EMULATING ENTERPRISE SOFTWARE ENVIRONMENTS

Enabling Observation, Analysis and Assessment of the Run-Time Properties of Enterprise Software Systems

BY

CAMERON HINE

A thesis submitted in total fulfilment of the requirements of the degree of Doctor of Philosophy.

2012

Faculty of Information and Communication Technologies
Swinburne University of Technology
Abstract

Enterprise software systems operate in complex distributed, heterogeneous environments of particularly large scale. Typical environments can include tens-of-thousands of individual software systems, hosted by a variety of different physical machines, scattered across the globe. To fulfil their roles within this environment, enterprise software systems must interact with one another, using different communication protocols where necessary to overcome heterogeneity. Certain enterprise software systems must interact with large portions of the environment, essentially requiring communication with tens-of-thousands of other systems using dozens of different protocols.

Developing software for enterprise environments is difficult and, even once a system is feature complete, assessing its non-functional quality remains significant challenge. For credibility, these assessments are best based on observations and measurements made whilst the system operates in environments which are representative of those that will be encountered upon deployment. The distribution, heterogeneity and scale typical of enterprise environments makes it difficult to produce the requisite, representative, test-beds. Specifically, no existing tool or approach exists which can produce test-beds that are both highly-scalable and well-suited to representing environments containing primarily server-class systems; enterprise environments often contain mostly server-class, as opposed to client-class, systems.

Interaction modelling and emulation is our approach to producing test-beds containing large numbers of server-class systems. The communication behaviour of real enterprise systems are specified using an interaction model, describing how the various endpoint systems react to different message receptions. The emulator then generates a test-bed environment from these specifications, executing them to drive interaction with external systems under test.

We define a framework allowing different concrete interaction models to target different levels of endpoint behaviour detail. The level of detail required varies depending on the type of testing. Within this framework we present two concrete model stacks: rudimentary, and expressive. The rudimentary stack is less powerful than the expressive, but is capable of specifying a wide range of synchronous interactions, so long as no attempt is made to independently verify modifications. The expressive, on the other hand, can specify asynchronous interactions, as well as data store persistency. Sample interaction specifications are developed alongside the model stacks demonstrating practical application and also highlighting the capability of our approach to specifying server-class interactions.

An architecture and proof-of-concept implementation of an enterprise software environment emulator is described. Experiments confirm the highly-scalable nature of our approach; ten-thousand endpoints can be emulated using a single physical host. Furthermore, experiments using our emulator to help test an industry-grade enterprise identity management suite demonstrate the wider validity of the work.
Acknowledgements

This thesis would not have been possible without the support and consultation generously provided by friends, family, colleagues and supervisors. I would like to take this opportunity to thank all of these people who have helped me along the way.

Firstly, thanks to my supervisory team, Dr. Jean-Guy Schneider, Prof. Jun Han and Dr. Steve Versteeg. The feedback, advice, and discussions provided by this team throughout the project has been immeasurably valuable. I am especially grateful for they way in which each has helped shape my approach to research and problem solving in general. Additionally, thanks to Dr. Tim Ebringer whose early guidance helped inform my understanding of the key issues and concerns in this area.

Thanks to my fellow PhD students at Swinburne and with CA Labs for helping maintain my sanity throughout the project: Garth Heward, Alda Aleti, Dana Zhang, and many more.

I have been fortunate in the organisations with which I have worked in this project. I would like to thank Swinburne University of Technology for the opportunity to undertake my PhD. Additionally, thanks to CA Technologies, in particular CA Labs, for providing a practical, yet compelling research topic to explore and the funding with which to explore it.

Thanks to my family for their love, encouragement and support throughout. And to my friends for helping lift my spirits when the end seemed out of reach, and will no doubt help me celebrate now that it is here.
Declaration

I, the candidate Cameron Hine, hereby declare that the examinable outcome embodied by this dissertation:

- contains no material which has been accepted for the award to the candidate of any other degree or diploma, except where due reference is made in the text of the examinable outcome;

- to the best of the candidate's knowledge contains no material previously published or written by another person except where due reference is made in the text of the examinable outcome; and

- where the work is based on joint research or publications, discloses the relative contributions of the respective workers or authors.

______________________________

Cameron Hine
Publications

During the course of this project a number of peer-reviewed publications were produced. They are presented here for reference and also to highlight the corresponding material in the thesis itself.

  
  - Corresponds to the rudimentary interaction stack which is presented in Chapter 6.

  
  - This is the protocol model used in the expressive interaction stack presented in Chapter 7.

  
  - Presents an early version of the enterprise software environment emulator described in Chapter 8.

  
  - Contains detailed analysis of the CA Identity Manager experiments mentioned in Chapter 9.
## Contents

1 Introduction .............................................. 3  
  1.1 Context ................................................. 4  
  1.2 Interactive Representations of Enterprise Software Operating Environments ................................................................. 6  
  1.3 Interaction Modelling and Emulation ................................................................. 9  
  1.4 Contributions ............................................. 11  
  1.5 Outline ................................................... 12  

2 Related Work ............................................. 14  
  2.1 Testing Distributed Software Systems ................................................................. 15  
  2.2 Interactive Representations of Distributed Software Environments ................................................................. 18  
    2.2.1 Physical Replication ......................................... 18  
    2.2.2 Programmatic ............................................... 19  
    2.2.3 Performance Testing and Load Generation Tools ................................................................. 21  
    2.2.4 Software Performance Engineering ................................................................. 21  
    2.2.5 System-Level Virtualisation ................................................................. 22  
    2.2.6 Network Emulation ........................................... 22  
    2.2.7 Overview .................................................. 24  
  2.3 Interaction Modelling and Specification ................................................................. 27  
    2.3.1 Automata .................................................. 27  
    2.3.2 Process Calculi Based ......................................... 28  
    2.3.3 Objects, Components and Service Protocols ................................................................. 29  
    2.3.4 Formal Specification Languages ................................................................. 29  
    2.3.5 Grammars .................................................. 30  
    2.3.6 Discussion .................................................. 30  
  2.4 Summary ................................................... 31  

3 Mathematical Preliminaries .................................. 32  
  3.1 Symbols .................................................... 32  
  3.2 Sets ......................................................... 33  
    3.2.1 Power Sets ................................................ 33  
    3.2.2 Set Builder Notation ......................................... 33  
    3.2.3 Natural Numbers ............................................ 33  
    3.2.4 Booleans ................................................ 34  
  3.3 Sequences .................................................. 34
3.4 Functions .......................................................... 34
3.5 Generative Grammar Notation ............................... 36

4 Enterprise Software Systems, Environments and Observation 37
  4.1 Enterprise Software Systems and Environments ............... 37
  4.2 Conceptual Model .............................................. 38
    4.2.1 Hosts ..................................................... 40
    4.2.2 Channels ............................................... 42
    4.2.3 Communication Infrastructure ............................ 45
  4.3 Observing Run-Time Properties of Open Software Systems ........ 45
    4.3.1 Open Software Systems ................................ 45
    4.3.2 Closure Environments .................................... 48
    4.3.3 Dynamic Systems and Environments ...................... 49
  4.4 Summary .................................................... 51

