Formal Specification of Distributed Services Management

G. P. A. Fernandes* and J. Derrick

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(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. The successful implementation and interoperability of these facilities depends to a large extent on a precise, but implementation independent, specification of their behaviour. This paper examines the use of formal techniques to describe managed objects. We describe a formal specification of a scheduling architecture for the management of services in a distributed system.

1 Introduction

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 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. This paper presents a case study in such a definition using an object-oriented variant of the formal technique Z.

The specification we develop is of a distributed systems manager[5], whose main goals are to manage the resources available in order to fulfil the quality of service (QoS) requirements of the application services and to release the application programmer from the job of allocating services to nodes. The successful implementation and interoperability of managed objects such as these depend to a large extent on a precise (but implementation independent) specification of their behaviour. 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. The aim of our work is to test the applicability of FDTs to managed object specification by formally specifying a realistic and large application.

The paper is structured as follows: in section 2 we describe the management model, section 3 explains informally the distributed systems manager, the formal specification is presented in section 4, and we conclude in section 5.

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

2 Management Model

The management model presented in this paper is a distributed object-oriented model based on the Open Distributed Processing (ODP) Reference Model [10] and the OSI management model [7, 6]. One of the most important ideas in OSI Systems Management is the use of object-oriented principles to define management information and interfaces. The OSI management model views the devices in the network that are subject to management as *managed objects*. 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.

Formal description techniques are playing an increasing role within ODP, and a number of proposals to specify managed objects formally have been made [13, 14, 11, 17, 9]. However, existing work in this area has concentrated on small scale case studies. [3] surveys some of the available techniques. Further work in the UK has produced guidelines on how to specify managed objects in Z [18], and derived a method for producing tests derived from these formal specifications. Z [15] is a state based 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. Z has proved to be one of the most enduring formal description techniques, and has significant industrial usage and support. Object-Z [4] and ZEST [12, 2] are similar object-oriented extensions of Z. Throughout the paper we assume the reader is familiar with the Z language.

3 The Distributed System Manager

In our architecture, distributed scheduling is used to locate a new application service on an appropriate node, taking into account the current state of the system and the quality of service (QoS) requirements of the service. The *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 compares the 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 *Node Manager* is an entity local to a node, responsible for the management of that node and reporting monitoring information to the DSM. The node manager monitors and controls the services instantiated on the node and collects information about the resources available.

All newly created services are instantiated by the DSM upon request by the *trader*. The 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 Formal Specification

This section illustrates how we have used ZEST to specify the scheduling architecture. We specify the collection of objects shown in Figure 1. The complete specification, called *MgtSystem*, will consist of a number of objects (*DSM*, *Nodes*, *Trader*) with a description of how they interact. To illustrate the specification of a single object, let us first consider the *DSM* object. We model a managed

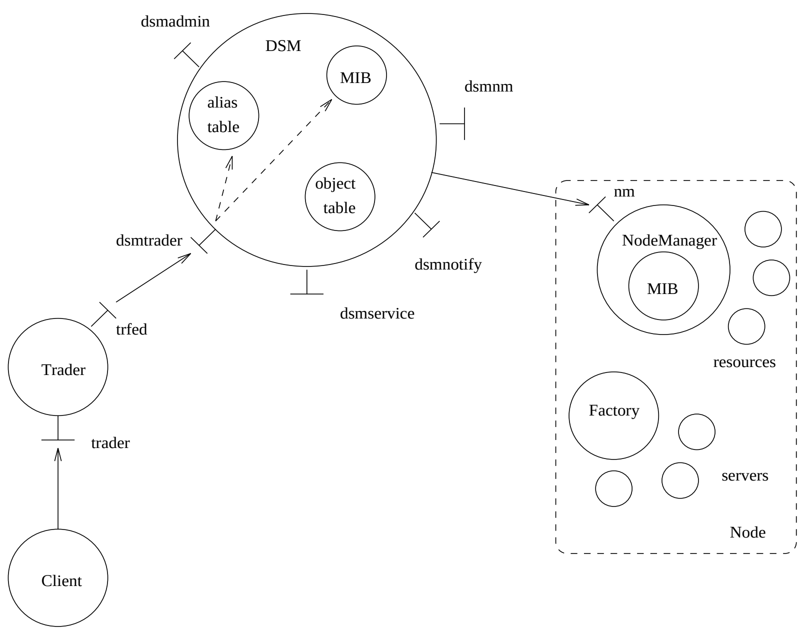

Fig. 1. The management architecture.

