David Narraway

AN EXAMPLE OF A SYNERGISTIC RELATIONSHIP BETWEEN SOFTWARE ARCHITECTURE AND A DOMAIN-SPECIFIC LANGUAGE
AN EXAMPLE OF A SYNERGISTIC RELATIONSHIP BETWEEN SOFTWARE ARCHITECTURE AND A DOMAIN-SPECIFIC LANGUAGE

David Narraway
Master's thesis
Spring 2013
Information Technology
Oulu University of Applied Sciences
PREFACE

Is it the mark of a lazy person to wish to avoid performing a tedious detail-oriented task more than once, or is it a sensible idea when such tasks can be automated and performed by computers? In 1932 Bertrand Russell wrote “In Praise Of Idleness” (Russell, 1932). His article suggests that automating tasks is not a good thing unless workers affected by the change benefit from an increase in leisure. Usually though, as he writes, time saved by improving productivity is used in some other task. That is generally as true today as it was in 1932.

Russell gives an example where an improvement in productivity is accompanied by an increase in unemployment. He describes a productivity improvement that in 4 hours allows workers to achieve that which previously took 8 hours. Rather than everyone benefiting from an increase in their personal leisure time half of the employees work as much as before and the other half become unemployed. His suggestion is that everyone would be happier if they all worked half as much. There is the same amount of new “leisure time”, but rather than everyone benefiting half of the workers are denied any and the other half forced to “enjoy” it all. There is an alternative point of view though. Automating a tedious task may not increase leisure, but it can improve a person’s day. The benefit is not necessarily time spent in leisure, but it is time not spent in drudgery.

Typically we do not ask ourselves if we should automate something. If we can do it then generally we will, but one of the hardest parts of automating is finding the time to do the work that will result in saving more time. Deciding if it is worth doing can also be a challenge. If a task will only need to be done once, then there is no point in investing in the time it will take to automate it. What if it will be needed twice? The three most important numbers in the computer world are 0, 1 and more than 1. Perhaps they are the only important numbers and the implication is that anytime that we do something twice we should expect that it will be necessary to do it a third time and a fourth and so on. So when we do anything with our computers, we should ask if we will need to do this activity again and consider ways to automate the operation.

Of course not all operations can or will be automated. A corollary to the “will” question is the “how” question. If I want to automate an operation, can I describe how it should
be done so that the computer understands my instructions? What programming language will I use?

The central theme of this thesis is a notation that can be readily understood by both Man and Machine. The notation is an example of a Domain Specific Language, a custom programming language created for a specific purpose, in this case to provide a formal description of a messaging protocol that can be used by code generators to automate the coding steps.

When faced with implementing message handlers for the second time without the benefit of specifications, my colleagues and I decided to try and avoid having to do it for a third time, even though we had and still have no expectation of there being a third time. Certain system features had not been explicitly documented the first time around and would need to be reverse engineered for the second version. Having been confronted with the “more than 1” case, we would clearly be remiss in assuming there would be no third or fourth time.

This thesis presents the work done to avoid another reverse engineering round. A specification is involved, but it is the nature of the specification and the various ways it is useful that hopefully make this document of interest.
ACKNOWLEDGEMENTS

The development activities described in this paper were performed during employment with NetHawk Oy (now part of EXFO). Several of my colleagues, Mr. Tapio Taipale, Mr. Ian de Souza and Ms. Dipti Adhikari, were involved in the earliest reverse engineering activities described in Chapter 5.2. Mr. Taipale in particular gave considerable support while I was finding my way around the original M5 implementation. The main body of the work, developing the notation and the code generators, was more of a solo effort on my part, but would not have been possible without the support of NetHawk managers and the patience of the development team while waiting for the code generators.

I would also like to thank Mr. Taipale, Dr. Kari Laitinen and Ms. Kaija Posio for their help in reviewing and improving this document.
ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Mobile Technology

Author: David Narraway
Title of thesis: An Example of a Synergistic Relationship Between Software Architecture and a Domain-Specific Language
Supervisor: Dr. Kari Laitinen
Term and year of completion: Spring 2013   Number of pages: 70 + 33 appendices

This Master's thesis work was carried out during a re-engineering project for NetHawk (now EXFO). The project's main aim was to replace the User Interface of NetHawk's M5 Network Analyzer, a tool used on PCs running Microsoft operating systems. An additional aim was to document the message interface used by cooperating parts of the M5. From the beginning of the project, it was clear that the lack of an explicit specification for the interface would give rise to problems when re-implementing the User Interface and the project team was required to fully document the interface. The specification was seen to have dual purpose. A human readable form of the specification was required to provide developers with a convenient reference source. A machine readable form was at least as important, if not more so, as it was intended that code generators would produce the code used to read and write the message contents.

The existing version of the M5 had been implemented using Delphi for the User Interface and C++ for the application logic. Automated and manual techniques were combined to extract information about the message structure from the codebase, producing human and machine readable documents in the process. The automated techniques included customized parsers operating directly on the codebase and also on information collected from the running M5. Those tools produced initial candidate specifications that could be manually edited into the final form.

The Software Architecture of the User Interface was designed to separate the processing of messages from the presentation of the information in the messages. This separation of concerns was a key factor in the successful outcome of the project. A suitable specification notation was developed as well as compatible code generators. Interface specifications, code generators, and Software Architecture evolved together as interface message requirements became clearer. The current version of the M5 Network Analyzer uses code generated from the specifications. Further development of the system could concentrate on separating the essential message structure from the issues related to the operating system (MS Windows) so that it could be used on other operating systems. The code generators could be developed further to produce code in different programming languages, such as C++.

Keywords: Domain Specific Language, Code generator, Software architecture, Machine readable specifications
# CONTENTS

Preface .............................................................................................................................................. 3  
Acknowledgements ............................................................................................................................ 5  
Abstract............................................................................................................................................... 6  
Contents............................................................................................................................................... 7  
List of Figures and Tables .................................................................................................................. 10  
Terms and Abbreviations................................................................................................................... 11  
1. Introduction ...................................................................................................................................... 12  
   1.1 Overview .................................................................................................................................... 12  
   1.2 Document Structure ................................................................................................................... 13  
   1.3 Result ........................................................................................................................................ 13  
2. The M5 Network Analyzer............................................................................................................... 14  
   2.1 M5 Multi-Analyzer Software Architecture Overview .............................................................. 15  
   2.2 Model-View-Controller Architecture ....................................................................................... 17  
   2.3 MVC and the M5 ....................................................................................................................... 19  
   2.4 The M5 Architecture: Interactions ......................................................................................... 23  
   2.5 Application Model ................................................................................................................... 25  
3. The M5 Message Protocol .............................................................................................................. 27  
   3.1 M5 Service Messages ............................................................................................................... 27  
4. Domain Specific Languages .......................................................................................................... 30  
   4.1 Identified advantages and disadvantages of DSLs .................................................................. 31  
   4.2 Designing a DSL ...................................................................................................................... 32  
   4.3 From DSL to implementation .................................................................................................. 33  
5. The M5 Specification Notation ....................................................................................................... 35  
   5.1 The Re-engineering challenge .................................................................................................. 35  
   5.2 Tools to aid the recovery of message protocols ....................................................................... 36  
   5.3 Why XML? ............................................................................................................................... 40
5.4 Information Item Types ................................................................. 42
5.5 Additional specification features .................................................. 44
5.5.1 Accessing manually written code from generated code .................. 44
5.5.2 Managing multi thread operations .............................................. 44
5.5.3 Property Locking ........................................................................ 50
5.5.4 Property Attributes ................................................................... 51
5.6 Generating code from the Specification .......................................... 52
5.7 Example input and output .............................................................. 52
5.8 Browsing the Specifications ............................................................ 54
5.9 M5 Specification Notation – DSL or not? ........................................ 54
6. Benefits from using Code Generators .............................................. 56
6.1 Message Identifier Collisions .......................................................... 57
6.2 Large array storage ...................................................................... 57
6.3 Application Model Memory Leak .................................................... 58
7. Alternatives to the M5 Message Specification .................................... 60
7.1 JavaScript Object Notation (JSON) ................................................ 60
7.2 Serialization using XML .................................................................. 61
7.3 Google Protocol Buffers ................................................................... 62
7.4 Comparison ..................................................................................... 63
8. Conclusions ..................................................................................... 67
9. References ....................................................................................... 68
Bibliography ........................................................................................ 68
Appendix A Message Operation Search Tool Output .............................. 71
Appendix B Message Log Parser .......................................................... 72
B.1 The Menu Bar ............................................................................... 73
B.1.1 File Menu ................................................................................ 74
B.1.2 Tools Menu ................................................................................ 74
B.1.3 Window Menu .............................................................................................................. 74
B.2 Message Operation View.................................................................................................. 75
B.2.1 Important notes about the XML description of the Application Event content...... 78
B.3 Using Notes and Bookmarks ......................................................................................... 79
B.4 Comparing Logs ............................................................................................................. 80
B.5 Log File Formats ........................................................................................................... 80
B.5.1 Message log file ......................................................................................................... 81
B.5.2 Message Operation log file ....................................................................................... 82
Appendix C An Event based Message Protocol................................................................. 83
Appendix D Message Specification XML-Schema .............................................................. 87
Appendix E Message Notation Style Sheet .......................................................................... 99
Appendix F A Browser View of a Message Specification.................................................... 103
LIST OF FIGURES AND TABLES

FIGURE 1. M5 Architecture Overview ................................................................. 17
FIGURE 2. Model-View-Controller paradigm (Madeyski & Stochmiałek) .......... 18
FIGURE 3. M5 Connection Configuration .............................................................. 21
FIGURE 4. Applying Deacon’s modified M2M,VC paradigm to the M5 Architecture 22
FIGURE 5. Domain Model updates Application Model .......................................... 24
FIGURE 6. View updates Model ........................................................................... 24
FIGURE 7. Partitioning of the Application Model .................................................. 26
FIGURE 8. Service Message and Application Event Structure .............................. 28
FIGURE 9. Example Application Event ................................................................ 29
FIGURE 10. Metacase illustration of productivity gains (Metacase, 2012) .......... 30
TABLE 1. Comparison of operation layout ......................................................... 35
FIGURE 11. UI and Delivery Thread interactions ................................................... 47
FIGURE 12. Deadlock condition .......................................................................... 49
FIGURE 13. Multi-threaded Domain Model updates Application Model ............. 50
TABLE 2. Breakdown of Message Operations ....................................................... 56
TABLE 3. Comparison of notation features ........................................................ 63
FIGURE 14. Screenshot of message parser ............................................................ 72
FIGURE 15 Message Operation View .................................................................. 76
FIGURE 16. Asynchronous Action - Response Sequence Diagram ..................... 85
FIGURE 17. Synchronous Action -Response Sequence Diagram ........................ 86
FIGURE 18: Specification presentation in browser .............................................. 103
TERMS AND ABBREVIATIONS

.NET  Microsoft Software Framework
ASN.1  Abstract Syntax Notation One
C#  Programming Language (Ecma, 2006)
C++  Programming Language (JTC1/SC22/WG21, 2003)
Delphi  Programming Language (Embarcadero, 2009)
DSL  Domain Specific Language
DTD  Document Type Definition
ECMA  European Computer Manufacturers Association
IDE  Integrated Development Environment
IETF  Internet Engineering Task Force
LOH  Large Object Heap
LOTOS  Language Of Temporal Ordering Specification
M5  EXFO (NetHawk) Network Analyzer
MVC  Model-View-Controller
OS  Operating System
UI  User Interface
UML  Unified Modelling Language
VS2005  Visual Studio 2005 (an IDE)
WPF  Windows Presentation Foundation
XML  eXtensible Markup Language
XSL  eXtensible Stylesheet Language
1. INTRODUCTION

1.1 Overview

The current User Interface (UI) of the M5 Network Analyzer (EXFO M5 Analyzer, 2012) from EXFO (EXFO Inc.) is the second to have been developed for it. The first was built using the Delphi programming language (Embarcadero, 2009), but uncertainty over the language’s future combined with the development of new analysis techniques implemented using the C# programming language (Ecma, 2006) gave rise to a project to re-engineer the User Interface completely using C# and Microsoft’s .NET framework (Microsoft Corporation).

The original M5 implementation had partitioned the system architecture into two main components. The User Interface was the smaller of these two components and was implemented using Delphi. The larger component, referred to as the Application Engines, was implemented using the C++ programming language (JTC1/SC22/WG21, 2003). The re-engineering effort replaced the User Interface component, but reused the C++ part without change.

Application Engines in the C++ code-space communicate with their User Interface peers by serializing information into messages to be sent to the User Interface and deserializing messages received from the User Interface. As the new User Interface was expected to work with the old Application Engines, then obviously existing message formats would be used, but unfortunately no explicit message specifications existed at the start of the re-engineering project.

Rather than describe a research activity this document is more in the nature of a review of a successful approach. After giving some background context, it will describe the approach taken to develop the specifications and will present the specification notation. It will provide descriptions of code generators used to transform the specifications into code and will draw attention to other relevant points. The notation and its supporting tools are in active use to this date.

The basic form of the specification notation was determined very quickly without much reference to existing protocol specification languages or other alternatives. To some

---

1 As of the date of publication
extent that was because of the approach already taken by the M5 development team. In
the later chapters of this thesis the notation is compared to some alternative approaches.
Perhaps unfairly, one of the alternatives considered did not exist when the project
started, but it is included because it addresses some of the same issues as the M5
internal communication protocol does and has since been used by other project teams
within EXFO.

1.2 Document Structure

Chapter 2 will introduce the M5 Network Analyzer in sufficient detail to provide
context for the rest of the document and will include a description of the macro
architecture of the product. We consider the goals of the project and how they influence
the architecture of the new User Interface.

Chapter 3 will provide an overview of the protocol used by the M5 to share information
internally.

Chapter 4 takes some time to introduce the topic of Domain Specific Languages and
later in the document we will consider whether the M5 specification notation is an
example of a Domain Specific Language.

Chapter 5 presents the essential features of the specification notation and the various
approaches used to extract message structure from the codebase and Chapter 6 considers
the benefits of using code generators with the message specifications.

We then introduce the second theme, asking the question “What should or could have
been done differently”. Chapter 7 looks at a limited set of alternative message protocol
specification notations and compares the notation features.

The closing chapters will summarize and suggest opportunities for further development
of the notation and tools that benefit from the notation.

1.3 Result

An important result of the re-engineering project is a set of explicit and correct
specifications used to generate production code and to provide reliable reference
material for developers.
2. THE M5 NETWORK ANALYZER

The EXFO M5 Analyzer is a wireless network analyser that can be used to monitor 2G, 3G and 4G wireless networks. The main software partitioning paradigm splits the functionality into “Applications” and their User Interfaces. An example of an Application is the component used to analyse and display protocol interactions as message sequence charts or tables of decoded messages. Another example is a component used to analyse and correlate different phases of calls.

The M5 Application and UI parts of the system serialize information into messages for sending and de-serialize messages when receiving information. The system implements message classes with Add and Get operations for various information types.