5 Modelling and Emulation ........................................ 52
  5.1 Interaction Modelling and Emulation ......................... 52
    5.1.1 Behavioural Approximation .................................. 55
    5.1.2 Expected Benefits and Limitations ......................... 56
    5.1.3 Discussion ................................................ 57
  5.2 Interaction Modelling Framework .............................. 58
    5.2.1 Principals .................................................. 60
    5.2.2 Interaction Layers ......................................... 61
    5.2.3 Stacks and Specifications .................................. 65
  5.3 Server-Class Endpoints ....................................... 65
  5.4 Summary ..................................................... 66

6 Rudimentary Interaction Stack .................................. 67
  6.1 Protocol ..................................................... 68
    6.1.1 Deterministic Finite State Automata ....................... 68
    6.1.2 Next Operation ............................................ 70
    6.1.3 Sample Specification ...................................... 71
    6.1.4 Emulation ................................................ 75
  6.2 Behaviour ................................................... 75
    6.2.1 Placeholder Values and Message Skeletons ................. 75
    6.2.2 Request Handling ........................................ 81
    6.2.3 Emulation ................................................ 83
  6.3 Data Store .................................................. 84
    6.3.1 Sample Specification ...................................... 85
    6.3.2 Emulation ................................................ 86
  6.4 Discussion .................................................. 86
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1.4 Environment</td>
<td>141</td>
</tr>
<tr>
<td>10.2 Results</td>
<td>141</td>
</tr>
<tr>
<td>10.2.1 Memory Consumption</td>
<td>142</td>
</tr>
<tr>
<td>10.2.2 CPU</td>
<td>146</td>
</tr>
<tr>
<td>10.2.3 Workload Timing Characteristics</td>
<td>148</td>
</tr>
<tr>
<td>10.3 Discussion</td>
<td>152</td>
</tr>
<tr>
<td>11 Conclusions and Future Work</td>
<td>153</td>
</tr>
<tr>
<td>11.1 Conclusion</td>
<td>153</td>
</tr>
<tr>
<td>11.2 Future Work</td>
<td>157</td>
</tr>
<tr>
<td>A Listings</td>
<td>167</td>
</tr>
<tr>
<td>A.1 Sample Kaluta Configuration by CSV File</td>
<td>167</td>
</tr>
<tr>
<td>A.2 Haskell</td>
<td>171</td>
</tr>
<tr>
<td>A.2.1 Message Shape Handlers</td>
<td>171</td>
</tr>
<tr>
<td>A.2.2 Expressive LDAP Data Store</td>
<td>177</td>
</tr>
<tr>
<td>A.3 Java</td>
<td>177</td>
</tr>
<tr>
<td>A.3.1 LDAP Service</td>
<td>177</td>
</tr>
<tr>
<td>A.3.2 LDAP Conduit</td>
<td>178</td>
</tr>
<tr>
<td>A Numerical Experiment Data</td>
<td>187</td>
</tr>
<tr>
<td>B.1 Memory</td>
<td>187</td>
</tr>
<tr>
<td>B.2 CPU</td>
<td>189</td>
</tr>
<tr>
<td>B.3 Workload Characteristics</td>
<td>191</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Interaction Modelling and Emulation ........................................ 9
4.1 Conceptual Representation of Enterprise Software Environments .... 39
4.2 High-Level Depiction of Endpoints in an Enterprise Software Environment 46
4.3 Individual System in an Enterprise Environment ........................... 47
4.4 Closure Environment for an Open Software System ......................... 49
5.1 Emulation ............................................................................. 54
5.2 Interaction Modelling Framework ............................................. 59
7.1 Expressive Protocol Specification for an LDAP Directory Server ...... 93
7.2 Expressive Protocol Specification of the BitTorrent Peer Wire Protocol . 95
8.1 Architecture for an Enterprise Software Environment Emulator ...... 115
8.2 LDAP Conduit Example ......................................................... 117
10.1 Peak Memory - Network Interface .......................................... 142
10.2 Peak Memory - Engine Log-Log Scale ...................................... 143
10.3 Peak Memory - Engine .......................................................... 144
10.4 Memory Use - 10,000 Expressive LDAP Servers ......................... 145
10.5 Summary - CPU Use ............................................................ 147
10.6 Engine CPU Use (sec) : Network Traffic (bytes) ......................... 148
10.7 Summary - Average Time (ms) to Complete Total Workload .......... 149
10.8 Median Time (ms) to Complete Workload Tasks - 1000 Emulated Endpoints 150
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Capabilities of Existing Approaches and Tools Providing Interactive Representations of Distributed Software Environments</td>
<td>25</td>
</tr>
<tr>
<td>6.1</td>
<td>LDAP Directory Server Protocol Transition Table</td>
<td>74</td>
</tr>
<tr>
<td>6.2</td>
<td>Sample Choice Resolution Map for an LDAP Server</td>
<td>85</td>
</tr>
<tr>
<td>B.1</td>
<td>Peak Memory - Network Interface - Rudimentary Endpoints</td>
<td>187</td>
</tr>
<tr>
<td>B.2</td>
<td>Peak Memory - Network Interface - Expressive Endpoints</td>
<td>188</td>
</tr>
<tr>
<td>B.3</td>
<td>Peak Memory - Engine - Rudimentary Endpoints</td>
<td>188</td>
</tr>
<tr>
<td>B.4</td>
<td>Peak Memory - Engine - Expressive Endpoints</td>
<td>188</td>
</tr>
<tr>
<td>B.5</td>
<td>CPU Use - Engine - Rudimentary Endpoints</td>
<td>189</td>
</tr>
<tr>
<td>B.6</td>
<td>CPU Use - Engine - Rudimentary Endpoints</td>
<td>189</td>
</tr>
<tr>
<td>B.7</td>
<td>CPU Use : Processed Network Traffic - Rudimentary Endpoints</td>
<td>190</td>
</tr>
<tr>
<td>B.8</td>
<td>CPU Use : Processed Network Traffic - Expressive Endpoints</td>
<td>190</td>
</tr>
<tr>
<td>B.9</td>
<td>Average Time (ms) to Complete Total Workload - Rudimentary Endpoints</td>
<td>191</td>
</tr>
<tr>
<td>B.10</td>
<td>Average Time (ms) to Complete Total Workload - Expressive Endpoints</td>
<td>191</td>
</tr>
<tr>
<td>B.11</td>
<td>Median Time to Complete Workload Tasks - 1000 Emulated Endpoints</td>
<td>192</td>
</tr>
</tbody>
</table>
List of Definitions