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. A ZEST object may have several interfaces, and each interface defines what is visible at that particular interface of the object. In the formal specification, the interfaces just serve to document design decisions, and do not effect the semantics of the specification.

Data concerning aliases (i.e. service descriptions), objects created and the results of node monitoring (the MIB) are held in the DSM. In our object-oriented paradigm these are specified as instances of appropriate classes. For example, 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 (complex) data stored in the MIB together with operations to access and update that data.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\,DSM\,\rule{20em}{0.4pt}} \\
\textbf{interface}\ \mathit{dsmnm}\ \textbf{add}\ \mathit{UpdateNode}, \mathit{NoResources} \\
\textbf{interface}\ \mathit{dsmservice}\ \textbf{add}\ \mathit{InstallAlias}, \mathit{RemoveAlias}, \ldots \\
\textbf{interface}\ \mathit{dsmtrader}\ \textbf{add}\ \mathit{LookupConstraintsSuccess}, \mathit{LookupConstraintsFailed} \ldots \\
\textbf{interface}\ \mathit{dsmnotify}\ \textbf{add}\ \mathit{CapsuleTerminated} \\
id : DSMId \\
\quad \begin{array}{|l}
\hline
alias\_table : DSMTable[Alias, AliasData] \\
object\_table : DSMTable[Handle, NotificationData] \\
dsm\_mib : DSM\_MIB \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\multicolumn{1}{l}{\_\,INIT\,\rule{15em}{0.4pt}} \\
alias\_table.INIT \land object\_table.INIT \land dsm\_mib.INIT \\
\hline
\end{array} \\
\texttt{The ZEST operations come here} \\
\hline
\end{array}
$$

The $alias\_table$ and $object\_table$ are objects containing abstractions of hash tables, but storing elements of different types. Z allows for re-use in this situation by defining the *DSMTable* class in terms of two generic parameters which can be instantiated with particular types (*Alias*, *Handle*, etc) in differing contexts. The (unnamed) state schema contains all the variable attributes ($alias\_table$ etc). The schema *INIT* then specifies the initial state of the DSM. The behaviour of the DSM 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.

The ability to specify policies (e.g. scheduling or general management policies) in a management application is an important area of concern. We illustrate here how we can specify a scheduling policy as part of a Z operation. Scheduling is concerned with matching service QoS requirements with resource provision. For example, consider a distributed system which supports scheduling with respect to storage type and availability, network devices (including bandwidth etc) and other devices (e.g. audio, video). To define the types in Z, we define a QoS entry to consist of a label (given by *QoSType*) together with actual parameters specifying the QoS level required. The *ServerQoS* type represents the sequence of QoS requirements the service needs in order to be instantiated. For example, the sequence

$$\langle\langle STORAGE, \langle RAM, 4\rangle\rangle, \langle STORAGE, \langle CompactDisk, 6\rangle\rangle\rangle$$

represents a requirement for a RAM with available storage 4 and compact disk with available storage 6 (in suitable units).