The Delphi programming language was used to implement the User Interface in the first two major versions (1.x and 2.x). For a new version NetHawk (now EXFO), in cooperation with (VTT Technical Research Centre of Finland), had been planning to introduce new analysis techniques. Tools based on those techniques were implemented using C#. At the same time the future of the Delphi programming language was uncertain as Borland had announced it was looking for a buyer for some products, including Delphi. NetHawk (now EXFO) had already started a project to replace the Delphi UI with a Java UI, but after a review of the results of the VTT project NetHawk’s technical management team decided that the new User Interface would be developed using C# to more effectively incorporate those new analysis tools. Fortunately, a relatively small amount of effort had been put into the Java User Interface, but beneficially that had shown that one of the difficulties for any re-engineering effort would be that no explicit specifications for the messages exchanged by the User Interface and Application parts existed.

Discussions involving the staff from the original M5 project, the Java User Interface project and the new C# User Interface project resulted in an agreement that an essential output from the new project would be explicit specifications for the M5’s internal protocol. The following chapters describe the resulting human and machine readable specifications, the use of code generators to produce the serialization / de-serialization code and assorted additional benefits.

This document should make clear that the use of machine generated code influenced the architecture of the User Interface considerably, but whether or not a machine writes our
code, separating the handling of the internal messages from the use we make of the information in them ought to be an obvious tactic. If we use a machine to write code for us, we would be optimistic to expect that the machine can understand and implement our intention, i.e. the semantics of our messages, but it is entirely reasonable to expect that it can cope with a consistent syntax. It becomes unreasonable to expect a machine to achieve that in a chaotic or unstructured environment, so we can deduce that we need a properly defined structure that accommodates the generated code in order to gain any benefit.

A suitable structure, or software architecture, should isolate the exchange of information between the User Interface and the Application engine from the presentation of that information to the User and should provide a way of coordinating access to the necessary information stores. A well-known architectural paradigm that aims to provide the required separation is the Model-View-Controller (MVC) concept (Reenskaug T., Trygve/MVC, 1979).

As well as being well-known, the MVC paradigm is also widely misused and somewhat misunderstood so Chapter 2.2 will review the paradigm and consider how well it fits the M5 requirements while Chapter 2.3 will present the use of MVC in the M5 User Interface. Before that, in the next Chapter, 2.1, we will briefly review the M5 overall architecture before looking at the User Interface in more detail in the subsequent chapters.

2.1 M5 Multi-Analyzer Software Architecture Overview

The Architecture of a software system can be considered to be the result of applying one or more partitioning paradigms to the system components. It is neither necessary nor desirable to repeatedly apply a single paradigm as the roles of components differ and a paradigm suitable for one part is not certain to be suitable for another. For example, some systems work well using a Server-Client approach whereas others are better constructed using a layered approach. The M5 is no different from any other system in that respect and several different paradigms are used to partition the system.

The first, rather crude paradigm separates the M5 into two separate code-spaces. The primary reason for this is to maintain a strong separation between the User Interface and the rest of the system. A secondary reason was to simplify reuse of legacy analysis and decoding code, which was written in C++, whereas the User Interface in the original M5
was written in Delphi. A system design goal was to be able to replace the User Interface easily, run the User Interface on a remote system or even to run the analyser with no User Interface at all.

The second partitioning paradigm divides system functionality into sets of Applications and Tools. A running system may have several Applications open, but only one Tool at any time and if a Tool is open, it prevents use of the open Applications (i.e. is modal). Examples of these are a Protocol Monitor Application, a Call and Session Analysis Application and various Configuration Tools for data sources, filters, databases etc.

Each Application / Tool has a User Interface and an Engine. Not surprisingly the User Interface is implemented in the User Interface code-space and the Engine is implemented in the second code-space. Each code-space contains framework code to support the Application and Tool implementations and for the sake of simplicity we shall henceforth refer to the code-spaces as the UI and the Engine.

A “gateway” component is used to connect the UI and Engine code-spaces. FIGURE 1 illustrates the result of applying these partitioning paradigms.

The UI and Application Engine are further partitioned by applying additional paradigms. The internal structure of the Application Engines is of no particular interest in the context of this document and will not be reviewed further. The UI Architecture is profoundly relevant though and will be discussed in more detail in forthcoming sections.

Information is passed between the two code-spaces using messages stored in the shared system memory. References to the messages are exchanged via the gateway. The structure of the exchanged information follows defined protocols and the memory contents are written and read according to those protocols.

Information is sent from one code-space to the other as the result of a user action in the UI or a state change in the Engine. An example is when the User starts an application to analyse network traffic. The UI will send appropriate messages to the Engine to instruct it to start monitoring the network and soon after the Engine then begins to send analysis results to the UI.
In the context of this re-engineering project the gateway component provided an important advantage. Since all messages in both directions pass through the gateway, it is an obvious point to add monitoring to, making it simple to collect data to use while reverse engineering the message structures. This will be discussed in Chapter 5.

2.2 Model-View-Controller Architecture

MVC is an elegant example of the separation of concerns concept, where application or business logic is separated from the presentation of information. The concept was introduced by Trygve Reenskaug (Reenskaug T., Trygve/MVC, 1979). The problem that MVC was introduced to solve was bridging the gap between a user’s mental model of the system in use and its underlying computer model. The intention was to support the idea of the user interacting with the domain information directly while hiding system implementation details and to be able to provide different perspectives of the same system data set without modifying the whole system.


**Model:** A Model is an active representation of an abstraction in the form of data in a computing system.

**View:** To any given Model there is attached one or more Views, each View being capable of showing one or more pictorial representations of the Model on the screen and on hardcopy. A View is also able to perform such operations upon the Model that is reasonably associated with that View.
**Controller:** A Controller is the link between a user and the system. It provides the user with input by arranging for relevant Views to present themselves in appropriate places on the screen. It provides means for user output by presenting the user with menus or other means of giving commands and data. The Controller receives such user output, translates it into the appropriate messages and passes these messages on to one or more of the Views.

(Reenskaug T., 1979-05-MVC.pdf, 1979)

According to (Madeyski & Stochmialek, 2005) the MVC concept is frequently applied or used inappropriately by confusing the semantics of the different parts, resulting in the Model being considered to be a simple repository of data and the controller having responsibility for the application behaviour whereas the MVC concept demands that the Controller maps user actions to Model operations, changing system state or to View changes. FIGURE 2 shows the relationships between the system parts.

Madeyski and Stochmialek write that the value of MVC is based on two essential rules:

1. Separation of Model and Presentation
2. Separation of View and Controller

In (Madeyski & Stochmialek, 2005), the authors write that the most common degeneration of the classic form breaks the 2nd rule by tightly coupling the View and Controller. In some systems this is a deliberate choice. For example, the architecture of Sun’s Java Swing platform has its roots in MVC, but the classic approach did not work well because of difficulties in maintaining rule 2 (Fowler, Amy, 1999).

![FIGURE 2. Model-View-Controller paradigm (Madeyski & Stochmialek)](Image)
Similarly the controller component is not very visible in Microsoft’s WPF platform (Kratochvil, 2011, pp. 28-38). WPF controls have absorbed the controller and handle mouse and keyboard events themselves and deliver events to the application via the Microsoft Event Pattern.

2.3 MVC and the M5

As a partitioning paradigm, MVC can be applied at different system abstraction levels. It is possible to partition the entire system into parts or to partition subsystems into the MVC parts. Chapter 2.1 introduced the first two partitioning paradigms used by the M5 architects. The 2nd paradigm splits functionality into Applications or Tools (essentially the same in the context of this document). In this section we present MVC as a 3rd paradigm used to partition those applications or tools, but the discussion that follows could be applied to the entire UI of a system.

Following the established tradition of not using the concept as originally conceived, the M5 implementation does not follow the intended pattern. However, rather than blur the boundary between the View(s) and the Controller(s) the M5 architecture modifies the role of the Controller.

Before looking at this in more detail, it is useful to review (Deacon, 2009). In section 5, Deacon introduces an interesting and relevant concept. He suggests that the Model introduced in the original paradigm is itself confusing and blurs important distinctions. He suggests that the Model is actually composed of two parts, the Domain Model and the Application Model, giving rise to a modified acronym MDMVC.

The Domain Model is the part that system analysts and designers would think of as the system model. It should be the collection of classes that capture and represent the essence of the problem for which the application is a solution. The Application Model is an object that “knows” views exist and that those views need some way of obtaining information and notification”. Deacon suggests that the M in MVC really refers to the Application Model.

2 “View” of an individual application or tool; ”Views” of all applications or tools, i.e. ”Views” of the system

3 ”Controller” of an individual application or tool; ”Controllers” of all applications or tools, i.e. ”Controllers of the system.
Deacon’s insight is interesting in the M5 context because of the division of the M5 into two distinct parts (Chapter 2.1). The larger part performs all the hard work of providing the “solution” and the smaller part (the User Interface) handles the interaction with the user. This smaller part must obviously use information maintained by the larger part though the larger part also maintains detailed information that is neither needed by, nor is useful to, the User. We can identify this larger part as Deacon’s Domain Model and it therefore follows that his Application Model is part of the M5 User Interface.

Some, but not all information in the Domain Model is relevant to the User. The UI uses some information not essential to modelling the problem domain, such as visual themes, window positioning and so on. We can therefore consider the model to be composed of two overlapping information sets. The intersection of the two sets is the information that must be exchanged between the M5 UI and Application Engine components.

A design goal of the original system was to be able to easily change the UI or remove it completely. This is most easily achieved by isolating the functionality related to interacting with the User in distinct components with simple, limited interfaces and the View part of the MVC paradigm fits that requirement.

What then of the Controller? As noted in Chapter 2.2 when using the Microsoft WPF platform the View and Controller are somehow blended into WPF controls, but that does not mean that our system can or should no longer have a Controller. In the early MVC concept the Controller translates keystrokes, mouse actions etc. into more abstract user actions affecting the system state. Since the M5 UI is implemented using WPF components, we are not especially interested in the low level keyboard/mouse interactions, but we are interested in more abstract actions and can identify a Controller role for managing them.

To illustrate, consider an editor used to edit connection configurations. FIGURE 3 shows a screenshot for such an editor in the M5. Connections are presented in a table with individual connections in rows and configurable values in columns.
There is no particular value in updating the Domain Model while the User is modifying values in the various fields so Controllers in the M5 applications or tools have no interest in the details of those interactions. When the User has completed the configuration, the system state must be modified. In this example, the User will indicate completion by clicking on the “Set Connections” or “Apply” buttons shown in FIGURE 3. Now we have a User Action for a Controller to care about. The editor View notifies the editor Controller that a User Action has occurred and the Controller causes the Domain Model to be updated and that in turn may give rise to further changes in the View.

We have noted that the UI contains the Application Model and the Engine the Domain Model. In the given editing scenario modifications made by the User must be affecting some part of the UI. Rather than apply those changes immediately to the Domain Model, the View maintains the modifications in the Application Model. When the
Controller is informed that the User Action has been performed (is complete), it will initiate the Domain Model update using the modified Application Model information.

We choose to give the M5 Controllers the additional role of managing the lifecycle of the Application Models and the Views in the User Interface (each Application / Tool has one Controller-Application Model-View triad). This is consistent with Reenskaug’s early definition of the Controller – “… by arranging for relevant views to present themselves …”. In general the Application Model is only needed by an active View and the View is only needed when the User wants to perform a relevant task. At other times there is no need for either component to exist. Application Models are created by their Controllers before the corresponding Views and it is possible, though perhaps not useful, for there to be no View at all. This additional lifecycle management role modifies the relationships between the parts so that the Controller notifies the View when state changes occur instead of the Model providing the notification as shown in FIGURE 2.

We can conclude this section by confirming that the MVC paradigm, in the form suggested by Deacon (MaMaVC) is a valid tool to partition the M5 User Interface into cooperating parts. FIGURE 4 shows the resulting M5 system architecture, noting that there is no attempt to refine the Engine beyond identifying the Domain Model.
2.4 The M5 Architecture: Interactions

In earlier chapters we have determined that the User Interface of an M5 Application or Tool can usefully be partitioned into three parts, an Application Model, a Controller and a View. A 4th part, the Domain Model is implemented in the Application or Tool Engine. The set of information items in the Application Model partially overlaps the set of information items in the Domain Model. The UI Controller facilitates the operations that synchronise the Application and Domain Models. In this section we will briefly consider these synchronisation operations.

In the traditional MVC approach the Controller abstracts low level actions into more abstract operations on the Model, potentially making changes to it. The View is notified when Model modifications occur and the View can query Model state at any time.

In the M5 this interaction approach has been somewhat modified. The notifications of changes to the Domain Model are delivered via the Controller to the Application Model. The change notification message is processed by the Application Model and the message contents are used to update the Model state. The Controller will then notify the View that the event associated with the received message has occurred. The View can access the related information in the Application Model so that it can update its presentation components appropriately. This sequence of actions is shown in FIGURE 5.

FIGURE 6 shows the alternative scenario where the View makes changes to the Application Model when the User interacts with the View. Aggregated changes to the Application Model are sent to the Domain Model as the response to certain abstracted User Actions. In these cases the View modifies the Application Model and then notifies the Controller that a particular User Action has occurred. The Controller asks the Application Model to build a change notification message that it can send to the Domain Model.

The Controller is involved in delivering Domain Model changes to the Application Model so that it can manage the Application Model lifecycle appropriately by creating Application Models as they are needed and destroying them when they are no longer needed. Views are managed in a similar way, so rather than have the Domain or Application Model directly notify the View of state changes, the M5 Controller delivers the notifications to the View so that it can manage the View lifecycle effectively.
FIGURE 5. Domain Model updates Application Model

FIGURE 6. View updates Model
2.5 Application Model

Before leaving the topic of the M5 Architecture, it is useful to look more closely at the Application Model. The previous chapter describes how information in the Application Model is included in change notification messages that are sent to the Domain Model and how similarly, information in the Domain Model is included in the change notification messages that are sent to the Application Model. The View is able to access information in the Application Model so that it can modify it or present state changes to the User. The terms serialization and de-serialization are generally used to refer to the action of building and processing change notification messages. It is useful to draw attention to two key characteristics of the Application Model. One is its ability to serialize / de-serialize information into and from change notification messages and the other is that information stored in the Application Model is accessible to the View.

To provide these two characteristics the Application Model is partitioned into two parts. One part handles the serialization / de-serialization activity and the other part provides the information stores that are accessed by the View. Those same information stores are used by the serialization / de-serialization component. We note that the View is in no way affected by how information in the Application Model is updated. The View’s only interest is that the information is available. The message handler part of the Application Model is indifferent to the purpose of the information stores. Its only interest is the message structure. With these distinctions it can be stated that the message handler is only interested in the syntax of the notification messages and the View is only interested in the semantics of those messages. This seems an admirable characteristic and in fact it is an important factor in the success of the Message Specification notation described in the forthcoming chapters. We call the collection of information stores the “Property Set” and we call the message handler the Model\(^4\). The separation of the Application Model into the two parts is shown in FIGURE 7.

\(^4\) This seems a poor choice of name in the context of this discussion, but hopefully won’t cause too much confusion.
The document Overview in Chapter 1.1 refers to code generators. Chapter 5 will go into more detail, but briefly we can note here that the Property Sets and the Models of each M5 Application or Tool are produced by code generators, using the M5 Message Specifications as their input.
3. THE M5 MESSAGE PROTOCOL

Chapter 2 looked at the Software Architecture of the M5 Analyzer and showed how the Model-View-Controller paradigm is appropriate for the M5 when the System Model is itself separated into an Application Model and a Domain Model. To briefly recap, the separate parts of the Model exist in the UI and Application Engine code-spaces, with the Application Model in the UI code-space and the Domain Model in the Application Engine code-space (FIGURE 4). Some information is duplicated in the two Model parts, but each part also has information that is not relevant to the other part. For example, the Domain Model will contain low-level system state information that is of no interest to the user, so it is not needed in the Application Model and the Application Model may contain information that helps managing the presentation of system state to the User, but is of no particular use in representing the underlying system. An example of this is the selection of colours for the views in the UI.