3.1 Functions on Sequences ............................................ 36
4.1 Operations on Hosts ............................................... 40
4.2 Operations on Endpoints ........................................... 42
4.3 Operations on Channels .......................................... 44
4.4 Operations on Software Systems ............................... 48
4.5 Closure Environments of Open Software Systems .......... 49
4.6 Dynamic Endpoints and Systems .............................. 50
4.7 Closure Environments of Dynamic Software Systems .... 50
5.1 Messages ............................................................ 60
5.2 Operations on Interaction Modelled Endpoints .......... 61
5.3 Protocol Model Operations ...................................... 62
5.4 Transitions for Interaction Modelled Endpoints .......... 64
5.5 Data Stores ....................................................... 65
5.6 Behaviour of Modelled Server-Class Endpoints .......... 66
6.1 DFA based Protocol Model ...................................... 69
6.2 Rudimentary Protocol Model - Next ......................... 70
6.3 LDAP Message Shapes ........................................... 72
6.4 LDAP Directory States Specification ......................... 72
6.5 Placeholder Values ............................................... 76
6.6 Request Offset Based Placeholder Value ................. 76
6.7 Resolve Request Offset Based Placeholder Value ......... 77
6.8 Message Construction from a Skeleton ..................... 78
6.9 Sample Value Sequence for an LDAP Bind Request ...... 79
6.10 Sample Value Sequence for an LDAP Bind Response Skeleton 80
6.11 Sample Value Sequence for an LDAP Bind Response .... 80
6.12 Rudimentary Handle Request Operation .................. 81
6.13 Rudimentary Data Store ........................................ 84
6.14 Skeleton Sequence Operation .................................. 85
7.1 Expressive Protocol Model - Abstract Syntax ............ 90
7.2 Bound and Free Variables in Expressive Protocol Model Specifications 97
7.3 Closed Expressive Protocol Model Specifications ....... 98
7.4 Expressive Protocol Model Context .......................... 99
7.5 Evaluate-able Expressive Protocol Model .................. 100
7.6 Expressive Protocol Model Evaluation ....................... 102
7.7 Next for an Expressive Protocol Model ........................................ 104
7.8 Shape Specific Handlers ...................................................... 106
7.9 LDAP Add Request Handler .................................................. 107
7.10 Expressive Handle Request Operation ................................. 109
7.11 Expressive Data Store ...................................................... 111
7.12 LDAP Directory Expressive Data Store ............................... 112
Chapter 1

Introduction

Interaction modelling and emulation of enterprise software systems is an approach to providing operating environments, for such systems, which approximate both the behaviour and properties of the environments in which the system is expected to operate once deployed. Such emulation environments may be used by quality assurance teams to assess the run-time characteristics of enterprise software systems; the system of interest being observed and measured whilst it interacts with environments that exhibit the approximate behaviour and properties of those anticipated in deployment. The results of such investigations can provide developers with an improved understanding of the run-time properties of their software and is likely to suggest areas for improvement. Non-technical members of an enterprise software development organisation may also find use for the results of such investigations. Sales teams, for instance, may be able to make stronger assurances regarding performance when deployed into a potential client’s specific environment. Advertising departments may be able to leverage positive results to improve the position of the software within the market. The direct and indirect usage scenarios we have mentioned, as well as those which may be conceived in the future, motivate the work described in this thesis.

This chapter introduces enterprise software and the environments in which it operates. We highlight the key characteristics of these operating environments and the necessity of having some interactive stand-in to be able to analyse the run-time properties of (enterprise) software systems which operate in such environments. We touch on the popular tools and approaches currently used in practise to provide these operating environment analogues calling attention to their strengths and limitations. This survey shows that there does not yet exist any tool or approach which is both highly-scalable; capable of representing many-thousands of individual software systems using a single physical host, and well-suited to the representation of complex server-class interactions. We outline interaction modelling and emulation as an approach to providing interactive representations of enterprise software operating environments which is potentially both highly-scalable and well-suited to exhibiting complex server-class interactions. The chapter is concluded with a description of the structure and content of the rest of the dissertation taking care to highlight the location of key contributions.
1.1 Context

Enterprise software systems can be defined, most freely, as software systems which exist in order to address some enterprise problem. Typically, enterprise problems arise out of the need to conduct essential business activities, possibly straightforward in small organisations, in larger scale corporations. Tasks such as identity management, supply chain management, customer relations and asset management can be substantially more difficult when a business has a complex hierarchy of personnel, organised into various departments, and physically located at different sites across the globe. Enterprise software systems are intended, and actively used, to help manage these kinds of critical business activities for large enterprises. Large businesses rely on these enterprise software systems to help manage critical aspects of day-to-day activity, supporting operations which are of particularly high business value. A point evidenced by the amount of money organisations are willing to spend on this software; sales and service contracts measured in the millions-of-dollars being, not the exception, but the rule.

The typical characteristics of enterprise software system operating environments is due largely to the kinds of problems they are intended to solve and the businesses likely to have such problems: large, potentially multi-national, organisations with complex internal structures, and complex external relationships with a multitude of suppliers and customers. Such organisations tend to have software operating environments which to a certain extent mimic their own structure: large-scale, distributed, heterogeneous environments. Tens-of-thousands of individual semi-independent software systems may be operating simultaneously in this environment, being run on a variety of different physical hosts scattered across the globe.

Enterprise software systems then, in order to fulfil their particular role within an enterprise’s business operations, must operate within this complex heterogeneous, large-scale, distributed environment. This means interacting with the other software systems in the environment. Accepting requests for communication from other systems and establishing channels with others as necessary. Receiving, processing and transmitting messages over established channels, all the while taking care to interact in accordance with whichever of the many application-layer protocols, typically used for communication between enterprise software systems, happens to be in place.

Assessing the run-time properties of enterprise software systems, such as its scalability and performance, remains a significant engineering challenge. Such assessments can be considered a priority given the often critical nature of an enterprise software system’s function as well as the substantial costs involved in their sale and ongoing service. For credibility, this assessment is best undertaken by observing and measuring the relevant run-time properties of an enterprise software system as it operates in a range of interactive environments exhibiting behaviour and characteristics representative of those encountered in production deployments. The interactive representation of the operating environment must allow the software under investigation to communicate with it at run-time in the same manner that it would a real environment and must perform these communications in real(istic)-time. While the software under investigation operates, interacting with the
stand-in environment in the manner it would a real environment, measurements can be made which help assess run-time properties such as, for example, the software’s scalability and performance when operating in different environment configurations. It is crucial that the environment has characteristics that are representative of a real enterprise environment. In particular, it must be of sufficient scale in order to ascertain whether the software can operate in environments containing thousands of individual semi-independent software systems. Currently, the production of such environments, with behaviour and characteristics representative of those typically encountered upon deployment, is both non-trivial and inadequately supported by existing tools and approaches.

Concrete Industry Scenario

The preceding few paragraphs introduce the context of this work at a somewhat abstract level. We now outline a concrete industry scenario helping clarify abstract concepts through example and also to provide a practical grounding for the work. The scenario outlined here forms the basis of the case study described in Chapter [9].

CA Identity Manager (IM) [Gar06] is an enterprise-grade identity management suite which helps large organisations manage employee identity and access permissions on the various computer systems present in the enterprise’s environment. It is typically purchased by large corporations, such as banks and telecommunications companies, who use it to manage the digital identities of personnel and control access of their vast and distributed computational resources. IM provides a central authoritative location where the identities and permissions of users can be specified by an organisation. These identities and permissions are then applied to the environment by IM interacting with effected resources using whichever application-layer protocol is supported by each resource.

