Developing Agents with the Managed Object Toolkit

GORDON MCNAIR AND JASON ETHERIDGE

***Abstract*—Software agents are critical to the management of network elements as they present these devices to managing applications using a structured interface and use a standards conformant protocol for communications which allows for interoperability.**

For a software development organisation the key is to be able to efficiently deliver software agents for network elements to its customers. Toolkits provide a method for improving the delivery of the agents.

This paper describes our experiences in constructing OSI software agents with the Hewlett Packard OpenView Managed Object Toolkit (MOT) and the lessons learned.

***Keywords*—OSI, TMN, CMIP, CMIS, OpenView, DM, MOT**

***Source of Publication*—Proceedings of the International OpenView Forum Conference, Anaheim, USA, June 1997.**

1 INTRODUCTION

Software agents are the point where the network management system meets the network elements being managed. The defined and structured world of standard interfaces and protocols used by managers must be mapped onto the particular implementation of the devices the software agent is responsible for.

1.1 REQUIREMENTS ON SOFTWARE AGENTS

Software agents exist in a hostile world where continual demands are being made of them from manager applications and the device being managed is generating sometimes large volumes of data and alarms. Figure 1 depicts a typical agent environment:

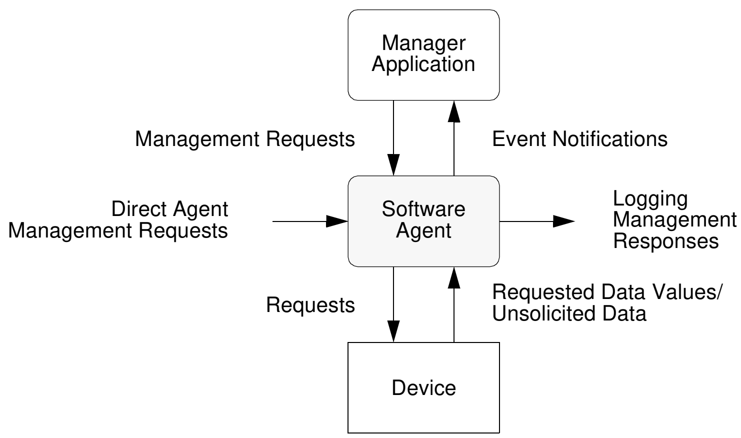

Figure 1: The Agent Environment

The agent must have an infrastructure for dealing with:

- communication with the manager application using the appropriate protocol
- communication with the device using appropriate method - most often proprietary
- dealing with ad-hoc management requests from the manager application
- handling of unsolicited data from the device
- collecting requested data from the device
- activity and error logging
- direct management of the agent - for example, configuration of the agent's behaviour

In addition the agent must manage the data associated with the device from the manager application's perspective and deal with the collection and caching of this information from the device in a suitable way. A containment tree built from the MIBs being presented to the manager application is the usual approach to performing this function in OSI software agents.

Many of these features are the same or use a similar pattern in all agents. This is a prime candidate for reuse of components from previous agents, use of existing frameworks or toolkits to generate components of the agent.

1.2 POSSIBLE APPROACHES TO BUILDING SOFTWARE AGENTS

Building a complete agent each time one is required is expensive and time-consuming. It involves re-implementing many components that are the same or similar to previously built agents. Reducing this "reinventing of the wheel" and thus reducing the cost and effort required to build agents is an important goal.

The possible alternative approaches to building software agents are:

- Reuse previous agents as models for future development
- Flexible or generic implementations of agents that can be reused
- In-house toolkits
- Commercial toolkits

1.2.1 REUSE OF AGENTS

Reuse previous agents as models for development improves the situation. The issues are:

- organisation still has to build the complete agent the first time
- each agent was probably tuned to situation so reuse may be difficult
- the specifics of a particular agent will be mixed with the framework of the agent

1.2.2 FLEXIBLE IMPLEMENTATIONS

Flexible or generic implementations will make reuse more likely. The problem to overcome is that of getting the first project to be produced flexibly. This cost must be borne by the developer organisation or the customer, and neither may be willing to pay for it. Also the organisation is now responsible for maintaining a body of code.