Chapter 2.5 showed how information is sent from one part to the other in response to system state changes or as the result of User Actions. Information is serialized into change notification messages in the sender and de-serialized into information stores in the receiver. This Chapter will look at the basic structure of the notification messages that synchronise information in the two model parts.

3.1 M5 Service Messages

With a few exceptions, all interactions between components in the User Interface and the Application Engine parts use M5 Service Messages. There are several service types, but only\(^5\) the “Application Event” type is relevant to this thesis.

An M5 Service Message contains 5 mandatory fields:

- Service Id
- Method Id
- Local Reference
- Remote reference
- Message Body

---

\(^5\) One of the service types is used to manage resource lifecycles and as mentioned in Chapter 2.3 the Controller performs that task in the UI, but that will not be covered in any detail in this document.
FIGURE 8. Service Message and Application Event Structure

The Service and Method Ids are used to indicate the type of operation that the sender requires the receiver to perform. The Local and Remote References identify the sender and receiver and the Message Body depends on the Service and Method ids. The top part of FIGURE 8 shows the general Service Message structure.

Application Events are identified by a particular Service id / Method id pair. The Application Event Message Body contains an Application Event header and an Application Event payload. The lower part of FIGURE 8 shows the structure of the Message Body for an Application Event Service Message.

As shown in FIGURE 8 the Message Body of an Application Event Service Message contains an Application Event header and the Application Event Content. The header contains four mandatory fields:

- Event Id
- Event Length
- Target Id
- Sequence Number.

The Content part of the Application Event is where the system serializes the information to be sent to the peer. A system implementation library provides a Service message class with primitive operations to add and read fields of different types. The primitive types are:

- Boolean
- Integer (8, 16, 32 and 64 bits)
- Double
- String (a length followed by a zero-terminated data block)
- Data (a length followed by the specified number of octets)
- Local Pointers
- Remote Pointers
FIGURE 9 shows an example Application Event containing four information fields: a 32-bit integer followed by two strings and a Boolean.

The M5 system uses Application Events to share information between the cooperating UI and Application Engine peers. Each Application Event has its own unique structure. As mentioned in the Introduction, no explicit specifications of these events were produced for the earlier versions of the M5 and one of the goals of the re-engineering project was to redress that omission. An informal approach to specify the messages was rejected in favour of a machine-readable format that could be processed by code-generators to produce the Property Sets and Models referred to in Chapter 2.5.

As well as describing the approach taken, this thesis sets out to ask if the specification notation can be considered to be an example of a Domain Specific Language. Chapter 4 will review that concept and then Chapter 5 will look at the notation in more detail.
4. DOMAIN SPECIFIC LANGUAGES

Domain specific languages are not an especially new concept. Backus-Naur Form was described in 1959, though perhaps it is a fairly recent trend to use the term Domain-Specific Language (DSL). Two of the many definitions of the term are:

A domain-specific language (DSL) is a language designed to provide a notation tailored toward an application domain, and is based only on the relevant concepts and features of that domain.
(Kosar, Martinez Lopez, Barrientos, & Mernik, 2007, p. 1).

[A] domain specific language (DSL) is a computer language that’s targeted to a particular kind of problem, rather than a general purpose language that’s aimed at any kind of software problem.
(Fowler, Domain Specific Languages, 2010).

(Metacase), a company based in Jyväskylä, Finland offers a Domain Specific Modelling environment. This product corresponds to what Fowler termed a “Language Workbench” (Fowler, Language Workbenches: The Killer-App for Domain, 2005). FIGURE 10 is an illustration of productivity gains that Metacase has derived from customer cases.

**FIGURE 10.** Metacase illustration of productivity gains (Metacase, 2012)
Resources (whitepapers etc.) provided on the Metacase website give examples of the productivity gains that can come from the use of domain specific tools. Metacase ascribes the potential gains to the fact that DSLs “fundamentally raise the level of abstraction, while at the same time narrowing down the design space” (Metacase, 2012). This narrowing down of the design space is also likely a key factor in the success of the so-called “mini or little languages” of UNIX. Eric S. Raymond writes about these in (Raymond, 2003, p. 183). Raymond references studies showing that error rates in code are largely independent of the language used, so the higher level the language, the fewer lines are needed and so the fewer defects there are. By focussing on domain concepts and introducing appropriate abstractions, DSLs can therefore boost productivity significantly, not least by cutting down on mistakes.

In his articles Fowler tends to focus on “language” whereas Metacase focuses on “model”. The common ground is that they write about “domain specific” tools. By that they mean that instead of using general purpose programming languages or general purpose modelling languages (e.g. UML), they recommend capturing essential domain concepts into the notation, i.e. a notation using concepts taken from a business’s products, focussing on the problems that the products solve.

The potential productivity gains are of course not free. Capturing the necessary domain knowledge and expressing it as a DSL requires effort and commitment. The definition of the language is best done by a domain expert, someone that already has the necessary knowledge and who also has the skills to express that knowledge appropriately. That can mean taking a person away from their normal duties and allowing them to concentrate on the DSL, which in turn implies a loss of productivity while the DSL is being created. This will only make sense when subsequent productivity gains make it worthwhile, but there can be no guarantees of gains. Nevertheless Metacase provides customer testimonials reporting productivity increasing by factors of 5 to 10.

### 4.1 Identified advantages and disadvantages of DSLs

Van Deursen (van Deursen;Klint;& Visser, 2010) surveys the literature and discusses Risks and Opportunities, DSL Design Methodology and DSL Implementation and also summarizes some 75 other papers. The benefits identified from the survey are:

1. DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain.
2. DSL programs are concise, self-documenting to a large extent and can be reused for different purposes.

3. DSLs enhance productivity, reliability, maintainability and portability.

4. DSLs embody domain knowledge, and thus enable the conservation and reuse of this knowledge.

5. DSLs allow validation and optimization at the domain level.

6. DSLs improve testability

Disadvantages (from the same paper) are:

1. The costs of designing, implementing and maintaining a DSL.

2. The costs of education for DSL users.

3. The limited availability of DSLs.

4. The difficulty of finding the proper scope for a DSL.

5. The difficulty of balancing between domain-specificity and general-purpose programming language constructs.

6. The potential loss of efficiency when compared with hand-coded software.

Additionally it is sometimes suggested that once the DSL is available, anyone familiar with the domain can use it without needing to be expert in programming. Fowler is sceptical of this notion as he writes “The lay programmer argument is a high stakes bet. If someone justifies some technology based primarily on enabling large scale user programming I overflow with skepticism. Yet if such an approach could succeed it would provide an enormous benefit. This wouldn't come from eliminating professional programmers, but on improving the often dire state of communication between domain experts and programmers. This lack of communication is often the biggest roadblock in software development projects.” (Fowler, Language Workbenches: The Killer-App for Domain, 2005, p. 15)

4.2 Designing a DSL

Raymond suggests that DSLs (minilanguages) can be designed in “at least three ways, two of them good and one of them dangerous”. The two good techniques are essentially about taking an existing format and formalising it, whereas the dangerous technique is
to “extend your way to it, one patch and crufty added feature at a time” (Raymond, 2003, p. 183).

In (van Deursen; Klint; & Visser, 2010, s. 3) the authors identify a series of steps, grouped into Analysis, Implementation and Use.

Analysis steps:

1. Identify the problem domain.
2. Gather all relevant knowledge in this domain.
3. Cluster this knowledge in a handful of semantic notions and operations on them.
4. Design a DSL that concisely describes applications in the domain.

Implementation steps:

1. Construct a library that implements the semantic notions.
2. Design and implement a compiler that translates DSL programs to a sequence of library calls.

Use steps:

1. Write DSL programs for all desired applications and compile them.

Fowler suggests there are three main steps to define a new DSL:

- Define the abstract syntax, i.e. the schema of the abstract representation.
- Define an editor to let people manipulate the abstract representation through a projection.
- Define a generator. This describes how to translate the abstract representation into an executable representation. In practice the generator defines the semantics of the DSL.

(Fowler, Language Workbenches: The Killer-App for Domain, 2005, pp. 21-22)

4.3 From DSL to implementation

The domain specific notation must be converted into an implementation. Fowler identifies two techniques: interpretation at runtime and by using code generators to convert the notation into another form (Fowler, Domain Specific Languages, 2010, p. 19). Mernik (Mernik; Heering; & Sloane, 2005, ss. 13-15) identifies more implementation patterns:
• Interpreter: DSL constructs are recognized and interpreted using a standard fetch-decode-execute cycle.

• Compiler / Application generator: DSL constructs are translated into base language constructs and library calls.

• Pre-processor: DSL constructs are translated into constructs in an existing language. Various sub-patterns exist:
  o Macro processing: Expansion of macro definitions.
  o Source to source transformation: DSL source code is transformed to base language source code.
  o Pipeline: Processors successively handling sublanguages of a DSL and translating them into the input language of the next stage.
  o Lexical Processing: Only simple lexical scanning is required, without complicated tree-based syntactic analysis.

• Embedding: DSL constructs are embedded into an existing GPL by defining new abstract data types and operators.

• Extensible compiler / interpreter: A GPL compiler or interpreter is extended with domain-specific optimization rules and/or domain-specific code generation.

• Commercial Off-the-Shelf (COTS): Existing tools and / or notations are applied to a specific domain.

• Hybrid: A combination of the above approaches.
5. THE M5 SPECIFICATION NOTATION

In this chapter we will look at the development of the specification notations, starting with some of the challenges of recovering information from the original system (5.1) followed by a discussion of the tools used to help with re-engineering (5.2).

Chapters 5.3 through to 5.5 cover the choice of specification notation, data types and other specification features. Chapters 5.6 and 5.7 describe the code generators and give examples of their output. Finally, Chapter 5.8 describes an additional benefit of using XML as the specification notation.

5.1 The Re-engineering challenge

The first version of the M5 was implemented without formally documenting the protocols used by Application Engine and User Interface components. This lack of specification meant that the team developing the second version of the User Interface components had no reference sources for the protocols other than the code implementing them. Reverse engineering the protocols from the code is not an especially difficult task, but mistakes can easily be made, particularly when implementations of core system behaviour are interleaved with message handling.

As a quick example, consider the two columns of TABLE 1. The columns show message operations in bold-italic and other operations in blue-italic with operations interleaved in the left column and grouped in the right column.

**TABLE 1. Comparison of operation layout**

<table>
<thead>
<tr>
<th>Determine how many contacts exist</th>
<th>Determine how many contacts exist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write number of contacts into message</td>
<td>Build contact list: For each contact</td>
</tr>
<tr>
<td>Get names</td>
<td>Get name</td>
</tr>
<tr>
<td>For each name</td>
<td>Get Addresses</td>
</tr>
<tr>
<td>Write each name into message</td>
<td>Write number of contacts into message</td>
</tr>
<tr>
<td>Get Address</td>
<td>For each contact</td>
</tr>
<tr>
<td>Write address into message</td>
<td>Write name into message</td>
</tr>
<tr>
<td></td>
<td>Write address into message</td>
</tr>
</tbody>
</table>
The column on the right appears to do more. Certainly it contains two loops instead of one as in the column on the left. For this reason, if no other, it might seem quite natural to use the approach on the left and in fact that is the approach commonly found in the original M5 implementation. Although it may seem less efficient, the approach on the right has important advantages, one of which is that the two groups of operations can easily be factored into separate methods that can be tested in isolation from each other. That is clearly not as simple with the approach on the left. Regardless of the relative merits of the two approaches, the fact that the function and message handling was interleaved in the reference implementation meant that recovering the protocols from the code would be a painstaking task.

5.2 Tools to aid the recovery of message protocols

During the earliest days of the re-engineering project some simple logging code was implemented into the gateway component. Its purpose was to help to understand the way the message exchange system worked by illustrating some of the protocols used to connect Application Engines to Application User Interfaces. It would have been marvellous to be able to show message fields as well, but that would have needed a specification documenting the fields and we did not have one.

There is a hint of the “Hole in My Bucket” (Various, accessed 19th April, 2013) situation here:

1) We need a message specification
2) We decide to capture messages to examine their fields
3) We therefore need to implement a message parser
4) We therefore need a specification for the messages

One of the team members explored another possibility – to parse the implementation files, looking for message operations. Each operation would be of the form Addxxx or Getxxx where xxx would be replaced by the field type, so in principle these operations could be found from the code. Appendix A contains an example of the search tool output.

The approach was only partially successful, because of another kind of code layout issue; sometimes implementations would operate on different messages in the same method.
The following code fragment provides an example:

```java
Message someMethod(Message m1) {
    Message m2 = new Message();
    string name = m1.GetString();
    m2.AddString(name);
    bool state = m1.GetBoolean();
    m2.AddBool(state);
    return m2;
}
```

It would of course be possible for the parser to distinguish the operations on m1 from those on m2, but that would have required more effort and would not guarantee that no other problems existed. At this stage it seemed that there was little alternative to extracting the structure of the messages from the code one by one.

During this time however, the early ideas about logging messages in the Gateway component had evolved. Rather than show the results as lines of text, a form based tool had been developed. Initially this log parser was only able to show the system management messages used to establish functional bindings between components and those messages are outside the scope of this document, but as the Application Events were specified the parser became more useful. Also, by this point XML had been chosen as the specification notation. Chapter 5.3 contains a discussion on that choice.

Appendix B contains some screenshots and descriptions of the Parser. The Parser incorporated a useful analytical feature. By using an instrumented version of the service message class, it was possible to correlate operations with particular message instances and avoid the problem mentioned above where operations on different messages were interleaved. The reader is reminded that a functional version of the M5 Analyser already existed, using a User Interface implemented in Delphi, and message logs could be captured while performing tasks with the original M5 Analyser and then analysed with the Message Log Parser.