Although IM is amenable to conventional functional testing techniques, non-functional testing is somewhat more challenging due to the scale and complexity of typical production environments. Performance modelling and simulation techniques can predict IM’s run-time properties in production environments. The most credible assessment, however, is obtained by observing and measuring IM as a real executable operating in environments with properties representative of those encountered in production. This approach provides primary quantitative data on IM’s actual behaviour in these environments, increasing confidence in the results of non-functional quality assessments. Unfortunately, the scale of typical IM production environments raises some difficult practical challenges. Physically replicating production environments is usually prohibitively expensive and system-level virtualisation, although somewhat scalable, requires several server clusters to represent enterprise-scale production environments.
The IM quality assurance team needs some way to produce interactive test-bed environments which exhibit behaviour and characteristics representative of typical production conditions. This test-bed must allow IM to operate at run-time in the same manner that it would in a real production environment, communicating in the same way and in real(istic)-time. While IM operates in this test-bed, the quality assurance team can observe and measure IM’s run-time properties, ultimately using this data to assess IM’s non-functional properties; such as it’s scalability and performance when operating in a range of different production-like environments.

1.2 Interactive Representations of Enterprise Software Operating Environments

Assessing the run-time properties of an enterprise software system in a customer’s environment is generally unacceptable. Customers typically want some estimates regarding these properties prior to purchase, as will the developers of the software and other assorted members of the development organisation. Furthermore, faults in the software in question may cause irreversible damage to the customer’s environment and consequently harm their day-to-day operations. Alternative environments for assessing the run-time properties of enterprise software system must be sought.

There are existing tools and approaches used in practise to produce environments in which it is possible to assess the run-time properties of enterprise software systems. The popular existing tools and approaches include physical replication, hand coded and framework assisted fake environments [Gib87, MFC01, FMPW04], performance testing and load generation tools [Mic06, Loa, Fou] as well as system-level virtualisation [SVL01, Wat08]. Each of these tools and approaches are well-suited in certain situations, but each has its limitations and scenarios for which it is ill-suited.

Physical replication may be considered the most straightforward approach to providing interactive representations of distributed software environments. The environment in which a piece of software is intended to operate in is manually constructed and configured, from the hardware up. The system under investigation may then be placed into the replicated environment and its operation within that environment observed and measured.

Physical replication can be suitable when the software under investigation is expected to operate in environments of modest scale and complexity, but becomes unwieldy at medium and larger scales. Small operating environments are relatively straightforward to manually construct and configure. The behaviour and characteristics of such environments can be a very accurate reflection of those encountered in production deployment and therefore the results of analysis and studies undertaken in such an environment can be highly accurate. However, the creation and configuration of these physical replicas is generally a tedious and time consuming activity. The environments are rigid; single environment, single configuration and do not lend themselves to fine-grained control of the behaviour making it difficult to stage scenarios which may be of particular interest.
Some of the systems required in an enterprise environment may be legacy and therefore no longer available for purchase, or may be mainframe machines too expensive to justify purchase. Most significantly, the time and cost resources required for this approach make it prohibitively expensive to scale, especially considering that each system in the environment may require its own physical machine.

The natural instinct of many software developers, when faced with a problem, is to address it through programming. To enable assessment of an enterprise software system’s run-time properties, the developers of the software in question will sometimes produce small stand-alone programs or scripts which approximate the behaviour and appearance of the environment. Programmatic approaches can also be more tightly integrated with the software under development. Mock objects [FMPW04] and the associated frameworks for instance, are commonly used to provide behaviour of components or systems, necessary to enable testing of the software under development but is either unavailable or, for some reason using a real component or system is undesirable. We dub these and other approaches which explicitly involve programming or scripting the programmatic approaches.

The programmatic approaches provide a flexible means to producing interactive representations of enterprise software environments, they are not however, necessarily scalable nor reusable across multiple software development projects. A stand-alone program or script acting as an enterprise software environment can be defined to whichever level of accuracy is desired. The scalability of such a script or program is, however, dependent upon the skill and intentions of its developer. Furthermore, the stand-alone script may define behaviours for systems which would also be useful in assessment of run-time properties of other software developed by the same organisation. Re-use of this behaviour in other such projects is not necessarily straightforward, if possible at all, leading to duplicated and therefore wasted human effort in an organisation. Integrated techniques are not always feasible, typically requiring the source code of the system under investigation. Nor are they necessarily desirable as the instrumentation of the system in question may inadvertently affect behaviour decreasing accuracy of assessment results.

Alternatively, a developer may turn to some existing tool or technology to provide the requisite interactive environment. There are a number of performance testing and load generation tools which are used in practise to evaluate the performance and scalability of server software. Tools such as SLAMD Distributed Load Generation Engine [Mic06], HP’s LoadRunner [Loa] and the Apache Software Foundation’s JMeter [Fou] are capable of representing many thousands of concurrent clients, transmitting requests to some server system under investigation. The more advanced tools are capable of providing sophisticated performance diagnosis of the system under investigation, identifying the most critical bottlenecks in the system.

Performance and load generation tools are geared towards generating scalable autonomous client load against some reactive server system, measuring the performance of that reactive system under the different client loads. They are not, however, as well-suited to representing environments containing many reactive software systems, allow-
ing the performance of some client, i.e. active system to be measured while interacting its environment. Environments produced by performance and load generation tools are focused on systems with active, client type, interaction patterns. Active systems initiate interaction, first establishing a connection with the system under investigation and subsequently issuing a series of requests of that system. Representing the complex interactions exhibited by reactive (server) systems, i.e. systems which wait for connections to be established and subsequently respond to requests, is inadequately facilitated by existing performance and load generation tools.

System-level virtual machine technology such as VMWare Workstation [SVL01] and VirtualBox [Wat08] are commonly used in industry to provide interactive representations of enterprise software operating environments. Each virtual machine image constitutes a fully executable representation of a real physical machine. This model (image), along with others, is executed by a virtual machine host.

The system-level virtual machine approach is essentially superseding physical replication, overcoming some but not all of the limitations of physical replicas, whilst managing to maintain an interaction behaviour which is close or even equivalent to that of a real enterprise environment. Once properly configured, virtual machines are far easier to manage than their physical counterparts. By taking regular snapshots of the virtual machine state, a previous state can be easily recovered should anything go wrong. Furthermore, system-level virtual machines are easier to scale than their physical counterparts. A single (high-end) host is able to execute around 12 or so virtual machines simultaneously. Unfortunately this degree of scalability is insufficient. Larger scale enterprise environments consist of thousands or even tens of thousands of individual machines, which would require hundreds or thousands of expensive virtual machine hosts. This situation may not significantly improve seeing as virtual machine models constitute fully executable abstractions of physical machines themselves, the typical resources of which will likely increase with time possibly eating up performance gains achieved in the hardware.

Although multiple tools and approaches exist which can aid the production of environments in which the run-time properties of enterprise software systems may be assessed, there is not currently any which is highly-scalable, well-suited to the representation of complex server-class interactions and, decoupled from any implementation details of the system under investigation. By highly-scalable we mean that the tool or approach is capable of representing up to ten-thousand individual enterprise software systems using a single (or a few) physical host(s) and can therefore represent large scale enterprise environments using one (or limited number) of physical machine(s). To this end, physical replication and system-level virtual machines cannot scale sufficiently. Performance testing and load generation tools are not well-suited to representation of complex server-class interactions. The scalability of stand-alone programmatic approaches is not assured and they do not provide any structured of automated means to re-use aspects of their behaviour across projects. Integrated programmatic approaches, while well-suited to enabling unit-testing of enterprise software systems, are typically coupled to the imple-
mentation of the system under investigation itself and therefore are less well-suited to the system-level run-time assessment we are interested in enabling. Being able to produce environments which appear to be and act similarly to large scale enterprise environments containing server-class systems, decoupled from any implementation details of particular enterprise software system, better enables opaque(black)-box assessment of run-time properties of enterprise software systems. It allows enterprise software systems to be observed and measured as they interact with environments having similar behaviour and characteristics of real deployment environments.