1.2.3 IN-HOUSE TOOLKITS

Toolkits which provide a framework of components and possibly generate components from a specification language are a good approach. In-house toolkits have the problem of the investment required to create them in the first place. Also once built the toolkit must be maintained and supported, and very few organisations are in business to maintain tools.

1.2.4 COMMERCIAL TOOLKITS

Commercial toolkits have all the benefits of in-house toolkits without the maintenance load. They are also designed to be generic and support a number of usage scenarios. These commercial tools have to be competitive in terms of cost and performance for the manufacturer to gain market share. The manufacturer should have a support infrastructure in place and enhancements are being produced.

1.3 THE APPROACH WE ADOPTED

In light of the above, CiTR decided to use a commercial toolkit to provide us with the ready framework for our agent development activities.

Using commercial toolkits which provide such a framework leave the software development organisation with the ability to focus on the unique parts of the implementation which are specific to the agent being developed, such as obtaining values from the device and mapping management requests into actions on the device.

This paper now describes our experiences in constructing software agents with the Hewlett Packard OpenView Managed Object Toolkit (MOT) which provides a framework as discussed above, and the lessons learnt in the process.

2 DESCRIPTION OF THE PROJECT

2.1 PROJECT OVERVIEW

In this project, CiTR developed software agents for an ATM switch produced by a major vendor. The project involved implementing a software agent for the ATM switch using the ATM Forum M4 Interface Specification [1], and implementing an agent for the proposed ATM Forum M4 Public Network View Specification [2]. The agents are referred to as the Network Element View (NEV) agent and the Public Network View (PNV) agent.

The customers of the switch vendor are users of ATM switches who require a standard interface for the management of their ATM network and easy integration with any existing TMN environments by supporting the standard MIBs. This customer may purchase equipment from a number of vendors but want their management system to be able to use one management approach, hence the use of the M4 specifications for NEV agents and PNV agents.

2.2 PROJECT EXECUTION

The development of the agents was divided between CiTR and the vendor's development team. CiTR was responsible for building the agent software except for the specific software interface to the ATM switch which was built by the vendor.

2.3 SOLUTION ARCHITECTURE

The architecture of the NEV agent consists of the components described below. Figure 2 illustrates the relationships between the components.

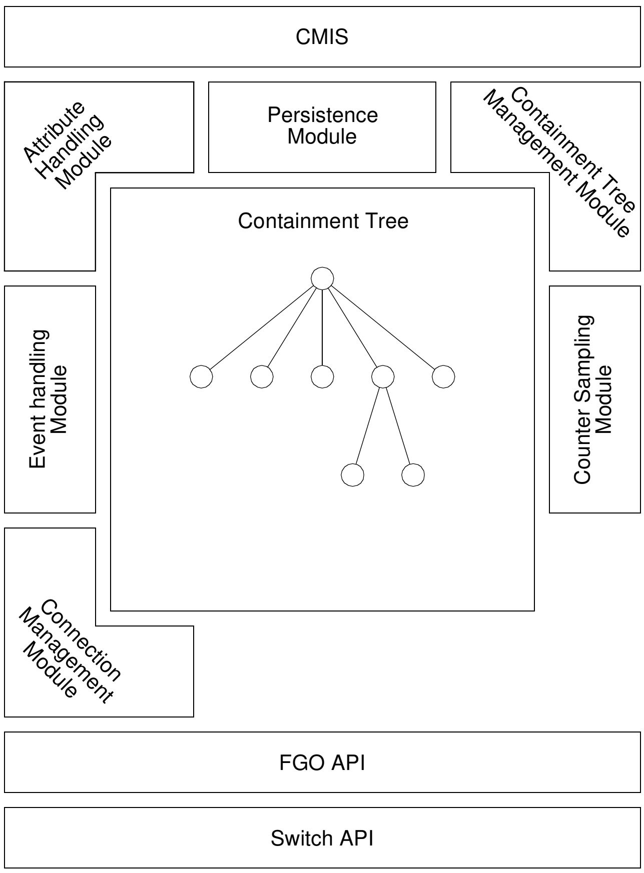

Figure 2: M4 NEV Agent Architecture Overview

The following components are defined in the figure 2:

- **CMIS**—MOT-provided layer encapsulating the CMIP stack; handles all CMIP communications
- **Containment tree management module**—Performs containment tree initialisation and ensures synchronisation with the network element being managed.
- **Persistence module**—Allows contents of the containment tree to be written into (and restored from) persistent storage. This was CiTR-developed, and made no use of the MOT persistence mechanism.
- **Attribute handling module**—Links CMIS M-Gets and M-Sets to the FGO API (for setting and retrieving values in the network element)
- **Containment tree**—MOT-provided infrastructure for managing the containment tree
- Event handling module—Catches events generated from the network element (via the FGO API), and translates them into M-Event-Reports (easily constructed and emitted using MOT helper functions)
- **Connection management module**—Implementation of cross-connection functionality; M-Actions sent to the agent's ATM fabric managed object are translated into calls on the FGO API, which creates cross-connections in the underlying ATM switch being managed. Also handles construction of appropriate managed objects to represent cross-connections in the containment tree.
- **Counter sampling module**—A background task that periodically samples the network element to track certain performance counters. Counter thresholds, established by the manager application by interacting by the managed objects in the containment tree, are monitored and M-Event-Reports are generated if thresholds are crossed.
- **FGO API**—Abstraction layer to allow easy interaction with the network element
- **Switch API**—API supporting network-element specific proprietary protocol

The architecture of the PNV agent consists of the components described below. Figure 3 illustrates the relationships between the components.

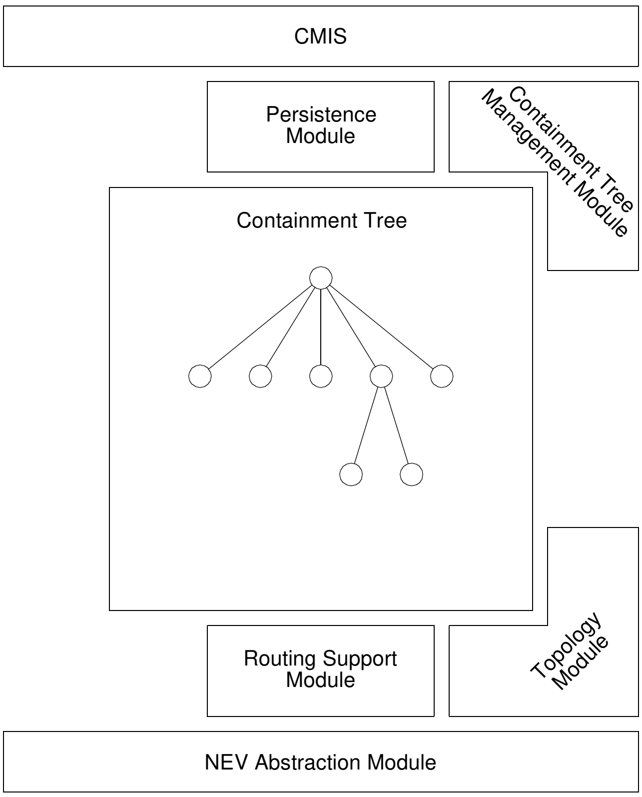

Figure 3: M4 PNV Agent Architecture Overview

The following components are defined in the figure 3:

- **CMIS**—Same as in the NEV agent above
- **Containment tree management module**—Same as in the NEV agent above
- **Persistence module**—Same as in the NEV agent above
- **Containment tree**—Same as in the NEV agent above
- **Topology module**—Maintains an abstraction, based on the contents of the containment tree, for representing the network topology. Also implements semantics and constraints imposed upon the topology.
- **Routing support module**—Implementation of path-creation M-Actions on managed objects in the containment tree. Using topology abstraction provided by Topology module, establishes most appropriate route. This route is then established by communicating with the underlying NEV agents using the NEV abstraction module.
- **NEV abstraction module**—An abstraction for the many NEV agents that are being managed by this PNV agent. Allows efficient management of resources used by the underlying communications infrastructure.

2.4 TESTING INFRASTRUCTURE

An important part of the project was to have an appropriate testing infrastructure to enable thorough testing of the agents. Due to the need for testing the breadth of the function that the software agents support, a flexible testing tool was also required.

CiTR built a test tool based on the Python scripting language. This approach provided the team with a manager application which could communicate with the software agents and had the flexibility of using scripts to perform the specific interactions with the agents. Using metaprogramming, our Python interpreter was constructed so that it can translate any ASN.1 types and GDMO constructs required for testing.

The Python language is a public domain scripting language [3]. It provides facilities which make integration of the scripting language and a set of language constructs for agent testing into a tool a relatively straightforward process.

The following is a short example of the test scripts the project was able to use:

```
agent_title = environ['M4_AGENT_TITLE']
root_moc_id = environ['M4_ROOT_MOC_ID']

# Create CMIP session and connect to agent
s = CMIPSession()
s.StartupWithResponder(agent_title)

sc = Scope()

managedElementAva = ("managedElementId", ("pString", root_moc_id))
switch_dn = [ managedElementAva ]
atmMpFabricAva = ("atmFabricId", ("numericName", 1))
atmMpFabric_dn = [ managedElementAva, atmMpFabricAva ]

sc.Set(2,0)
filter = ("equality", ("objectClass", "tcAdaptorTTPBidirectional"))
lines = getattribute(s, sc, "managedElementR1", switch_dn, filter,
                     ["tcTTPId"] )
line_dns = []
for line in lines:
  (moc, dn, time, attrs) = line
  line_dns.append(dn)

sc = Scope()

#Create a point to point connection
d1 = ("descriptor", [ ("interfaceId", uni1), ("vpi", 1), ("vci", 3)])
d2 = ("descriptor", [ ("interfaceId", intra1), ("vpi", 2), ("vci", 1)])
info = ("connect", [[("fromTermination", d1), ("toTermination", d2)]])
connectpoint = performaction(s, sc, "atmMpFabric", atmMpFabric_dn, None,
                             info)
```

```
try:
  start(connectpoint)
  print
except:
  print "ERROR: CONNECTION FAILED"

sc.Set(2,0)
filter = ("equality", ("objectClass", "atmCrossConnection"))
getatm = getattribute(s, sc, "managedElementR1", switch_dn, filter,
              atmCrossConnection)
try:
  start(getatm)
  print
except:
  print "ERROR: exception GET ATTRIBUTE FAILED - atmCrossConnection"
```

3 WHAT THE MANAGED OBJECT TOOLKIT OFFERS

CiTR used Release 1.0 of the Managed Object Toolkit on this project. As this paper was being written, Release 1.1 of MOT was released. The new release of the software extends the toolkit beyond what is described in this paper.

The following describes the features of MOT from our project team's perspective but does not completely describe the features of MOT. For full details of MOT's features consult Hewlett Packard documentation.

3.1 FEATURES OF MOT

The Managed Object Toolkit basically provides:

- an infrastructure which contains libraries to build agents with
- tools to generate components of the agent

MOT accepts your MIB (as GDMO/ASN.1 documents), and generates the code to build an agent supporting the supplied MIB. It can also build a partial framework for a manager application.