The operation analysis could be taken a step further to write specification fragments describing the Application Event of interest and those fragments could be copied to the specifications. There was a limit to the sophistication of the synthesis step though. Application Events can contain repeated groups of fields. The operation analysis isolated operations on a particular message instance, but did not identify repeated
sequences of operations. In theory that could have been done, but in practice there were too many possible problems for it to be considered practical. As an example, consider a message that will carry some number of information items where each item comprises a name, a personal identifier and a room number. The name is represented using a string, the personal identifier as a 32-bit number and the room number as a 16-bit number. Each item would then be added to the message using the following sequence of add operations:

```csharp
message.AddString(name);
message.AddInt32(PersonalIdentifier);
message.AddInt16(RoomNumber);
```

Usually the number of items in the group would immediately precede the group itself, but that is not necessary and is not always the case. The only requirement is that the number of items is known before the group is parsed. As long it precedes the group it can be anywhere in the message or it can be delivered in another message, as long as that message is delivered before the one containing the group. It could in fact simply “be known”, maybe because that number of items has been requested. Assuming the most likely case, the operations on a message containing three items would be:

```csharp
message.AddInt32(3);
message.AddString(name1);
message.AddInt32(PersonalIdentifier1);
message.AddInt16(RoomNumber1);
message.AddString(name2);
message.AddInt32(PersonalIdentifier2);
message.AddInt16(RoomNumber2);
message.AddString(name3);
message.AddInt32(PersonalIdentifier3);
message.AddInt16(RoomNumber3);
```

From these operations the tool would synthesise the following XML:

```xml
<Event>
  <PrimitiveType type="Int32"/>
  <PrimitiveType type="String"/>
  <PrimitiveType type="Int32"/>
  <PrimitiveType type="Int16"/>
  <PrimitiveType type="String"/>
  <PrimitiveType type="Int32"/>
</Event>
```

6 This is simplified for presentation. The PrimitiveType element contains several child elements that are not shown here.
The correct XML, given the (textual) description above is:

```xml
<Event>
  <PrimitiveType type="Int32">
  </PrimitiveType>
  <CountedGroup ...>
    <CountedGroup>
    </CountedGroup>
  </CountedGroup>
</Event>
```

Most occurrences of CountedGroup would be in the expected form so the synthesis step would have produced the correct XML quite often. Including the data values in the analysis step would have increased the number of times that the correct form of XML would have been produced, but as usual there is a joker in the pack to spoil the game. Messages can contain optional sequences of fields as well as repeated sequences. An optional sequence appears once or not at all, though it can itself include a repeated sequence. A valid construct would specify a sequence to be included if a given field has the value 1. This could give rise to operations:

```csharp
message.AddInt32(1);
message.AddString(name);
message.AddInt32(PersonalIdentifier);
message.AddInt16(RoomNumber);
```

This matches the previous specification syntax, but in this case the XML should be\(^7\):

```xml
<Event>
  <PrimitiveType type="Int32">
  </PrimitiveType>
  <OptionGroup ...>
    <OptionGroup>
    </OptionGroup>
  </OptionGroup>
</Event>
```

\(^7\) Simplified for presentation again
Luckily the synthesis output proved to be quite usable without that extra sophistication. In practice the specification writer would refer to the code to find field descriptions and comments to add to the specification document fields anyway, so a few additional editing operations to correctly format the XML was no great burden when the synthesis step had produced a closely matching template.

5.3 Why XML?

There are a number of notations that can be used to specify interfaces between components. In the telecommunications world, Abstract Syntax Notation One (ASN.1) (International Telecommunications Union, 2012) is a well-known approach. Another older protocol specification language is LOTOS (ISO, 1989).

With hindsight, we at NetHawk should probably have looked at available languages before starting our work. We were of course aware of ASN.1, and in fact part of the M5 code is produced by a commercial code generator using ASN.1 specifications. Nevertheless, at the time we thought that an XML notation would be more lightweight and more flexible while developing our own code generators for it and it would also be simpler (less challenging, shorter, shallower learning curve) for developers to use. An XML editor was included in the IDE that the team was using (Visual Studio 2005) and several other free editors were available. The editor in VS2005 could use a schema file to guide the developer by suggesting valid elements and attributes as the editing operation progressed. The commercial code generator that worked with ASN.1 produced C++ and we needed C#, which was not available, so we chose XML as our notation syntax.

XML is an interesting notation in that it is entirely semantic free, yet the freedom to define element and attribute names makes it easy to imply notation semantics. For example using element names such as Book and Author imply meaning that element names such as B and A cannot. The referencing of DTDs or XML-Schema within the documents also implies meaning, but they still only provide rules for the notation grammar, not the semantics of any particular “phrase”. This characteristic is useful to us in the M5 context, since it leaves us free to take information from the document and
use the code generators to express the semantics appropriate to each of the relevant code components.

A contrary opinion is expressed by (Markopoulos, 1997, p. 14) who wrote about formal specifications of user interfaces:

“A formal method is associated with a formal specification language, which has a formal syntax and a formal semantics, …” and “A formal specification is unambiguous because it has only one meaning; it should always be clear whether a particular object of the semantic domain belongs to the class of objects described by the specification. … A specification may address various levels of abstraction and a high-level prototyping language may look no less abstract. What distinguishes the formal specification is the mathematical definition of its semantics and the underlying inference system associated with them.”

Fowler is also of the opinion that formalised semantics form a vital part of a DSL, suggesting that his Semantic Model (Fowler, Domain Specific Languages, 2010, p. 21) maintains a clear separation of concerns between parsing a language and the resulting semantics.

In this document however, we have suggested that the lack of formal semantics is a reason to adopt XML as our specification notation. That seems to conflict with both Markopoulos’s and Fowler’s position, but for the M5 we use the same input document with different code generators to express different semantics for each output document, where an output document is the source code for a system component. Embedding semantics in the notation would prevent that flexibility.

Perhaps the most obvious component to produce with the code generator is the one that writes information into and reads information from a message, because those operations are quite straightforward to visualize. But such a component needs access to the information that should be written and also needs access to the data stores that will receive information read from the messages. Accordingly, one output from the code generation step is a component containing methods for reading and writing messages and another output is a component containing the information items. These parts were identified in Chapter 2.5 and both are considered to be part of the Application Model as shown in FIGURE 7.

The focus of this document is the creation of the two Application Model parts by providing code generators with the Message Specifications and the following sections describe the Notation features that support that activity, but for clarity it is useful to
remember that the User Interface implementation for each M5 Application / Tool contains all four of the components shown in FIGURE 7. Two of them are produced by code generators (the two parts of the Application Model), while one is a generic class, easily produced from a template when initially creating the UI (the Controller) and the last component, the View, is produced by the development teams.

Appendix D contains the full XML Schema describing the Message Specification Notation, but parts of it are reproduced where relevant to clarify the notation description.

5.4 Information Item Types

Chapter 3.1 listed the primitive operations provided by the Service Message class. The Specification Notation obviously has to support those primitive types and they are defined using an element named “PrimitiveType” that has a “type” attribute specifying the required type. The XML schema describing the XML type for the PrimitiveType element follows:

```xml
<xs:complexType name="PrimitiveDataType">
  <xs:sequence>
    <xs:group ref="CommonPropertiesGroup"/>
  </xs:sequence>
  <xs:attribute name="type">
    <xs:annotation>
      <xs:documentation>The primitive data type for this message</xs:documentation>
    </xs:annotation>
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="Boolean"/>
        <xs:enumeration value="Int8"/>
        <xs:enumeration value="Int16"/>
        <xs:enumeration value="Int32"/>
        <xs:enumeration value="Int64"/>
        <xs:enumeration value="Double"/>
        <xs:enumeration value="Data"/>
        <xs:enumeration value="String"/>
        <xs:enumeration value="LocalPointer"/>
        <xs:enumeration value="RemotePointer"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
```

Information messages are usually more than just a list of primitive types. A common requirement is for primitive types to be repeated in groups and another common requirement is for items to be optionally present. Accordingly, the specification allows
for structure to be specified using elements named “SimpleGroup”, “OptionGroup” and “CountedGroup”.

A “SimpleGroup” is interpreted as meaning that the remainder of the message has the specified structure, present at least once (i.e. an unspecified number, \( N > 1 \), of the groups are present in the message remainder).

```xml
<SimpleGroup>
  <Description>Adapters Name</Description>
  <PropertyName>GetAdaptersName</PropertyName>
  <PrimitiveType type ="String">
    <Description>All adapters Name</Description>
    <PropertyName>adaptersName</PropertyName>
  </PrimitiveType>
</SimpleGroup>
```

An “OptionGroup” specifies that the defined part may or may not be present and specifies the property used to decide that.

```xml
<OptionGroup>
  <testValue>true</testValue>
  <testResult>true</testResult>
  <Description>if file name exist</Description>
  <PropertyName>FileNameExist</PropertyName>
  <PrimitiveType type="String">
    <Description>If the trace path is empty use M5 directory, otherwise use as the path</Description>
    <PropertyName>FilePathAndName</PropertyName>
  </PrimitiveType>
</OptionGroup>
```

The “CountedGroup” references a property to be used as the limit count in a loop. This property is usually given earlier in the same message, but can be delivered in an earlier message.

```xml
<CountedGroup indexName="NumberOfConditions">
  <Description>Condition item</Description>
  <PropertyName>ConditionItem</PropertyName>
  <PrimitiveType type="String">
    <Description>Condition name</Description>
    <PropertyName>ConditionName</PropertyName>
  </PrimitiveType>
  <PrimitiveType type="String">
    <Description>Condition Visible name</Description>
    <PropertyName>ConditionVisibleName</PropertyName>
  </PrimitiveType>
</CountedGroup>
```

The message specification references an XML Schema that defines the allowed structure. This has the advantage that tools such as Visual Studio, used by EXFO to develop the M5 code, can guide and validate the specification, but validation in this way is optional and an invalid specification (according to the XML Schema) will still be
used in the build step. Syntactical errors in the specification will cause the generation step to fail.

5.5 Additional specification features

The original intention was that the specification would describe the structure of the messages, but the usefulness of additional features was obvious from the beginning. The following sections describe these additional features. Hopefully these are not the “crufty features” disparaged by (Raymond, 2003, p. 183)!

5.5.1 Accessing manually written code from generated code

It would have been foolish to expect that the code generator implementation would produce sufficiently functional code in every case in the early days of implementation. To handle anticipated exceptional cases or to work around errors the “useHelper” attribute was introduced to the MessageToUI and MessageToEngine elements. When this attribute is encountered the code generator writes code to invoke an Application Model operation in a helper class. Developers would manually write code in a known method in the helper class to handle the exceptional case. For example:

```xml
<Event>
  <Name>ET_DRAG_DROP_DATA</Name>
  <Id>0x2f</Id>
  <VersionNumber>1</VersionNumber>
  <TaskDescription>Activating connections.</TaskDescription>
  <Description>Accept and execute data drop from PM Engine</Description>
  <MessageToUI useHelper="true">
```

The fragment of specification above produces code:

```java
    case (int)PropertySet.Events.ET_DRAG_DROP_DATA:
        // Use the Model Helper to decode the message
        m_modelHelper.UpdateModel(eventId, message, PropertySet);
```

5.5.2 Managing multi thread operations

Each message can be part of a sequence of messages exchanged between the Application and Domain Models so, for example, a message from the Domain Model might trigger a number of responses from the Application Model. Each Event definition allows a list of responses to be defined using the <Response> and <ResponseList> elements. We might expect that a given sequence of events should always happen in which case this feature could be used to present an appropriate message sequence chart for the activity. It is, however, quite common to need to modify the sequence in order to
react to the information in one or more of the events. This feature might then be used to describe the normal flow of events. There must then of course be a way to modify that flow by removing events and/or adding new ones to the sequence. The application event handling framework provides a means for the developer to modify the response list without exposing detail that might compromise the separation of the MVC triad. As well as adding convenience, this feature affects the threading model used to handle application events.

```
<MessageToUI>
  <ResponseList>
    <Response>ET_CREATEUI</Response>
  </ResponseList>
</MessageToUI>
```

This specification fragment gives:

```c
    case (int)COCAnalyserPropertySet.Events.ET_CREATEUI:
       // register the required response codes - we don't prepare the result message yet
        responseInformation.AddAppEvent((int)COCAnalyserPropertySet.Events.ET_CREATEUI,
                                      AnalyserPropertySet.TargetId,
                                      false);
```

The Response list can be used to solve a thread synchronisation problem that can lead to deadlock conditions. Part of the problem is caused by a Windows platform feature that requires applications to use a single specific thread to update visual controls. Visual control update operations using information delivered by other “worker” threads require that the operation and information be “marshalled” onto the UI thread. The .NET framework (Microsoft Corporation) provides two marshalling operations for this purpose. One will block the calling thread and the other will not. The early versions of the M5 framework used the blocking version, but it quickly became apparent that more flexibility would be needed.

In order that the UI remain responsive to user activity, a set of “delivery threads” are used to deliver event from the UI to the Application Engines. Each Application UI is given a dedicated delivery thread to use. The Controller queues the associated Application Event in the input queue of an “Event Pump” when the View component informs the Controller that a User Action has occurred. The UI thread is normally used for the queuing operation. The delivery thread for that resource then picks up the Application Event from the queue and delivers it to the Application Engine. The event
elements in the Specification Notation include a Boolean attribute named “synchronousSend” to specify if the User Interface thread should be blocked for the duration of the delivery operation or not. When this is specified, the program control does not return to the caller until the Application Event has been delivered, though the UI remains responsive. Additionally, the Controller provides two types of User Actions; one that seems to block and one that does not, to provide developers with a manual means of controlling behaviour.

FIGURE 11 illustrates the interactions between the UI thread and other system threads including Application Engine “delivery threads”. The different threads are shown using different colours. Black represents the UI thread, green represents a delivery thread and blue represents an unspecified thread – it could be a thread belonging to one of the Application Engines or it could be a delivery thread. The four titles in the diagram can be considered to be identifying “swim lanes”, which also allow us to see the activities the threads are responsible for. The left-most lane is there to represent the User interacting with the View. There is a box in the centre labelled “Response List”. The UI thread writes into that list and the “unspecified thread” reads the list.

An example interaction scenario starts with the user performing some action. After starting the UI thread enters the “Handle Events” activity. One of the guarded exit paths is [User Action] and in this scenario we traverse that path to send the “AppEvent to Engine” signal. We then make a decision: Is this send operation synchronous or not? If it is, we enable waiting for the “Handled” signal and return to the Handle Events activity. If the send operation is asynchronous, we send the UI done signal and the return to the Handle Events activity. The “AppEvent to Engine” signal is accepted by the Delivery thread on the right. The thread performs the “Handle App Event” activity and then sends the “Handled” signal. If the UI thread is waiting for the “Handled” signal, which it does for synchronous send operations, it traverses the [Handled] path, sends the “UI done” signal and returns to the Handle Events activity. If the UI Thread is not waiting for the “Handled” signal, it is ignored.

The Application Engine may have sent a response to the UI from the “Handle App Event” activity processed on the App Engine delivery thread. We can follow that by referring to the “Unspecified thread” swimlane. The thread sends the “AppEvent to UI” signal. The thread can wait for the UI to handle the signal (blocking / synchronous operation, waiting for the “UI done” signal) or it can immediately continue.
FIGURE 11. UI and Delivery Thread interactions
If any responses have been defined in the Specification, they are processed by sending the “AppEvent to Engine” signal for each response.

The initial part of the scenario has the UI thread sending “AppEvent to Engine” signals but later we have the “Unspecified thread” sending responses specified in the “ResponseList”.

This implementation allows the UI thread to be used for synchronous and asynchronous operations by View components and also allows other threads to perform synchronous and asynchronous View updates.

The UI thread must always be able to process the Windows Message Queue because otherwise the User cannot interact with the User Interface at that time, but equally importantly notifications about Application Events from the Domain Model must be delivered to the View on that thread, even while a method in the View is blocked. This means that the View must be re-entrant and consequently care is needed in the View implementation and/or the message protocol.