### 1.3 Interaction Modelling and Emulation

Interaction modelling and emulation is the approach we propose to provide interactive representations of enterprise environments, enabling the assessment of enterprise software run-time properties. The approach consists of two key elements: (i) interaction models which describe the interactive behaviour of individual enterprise software systems and (ii) an emulator which provides a means to “execute” or “interpret” these models and thereby interact with a real enterprise software system under investigation.

![Figure 1.1: Interaction Modelling and Emulation](image-url)

**Figure 1.1: Interaction Modelling and Emulation**
The approach is presented diagrammatically in Figure 1.1. On the left we depict an enterprise software system whose run-time properties we would like to assess, shaded in grey, along with a possible deployment environment for that software system. The deployment environment contains three separate enterprise software systems with which the enterprise software system in question interacts in order to carry out its duties. It should be noted that this deployment environment containing only three individual systems does not depict the typical scale of environments our approach targets. We are more interested in environments containing thousands of individual systems with which the system under investigation (grey) will interact. Nevertheless the deployment environment depicted will suffice for illustration of the general ideas behind the interaction modelling and emulation approach. As already stated, the software system we wish to investigate interacts with the systems in its deployment environment, this is illustrated through the double-ended solid line arrows connecting the deployment environment systems and the other. Our approach focuses on these interactions, taking into consideration the messages being transmitted by both parties, and attempts to specify this behaviour using an interaction model. The interaction model contains information such as the valid temporal ordering of messages as well as means to process and generate message content, all defined from the perspective of the enterprise software system in the deployment environment. These interaction models are then fed into an emulator that is capable of simultaneously emulating (executing/interpreting) many such models. The enterprise software system whose run-time properties we set out to assess may then interact with the emulator in the same manner it would its deployment environment, unaware that it is communicating with an emulation environment rather than the deployment environment itself. While the enterprise software system interacts with the emulator the run-time properties of interest can be observed and measured, thereby enabling assessment of such properties.

Interaction modelling and emulation is an approach to providing interactive approximations of enterprise software operating environments which, we expect, is capable of addressing some of the key limitations of existing tools and approaches. Specifically, we hypothesise that this approach will be highly-scalable, well-suited to the representation of complex server-class interactions and decoupled from the implementation details of any particular system under investigation. We justify the scalability hypothesis though the observation that the interaction models we intend to use to define enterprise software system interactions will be targeted to be as light-weight as possible. Meaning that the computation resources required for their emulation (execution/interpretation) shall be as modest as is possible.
We expect this light-weight property will ensure that a single emulator, being executed on a single physical host, can simultaneously emulate thousands of individual model specifications and therefore can be branded, highly-scalable. Whilst developing the interaction models we will explicitly target server type systems to ensure the approach is well-suited to the representation of such complex behaviours. Finally, the interaction modelling and emulation approach explicitly treats the enterprise software system under assessment as an opaque(black)-box and as such is decoupled from implementation details of the external system under investigation.

1.4 Contributions

This dissertation demonstrates that:

Interaction modelling and emulation is an approach to producing interactive approximations of enterprise software environments capable of acting as and appearing to be real environments, enabling observation and measurement of the run-time properties of enterprise software systems. This approach is highly-scalable, well-suited to representing server-class interactions, and decoupled from implementation details of systems under investigation.

More specifically, and in the order they appear, this thesis contributes:

- An “Interaction Modelling Framework” described in Chapter\textsuperscript{5} which provides a foundational set of abstractions and a structure upon which a variety of emulate-able interaction models can be defined which utilise different levels of expressive power. By enabling different levels of expressive power this framework is able to facilitate different interaction models, each of which may be suitable in different practical scenarios.

- A “Rudimentary Model Stack” described in Chapter\textsuperscript{6} which demonstrates both, the a use of the interaction modelling framework, and also provides a simple model of interaction based on protocol conformity and output message skeletons. This model stack is essentially a refined and more detailed version of some of our previously published work [HSHV09].

- The protocol model of the expressive model stack is worth highlighting. This protocol model, appearing in Chapter\textsuperscript{6} Section\textsuperscript{6.1} introduces support for dynamic extension and contraction of protocols, as described in our previously published work [HSHV10]. By supporting dynamic extension and contraction this protocol model is able to model interaction patterns that exhibit subservient parallelism, a pattern often exhibited by enterprise software protocols.
1.5 Outline

An “Expressive Model Stack” is described in Chapter 7 which demonstrates the flexibility of the interaction modelling framework by defining a more powerful interaction modelling stack. This expressive stack overcomes some of the descriptive limitations of the rudimentary and is based on definition and invocation of specific request handlers. It also introduces the notion of data store persistency so that modification requests can be persisted for the duration of an emulation.

We provide an architecture and proof of concept implementation for an enterprise software environment emulator in Chapter 8. This architecture provides guidance for others who would like to build such an emulator. The emulator itself provides a concrete example of how the architecture can be applied through describing a real implementation.

A case study presented Chapter 9 provides the more practically inclined reader exactly how given a real scenario and an enterprise software environment emulator, one can produce an environment which enables observation and analysis of an enterprise software system through interaction modelling and emulation. This also demonstrates the practical utility of the approach showing that an emulation of a real enterprise software system’s environment can in fact interact with that software system and thereby enable its observation and measurement.

An investigation of the potential scalability of emulation is provided in Chapter 10. This shows that a single physical host can simultaneously emulate up to and in excess of ten-thousand software systems. We also show that the rudimentary model stack is, as expected, more computationally expensive to emulate than the expressive, but that this difference is not excessive and the choice of model stack is therefore more dependent on the requirements of the scenario and the amount of modelling effort that one is willing to invest.

In this chapter we have introduced enterprise software systems and environments as well as the key issue of producing interactive environments which allow assessment of the run-time properties of enterprise software system. We summarised the popular approaches to providing such environments highlighting the situations where they are appropriate and the situations in which they are inappropriate. We introduced interaction modelling and emulation as an approach to producing these assessment environments which we expect to be highly-scalable, well-suited to the representation of complex server-class interactions and decoupled from the implementation details of any particular system under investigation. Finally, we have introduced the main contributions made in this work and now provide an outline of its structure and content.
The remainder of this thesis is organised as follows: Related work is covered next in Chapter 2, which includes a more thorough analysis of the existing approaches to producing interactive assessment environments, as well as covering the major issues in testing distributed software systems, and the work on interaction modelling and specification in general. This is followed by a treatment of the mathematical preliminaries in Chapter 3, where the formal symbols and notations used throughout the thesis are described. A more rigorous treatment of enterprise software environments is provided in Chapter 4, providing a more complete and somewhat formal definition of the problem we are investigating. This is followed by a detailed account of the interaction modelling and emulation approach in Chapter 5, in which we also present the modelling framework on which our two interaction stacks are based. The rudimentary stack follows in Chapter 6. This rudimentary stack is a simple interaction model which is similarly simple in specification and in emulation. A consequence of this simplicity is that the rudimentary stack is not powerful enough to express all the interactions which we would like to emulate. In particular, it cannot express asynchronicity in protocols, nor persistency in modifications. To overcome these limitations we introduce a more powerful interaction model in Chapter 7, which is dubbed, the expressive interaction stack and is capable of modelling asynchronicity as well as data persistency. We follow the model stacks with an architecture and implementation of an enterprise software environment emulator in Chapter 8. A case study is provided in Chapter 9, demonstrating how given a real scenario and an enterprise software environment emulator, one may produce an environment that enables observation and measurement of an enterprise software system through interaction modelling and emulation. We follow this with experiments in Chapter 10, which investigate the scalability of interaction modelling and emulation as well as how each model stack scales in comparison with the other. We conclude in Chapter 11, presenting a summary of the main contributions as well as detailed treatment of the future work.
Chapter 2

