each time an action is applied it divides an operation into several parts, each of which will either be a test or be further divided. The division must satisfy the following rules (and heuristics enforce this), where Op is the operation under test and $\{T_i\}$ the collection of tests at this stage: *soundness* i.e. $\forall i \bullet Op \Rightarrow T_i$; *completeness* i.e. the collection of tests must cover the operation, so $Op = \bigwedge_i T_i$.

As an example, consider the operation *RemoveAlias* which is part of the complete DSM specification:

$$
\begin{array}{|l}
\hline
\textit{RemoveAlias} \\
\Delta(\textit{alias\_table}) \\
\textit{alias}? : \textit{Alias} \\
\textit{status}! : \textit{DSMServiceStatus} \\
\hline
(\textit{alias}? \notin \operatorname{dom} \textit{alias\_table.table} \Rightarrow \textit{status}! = \textit{NonExistantAlias}) \\
(\textit{alias}? \in \operatorname{dom} \textit{alias\_table.table} \wedge \textit{alias\_table.table}(\textit{alias}?).2 = \textit{TRUE} \Rightarrow \textit{status}! = \textit{AliasActive})
\end{array}
$$

$$
\begin{array}{l}
(alias? \in \mathrm{dom}\ alias\_table.table \land alias\_table.table(alias?).2 \neq TRUE \land \\
\quad alias\_table.table(alias?).3 = TRUE \Rightarrow status! = AliasPosted) \\
((alias? \in \mathrm{dom}\ alias\_table.table \land alias\_table.table(alias?).2 \neq TRUE \land \\
\quad alias\_table.table(alias?).3 \neq TRUE) \Rightarrow \\
\quad (status! = DSMServiceSuccess \land alias\_table.Remove\_from\_table[alias?/key?][TRUE/result!]))
\end{array}
$$

Partitioning the operation involves deriving a set of tests each covering a different aspect of the operations pre-condition. The partition is defined by predicates H_i, and for an operation Op defined in terms of a declaration D and a predicate P, we derive tests T_i given by

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{Op} \\
D \\
\hline
P \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\multicolumn{1}{l}{T_i} \\
D \\
\hline
H_i \Rightarrow P \\
\hline
\end{array}
$$

This will split the operation into a collection of tests such that $Op \mathrel{\widehat=} T_1 \land \ldots \land T_n$. For example, for the operation $RemoveAlias$ we could partition as follows:

H_1 - $alias? \notin \mathrm{dom}\ alias\_table.table$
H_2 - $alias? \in \mathrm{dom}\ alias\_table.table$

Choosing the predicates H_i to perform the partition is part of the testers skill, the aim is to choose a partition that simplifies what is being tested. The partitioning process generates (after simplification) tests T_1 and T_2 (where we will subdivide T_2 further) given by:

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{T_1} \\
\Delta(alias\_table) \\
alias? : Alias \\
status! : DSMServiceStatus \\
\hline
(alias? \notin \mathrm{dom}\ alias\_table.table \Rightarrow status! = NonExistantAlias) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{T_2} \\
\Delta(alias\_table) \\
alias? : Alias \\
status! : DSMServiceStatus \\
\hline
(alias? \in \mathrm{dom}\ alias\_table.table \land alias\_table.table(alias?).2 = TRUE \Rightarrow status! = AliasActive) \\
(alias? \in \mathrm{dom}\ alias\_table.table \land alias\_table.table(alias?).2 \neq TRUE \land \\
\quad alias\_table.table(alias?).3 = TRUE \Rightarrow status! = AliasPosted) \\
((alias? \in \mathrm{dom}\ alias\_table.table \land alias\_table.table(alias?).2 \neq TRUE \land \\
\quad alias\_table.table(alias?).3 \neq TRUE) \Rightarrow \\
\quad (status! = DSMServiceSuccess \land alias\_table.Remove\_from\_table[alias?/key?][TRUE/result!])) \\
\hline
\end{array}
$$

Repeated partitioning can be applied on the input e.g. on the value of $alias\_table.table(alias?).2$ etc, to produce four tests, the last being:

$$
\begin{array}{|l}
\multicolumn{1}{l}{T_4} \\
\hline
\Delta(alias\_table) \\
alias? : Alias \\
status! : DSMServiceStatus \\
\hline
((alias? \in \mathrm{dom}\ alias\_table.table \wedge alias\_table.table(alias?).2 \neq TRUE \wedge \\
\quad alias\_table.table(alias?).3 \neq TRUE) \Rightarrow \\
\quad (status! = DSMServiceSuccess \wedge alias\_table.Remove\_from\_table[alias?/key?][TRUE/result!])) \\
\hline
\end{array}
$$

Weakening can now be applied, which loosens the constraints on the output (and after-state). Weakening an operation *Op* produces a test T_w and a residual part T_r, which documents the aspects of the operation we are not testing, with $Op \mathrel{\widehat{=}} T_w \wedge T_r$. In *RemoveAlias* we are not interested in checking the output *status*!, just that the *alias_table* has been altered correctly, so we weaken the test T_4 to derive the weakening T_w. The remaining component, T_r, will now document the aspects of *RemoveAlias* not covered by the conformance testing.