It is possible to produce a nested sequence of events using a delivery thread and unfortunately it is possible to create lock conditions where the UI thread is waiting for the delivery thread, which is waiting for the UI thread. Such a sequence would start with a User Action. The UI thread sends the “AppEvent to Engine” signal as a synchronous operation, so starts to wait for the [Handled] signal. If the delivery thread now delivers an application event to the UI as a synchronous operation, it is blocked until the UI update is complete. If the UI should now send another application event using a synchronous operation, the system will enter a deadlocked state. Although the UI thread is still responsive to system events, including application events delivered on other threads, the original delivery thread is waiting for the UI update to complete and that cannot complete until the delivery thread handles the new UI request, which it cannot because it is waiting until its own update is handled.

FIGURE 12 shows the sequence chart for the deadlock scenario, with the deadlock condition shown in red and labelled in the centre of the sequence. The diagram lifelines are associated with actual system components. FIGURE 11 does not show the assignment of activities to any system components.
Deadlock conditions are obviously undesirable so some tuning is sometimes necessary. One technique is to use the ResponseList rather than to have the View explicitly sending response events. This is because the ResponseList is processed by the delivery thread and the Event Pump implementation allows the delivery thread to “pass through” and deliver the response directly.

FIGURE 12. Deadlock condition
5.5.3 Property Locking

The Windows requirement of marshalling operations onto the UI thread can introduce a performance problem. If more than one Application Engine thread delivers a given Application Event type, then it is possible that information in the Application Model is overwritten before the View can process it.

We see this situation in FIGURE 13.

![Diagram showing multi-threaded Domain Model updates Application Model](image-url)

FIGURE 13. Multi-threaded Domain Model updates Application Model
Once again we use different colours to identify different threads and introduce the Windows Queue to illustrate the marshalling operation.

The two Domain Model threads put AppEventX items into the Windows Queue (using the marshalling API). After some time the View is notified and processes the first event. Unfortunately, the 2\textsuperscript{nd} event has already overwritten the information delivered in the 1\textsuperscript{st} event, so we have a situation where information is lost.

A similar situation can occur if the Domain Model is using a single thread, but is delivering information at a high rate. This 2\textsuperscript{nd} situation can be handled by making the View updates synchronous, but that usually has a significant impact on performance.

The problem can be solved by defining a suitable message group, which causes received items to be stored in a list, and using an attribute called “lockProperty” in the message specification. The value of the attribute should be the name of a property to lock while handling the item list.

```
<MessageToUI lockProperty="IPNodeList">
```

By defining the message structure using one of the Group types, the code generator will emit code defining and using a list of items and will add each item to the list. The View component can then lock the same property and process the list of items when it executes.

An alternative, though not recommended, approach is to use the “useHelper” attribute and write code manually.

### 5.5.4 Property Attributes

The code generators produce C# (Ecma, 2006), a language used with Microsoft’s .NET platform (Microsoft Corporation). A useful feature of the language and platform is the ability to decorate classes, methods and properties with “attributes” that can be inspected at runtime. The M5 message specification provides information used to add documentation attributes to the generated code. Additional attributes document the messages that a given property is used in allowing another development tool to derive a filtered list of properties used in selected messages. For example, the fragment

```
<Event>
  <Name>ET_ADD_GRAPH_ITEM</Name>
  <Id>4009</Id>
  <VersionNumber>1</VersionNumber>
```
<TaskDescription>Update Network Graph</TaskDescription>
<Description>
Delivers information to update the network graph; requests the engine to send update information
</Description>

produces the following entry in the PropertySet’s Event class:

```csharp
/// <summary>
/// Delivers information to update the network graph; requests the engine to send update information
/// </summary>

public const int ET_ADD_GRAPH_ITEM = 4009;
```

The TaskDescription attribute uses the value of the TaskDescription attribute. The EventDirection attribute value indicates if the event is defined for messages to and from the UI or just in one direction.

### 5.6 Generating code from the Specification

The M5 build system has 3 code generators. One produces components that are incorporated into the Message Log Parser (Appendix B) and the other two produce code for the Application Model and the Property Set introduced in Chapter 2.5 and shown in FIGURE 7.

### 5.7 Example input and output

The following is an example specification of a single information message:

```xml
<Event>
  <Name>ET_PROTOCOL_INFO</Name>
  <Id>0x3ec</Id>
  <VersionNumber>1</VersionNumber>
  <TaskDescription>
    Delivering protocol information.
  </TaskDescription>
  <Description>
    Delivers protocol information to UI
  </Description>
  <MessageToUI>
    <PrimitiveType type="Int32">
      <Description>
        Number of protocol items listed
      </Description>
      <PropertyName>NumberOfProtocols</PropertyName>
    </PrimitiveType>
    <CountedGroup indexName="NumberOfProtocols">
      <Description>List of protocols</Description>
      <PropertyName>ProtocolList</PropertyName>
    </CountedGroup>
  </MessageToUI>
</Event>
```
This fragment describes an event named ET_PROTOCOL_INFO. The specification comprises some descriptive fields followed by the specification of the message sent to the UI. This contains a 32-bit integer primitive type, holding the number of items to follow and a counted group of string primitive types. The supplied loop count is stored in the property named “NumberOfProtocols” and each protocol name will be one item in a list of strings, the list being named “ProtocolList”.

One of the code generators produces serialization / deserialization code. As this fragment defines the message to the UI we only show the deserialization code:

```csharp
// Number of protocol items listed
AnalyserPropertySet.NumberOfProtocols = message.GetInt32();

// List of protocols
for ( int tNumberOfProtocols = 0; tNumberOfProtocols < AnalyserPropertySet.NumberOfProtocols; tNumberOfProtocols++ )
{
    COCAnalyserPropertySet.structProtocolList
    groupProtocolList = new COCAnalyserPropertySet.structProtocolList();

    // Protocol Name
    groupProtocolList.ProtocolName = message.GetString();

    // add to the list in the property set
    AnalyserPropertySet.ProtocolList.Add( groupProtocolList );
}

Another code generator produces the information stores:

```summary```
/// Number of protocol items listed
/// </summary>
private System.Int32 m_NumberOfProtocols;

```summary```
/// Number of protocol items listed
/// </summary>
[DescriptionAttribute("Number of protocol items listed")]
[COCEventAttribute("ET_PROTOCOL_INFO")]
public System.Int32 NumberOfProtocols
{
    get { return m_NumberOfProtocols; }
    set { m_NumberOfProtocols = value; }
}
5.8 Browsing the Specifications

The Message specification references an XSL style sheet (Appendix E), allowing the specifications to be conveniently browsed from internal network locations. Style sheet output for the example specified in Chapter 5.7 is shown below.

ET_PROTOCOL_INFO: 0x3ec

Delivers protocol information to UI

Message To UI

<table>
<thead>
<tr>
<th>Note</th>
<th>Field Type</th>
<th>Description</th>
<th>Associated Property</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Int32</td>
<td>Number of protocol items listed</td>
<td>NumberOfProtocols</td>
</tr>
</tbody>
</table>

Start of Counted Group using index NumberOfProtocols ProtocolList

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Description</th>
<th>Associated Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Protocol Name</td>
<td>ProtocolName</td>
</tr>
</tbody>
</table>

End of Counted Group using index NumberOfProtocols

Response Target Id

Message To Engine

Note Field Type Description Associated Property

Response Target Id

Appendix F shows an example captured as a screenshot from a browser.

5.9 M5 Specification Notation – DSL or not?

The intent of the M5 Specification Notation is to describe the messages exchanged by cooperating parts of the M5. It is:

- Focussed
- Solutions (messages) expressed at the level of abstraction of the problem domain (Chapter 4.1, point 1)
• Solutions are concise (allowing for the use of XML, which is considered by some to be too “noisy”) and are usable for different purposes (e.g. implementation, reference documentation, testing and analysis). (Chapter 4.1, point 2)

• Reuse, portability and maintainability are definitely enhanced compared to earlier implementations (Chapter 4.1, point 3).

• Some domain knowledge has been captured in the notation (Chapter 4.1, point 4)

• Not a general-purpose language. It is declarative rather than imperative, which is a common feature of DSLs.

• The semantic notions of the notation correspond to the problem domain. There are a small number of semantic notions.

If we consider Fowler’s “three parts” in 4.2 we find:

• There is an abstract syntax in the form of the notation’s XML-schema (Appendix D)

• We do not define an editor so much as notice that there is a wide choice of editors capable of checking and guiding the construction of the specification using the XML-schema. The development team will typically use Microsoft’s Visual Studio.

• There are three generators in use, each transforming the source with different semantics.

It seems quite reasonable to classify the M5 Specification Notation as a Domain-specific language. If the list above is not sufficient perhaps an appeal to authority is – Google refer to their protocol buffer notation as a DSL and the M5 Notation is even more domain specific than that. Chapter 7.3 gives a brief introduction to Google’s Protocol Buffers and Chapter 7.4 compares the concept to the M5 Message Specification Notation.

Chapter 4.3 lists a number of implementation patterns that can be used to generate code from a DSL. The approach in the M5 is to emit code lines directly based on the information type being processed. This implementation pattern is the “Pre-processor”
pattern; “Source to source” sub-pattern. This applies to the transformation of the notation by the M5 code generators and to the production of documentation by using XSLT. The selected approach fits well with an evolutionary approach to building the code generator. As the specification features evolved, so the code generators also changed.

6. BENEFITS FROM USING CODE GENERATORS

At the moment we have over 730 messages specified for 33 components defining 2800 properties with nearly 1300 message read operations and nearly 1100 message write operations, as detailed in TABLE 2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of Add operations</th>
<th>Number of Get operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8</td>
<td>14</td>
<td>38</td>
</tr>
<tr>
<td>int16</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>int32</td>
<td>638</td>
<td>732</td>
</tr>
<tr>
<td>int64</td>
<td>17</td>
<td>61</td>
</tr>
<tr>
<td>bool</td>
<td>147</td>
<td>287</td>
</tr>
<tr>
<td>double</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>string</td>
<td>446</td>
<td>698</td>
</tr>
<tr>
<td>data</td>
<td>37</td>
<td>2</td>
</tr>
</tbody>
</table>

Although there have been remarkably few problems with the generated implementations, some corrections were necessary. If the code had been manually written, corrections affecting many messages would have been quite time consuming, but with code generators all affected code is corrected in a single build operation.

Chapters 6.1 to 6.3 describe some problems that were encountered during the project and how the problems were dealt with.
6.1 Message Identifier Collisions

Originally, the PropertySet code generator used the C# “enum” construct as the type of the Message (Application Event) identifier. The given name was associated with the given event id using an expression of the form:

```csharp
public enum Events
{
    /// <summary>Delivers protocol information to UI</summary>
    ET_PROTOCOL_INFO = 0x3ec,
    ...
}
```

Unfortunately, one of the components had been implemented in such a way as to require different names for the same event id. That is not possible using the enum construct, so the generator was modified to produce expressions of the form:

```csharp
    /// <summary>Delivers protocol information to UI</summary>
    [COCTaskDescription("Delivering protocol information.")]
    [COCEventDirection(enumEventDirection.eToUI)]
    public const int ET_PROTOCOL_INFO = 0x3ec;
```

This change affected all message definitions, which typically have two parts:

- Message to User Interface
- Message from User Interface

With 730 message definitions and two parts to each definition this change would have required well over 1,000 manual edits if it could not be handled in the code generators.

6.2 Large array storage

A more interesting case arose due to the original implementation using the “struct” type to define the items in the Specification’s composite types (SimpleGroups, OptionsGroups and CountedGroups). The “struct” keyword defines a “value type” and a list of that type is stored in contiguous memory locations. The .NET runtime uses a multi-generation garbage collector to manage memory, but also uses a “Large Object Heap”. Generations 0 to 2 are compacted, meaning that when items are removed from the heaps the remaining items are moved so that there are no gaps between items. Additionally, the heaps can shrink as well as grow. The Large Object Heap (LOH) is different in two important regards. The LOH is not compacted, so when items are removed, gaps will be left and it also cannot shrink.

The M5 problem was caused by a large number (~1,000,000) of grouped items being sent in an Application Event message. The items were stored in a list, which would
grow as needed when more items were added. Typically, such a list doubles in capacity each time it needs to grow. In the problem cases the list would quickly grow to such a size that it would be moved to the large object heap where it would continue to grow, demanding twice as much memory as before each time it grew. As it grew, it would cause the large object heap to expand and would also cause an increasingly large gap in memory. After the message had been handled the list would be emptied and the memory would be available for other use, but as the heap would not shrink that memory would not be released back to the system. The effect of this was to sharply reduce the amount of memory available to the M5.

The solution was to redefine the list items as classes. A .NET class is a reference type, so entries in the list would be references to instances of the list item class. As the reference used less memory, than the struct item the new list was considerably smaller and was not always moved to the large object heap. A tactic used to improve performance was to specify the required list capacity when the list was created. For CountedGroups that is simple enough – the number of items is known at runtime, but for SimpleGroups it was necessary to estimate the number of items that would go into the list. The estimation algorithm was quite crude, simply summing the sizes of the primitive types in the message. The crudeness of the technique came from assuming that a string would have the length of a 32-bit integer. A string is serialized as a length followed by the string characters. Assuming the string was empty would over-estimate the required capacity of the list but nevertheless the technique was good enough to provide significant performance and memory management improvements.

The capacity estimation tactic and the redefinition of list items as classes rather than structs were both deployed through changes to the code generators. A manual edit approach would have required changes to over 1,000 group definitions in the Property Sets.

### 6.3 Application Model Memory Leak

In another case a memory leak present in all Application Model components was fixed by a modification to the generated code. A temporary message object created while synchronizing the Application and Domain Models was not released correctly, causing memory to be exhausted in a few extreme cases. The change was needed to each
Application Model component so that it would not have resulted in as many manual edit operations as the cases described in the previous 2 chapters, but nevertheless would have taken some hours to complete, whereas it was quickly fixed in the code generator.
7. ALTERNATIVES TO THE M5 MESSAGE SPECIFICATION

The M5 Message Specification is a notation used to describe the structure of messages exchanged between two cooperating peers. Web servers and clients behave in a similar fashion to manage context in a user session. Objects representing the system state are exchanged over the network connection and various techniques are used to serialize the object. This Chapter will briefly introduce three of those techniques. Two of them, in Chapters 7.1 and 7.2, are text based and the third, in Chapter 7.3, uses a binary format equivalent to the M5 messages.

The text based notations, JSON and XML, are presented here as alternatives to the binary formats, Google Protocol Buffers and the M5 Notation, because they are in common use and also because one of the comparisons for the efficiency of the Protocol Buffer concept is made against XML. Since the M5 Message Specification uses XML it is useful to include at least XML in this Chapter.

Chapter 7.4 will compare the M5 and the Protocol Buffer notations, but not JSON or XML since their messages are text, whereas M5 messages are not.

7.1 JavaScript Object Notation (JSON)

JSON is a lightweight, text-based language-independent data interchange format (Crockford, 2006). It is used to serialize structured data into a format suitable for sending over a network connection, typically between a web server and client application. Modern browsers are able to parse JSON, but it is also parsed using JavaScript. The notation supports basic types:

- Number, as a double precision floating point number
- String
- Boolean
- Array
- Object, as an unordered collection of key-value pairs

The Object type is used as the outer container of a JSON message. It uses curly braces as delimiters and contains a comma separated list of key-value pairs, with the key and value separated by the ‘:’ character. Arrays (or lists) are delimited using square brackets.
For example:

```json
{  "Author": {  
    "Name": "Martin Fowler",
    "Books": [  
      {  
        "Name": "Domain Specific Languages",
        "Year": 2010
      },  
      {  
        "Name": "Analysis Patterns: Reusable Object Models",
        "Year": 1996
      }
    ]
  }
}
```

As the information is text based, it is human-readable, but is also quite inefficient. It can also be prone to problems with numbers because strings or numeric literals can be used to represent the same number. For example, leading zeroes can be lost from phone numbers.

A web site dedicated to JSON (JSON.org) was launched in 2002 and the notation was in use by Yahoo! and Google by 2006. The first version of the M5 was launched before then. JSON would not have been considered a serious alternative to the approach taken for the M5 as it is not efficient enough, i.e. messages would have been considered to be too big.

### 7.2 Serialization using XML

Another commonly used technique for exchanging information between computer systems is to serialize the information as XML. The Author example given in the previous section might be represented as:

```xml
<Author>
  <Name>Martin Fowler</Name>
  <Books>
    <Book name="Domain Specific Language" year="2010" />
    <Book name="Analysis Patterns: Reusable Object Models" year="1996" />
  </Books>
</Author>
```

In general JSON is typically more efficient than XML. For the given examples as formatted, XML is slightly more efficient, but if the formatting is removed JSON is (166 characters vs. 179).
The same information can be represented in different ways using XML. Rather than be given as attributes to a Book element, as in this example, the name and year of the book could be contained in child elements. That would make the resulting XML less efficient because opening and closing tags for the name and year elements would also be needed. Like JSON, XML was not efficient enough to be used for the M5 messages in the original implementation.

It may be helpful to point out that the current use of XML is in the specification of the messages, not in the messages themselves. The serialization code produced by the code generators serializes the message contents into a binary format more similar to the Google Protocol Buffer approach outlined in the next Chapter.

7.3 Google Protocol Buffers

The M5 re-engineering project started in 2007 and the first versions of the code generators were already in use that same year. Google made the initial release of their protocol buffers available in 2008, well after the M5 Specification Notation was in use. Since the protocol buffers format was not around when the M5 decisions were being made, it could not have been a candidate. It is included for comparison here because it claims to address the same issues as the M5 notation, and is also in use at EXFO NetHawk.

Google states that “Protocol buffers are a flexible, efficient, automated mechanism, for serializing structured data.” (Google, Inc). The overview mentions XML, claiming that protocol buffers are “smaller, faster and simpler”. Since the M5 notation uses XML it does seem like an interesting comparison to make.

A protocol buffer message description specified a set of uniquely numbered fields, each of which has a name and a type. Allowed field types include various primitive types and nested message types. Fields can be optional, required or repeated. The Author example in the previous two Chapters could be represented with the following proto definition:

```proto
define Author {
  required string name = 1;
}
define Book {
  required string name = 1;
  optional year = 2;
}
repeated Book books = 2;
```
Code generators produce serialization and deserialization code in various programming languages, including C# and C++.

Protocol Buffers are more efficient than XML or JSON, primarily because they only contain the essential content (values) of the fields and not information about the fields, such as name or type. There is some overhead (for example, the field identifier), but not as much as JSON or XML. This comes at a cost though. Sender and receiver must use the same definitions so that fields are serialized and deserialized in the correct order, whereas JSON and XML both name the fields, so they are less sensitive to field ordering. This restriction applies to the M5 message system as well, but is perhaps less of an issue than for Protocol Buffers since the M5 is a single product.

### 7.4 Comparison

As stated in earlier, this Chapter will compare the M5 notation to the Google Protocol Buffer notation and will not include JSON or XML because they are text based.

TABLE 3 compares several feature sets for the two notations. The first is the set of primitive types that the notations support, followed by the features that support more complex types, the ability to include / exclude fields and support for other features such as thread synchronization and documentation.

**TABLE 3. Comparison of notation features**

<table>
<thead>
<tr>
<th>Primitive / scalar types</th>
<th>M5 Notation</th>
<th>Protocol buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>int16</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>int32</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>int64</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>uint32</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>uint64</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>sint32</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>sint64</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Field Type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>fixed32</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>fixed64</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sfixed32</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sfixed64</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>bool</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>float</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>double</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>string</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>data (bytes)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>enums</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Name-Value pairs</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Composite types**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Presence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Referencing other definitions</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Nested types</td>
<td>➕ (as groups)</td>
<td>✓</td>
</tr>
<tr>
<td>Groups</td>
<td>➕</td>
<td>Deprecated</td>
</tr>
<tr>
<td>Extensions</td>
<td>➕ (via inheritance)</td>
<td>✓ (message definitions)</td>
</tr>
<tr>
<td>Packed fields</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

**Presence**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Presence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>➕ (implied)</td>
<td>✓ (keyword)</td>
</tr>
<tr>
<td>Optional</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Repeated</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Field Ordering</td>
<td>➕ (implied)</td>
<td>✓ (explicit using tag ids)</td>
</tr>
</tbody>
</table>

**Other**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Presence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronization features</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Self documenting, browsable</td>
<td></td>
<td>✓ (with XSL stylesheet)</td>
</tr>
</tbody>
</table>
Protocol buffers provide more scalar types (referred to as primitive types in the M5 Notation) than the M5 Notation, but that is a relatively trivial issue since types are so easy to add. Rather than ask if they can be supported, it is better to ask if they need to be supported and so far the answer is “no”. The same point applies to the Packed fields feature in the Composite types section.

A more significant difference is the ability to reference other definitions. That feature could allow structuring of definitions to reduce the overall number of them, but equally it could lead to a tangled web of dependencies. The Extensions is also missing from the M5 Notation. This allows a developer to add fields to a new definition based on another.

The presence features are essentially the same, though the repeated field feature has some interesting differences. The M5 Notation allows for 2 approaches to the repeated fields. One uses the CountedGroup and the other the SimpleGroup. The first requires a count to be available when the message is de-serialized and the second expects that the rest of the message contains instances of the repeated field, so does not need a count. The Protocol buffers approach uses tags to identify fields so it is not necessary to include a count. Including such a count would however, make it possible to detect corruption of the data. The count can be included in the serialized data as part of the lower level protocol.

Protocol buffers are a clear winner in the language support stakes. The M5 code generators only produce C#. Adding C++ support is certainly possible, but the existing Application engines would need to be refactored to obtain any real benefits and that work would dwarf the effort to change the code generators, so, in other words, is not likely to happen.

Performance is another criterion worth comparing. Google claims that protocol buffers are “smaller, faster and simpler” than XML. Since the M5 Notation uses XML, then it would seem that protocol buffers win there too, but that comparison is not valid.
Google’s claim relates to serializing the data into XML, something that the M5 code
generators do not do. The availability of packed fields in Protocol buffers would
contribute to smaller sizes of messages at the cost of the packing / unpacking
operations. Several integer types are available in packed formats whereas the M5 uses
fixed length fields, so again smaller sizes are achievable with some cost to performance.

The “simpler” claim is somewhat disingenuous. Which is simpler for a human reader?
XML is human readable whereas a densely packed sequence of bytes is not. Google’s
notation is fairly simple to read and does not have the angled brackets that allegedly
hurt the eyes.

On balance Protocol buffers are probably more generally useful than the M5
Specification Notation, but they do not provide all features that are essential for the M5.
The features missing from the M5 Notation can be added since it is wholly in our
control, but it is unlikely that features essential to the M5 will appear in Protocol
buffers. Although both are examples of Domain Specific Languages the protocol
buffers notation is more general purpose than the M5 Notation.
8. CONCLUSIONS

The construction of a Domain-Specific Notation for describing application event messages in the M5 Analyzer has clearly been a success. The notation and its associated code generators have been in daily use almost since the beginning of the re-engineering project started in 2007 and have required very little maintenance in recent times.

The development of the notation was somewhat organic, with features added to the language as the need became apparent. This breaks Raymond’s recommendation given in Chapter 4.2, but perhaps in this case it was successful because the features were not “crufty”.

With the perspective given by preparing this document it becomes apparent that the language provides features that ought to be more distinct from each other. It is not merely a language for describing the structure of messages; it also conveys functional instructions. Even with that capability, it remains a limited language suitable only for the domain that it was intended for and the functional aspects of the language are applicable only to the Windows environment.

Given that the message definitions are orthogonal to concepts like the synchronization and locking features, it could make sense to separate them more explicitly, though the features do relate directly to the language items they are associated with.

The User Interface architecture is certainly a key factor in the success of the approach. The separation of data storage from data serialization operations is a significant benefit in isolating the UI behaviour into well-defined components, allowing them to be developed and tested without the rest of the system being in place.

Given the availability of certain pre-requisites, e.g. readily available domain expertise and management support, Domain specific languages and supporting tools can be an excellent approach to formally capture knowledge that might not otherwise be expressed so clearly.
9. REFERENCES

BIBLIOGRAPHY


Appendix A Message Operation Search Tool Output

File: coretex\Sources\COCMsgDataSubscription_U.pas
procedure COCMsgDataSubscription.Serialise(rProxyMsg:COCProxyMsg)
Msg: AddBool(m_bDataRequired);
Msg: AddBool(m_bUsePurchaseList);
Msg: AddInt32(integer(m_aElements.count));
Msg: AddInt32(PCOTSbsElem(m_aElements[i]).m_iSubscriptionId);
Msg: AddString(PCOTSbsElem(m_aElements[i]).m_strDescription);
Msg: AddInt32(PCOTSbsElem(m_aElements[i]).m_iConnectionId);

File: coretex\Sources\COCMsgData_U.pas
procedure Deserialise(rProxyMsg:COCProxyMsg)
Msg: SetMessageInfo(pMsgInfo:COCMsgInfo);
Msg: AddInt32(m_uDataLenInBits);
Msg: AddInt32(m_iConnectionId);
Msg: AddData(m_pData,uDataLenInOctets);
Msg: AddBool(true);
Msg: AddBool(false);

File: coretex\Sources\COCMsgInfo_U.pas
procedure COCMsgInfo.Serialise(rProxyMsg:COCProxyMsg)
Msg: AddInt64(m_u64_TimeStamp);
Msg: AddInt32(m_iSourceConnectionId);
Msg: AddInt32(m_iLineNumber);
Msg: AddInt32(m_uCallId);
Msg: AddInt32(integer(m_Direction));
Msg: AddInt32(m_iEventType);
Msg: AddInt32(m_iInterfaceUnitNumber);
Msg: AddInt32(m_iPathId);
Msg: AddInt32(m_iSubchannelId);

File: coretex\Sources\COCProxyAppEvent_U.pas
procedure COCProxyAppEvent.HandleEvent(rAppEvent:COCAppEvent)
Msg: AddInt8(ShortInt(m_eProxyType));
Msg: AddInt8(ShortInt(eAEHandleEvent));
Msg: AddRemotePointer(m_vRemoteObject);
Msg: AddLocalPointer(nil);
Msg: AddInt32(rAppEvent.m_iEventType);
Msg: AddInt32(rAppEvent.m_iLen);
Msg: AddInt32(rAppEvent.m_iTargetId);
Msg: AddInt32(rAppEvent.m_iSequenceNumber);
Msg: AddData(rAppEvent.m_pData,rAppEvent.m_iLen);
Appendix B Message Log Parser

The Message Log Parser is a tool that was developed early in the M5 UI re-engineering project. It is used to parse files produced by the running M5 when in logging mode.

The screenshot in FIGURE 14 shows the main view of the Message Log Parser. The red overlays number each table in the screenshot and the table descriptions follow.

Table 1 (in the screenshot) shows the “raw” message contents as a single line view. Table 2 shows several lines of partially decoded service messages. In this table we only decode the standard message frame: Service Id, Method Id, Local Reference and Remote Reference. The table also shows the direction of the message and the remaining length (after decoding the standard frame). The remaining octets of the message are shown in the Message Body column of the table.

FIGURE 14. Screenshot of message parser
Table 3 enhances the standard message decoding by adding the name for the UI resource involved in the message transaction. Rather than show the Service and Method IDs as numbers (as in Table 2), this table shows the Service and Method names. The message direction is shown with the Method name in the Message column. The Local and Remote References are now shown in the UI and Engine Instance columns.

Table 4 shows the message body, if that information is available. This is currently done using two approaches. The message body for Application Events is determined by using helper classes built by the M5 Code Generator using the Message Specification. For other message types the content is decoded by manually written code. The columns of Table 4 show the Field Name, Field Type and Field Contents for the fields in the message body. The Field Contents for the currently selected row are also shown in Table 5 (because there is more space to show them).

Table 6 shows the current references for the services that have been “discovered” by parsing the log file.

Each tabbed view has some simple controls. There are five buttons or markers. They are labelled “N”, “B”, “MO”, “PD” and “PN”. The “N” button will be enabled and will change colour to yellow when the selected message has a note attached to it (see Appendix B.3). The “B” marker will be enabled and change colour to green when the selected message has a bookmark. Notes are also regarded as bookmarks. (Their difference is primarily in their effect on the log file.)

When there is a message operation log file in the same location as the current log file (i.e. the log file whose contents are shown in this view), then the “MO” button is enabled. Pressing this button will open the Message Operation view (see Appendix B.2).

The “PD” and “PN” buttons are used to step between “differences” in the message list. See Appendix B.4 for information on comparing the contents of two log files.

In addition to the buttons and markers on the tabbed view there are also some operations available on the menu bar and these are described in Appendix B.1.

B.1 The Menu Bar

There are three main topics on the menu bar: File, Tools and Window.
B.1.1 File Menu
This offers the usual file operations:

Open – looks for and offers files with “.m5t” extension

Save – allows changes to the log information to be saved. This is relevant after adding notes or bookmarks (see Appendix B.3)

Save As – allows the log information to be saved using a different filename.

Close – closes the log view and associated file. If notes or bookmarks have been added, the user is asked if the changes should be saved.

Exit – closes the parser tool

B.1.2 Tools Menu
This menu allows the user to add or remove notes or bookmarks and step between them. Refer to Appendix B.3.

B.1.3 Window Menu
The parser view can be tiled (or split) horizontally or vertically to assist in comparing two log files. If more than one file is currently opened, then the selected view will be moved into the right hand view when vertically tiled, or the lower view when horizontally tiled. The tile type can be changed by clicking the appropriate choice in the menu. The tile view is closed by clicking on the selected tile type.

If there is only one file open, selecting a tiled view will cause a new view of the log file to be made – i.e. the log file will be parsed again.

File, bookmark and note operations currently only act on the left hand / upper tile. When the tiled view is closed, the view in the right or lower tile will be moved to the remaining view.

Several additional operations become possible when a tiled view is open. First of these is Synchronised Scrolling. Scrolling or selecting a message in one tile causes the same effect in the other. Note that if there are different sized logs in the two tiles the effect may be different from what is expected.
The selected views in the two tiles can be compared (Compare all messages). Alternatively, the selected messages in the two selected views can be compared (Compare selected messages). See Appendix B.4.

### B.2 Message Operation View

The Message Operation view can be opened by pressing the “MO” button, which is enabled when there is a message operation log file with the same name as the current main view in the same folder as the log file for that main view. So, for example, if the main view shows the message trace from a file named “StartStop.m5t” and there is a file called “StartStop.pmt” in the same folder as “StartStop.m5t”, then the “MO” button on the main view is enabled.

This view shows the operations performed on service messages from creation to destruction. There are four tables and one text area in the view, shown in FIGURE 15.

Once again the red overlays number the tables referred to in the following description. Table 1, the Message List, shows the message operations on all types of messages. There are two columns, one showing the message reference, the second showing the number of operations on the message. Messages can appear more than once in the table if they are reused in another context⁸.

---

⁸ The first operation of a reuse activity is Clear rather than Create.
Table 2, the top Operation list, shows the lifecycle for the message selected in Table 1. The last operation in the lifecycle will usually be Destroy, but if the message is reused (i.e. if a Clear operation is performed later) then the last operation will probably be GetMessageData, but no guarantees are offered! This table has four columns. The first shows the operation by name. The second column shows the index\(^9\) of the operation in the log file. This column can be useful when looking to confirm that a suggested pair of Application Event and Content messages makes sense. It also hints at the relative lengths of message lifetimes. Take some time to look at the differences in index values for Create and Destroy operations on different messages (and come up with some explanations). The last two columns in the table (currently) only really have meaning for the AddData, GetData, GetMessageData and SetMessageRef operations. These are

\(^9\) The index of the very first operation in the file is 0.
the operations we are interested in to link the Application Event and Content messages. So for those operations the Buffer Reference column shows the address of the buffer accessed in the operation and the Buffer Length column shows its length.

Tables 3 (the Application Event List) and 4 (the bottom Operation list) show the same information as Tables 1 and 2, but only for Application Event messages. The number of operations on Application Events from the UI to the Engine is 14 and from the Engine to the UI is 17. When an Application Event is selected by clicking on it in the Application Event list, the tool looks for the corresponding Content message. For an Application Event sent by the Engine to the UI, the Content message will be built before the Application Event is and vice versa for Application Events from the UI to the Engine. Note that the previous statement does not mean that the Content message is necessarily created first in the Engine => UI case. It means that all additive operations on the Content message are completed before the AddData operation on the Application Event message. In particular, for the Engine to UI case, we should see that the index of the GetMessageData operation on the Content message (in table 2) is lower than the index of the AddData operation on the Application Event (in table 4). To help with checking that the suggested pair is a sensible selection the Buffer Reference cell in the appropriate row in tables 2 and 4 is marked with a different colour.

Some Application Events do not have any content, meaning that the AddData operation uses an empty buffer, resulting in a 0 value reference and 0 value length. In that case the tool may still suggest a content message. The suggestion may still make sense, but can be ignored since there is no useful information provided. If the buffer reference is 0, then a reminder of this is given by turning a text field between the Message List and the Application Event List red. (The text is always shown as a reminder, but it is emphasised when it matters most.) If the indexes of the operation pairs GetMessageData/AddData / GetData/SetMessageRef are properly related, then the suggested pair may be valid. If there is a very large difference in the index values, then the suggested pair is probably not valid.

The view also shows a text area (marked as 5 in FIGURE 15). Since the purpose of this tool is to help reverse engineer the message interface and we want the message interface

---

10 This difference is an artefact of logging the information from the engine side.
to be described using XML format Message Specifications, the tool makes an effort to provide an XML description of the content of the Application Event. Please read the next section carefully before using this information.

B.2.1 Important notes about the XML description of the Application Event content

The format of the XML Message Specification is formally described in the Message Specification schema file, MessageInterface.xsd, reproduced in Appendix D. The content of the text area in the Message Operation view does not properly conform to that schema because some elements required in a valid specification are missing. The suggested XML cannot therefore be pasted into a file and be used directly. Parts of the suggested XML can however be pasted into a properly formed specification document. Specifically you can copy the “Event” element and paste it into your document. However, you will have to do some editing before you can use it.

An Event element in the message specification should provide a description of the content of the Application Event for both directions. The Event element has several children, one of which is MessageToUI and another is MessageToEngine. The suggestion given in the text area is only for one of those cases – the case selected in the Application Event list. When you copy the information you will need to make sure that you do not create another event with the same id as then the code produced by the M5 Code Generator will not compile.

Several elements in the suggestion will need additional information. If you choose to paste the Event element and its children into your document, you will need to add the event name in the Name element, edit the Id element to give the correct value if that is missing or wrong and provide the event description. For the presented case, (MessageToUI or MessageToEngine) you will also need to provide a description. All content types also need descriptions and property names. The latter is essential as it is used to create the property set for your UI resource. The description is used as a comment, so is helpful, but the code generator can work without it. Various attributes on elements will also need to be added as required. Read the Message Specification guide for help on this matter.
If you already have an event element with the correct id for the proposal, then presumably you have already edited all the necessary information for the Event element. You should paste this proposal into the correct child of Element.

Some operations in the Operation table are the result of another operation and are not true content in the sense we are interested in. For example, a String field in a service message is stored as a length followed by the string characters. For an AddString operation the operation lifecycle also shows an AddInt32 operation. The same point applies to the GetString, AddData and GetData operations. These additional length operations are consumed when producing the XML description.

The XML suggestion is very basic and does not apply any inference rules to the case. You will need to read the Message Specification guide to properly form the content description. A specific issue you will need to address is that many messages include so-called counted groups. These are groups of content that appear in the message a particular number of times. The message should contain a field that holds the count and then that number of repeats of the group content. For example, a message listing logical lines contains a count of the number of lines and then four strings for each line. The suggested XML for a case with 4 lines will show the content as an integer followed by 16 strings, whereas it should be shown as a counted group containing 4 strings.

The “Application” part provides some help by giving the UIResource name, but that should only appear once for each application in a specification document.

**B.3 Using Notes and Bookmarks**

Messages can be annotated or bookmarked to help movement through the file. The Add Note or Add Bookmark options in the Tools menu are used for this. Add Note will open a dialog to allow text to be entered. Add Bookmark will simply mark the message.

Movement between notes and bookmarks is achieved by using the Next and Previous functions in the menu and a note or bookmark can be removed with the Remove function.

Saving notes in a log file increases the size of the file. The note is inserted before the message entry in the log file. Saving bookmarks has no effect as it simply changes the case of the direction marker in the file (see Appendix B.5 for details of the file formats).
B.4 Comparing Logs

Message comparison is a field by field comparison. Service Ids are compared. If they are the same, then Method Ids are compared. There is no point in continuing compare operations if the Service or Method Ids are different, but if not then the message bodies are compared. The Local and Remote References are not compared, since we expect that they can legitimately be different for different runs of the M5.

Similarly some fields in the message body can change between runs. Currently, when comparing message bodies, the tool ignores fields specified as Local or Remote Pointers. Some other field types will need to be ignored, but are currently throwing up differences.

Our intention is to include an attribute on the fields of Application Events to indicate if they should be included in a comparison operation.

Differences are marked by changing the colour of the cells corresponding to the difference. So, for example the Service Id column in Table 2 and the Service Name column in Table 3 will be marked if the Service Ids are different. Differences in the message bodies will result in markers in the Message Body cell of Table 1 and the corresponding row of Table 4.

B.5 Log File Formats

The logging enhancements will produce two log files from a running M5. The first contains the messages exchanged between the UI resource and Application engine subsystems. The second contains the operations performed on service messages (implemented in the COCProxyMsg class).

1. We use two files for a number of reasons:
   - We do not necessarily need both files.
   - The message log file provides a lot of useful information for the cost of changing one deployed M5 file (NHCore.dll).
   - The operation log file changes impact on a lot of deployed components (all that depend on COCProxyMsg).
Two class methods are defined in the header file, so they may be inlined wherever the COCProxyMsg class is used. Changing COCProxyMsg.h directly affects all those components.

Those developers that can live with the message trace only need to change one file; the rest may as well replace all dlls (because there are so many of them, it is quicker that way!).

2. The files are written by different components, so it is easier, better and safer to use different file references and different files than to use one and cross-connect otherwise independent components.

3. The information in the two files can be usefully analysed independently.

4. The message log needs variable length entries, the operation log needs fixed length entries

5. We started with the message log and added the operation log later, i.e. it is just an operational convenience.

Reasons 1, 2 and 5 are the most compelling justifications for separate files.

B.5.1 Message log file

Messages are stored into a file named UIEngMsg.m5t. That filename is always used to keep the impact of the logging system on the M5 code as light as possible. Entries are appended to the file, so the file should be renamed, moved or deleted between “runs” of the M5.

The format of the file is fairly simple. A marker octet is followed by other information. The tool currently recognises 6 markers:

- ‘V’ – for the logger version
  - This marker is followed by a single octet containing a version number

- ‘I’ for messages going to the UI from the Engine
  - This marker is followed by:
    - the message buffer reference
    - the message length
    - the message contents

- ‘O’ for messages going from the Engine to the UI
  - This marker is followed by:
the message buffer reference
- the message length
- the message contents

- ‘i’ for **Bookmarked** messages going to the UI from the Engine
  - This marker is followed by:
    - the message buffer reference
    - the message length
    - the message contents

- ‘o’ for **Bookmarked** messages going to the Engine from the UI
  - This marker is followed by:
    - the message buffer reference
    - the message length
    - the message contents

- ‘N’ for a note
  - This marker is followed by:
    - the note length
    - the note contents

The logger will only write the first three markers. The parser tool will write all of them.

### B.5.2 Message Operation log file

The message operation log file has fixed size entries:

- Message reference – 4 octets, the least significant first
- Additional Information 1, a reference – 4 octets, the least significant first
- Additional Information 2, an integer – 4 octets, the least significant first
- The message operation identifier – refer to the code for the values (in case changes are made to make this out of date).

As can be seen from the above description, the additional information fields are the same size and the octet position significance is the same, so it is only the tool’s convention that the first is a reference and the second is an integer.
Appendix C An Event based Message Protocol

As has been described, Application Events are used by the Application Engines and UIs to exchange information. The M5 was intended to be built as an example of an event driven architecture. The UI would send an event to the Engine and would process any responses when they arrived. Similarly, the Engines would wait for events and act accordingly when they occurred.

Unfortunately, this design intention was not completely fulfilled by the implementation. For whatever reason (the use of legacy code perhaps), sequences of initiating event, response to the event, response to the response, etc. were treated as atomic operations. Instead of splitting behaviour into chunks triggered by events the implementation would assume that after sending the initiating event, the entire sequence would be complete.

FIGURE 16 shows a stylized view of the asynchronous approach. A User Action to get information about the system analysers is sent to the Controller. The Controller instructs the Application Model to build an application event and that event is sent via the Gateway to the Domain Model.

In this approach the UI thread is involved in delivering the User Action from the View to the Controller and will also be used to get the application event. The event is delivered to the Gateway using another thread and the UI thread is available for other activity.

The Domain Model sends the response, the notification of which reaches the View after the Application Model has been updated. The View Update operation must use the UI thread. In this example the View sends another User Action\(^{11}\) to the Controller and the process is repeated.

In contrast, FIGURE 17 shows the same scenario using synchronous operations from the UI. From a system perspective, FIGURE 17 is remarkably more complicated, but from a developers perspective it can seem simpler because the View will not continue operating until control returns from the UserActionX invocation and the

\(^{11}\) The label "User Action" is used to refer to any change notification or information request sent by the View to the Domain Model. The system user may cause a single "User Action" or cause a sequence of them.
“UpdateViewY” notification can be ignored or assumed to have happened. For an apparent simplification for the developer, we pay some system costs. By nesting as few as 2 event-responses we can see that operations in Controller have been invoked 4 times with an obvious effect on the system call stack. At least as significant is the fact that the UI thread is blocked for the entire update cycle.

Referring again to FIGURE 16, the comment box in the diagram marks the three periods when the UI thread is blocked. This arrangement is likely to be more responsive to the User, but for short update cycles that change may not be apparent. On the other hand, the impact on the system call stack is greatly reduced in the asynchronous approach.

Post-processing of Responses X & Y is implemented and performed differently in these two cases. In the synchronous case (FIGURE 17) post-processing will occur when control returns from the invocation of operation UserActionX on the Controller. The implementation is likely to be in the same method as the UserActionX invocation. In the asynchronous case (FIGURE 16) post-processing will be performed in the ResponseY handler.
FIGURE 16. Asynchronous Action - Response Sequence Diagram

UI thread is busy during these operations
FIGURE 17. Synchronous Action -Response Sequence Diagram
Appendix D Message Specification XML-Schema

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <!-- some pattern and enumerated types -->
  <xs:simpleType name="EventCode">
    <xs:restriction base="xs:string">
      <!-- an event code can be expressed as a hex number or as a decimal number -->
      <!-- although there are built in types for those numbers we want to control the form -->
      <!-- so we define the code as a string and use pattern restrictions -->
      <!-- a hex number must start with 0x and have at least two digits (with leading zero if needed) -->
      <!-- e.g. 0x01, 0x22, 0x1234 etc -->
      <!-- letters can be lower or upper case, but must be consistent within the number -->
      <!-- i.e. 0x1A2b is not allowed -->
      <xs:pattern value="0x[0-9a-f][0-9a-f]+"/>
      <xs:pattern value="0x[0-9A-F][0-9A-F]+"/>
      <xs:pattern value="[0-9]+"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="MessageFieldEnumeration">
    <xs:restriction base="xs:string">
      <xs:enumeration value="MT_STR"/>
      <xs:enumeration value="MT_INT8"/>
      <xs:enumeration value="MT_INT16"/>
      <xs:enumeration value="MT_INT32"/>
      <xs:enumeration value="MT_INT64"/>
      <xs:enumeration value="MT_BOOL"/>
      <xs:enumeration value="MT_DATA"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- now some more sophisticated types -->
  <xs:complexType name="PrimitiveDataType">
    <xs:sequence>
<xs:group ref="CommonPropertiesGroup"/>
</xs:sequence>
<xs:attribute name="type">
  <xs:annotation>
    <xs:documentation>The primitive data type for this message field</xs:documentation>
  </xs:annotation>
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Boolean"/>
      <xs:enumeration value="Int8"/>
      <xs:enumeration value="Int16"/>
      <xs:enumeration value="Int32"/>
      <xs:enumeration value="Int64"/>
      <xs:enumeration value="Double"/>
      <xs:enumeration value="Data"/>
      <xs:enumeration value="String"/>
      <xs:enumeration value="LocalPointer"/>
      <xs:enumeration value="RemotePointer"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
</xs:complexType>

<xs:complexType name="CountedGroupType">
  <xs:group ref="ContentGroup"/>
  <xs:attribute name="indexName" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="EnumerationType">
  <xs:sequence>
    <xs:group ref="CommonPropertiesGroup"/>
    <xs:element name="Entry" maxOccurs="unbounded">
      <xs:complexType>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="value" type="xs:int" use="optional"/>
        <xs:attribute name="description" type="xs:string" use="optional"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="ItemType">
   <xs:group ref="ContentGroup"/>
</xs:complexType>

<xs:complexType name="ResponseType">
   <xs:simpleContent>
      <xs:extension base="xs:string">
         <xs:attribute name="targetId" use="optional">
            <xs:simpleType>
               <xs:restriction base="xs:string">
                  <xs:enumeration value="-1 (Determined at runtime)"/>
                  <xs:enumeration value="TT_INIT"/>
                  <xs:enumeration value="TT_DEFAULT"/>
                  <xs:enumeration value="TT_STARTINGUP"/>
                  <xs:enumeration value="TT_SHUTDOWN"/>
               </xs:restriction>
            </xs:simpleType>
         </xs:attribute>
         <xs:attribute name="synchronousSend" type="xs:boolean" use="optional"/>
      </xs:extension>
   </xs:simpleContent>
</xs:complexType>

<xs:complexType name="MessageType">
   <xs:sequence>
      <xs:element name="ResponseList" minOccurs="0" maxOccurs="1">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="Response" type="ResponseType" minOccurs="1" maxOccurs="unbounded"/>
            </xs:sequence>
         </xs:complexType>
      </xs:element>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
         <xs:group ref="TypeGroup"/>
      </xs:sequence>
   </xs:sequence>
</xs:complexType>
<xs:complexType>
<xs:sequence>
<xs:attribute name="useHelper" type="xs:boolean" use="optional"/>
<xs:attribute name="MonitorActivityDuration" type="xs:boolean" use="optional"/>
</xs:complexType>

<xs:complexType name="NameValueGroupType">
<xs:sequence>
<xs:group ref="CommonPropertiesGroup"/>
<xs:element name="Name" type="xs:string" minOccurs="0"/>
<xs:element name="FieldType" type="PrimitiveDataType" minOccurs="0"/>
<xs:element name="Value" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="OptionGroupType">
<xs:sequence>
<!-- specify the test value to be used in the if clause -->
<xs:element name="testValue">
<xs:complexType>
<xs:simpleContent>
<xs:annotation>
<xs:documentation>Used to indicate that the test Value is a string literal (not a class member)</xs:documentation>
</xs:annotation>
<xs:extension base="xs:string">
<xs:attribute name="stringLiteral" type="xs:boolean" use="optional" default="true"/>
<xs:attribute name="isEnum" type="xs:boolean" use="optional" default="true"/>
</xs:extension>
</xs:simpleContent>
</xs:complexType>
</xs:element>
<!-- specify the result to be used in the if clause -->
<xs:element name="testResult" type="xs:boolean"/>
<!-- the basic group includes the description and more importantly the property name to use in the test -->
<xs:group ref="ContentGroup"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="PropertyNameType">
    <xs:simpleContent>
        <xs:annotation>
            <xs:documentation>Used to indicate that the property is a string literal (not a class member)</xs:documentation>
        </xs:annotation>
        <xs:extension base="xs:string">
            <xs:attribute name="hides" type="xs:boolean" use="optional" default="false"/>
            <xs:attribute name="useLock" type="xs:boolean" use="optional" default="false"/>
            <xs:attribute name="emitAsProperty" type="xs:boolean" use="optional" default="false"/>
            <xs:attribute name="value" type="xs:string" use="optional" default=""/>
            <xs:attribute name="stringLiteral" type="xs:boolean" use="optional" default="true"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="SimpleGroupType">
    <xs:sequence>
        <!-- the basic group includes the description and more importantly the property name to use in the test -->
        <xs:group ref="ContentGroup"/>
    </xs:sequence>
</xs:complexType>

<!-- some groups -->
<xs:group name="ContentGroup">
    <xs:sequence>
        <xs:group ref="CommonPropertiesGroup"/>
        <xs:group ref="TypeGroup" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:group>

<xs:group name="CommonPropertiesGroup">
    <xs:sequence>
        <xs:element name="Description" type="xs:string"/>
        <xs:element name="PropertyName" type="PropertyNameType"/>
    </xs:sequence>
</xs:group>
<xs:group name="TypeGroup">
    <xs:choice>
        <xs:element name="CountedGroup" type="CountedGroupType"/>
        <xs:element name="NameValueGroup" type="NameValueGroupType"/>
        <xs:element name="OptionGroup" type="OptionGroupType"/>
        <xs:element name="PrimitiveType" type="PrimitiveDataType"/>
        <xs:element name="SimpleGroup" type="SimpleGroupType"/>
    </xs:choice>
</xs:group>

<!-- The definition proper -->
<xs:element name="MessageInterface">
    <xs:annotation>
        <xs:documentation>A description of the message interface implemented between the UI and Core parts of the M5 Analyser</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Application">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="ApplicationInformation">
                            <xs:complexType>
                                <xs:sequence minOccurs="0" maxOccurs="unbounded">
                                    <xs:element name="PrimitiveType" type="PrimitiveDataType"/>
                                </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                        <xs:element name="UIInformation">
                            <xs:complexType>
                                <xs:sequence minOccurs="0" maxOccurs="unbounded">
                                    <xs:element name="PrimitiveType" type="PrimitiveDataType"/>
                                </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="Extends" minOccurs="1" maxOccurs="1">
  <xs:complexType>
    <xs:attribute name="namespace" type="xs:string" use="required">
      <xs:documentation>Gives the namespace for the base class</xs:documentation>
    </xs:attribute>
    <xs:attribute name="classnameRoot" type="xs:string" use="required">
      <xs:documentation>Gives the classname root for the base class</xs:documentation>
      <xs:documentation>classname is constructed by prepending COC or COI and postpending class type</xs:documentation>
    </xs:attribute>
  </xs:complexType>
</xs:element>

<xs:element name="EnumerationList" minOccurs="0" maxOccurs="1">
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="Enumeration" type="EnumerationType"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Event" minOccurs="0" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name">
        <xs:complexType>
          <xs:simpleContent>
            <xs:annotation>
              <xs:documentation>Used to indicate that the event hides an inherited member</xs:documentation>
            </xs:annotation>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:attribute name="messageCounter" type="xs:string" use="optional"/>
<xs:attribute name="targetId" use="optional">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="-1 (Determined at runtime)"/>
      <xs:enumeration value="TT_INIT"/>
      <xs:enumeration value="TT_DEFAULT"/>
      <xs:enumeration value="TT_STARTINGUP"/>
      <xs:enumeration value="TT_SHUTDOWN"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute name="synchronousSend" type="xs:boolean" use="optional" />
</xs:extension>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="EngineName">
  <xs:annotation>
    <xs:documentation>The name of the Engine component for this Application</xs:documentation>
  </xs:annotation>
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="APAbisScannerEditor"/>
      <xs:enumeration value="APAdapterOptions"/>
      <xs:enumeration value="APAnalyser"/>
      <xs:enumeration value="APAnalyserList"/>
      <xs:enumeration value="APApplicationManagement"/>
      <xs:enumeration value="APCapture"/>
      <xs:enumeration value="APCommon"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:enumeration value="APConnectionConfiguration"/>
<xs:enumeration value="APCounter"/>
<xs:enumeration value="APCSAPrefilter"/>
<xs:enumeration value="APCSStatisticsUIEngine"/>
<xs:enumeration value="APCTUserApp"/>
<xs:enumeration value="APFileDialog"/>
<xs:enumeration value="APGeneralOptions"/>
<xs:enumeration value="APGraphicalNetwork"/>
<xs:enumeration value="AP2GHOTracker"/>
<xs:enumeration value="APMessageDialog"/>
<xs:enumeration value="AppEngineBase"/>
<xs:enumeration value="APPrefilter"/>
<xs:enumeration value="APProgressDialog"/>
<xs:enumeration value="APProtocolMonitor"/>
<xs:enumeration value="APQoSConfig"/>
<xs:enumeration value="APRCAEngine"/>
<xs:enumeration value="APRecorder"/>
<xs:enumeration value="APReferenceConfig"/>
<xs:enumeration value="APStateMonitor"/>
<xs:enumeration value="APStats"/>
<xs:enumeration value="APTfsEditor"/>
<xs:enumeration value="APTraceList"/>
<xs:enumeration value="APTRAUStreaming"/>
<xs:enumeration value="APTrigger"/>
<xs:enumeration value="APWorkspace"/>
<xs:enumeration value="NoEnginePeer"/>

</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="UIName">
  <xs:annotation>
    <xs:documentation>The name of the UI component for this Application</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="CSStatisticsConfiguration"/>
    <xs:enumeration value="UIAbisScannerEditor"/>
  </xs:restriction>
</xs:simpleType>
<xs:restriction>
  <xs:enumeration value="UIAboutWindow"/>
  <xs:enumeration value="UIAdapterOptions"/>
  <xs:enumeration value="UIAnalyser"/>
  <xs:enumeration value="UIApplicationManagement"/>
  <xs:enumeration value="UIBase"/>
  <xs:enumeration value="UICapture"/>
  <xs:enumeration value="UICounter"/>
  <xs:enumeration value="UIConditionList"/>
  <xs:enumeration value="UIConfiguration"/>
  <xs:enumeration value="UIConfigurationBase"/>
  <xs:enumeration value="UIConnectionConfiguration"/>
  <xs:enumeration value="UICounter"/>
  <xs:enumeration value="UICSAPrefilter"/>
  <xs:enumeration value="UICSAPrefilter"/>
  <xs:enumeration value="UICTUserApp"/>
  <xs:enumeration value="UIFileDialog"/>
  <xs:enumeration value="UIGeneralOptions"/>
  <xs:enumeration value="UIGraphicalNetwork"/>
  <xs:enumeration value="UI2GHOTracker"/>
  <xs:enumeration value="UILayerSettings"/>
  <xs:enumeration value="UILicenseServer"/>
  <xs:enumeration value="UIMessageDialog"/>
  <xs:enumeration value="UIOfflineAnalyser"/>
  <xs:enumeration value="UIPrefilter"/>
  <xs:enumeration value="UIProgressDialog"/>
  <xs:enumeration value="UIProtocolMonitor"/>
  <xs:enumeration value="UIStats"/>
  <xs:enumeration value="UITfsEditor"/>
  <xs:enumeration value="UITRAUStreaming"/>
  <xs:enumeration value="UITrigger"/>
  <xs:enumeration value="UIWorkspace"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="namespace" type="xs:string" use="required">
  <xs:annotation>
    <xs:documentation>Specifies the namespace of the UI resource</xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name="uiIsSingleton" type="xs:boolean">
  <xs:annotation>
    <xs:documentation>Specifies if the UI is a singleton resource</xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name="invokeTestRequired" type="xs:boolean">
  <xs:annotation>
    <xs:documentation>Specifies if a test for using the Invoke method is required when updating the UI</xs:documentation>
  </xs:annotation>
</xs:attribute>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
### Appendix E Message Notation Style Sheet