Related Work

Emulating enterprise software systems is, to the best of our knowledge, a new research topic in computer science and software engineering. This novelty, although satisfying, poses some additional challenges for our work. Convincing fellow researchers that the problem exists at all is one such challenge. Carrying out and producing a respectable literature review is another. Without any directly relevant body of work it is difficult to find a point at which to start, and equally difficult to know when it is safe to stop. The approach we have adopted is to review literature from range of different areas, selecting work from each which appears most relevant considering our topic. We also investigate work conducted outside of academia. A number of tools exist capable of achieving some of our goals and are reviewed as part of this chapter.

This chapter begins with discussion on the state of the art in testing distributed software systems in Section 2.1. This is followed, in Section 2.2, by a detailed treatment of the existing tools and approaches to providing interactive representations of distributed software environments. Some of these approaches, like our own, are intended to enable testing activities, while others are intended to address entirely different needs. Modelling and specifying the interactions of enterprise software systems is a key element of our approach and is investigated in Section 2.3. The chapter is concluded by Section 2.4 which summarises its main points.
2.1 Testing Distributed Software Systems

Although our work is not explicitly on conducting testing of distributed software systems, enabling assessment of run-time properties of such software systems is a fundamental motivating factor. As such it is reasonable to review the existing work on testing distributed software systems so that we are aware of the major issues in the area and can be sure our work will be of value and positioned appropriately.

In the past dozen years, two noteworthy articles describing the current status and future direction for research in software testing [Har00, Ber07] have been published. Both articles raise issues which motivate and suggest the necessity of our own work. Mary Jean Harrold provides in [Har00] a “roadmap” from the status of testing, methods, tools and processed in 2000, and the fundamental research topics which have the potential to lead to practical testing methods, tools and processes for development of high-quality software. These fundamental research topics cover issues such as testing based on pre-code artefacts, testing evolving software, using testing artefacts, and most relevantly to our problem, testing (distributed) component-based systems. With respect to the issue of testing these component-based systems, Harrold highlights two perspectives, that of the component-provider and, the component-user. Our work can be seen to take the component-provider perspective but with a focus on run-time (extra-functional) properties and therefore, unlike Harrold’s characterisation, we do not assume nor require access to the underlying source code of the component under test.

More recently (2007), Antonia Bertolino developed a roadmap [Ber07] also covering the testing and quality assurance domain. Bertolino introduces 6 faces of software testing to help classify testing activities: why, how, how much, what, where and when. The roadmap presented consists of the current achievements, ultimate dreams, and key research challenges which map out a potential research path from these achievements towards realisation, to a limited extent, of the stated dreams. Considering the perspectives mentioned by Bertolino, the problem we investigate is most closely associated with the where of testing: providing an interactive representation of large scale, distributed environments where the run-time properties of a software system can be observed (tested). Considering the research challenges outlined by Bertolino, the problem we investigate is most closely associated with coherent testing of functional and, in particular, extra-functional properties: an interactive representation of an enterprise environment enables observation of these extra-functional properties of enterprise software systems and thereby help to enable testing of such properties.
There are a couple of papers which focus on specific issues in testing distributed software systems \[GM99\] \[CP06\]. Ghosh and Mathur \[GM99\] provided the earlier work in 1999, presenting a broad outline of the issues involved in testing such systems, nine in total:

1. Scalability of test adequacy activity
2. Redundant testing during integration of components
3. Availability of source code
4. Heterogeneity of language, platforms and architectures
5. Monitoring and control mechanisms in distributed software testing
6. Reproducibility of events
7. Deadlocks and race conditions
8. Testing for system scalability and performance
9. Testing for fault tolerance

Three of these issues in particular, are closely associated with the problem we investigate:

**Availability of source code:** The interactive distributed environment we investigate treats external systems, i.e. those under test, as a black-box and thereby does not require access to the source code.

**Heterogeneity of language, platforms and architectures:** Distributed enterprise software systems interact with one another using a variety of different application-layer protocols and are implemented in a variety to different languages.

**Testing for system scalability and performance:** These are precisely the kinds of run-time properties which we would like to enable observation and measurement of.

In 2006, Canfora and Di Penta presented similar issues with their paper on testing service-centric systems \[CP06\]. Their focus was on service oriented systems, examined from a range of different testing perspectives and levels. The perspectives considered were that of the service developers, providers, and integrators, the third-party certifiers, as well as the users. They considered service testing at levels: functional, non-functional, integration and regression. The testing perspective and level to which our problem is most closely related is that of the service integrator and service integration testing. An interactive enterprise environment mimics the behaviour of certain real services that are part of a composition but are either unavailable or undesirable to be used directly in integration testing. An emulated environment could enable observation of services of a composition, even when not all of the services of the composition are available.
There are a few approaches to testing distributed software systems which are related to our problem area. Mike Barnett and Wolfram Schulte while working at Microsoft Research in 2001 [BS01a] describe a run-time verification technique for COM components based on concurrently executing a component under test along with an executable specification representing its interface. The specifications are defined using the Abstract State Machine Language [GRS05] and the run-time communications of the component under test are compared against this specification as both are executed. Unfortunately, these executable specifications are not used as the basis for the communication itself, just as a means of tracking and verification.

Model checking techniques have been used extensively to statically verify the properties of software systems. Work by Sylvain Hallé and Roger Villemaire [HV08] looks to extend the application of model checking techniques into the dynamic domain, enabling run-time monitoring and analysis. They define a linear temporal logic capable of expressing data dependencies between messages in XML traces and an algorithm capable of checking conformance of systems efficiently at run-time. Their aim is to model the properties and constraints of software systems and then monitor the software system to ensure these properties and constraints are met at run-time. Unfortunately, like the work by Barnett and Schulte, the models are not used as the basis for the communication itself, just as a means of tracking and verification.

More recently, Immo Grabe, Marcel Kyas, Martin Steffen and Arild B. Torjusen have used executable interaction specifications as a foundation for a formal approach to black-box testing interface conformance of asynchronous Creol components [GKST10]. This approach allows checking of a components conformance to typing, scoping and communication pattern rules, as well as conducting the testing activity itself. In a nutshell, this work can be thought of as property based testing with mock objects. This work however, focuses on functional testing, generating input for a component under test and checking its output for conformance to the specification. This is different from our problem where an interactive representation of a component under test’s environment is required so that the component’s run-time properties may be observed.