We are now in a position to specify a simple node selection policy. The operation *SelectSuitableNodes* returns the set of nodes which match all the requirements made by the inputted $serverQoS?$ For each type of QoS requirement ($STORAGE$, $DEVICE$ etc) it selects all nodes that meet those particular requirements, and then forms the intersection over all $QoSTypes$. To select nodes that meet particular requirements, the operation invokes operations ($Lookup\_Storage$ etc) within the dsm_mib to access information about the nodes, for example, $dsm\_mib.Lookup\_Storage$ returns the set of nodes satisfying the storage requirements.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SelectSuitableNodes}} \\
\hline
serverQoS? : ServerQoS \\
suitableNodes! : \mathbb{P}(NMId \times \mathbb{N} \times AllocationTime) \\
serverQoS! : ServerQoS \\
match? : Match \\
\hline
suitableNodes! = \\
\{dsm\_mib.Lookup\_Storage \mid \langle STORAGE, qosRecord \rangle \in \operatorname{ran} serverQoS? \\
\quad \wedge match? = Exact \wedge qosRecord.1 = storageType? \\
\quad \wedge qosRecord.2 = storageRequirement? \wedge x \in nodeSet! \bullet x\} \\
\cap \\
\{dsm\_mib.Lookup\_Device \mid \langle DEVICE, qosRecord \rangle \in \operatorname{ran} serverQoS? \\
\quad \wedge match? = Exact \wedge qosRecord.3 = deviceType? \\
\quad \wedge qosRecord.4 = deviceRequirement? \wedge x \in nodeSet! \bullet x\} \\
\cap \\
\{dsm\_mib.Lookup\_Network \mid \langle NETWORK, qosRecord \rangle \in \operatorname{ran} serverQoS? \\
\quad \wedge match? = Exact \wedge qosRecord.5 = networkType? \\
\quad \wedge qosRecord.6 = networkRequirement? \wedge x \in nodeSet! \bullet x\} \\
serverQoS! = serverQoS? \\
\hline
\end{array}
$$

The output of this operation is a set of node identifiers ($NMId$) together with a value indicating the current level of resource availability and latest allocation time. We also output the $serverQoS?$ for reasons we describe below, for a simple policy this would not be necessary. The predicate in this operation represents the selection *policy* of this DSM. It is now a simple matter to specify more sophisticated policies, which can then be combined using the Z schema calculus in a rather elegant fashion.

If no node was found that can provide all the QoS requirements specified for a service (i.e. $suitableNodes! = \varnothing$), alternative scheduling policies can be followed. For example, we can specify a policy to select nodes according to their allocation time. This policy is applied when the previous policy failed to find a collection of suitable nodes (i.e. $suitableNodes? = \varnothing$). Under these circumstances it selects another set of suitable nodes as output ($suitableNodes! = \ldots$), now selected according to their allocation time. We pass on the inputs as outputs in each policy so that the policies can be combined by specifying a sequence of policies composed together:

$$SelectSuitableNodes \fatsemi Policy_2 \fatsemi \ldots Policy_n$$

This composite policy will implement $SelectSuitableNodes$ first, then if and only if this fails it will try $Policy_2$, then iff this fails it will try $Policy_3$ etc. A conjunction of policies could be specified by $Policy_1 \wedge \ldots \wedge Policy_n$, and the disjunction allowing alternative strategies to be deployed.

The complete specification (which is too long to repeat here) contains definitions of the trader class (*Trader*) and a *Nodes* class. The interactions between objects of these classes are given by *MgtSystem*. We have omitted some of the operations and the type definitions.

MgtSystem

$$
\begin{array}{l}
trader : Trader \\
dsm : DSM \\
nodes : Nodes \\
\hline
\vdots \\
NewActivation \mathrel{\widehat=} \\
\quad (DSMCreateNewActSuccess \fatsemi \\
\qquad (PollNodesFailed \\
\qquad \wedge [result! : DSMServiceStatus \mid result! = NoSuitableNode]) \\
\quad \vee (PollNodesSuccess \fatsemi \\
\qquad ((InstantiateServiceFailed \wedge \\
\qquad [result! : DSMServiceStatus \mid result! = FailedToCreateServer]) \\
\quad \vee (InstantiateServiceSuccess \fatsemi \\
\qquad DSMLookupUpdate \wedge \\
\qquad [result! : DSMServiceStatus \mid result! = DSMServiceSuccess])))) \\
TraderLookup \mathrel{\widehat=} trader.Lookup \\
RequestService \mathrel{\widehat=} \\
\quad TraderLookup \fatsemi (DSMLookupConstraints \fatsemi \\
\qquad (DSMCreateNewActFailed \vee NewActivation \\
\qquad \vee (DSMReturnExistingAct \wedge \\
\qquad [result! : DSMServiceStatus \mid result! = DSMServiceSuccess])))
\end{array}
$$

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. Operations are defined in *MgtSystem* which describe how objects in the class interact and communicate.