```xml
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/"
  >
    <HTML
    >
      <A name="top"/>
      <xsl:for-each select="MessageInterface/Application"
      >
        <TABLE
        >
          <TR>
            <TD><IMG border="0" src="NH_M5logo.jpg" width="128" height="127" alt="" /></TD>
            <TD><H1><xsl:value-of select="@UIName" /> Message Specification</H1></TD>
          </TR>
        </TABLE>
        <TABLE BORDER="1"
        >
          <TR STYLE="font-weight:bold">
            <TD>Event Name</TD>
            <TD>Event Id</TD>
            <TD>Description</TD>
            <TD>Details</TD>
          </TR>
          <xsl:for-each select="Event"
          >
            <TR>
              <TD><xsl:value-of select="Name" /></TD>
              <TD><xsl:value-of select="Id" /></TD>
              <TD><xsl:value-of select="Description" /></TD>
              <TD><A HREF="#{generate-id(Name)}">More</A></TD>
            </TR>
          </xsl:for-each
        </TABLE>
        <xsl:for-each select="Event"
        >
          <A name="#{generate-id(Name)}"/>
          <H3><xsl:value-of select="Name" /></H3>
          <B><I><xsl:value-of select="Description" /></I></B>
          <H3>Message To UI</H3>
          <TABLE
          >
            <TR STYLE="font-weight:bold">
              <TD>Note</TD>
              <TD>Field Type</TD>
              <TD>Description</TD>
              <TD>Associated Property</TD>
            </TR>
            <xsl:call-template name="fields"
            >
              <xsl:for-each select="MessageToUI/*"
              >
              </xsl:for-each
            </xsl:call-template
          </TABLE>
          <TABLE
          >
            <TR STYLE="font-weight:bold">
              <TD>Response</TD>
            </TR>
            <TD/>
            <TD>Target Id</TD>
          </TABLE>
        </xsl:for-each
      </TABLE>
      <HR/>
    </xsl:for-each
    </xsl:template
  >
  </xsl:stylesheet>
```
### Message To Engine

<table>
<thead>
<tr>
<th>Note</th>
<th>Field Type</th>
<th>Description</th>
<th>Associated Property</th>
</tr>
</thead>
</table>

### Response

<table>
<thead>
<tr>
<th>Response</th>
<th>Target Id</th>
</tr>
</thead>
</table>

---

Return to top

Go back to main resource message specification page
<xsl:template name="fields">
  <xsl:choose>
    <xsl:when test="@type">
      <xsl:call-template name="primitiveType"/>
    </xsl:when>
    <xsl:when test="@indexName">
      <xsl:call-template name="countedGroup"/>
    </xsl:when>
    <xsl:when test="testValue">
      <xsl:call-template name="optionGroup"/>
    </xsl:when>
    <xsl:when test="self::SimpleGroup">
      <xsl:call-template name="simpleGroup"/>
    </xsl:when>
  </xsl:choose>
</xsl:template>

<xsl:template name="primitiveType">
  <TR>
    <TD></TD>
    <TD><xsl:value-of select="@type"/></TD>
    <TD><xsl:value-of select="Description"/></TD>
    <TD><xsl:value-of select="PropertyName"/></TD>
  </TR>
</xsl:template>

<xsl:template name="countedGroup">
  <TR>
    <TD><I>Start of</I></TD>
    <TD><I>Counted Group</I></TD>
    <TD><I>using index <xsl:value-of select="@indexName"/></I></TD>
    <TD><xsl:value-of select="PropertyName"/></TD>
  </TR>
  <xsl:for-each select="*">
    <xsl:call-template name="fields"/>
  </xsl:for-each>
  <TR>
    <TD><I>End of</I></TD>
    <TD><I>Counted Group</I></TD>
    <TD><I>using index <xsl:value-of select="@indexName"/></I></TD>
    <TD></TD>
  </TR>
</xsl:template>

<xsl:template name="optionGroup">
  <TR>
    <TD><I>Start of</I></TD>
    <TD><I>Option Group</I></TD>
    <TD><I>testing <xsl:value-of select="PropertyName"/> against value <xsl:value-of select="testValue"/></I></TD>
  </TR>
  <xsl:for-each select="*">
    <xsl:call-template name="fields"/>
  </xsl:for-each>
  <TR>
    <TD><I>End of</I></TD>
    <TD><I>Option Group</I></TD>
    <TD><I>testing <xsl:value-of select="PropertyName"/> against value <xsl:value-of select="testValue"/></I></TD>
  </TR>
</xsl:template>
<xsl:template name="simpleGroup">
  <TR>
    <TD><I>Start of</I></TD>
    <TD><I>Simple Group</I></TD>
    <TD></TD>
    <TD><xsl:value-of select="PropertyName"/></TD>
  </TR>
  <xsl:for-each select="*">
    <xsl:call-template name="fields"/>
  </xsl:for-each>
  <TR>
    <TD><I>End of</I></TD>
    <TD><I>Simple Group</I></TD>
    <TD></TD>
    <TD><xsl:value-of select="PropertyName"/></TD>
  </TR>
</xsl:template>

<xsl:template name="fields">
  <xsl:for-each select="*">
    <xsl:template name="fields"/>
  </xsl:for-each>
</xsl:template>
Appendix F  A Browser View of a Message Specification

FIGURE 18 shows how an example definition looks when viewed in a browser. The appearance is controlled by an XSL stylesheet referenced in the message specification.

FIGURE 18: Specification presentation in browser