Testing distributed software systems is a challenging research area under active development. In particular, there is work to be done on tools and approaches which facilitate assessment of extra(or non)-functional, i.e. run-time properties, of distributed software systems: a fundamental driving force for our work. There are existing model based approaches for testing of distributed software systems. To date, however, these approaches have targeted functional verification and cannot be directly applied to verification of extra-functional, or run-time properties of distributed software systems.
2.2 Interactive Representations of Distributed Software Environments

There are some existing tools and approaches which provide interactive representations of distributed software environments. By interactive we mean that the environment interacts with real external software systems directly, in the manner native to those systems, and in realistic-time. Each existing tool and approach is, in general, well suited to some subset of the scenarios in which such an environment is required. The approaches we have come across through discussion with industry practitioners and investigation of the literature include: (i) physical replication of the deployment environment, (ii) ad-hoc and structured code based approaches, (iii) performance and load generation tools, (iv) system-level virtualisation and (v) network emulation. The remainder of this section describes each approach surveying the corresponding literature where available.

2.2.1 Physical Replication

Physical replication involves construction and configuring real distributed software environments from the ground up. This includes setting up networking infrastructure, procuring host machines and initialising them with the appropriate operating systems and service software. Enterprise software systems can be deployed into these physical replica environments and, whilst operating, have their run-time properties observed and measured.

The financial and human resources necessary to produce physical replicas of enterprise environments limits its appeal. However, if the software under investigation is expected to operate in environments of relatively modest scale and complexity, then physical replication can be the best option. The behavioural accuracy of physical replication is the highest possible without resorting to using live client environments. High behavioural accuracy increases confidence in results obtained from experiments conducted in these environments; QA teams have a higher level of certainty that their results will reflect customer reality.

In many situations however, particularly in the enterprise context, physical replication is not viable. The scale and complexity typical of these environments means physical reproduction is prohibitively expensive much of the time. An additional concern is the rigidity of the resulting environments. Altering a physical environment so that it reflects some other deployment scenario can require substantial human effort.
2.2.2 Programmatic

Programmatic approaches can be used to produce interactive representations of distributed software environments. In this approach the environment’s behaviour is programmed by humans using general purpose, or domain specific, languages. Programmatic approaches can be further classified as those which stand-alone and those which are integrated. Stand-alone approaches are isolated from the implementation (code) of systems under test. Integrated approaches, on the other hand, are typically part of the same code base as the systems under test, and consequently, may be inoperable in other contexts.

Stand-Alone

The standalone programmatic approach involves part of the development and/or quality assurance team developing some program or script which provides an interactive representation of the operating environment for the software under investigation. This approach affords a substantial degree of flexibility. The developers of the standalone program or script are free to specify the interaction behaviours of the precise range of environments in which they are interested and to whichever level of accuracy deemed necessary. This strategy can be pragmatic leading to an interactive representation which is especially well-suited for enabling observation of the software under investigation. Furthermore, the developers of the standalone program or script only need build the features required for enabling observation in the scenarios in which they are interested.

The flexibility and specialisation afforded by this approach means it is difficult to make concrete statements regarding the unfavourable characteristics of these environments. We can however, highlight a couple of limitations. The specialisation of standalone environments can make it difficult to reuse common aspects of the environment across development projects both within and between organisations and therefore can lead to duplicated efforts. We are thinking, in particular, of the interaction behaviour of specific system types that may be useful in other projects which also interact with systems of that type. This is not to say that code cannot be harvested from the standalone programs or scripts, only that such harvesting needs to be performed manually. Another general limitation of this approach is the potential scalability of the script or program. By adopting the standalone approach scalability is determined primarily by the skills, intentions and choices made by of its developer. High-scalability is not guaranteed.

Integrated

Programmatic approaches, as an alternative to the standalone programs or scripts, can be more tightly integrated with the software under development. Mock objects [FMPW04] and the associated frameworks, for instance, are increasingly used to provide behaviour of components or systems, necessary to enable testing of the software under development but is either unavailable, or for some reason use of a real component or system is undesirable.

The integrated programmatic approach has been widely adopted in research and par-
2.2 Interactive Representations of Distributed Software Environments

tically by industry. From as far back as 1987, *method stubs* have been used to represent some basic interaction behaviour of remote systems [Gib87]. Lately, techniques such as *fake* [MFC01] and *mock* [FMPW04] objects have become the common, perhaps even standard, means to mimic the interactive behaviour of real remote systems.

The test driven development movement, through its focus on functional unit testing, has popularised the mock object technique in particular, as an approach to providing interactive representations of distributed software environments. There are currently mocking frameworks available for practically all popular programming languages: Mockito [Mocb] for Java, rspec [rsp] for Ruby, and Mockery [Moca] for PHP to name but a few. By applying these integrated programmatic techniques developers can have representations of distributed software environments which are highly controllable and immensely useful for performing functional unit testing.

There are however, a couple of key limitations when interactive representations of distributed environments are provided through integrated programmatic approaches. Firstly, the code of integrated programmatic approaches are typically coupled with the code of the software under investigation. A mock object, for example, is generally a local object in the object-oriented programming language sense, which the software under (test) investigation interacts with in place of another object that is used in deployment that provides an interface for interaction with the real distributed system. Coupling the interactive environment with the code under investigation is only possible if the code itself is available, which is not necessarily the case. A second, and perhaps more significant, limitation of the integrated programmatic approach is the general lack of communication taking place over a computer network. The underlying communication infrastructure, even at the application-layer, between two software systems introduces issues which are simply absent in the synchronous communication occurring between synchronous object-to-object interactions. Issues which a software system operating in a distributed environment is expected to handle gracefully. Furthermore, the results of an analysis of the run-time properties of a software system intended to interact with distributed software systems, while it interacts with local code in lieu of real distributed systems, are likely to be considerably inaccurate. All tasks, establishing and terminating connections, sending and receiving messages, are more complex and can take unpredictable lengths of time longer when being carried out on a distributed computer network.
2.2.3 Performance Testing and Load Generation Tools

There are a number of performance testing and load generation tools which are used in practice to evaluate the performance and scalability of server software. Tools such as SLAMD Distributed Load Generation Engine [Mic06], HP’s LoadRunner [Loa] and the Apache Software Foundation’s JMeter [Fou] are capable of representing many thousands of concurrent clients, transmitting requests to some server system under investigation. The more advanced tools being able to provide sophisticated performance diagnosis of the system under investigation, identifying the most critical bottlenecks in the system.

Performance and load generation tools are particularly well-suited to generating a large scale autonomous active, client, or user load against some other reactive, server-class software system, measuring the performance of that system under the different active loads. They are not, however, well equipped to support scenarios where the roles of the system under test and the test bed environment are reversed. In such scenarios the test bed environment needs to contain a large number of reactive, server-class software systems to enable performance measurement for a client-class, i.e. active, software system under test. Although enterprise software systems are executed on powerful server machines, their interactions are often better characterised as client-class as they initiate and drive communications with large environments consisting predominantly of server-class systems. Consequently, existing tools of this kind are not well-suited to enabling observation and measurement of many enterprise software systems.