MOT builds a directory structure containing all of its generated files, including makefiles. Once the generation is complete, you can build a default agent without changing anything.

All generated code is C++. Classes are generated for:

- ASN.1 types—makes manipulating and traversing ASN.1 values easy
- GDMO attributes—can be specialised to offer specific functionality on an attribute-by-attribute basis
- GDMO managed object classes—can be specialised to offer functionality for a given managed object, including providing an implementation for M-Actions that the managed object supports

The generated code is linked against the MOT libraries, which provides the basic agent infrastructure, including all DM-level initialisation and complete containment tree management.

MOT also provides facilities for persistent storage of the contents of the containment tree. This enables the loading of containment tree data at agent start-up.

When building manager applications the features of MOT described above that can be made use of are:

- wrapped interfaces for DM-level initialisation
- the ASN.1 and GDMO types and classes

3.2 USING MOT

The steps in using MOT are:

- Prepare MIBs to be supported in the agent for processing.
- Process the MIBs, generating code and source directories.
- Build the basic agent from the generated code.
- Specialize and extend the generated code - making use of the provided libraries and generated code - to produce the final agent.

In the course of building the agent the above steps may be iterated through a number of times due to changes in the MIBs supported. This is shown graphically in Figure 4.

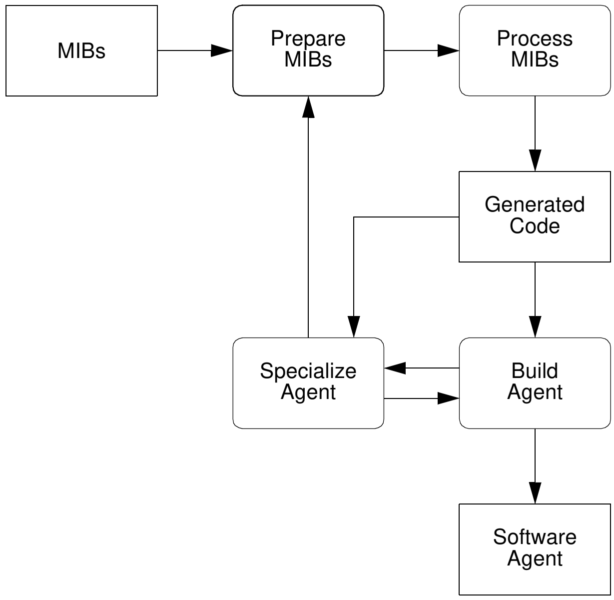

Figure 4: Workflow When Using MOT

4 OUR APPLICATION OF THE TOOLKIT

The process of using MOT on the project was basically as outlined above. Due to changes in the MIBs supported, conditional packages supported and changes to attribute behaviour, the project team executed MOT a number of times to regenerate the code.

Some specific actions the project team had to take were:

- Edit the supported MIBs to resolve cross-referencing errors due to inconsistent reference methods
- Combine all the MIBs being supported by the agent into a single file to prevent exceeding a size limit in the running agent.
- Implement our own persistence solution due to the need to handle complex types (discussed further in Section 6.1.1).

The diagrams Figure 5 and Figure 6 show the impact of MOT on the built agents. The components of the agents are shown in three categories:

- those generated completely by MOT
- those which involved specialization of MOT provided classes
- those which were produced by the development team, some of which made use of libraries provided by MOT.