$$
\begin{array}{|l}
\multicolumn{1}{l}{T_w} \\
\hline
\Delta(alias\_table) \\
alias? : Alias \\
status! : DSMServiceStatus \\
\hline
((alias? \in \mathrm{dom}\ alias\_table.table \wedge alias\_table.table(alias?).2 \neq TRUE \wedge \\
\quad alias\_table.table(alias?).3 \neq TRUE) \Rightarrow \\
\quad alias\_table.Remove\_from\_table[alias?/key?][TRUE/result!] \\
\hline
\end{array}
$$

This method works well on the individual schema level, however, most of the interesting behaviour in a managed object results from a process of combining operations using the schema calculus. Current test generation methods such as the one outlined above need to be extended to produce tests from operations defined using the schema calculus. For example, if the operations Op_1 and Op_2 produce complete and sound tests $\{T_i\}$ and $\{R_j\}$ respectively, can we derive a suitable collection of tests for the operation $Op_1 \mathbin{⨾} Op_2$?

6 CONCLUSIONS

The use of formal description techniques is increasing within ODP, and a number of proposals to specify managed objects formally have been made [14, 15, 12, 19, 10]. However, existing work in this area has concentrated on small scale case studies involving just one managed object (often the Sieve or LOG managed object). At ISINM'95 we reported on differing proposals to the formal specification of managed objects [4]. Further work in the UK has produced guidelines on how to specify managed objects in Z [20], and derived a method for producing tests derived from these formal specifications.

The aim of our work has been to test the applicability of these methods by specifying a larger scale case study of a new application (rather than a behaviour that is well documented). While Z is not necessarily a perfect vehicle for managed object specification, it does offer considerable benefits over current practice. For specifications where behaviour is important or subtle, GDMO clearly needs enhancement, and Z offers a wide user base and suitable facilities for abstraction. The ability to derive tests from formal specifications adds another dimension to the usefulness of the technique, although further work is needed in this area as outlined above.

REFERENCES

D. Carrington and P. Stocks. A tale of two paradigms: Formal methods and software testing. In J.P. Bowen and J.A. Hall, editors, *ZUM'94, Z User Workshop*, pages 51–68, Cambridge, United Kingdom, June 1994.

E. Cusack and G. H. B. Rafsanjani. ZEST. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 113–126. Springer-Verlag, 1992.

E. Cusack and C. Wezeman. Deriving tests for objects specified in Z. In J. P. Bowen and J. E. Nicholls, editors, *Seventh Annual Z User Workshop*, pages 180–195, London, December 1992. Springer-Verlag.

J. Derrick, P.F. Linington, and S.J. Thompson. Formal description techniques for object management. In A. S. Sethi, Y. Raynaud, and F. Faure-Vincent, editors, *Fourth IFIP/IEEE International Symposium on Integrated Network Management (ISINM '95)*, pages 641–653. Chapman and Hall, May 1995.

R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, September 1995.

G. P. A. Fernandes and I. A. Utting. An Object-Oriented Model for Management of Services in a Distributed System. To appear in the ECOOP'96 workshop on Object Oriented Technology for Service and Network Management, 1996.

H-M. Horcher. Improving software tests using Z specifications. In J. P. Bowen and M. G. Hinchey, editors, *Ninth Annual Z User Workshop*, LNCS 967, pages 152–166, Limerick, September 1995. Springer-Verlag.

ISO/IEC 10040. *Information Technology - Open Systems Interconnection - Systems Management Overview*, 1992.

ISO/IEC JTC1/SC21/WG4 10165-4 (X.722). *Information Technology - Open Systems Interconnection - Structure of Management Information - Part 4: Guidelines for the Definition of Managed Objects*, 1991.

ISO/IEC JTC1/SC21/WG4 N1644. *Liaison to CCITT SG VII concerning the use of Formal Techniques for the specification of Managed Objects*, December 1992.

ITU Recommendation X.901-904 — ISO/IEC 10746 1-4. *Open Distributed Processing - Reference Model - Parts 1-4*, July 1995.

N D North. RSL specification of the log managed object. Technical report, National Physical Laboratory, UK, 1992.

G. H. B. Rafsanjani. ZEST - Z Extended with Structuring: A users's guide. Technical report, British Telecom, June 1994.

S. Rudkin. Modelling information objects in Z. In J. de Meer, V. Heymer, and R. Roth, editors, *IFIP TC6 International Workshop on Open Distributed Processing*, pages 267–280, Berlin, Germany, September 1991. North-Holland.

L. Simon and L. S. Marshall. Using VDM to specify OSI managed objects. In K R Parker and G A Rose, editors, *Formal Description Techniques 1991*. North Holland, 1992.

J. M. Spivey. *The Z notation: A reference manual.* Prentice Hall, 1989.

S. Stepney. Testing as Abstraction. In J. P. Bowen and M. G. Hinchey, editors, *Ninth Annual Z User Workshop*, LNCS 967, pages 137–151, Limerick, September 1995. Springer-Verlag.

P. Stocks and D. Carrington. Deriving software test cases from formal specifications. In *6th Australian Software Engineering Conference*, pages 327–340, July 1991.

C. Wezeman and A. J. Judge. Z for managed objects. In J. P. Bowen and J. A. Hall, editors, *Eighth Annual Z User Workshop*, pages 108–119, Cambridge, July 1994. Springer-Verlag.

H. B. Zadeh. Using ZEST for Specifying Managed Objects. Technical report, British Telecom, January 1996.