The transaction corresponding to a sequence of operations that need to be performed in order to provide a service requested by a client is specified by the *MgtSystem RequestService* operation. Communication in Object-Z/ZEST is illustrated using the operator $\fatsemi$. It 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 [4]. A client may request a service by calling the operation *Lookup* provided by the trader. The trader recognises the service offer as a proxy and forwards the request to the DSM calling its *LookupConstraints* operation. This operation looks up in the *alias_table* the alias specified whose properties match the constraints specified by the input. An activation is an instance of the service described by the alias. If there are no activations for that alias or the limit of activations has not been reached yet, a new activation is allowed. *DSMCreateNewActFailed* specifies the situation where no suitable node was found to allocate the new activation, and *DSMCreateNewActSuccess* specifies when at least one suitable node was found. The last case is specified by *NewActivation*. The suitable nodes are polled to check if they can still provide the specified requirements. *PollNodesFailed* specifies when none of the nodes can provide the requirements while *PollNodesSuccess* returns the most suitable node of those polled. The new activation is allocated to this node by calling the operation *Instantiate* on its local node manager. The node manager may not succeed to create the new server (specified by *InstantiateServiceFailed*), otherwise the information stored in the *DSM* is updated by *DSMLookupUpdate* with the information returned by *InstantiateServiceSuccess* after creating the new activation. When the activations limit for an alias has already been reached, a reference to one of the existing activations can be returned to the client. This behaviour is specified in *DSMReturnExistingAct*.

In this fashion the schema calculus can be used to mirror the structure of a potential implementation, yet remain at a suitable level of abstraction.

5 Conclusions

No design notation is perfect, and Z suffers from its idiosyncrasies. However, for this type of specification it offers the correct level of abstraction and suitable facilities. Z seems to offer particular advantages for the specification of managed objects. In particular, managed object definitions involve heavy use of state making a state based language particularly suitable. Complex sequences of manipulations of state can be succinctly represented using the schema calculus (e.g. by specifying sequencing using $\fatsemi$ or composition using conjunction). Using a process algebra, such as LOTOS [1] would be feasible but, with emphasis on the above aspects (particularly the need to specify state), more clumsy. When specifying the interaction of more than one managed object (e.g. the DSM and Node Manager) it is necessary to be able to specify communication between objects. Using Z would not be sufficient here as it has no standard way of expressing communication (or concurrency), however, both object-oriented variants ZEST and Object-Z offer similar support for describing communication between objects.

Many of the case studies using object-oriented variants of Z have been undertaken by the designers of the languages, part of our aim was to test the applicability of these techniques when not versed in the subtleties of the language. To that extent the application was a success. The designer of the scheduling architecture (the first named author) found the language a reasonably natural vehicle to express an abstraction of the implementation.

The specification in this paper is currently being implemented in ANSA. Once completed we plan to test the implementation with tests generated from the formal specification according to the guidelines developed in [16]. The method developed in [16] does not support *automatic* test generation, but heuristics provide a means to derive a complete and orthogonal collection of tests.

References

1. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1988.
2. 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.
3. 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.
4. 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.
5. 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.
6. ISO/IEC 10040. *Information Technology - Open Systems Interconnection - Systems Management Overview*, 1992.
7. ISO/IEC 7498-4 | CCITT REC. X.700. *Information Processing Systems - Open Systems Interconnection - Basic Reference Model - Part 4: Management Framework*, 1989.
8. 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.
9. 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.
10. ITU Recommendation X.901-904 — ISO/IEC 10746 1-4. *Open Distributed Processing - Reference Model - Parts 1-4*, July 1995.
11. N D North. RSL specification of the log managed object. Technical report, National Physical Laboratory, UK, 1992.
12. G. H. B. Rafsanjani. ZEST - Z Extended with Structuring: A users's guide. Technical report, British Telecom, June 1994.
13. 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.
14. 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.
15. J. M. Spivey. *The Z notation: A reference manual*. Prentice Hall, 1989.
16. 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.
17. 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.
18. H. B. Zadeh. Using ZEST for Specifying Managed Objects. Technical report, British Telecom, January 1996.

This article was processed using the LaTeX macro package with LLNCS style