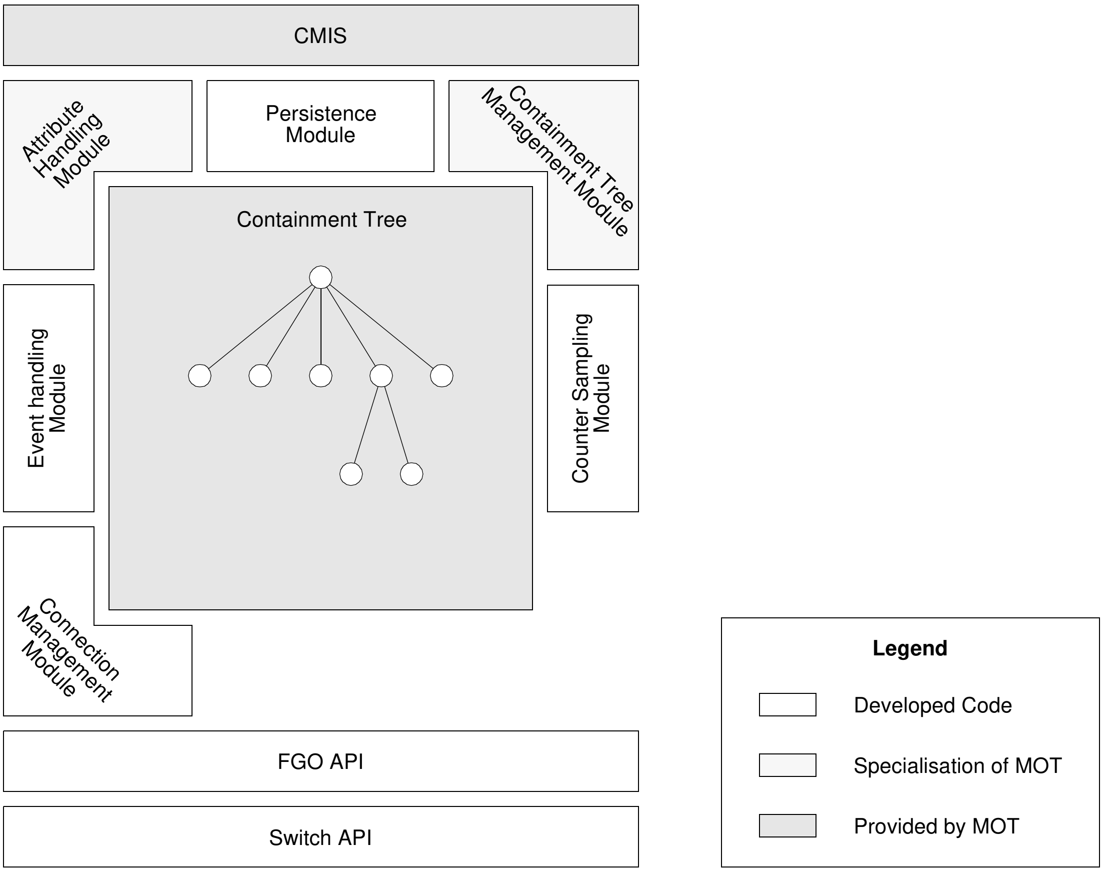

Figure 5: M4 NEV agent architecture—MOT impact

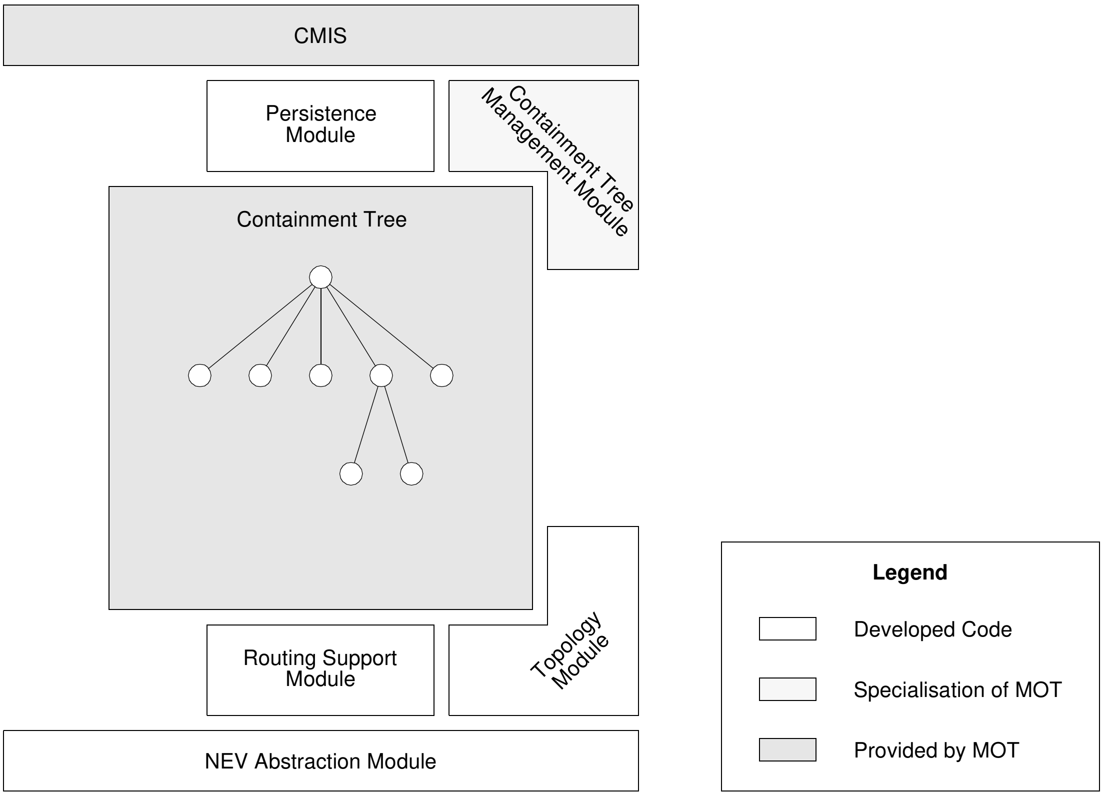

Figure 6: M4 PNV agent architecture—MOT impact

In building the test tool, the team was able to make use of the MOT infrastructure and libraries which simplified the development of the tool.

5 EXPERIENCE GAINED AND LESSONS LEARNT

5.1 PROJECT EXECUTION

Due to the use of MOT the project team was able to concentrate on the core of the project - the specifics of the NEV and PNV agents. There were a number of activities that could be removed from the schedule, reducing the scheduled time to complete and size of the team.

While unable to give an exact measure the team believes that the use of MOT approximately halved the total effort required for the project compared to producing all of the code for the agents without any toolkit or framework. The actual saving may vary depending on the skills of the development team. For example, the skill level of developers in using C++ will have an impact. Our project team had a high level of C++ expertise.

The summary of this is that MOT paid for itself in saved time and effort on a single project. Considering the cost of such a toolkit this is quite an achievement.

5.2 PERFORMANCE

Performance is always an issue when it comes to software such as these agents. The use of a toolkit with its libraries and additional layers of software encapsulating various functions always raised the prospect of performance degradation. Also the use of C++ adds another potential performance load.

Our investigation of the performance of the agents built with MOT were very reassuring. The data was collected using the Quantify performance analysis tool from Pure Atria. The results of the investigation, while not comprehensive or statistically validated, were that the MOT libraries and generated code added very little to the execution time of operations.

5.3 PRODUCING THE CODE

Due to the libraries and generated code that MOT provided the project team, they were able to concentrate on unique aspects of the agents. The developers were concentrating on the interesting and challenging work of developing the agents without having to produce the framework of the agents.

The amount of code generated by MOT is substantial and produces an executable agent very early in the development process. Having this executable framework so early in the project was extremely valuable as it enables testing to start much earlier.

During the development MOT proved to be very robust and of excellent quality. There were only a only a handful of places where the developers had to work around a problem—refer to Section 6.

Hewlett Packard offer the DevAssist service for DM and MOT developers to assist when using the platform. Our project did not have access to this service. In reviewing the project, it is unclear if having the service would have made any impact for this particular project.

6 SOME ISSUES, CONCERNS AND ENHANCEMENTS

6.1 SOME KNOWN PROBLEMS AND WORK-AROUNDS

The following are areas where the toolkit was unable to provide the functionality the project team required in the agent being implemented. The first is a weakness in a subsystem, the remaining two are specific functional problems which have been described in detail.

6.1.1 PERSISTENCE MECHANISM SUPPORT OF COMPLEX DATA TYPES

The MOT 1.0 persistence mechanism does not support all of the complex ASN.1 types that can be used by the attributes of managed objects in the agent. As a result the supplied persistence framework could not be used to generate the data files for the agent's persistent data store. A proprietary solution was implemented to provide the required functionality.

6.1.2 CANNOT GENERATE THE "INVALID ATTRIBUTE VALUE" CMIS SERVICE ERROR

Attempts to directly or indirectly generate an "Invalid attribute value" CMIS service error on a M-Create result in the requester blocking perpetually; it appears that the agent is unable to propagate this CMIS service error back to the responder, though no error message is indicated at the time the error is generated.

The chosen work-around is to generate "Access denied" CMIS service errors instead of "Invalid attribute value" errors.

6.1.3 MOT CANNOT ACCEPT THE ATTRIBUTES PACKAGES, NAMEBINDING OR OBJECTCLASS WHEN PROCESSING M-GET REPLY

MOT implements the X.721 attributes Packages, NameBinding and ObjectClass with specific C++ classes (rather than auto-generating implementations with **`ovmotccgen`**). However, when one of these three attributes is received by a manager from the agent, the resulting OVmotAttC contains a null value.

Note that sending values for these attributes from the manager to the agent does function correctly with the values being updated. Thus only manager-side testing was affected by this limitation.

6.2 FEATURES WE WOULD HAVE LIKED TO HAVE

The following is a list of features that the development team believe would make MOT and its use even more effective:

- A fully functional persistence framework supporting all data types (Release 1.1 of MOT addresses this).
- Provision of an integrated logging framework for messages from within the MOT provided code as well as developer coded messages.
- The containment tree produced by MOT is kept totally in memory when the agent is running. This limits the size of containment tree the agent can support depending on the available system resources of the machine the agent is executing on. Facilities to allow managed object instances to be loaded and unloaded from memory would be useful.

This was not an issue for this project but needs to be considered when developing agents that may have very large containment trees.

- The software developers believe that the agents produced with MOT would benefit from the use of real threads rather than the current use of the USL Task Library. The task library introduces a number of problems, due to the understandable difficulty of producing tasking behaviour in client code without kernel support.
  MOT's framework-based approach makes it easy to specialise its coordinator mechanism to use kernel threads, though for this to work all MOT and DM libraries, as well as the generated code, would have had to have been thread-safe.
- Access to some of the product core which is currently totally wrapped. While this encapsulation made getting up-and-running easier (all the work was done for us), when we actually wanted to access the details of session initialisation at the DM level, we were unable to access it.

7 SUMMARY

Our use of the Managed Object Toolkit in producing software agents for our customer has proved very worthwhile. The toolkit has improved our ability to produce OSI software agents and at the time of writing this paper is starting to be used on other projects within CiTR. This toolkit is a valuable addition to the tools of the trade available within our organisation and could be used on any future OSI software agent development.

ACKNOWLEDGEMENTS

The authors would like to thank the many people who have contributed to this paper both in review and in providing the source material. In particular, the project team who used the tool and showed its value to the organisation.

REFERENCES

[1] af-nm-0027.001, CMIP Specification for the M4 Interface, September 1995

[2] ATM_Forum/96-0973R2, M4 Network View CMIP MIB Specification 1.0, Letter Ballot DRAFT, October 1996

[3] See **`http://www.python.org`** for detailed reference documentation Python and access to the Python language source files