2.2.4 Software Performance Engineering

The field of software performance engineering (SPE) investigate distributed software system scalability and performance. Modelling is a key technique applied in this field and, to a certain extent, can be automatically extracted [BHK11]. Particularly popular are closed-world predictive models of performance, constructed early on in the development process using formalisms such as queuing networks, layered queues, and stochastic process algebras [WFP07]. Through simulation, the performance of a software system can be predicted based on different architecture, design and deployment choices available [CLGL05, LG05]. This early-cycle, design-oriented modelling is not directly useful in the context of our work. The closed-world nature of simulation means that the modelled environment is not intended, or even able, to interact with real software system executables.

Not all of modelling in the SPE field is closed-world. The SoftArch/MTE [GCL05] system, for example, is able to generate executable distributed test-beds from high-level models of the system. Similarly to the predictive early-cycle performance models, the SoftArch/MTE test-bed is intended help evaluate the performance of different architecture, design and deployment choices for a system while it is being developed. A limitation of the SoftArch/MTE work is its scalability; many physical hosts are required to represent environments with many systems. MaramaMTE [DGH+06, GHLL06] is another tool with similar capabilities. The architecture view of MaramaMTE allows specification of relationships between proposed services and a database. This information is used in
the synthesis of code which mimics proposed service behaviour in lieu of a real service implementation for testing purposes. A limitation of MaramaMTE is its lack of support for asynchronous service-oriented architectures, present in enterprise environment protocols such as LDAP and often necessary to achieve high system scalability.

### 2.2.5 System-Level Virtualisation

System-level virtual machine technology such as VMWare Workstation [SVL01] and VirtualBox [Wat08] is finding increasing use in industry as a means to providing interactive representations of distributed software environments. Each virtual machine image constitutes an abstract representation of a full physical machine. This model (image) being subsequently executed by the virtual machine host.

System-level virtual machines are essentially superseding the physical replication approach to providing interactive representations of distributed software environments. Virtual machines having numerous advantages over the physical replication approach whilst managing to maintain an interaction behaviour which is close or even equivalent to that of a real operating environment. Once properly configured, virtual machines are far easier to manage than their physical counterparts. By taking regular snapshots of the virtual machine state a previous state can be easily recovered should anything go wrong. Furthermore, system-level virtual machines out scale their physical counterparts. Anecdotal evidence suggests that a single dedicated (high-end) host is typically able to execute approximately 12 virtual machines simultaneously.

While system-level virtualisation does improve on the scalability possible using physical replication it is still somewhat limited. Representing medium to larger scale enterprise environments, consisting of thousands or even tens-of-thousands of individual machines, is still not particularly practical using system-level virtualisation. This situation may not significantly improve seeing as virtual machine models constitute fully executable abstractions of physical machines themselves, the typical resources of which will increase with time and likely eat up any performance gains made in the hardware of the physical hosts.

### 2.2.6 Network Emulation

Computer network researchers have a history of using simulation and emulation techniques to experiment with real distributed protocol implementations [ADLY95]. It is through investigation of computer networking research that we have settled on the term emulation rather than simulation. Network emulators, as opposed to simulators, are used to enable testing of software implementations under controlled and repeatable conditions which mimic those found in real deployments. As such, a network emulator is open and actually interacts with real network traffic. A simulator on the other hand is closed and does not interact with any real external network devices.
The original WAN emulator [ADLY95] (based on hitbox) was used by researchers to perform part of an evaluation of the TCP Vegas congestion control algorithm. This emulator made use of a dozen workstations and required manual configuration. Two years later Dummynet [Riz97] and ONE [ACO97] were introduced. Both systems enable the emulation of multi-hop network topologies on a single physical machine as opposed to the earlier approach which required many physical machines. Similarly, the emulation capabilities of the widely used network simulation tool NS [Fal99] allows for multi-hop emulation on a single physical machine. The key feature of the NS emulator comes from its ability to leverage the rich simulation capabilities of the underlying NS simulator. The limitation, however, is in the simulation’s scalability; the simulation scheduler may not be able to keep up with the input and output load being placed on the system. In 2002, researchers from Duke University introduced ModelNet [VYW+02] which is a direct attempt at addressing issues of scalable network emulation. ModelNet runs atop a cluster infrastructure and enables researchers to deploy unmodified distributed application implementations into large-scale network environments. Introduced in a paper the following year NIST Net [CS03] features emulation of sophisticated probabilistic performance dynamics at high levels of system load whilst using only a single commodity class Linux machine.

Although the work in computer network emulation provides a useful guide for our work, especially when it comes to terminology, it is not directly applicable to our problem area. Network emulators operate at a lower level of abstraction than that which we are interested. Their focus is on modelling and emulating the underlying communication infrastructure of networks. Our problem however, is focused on the interactions at the service or application-layer of the OSI reference model [Zim80]. The difference in level of abstraction means that some of the main issues are quite different. In particular, the actual payload of packet content handled by network emulators does not particularly matter, so long as it is consistent, i.e. has a valid checksum. At the application-layer however, more care is typically necessary. The message identifier of a response for instance, may need to match the message identifier of the corresponding request.
2.2.7 Overview

We now provide an overview of the investigated approaches and tools for producing interactive representations of distributed software environments. The key capabilities of each are summarised in Table 2.1. Each row in the table summarises the capabilities of one of the approaches or tools we have considered, the capabilities of each are assessed in the following dimensions:

**Scalability** The number of interactive systems capable of being represented using a single physical host.

**Reuse** Whether and how the models of the system interactions used by an approach can be reused in other projects with scenarios that require similar environment configurations.

**Decoupled** Does the approach constitute an environment that is entirely isolated from the implementation details of the software which is intended to be observed?

**Accuracy** The degree to which the interactions of each approach are accurate with respect to the real environment on which the behaviour is based.

**Abstraction** The underlying abstraction on which the mimicked interactions are based.

**Modality** Whether the interactive representation is able to mimic behaviour of a client (active), server (passive) or both, i.e. dual.

Physical replication, the first row of Table 2.1, is capable of providing a behavioural accuracy which is approximately equivalent to that of a real environment. Scalability however, can be a problem as a physical host machine is required for every physical system present in the real environment. Additionally interaction model reuse is difficult, although possible, however it can require physically moving and manually re-configuring each machine.

The flexibility afforded by the programmatic standalone approach makes it difficult to provide general statements regarding its capabilities. Specifically, it is not possible to say for certain what potentially for scalability such an approach has as well as what the underlying base abstraction may be. The two are in fact related; a light-weight abstraction generally leads to higher potential scalability. A significant limitation of the programmatic standalone approach is the lack of automated reuse of interaction models. These models being embedded within the interactive representation itself and thereby requiring manual extraction for reuse in other situations. Although through proper design and organisational processes the tedium of this task can be mitigated, it is still preferable to have some (semi-)automated means to reuse these abstractions in alternative scenarios.

Although scalability is not generally the aim of the integrated programmatic approaches, it is capable of providing thousands of system interactions using a single physical host. This is primarily a side effect of representing each distributed system as a local piece of code and thereby not having to actually communicate with thousands or remote systems.
### Table 2.1: Capabilities of Existing Approaches and Tools Providing Interactive Representations of Distributed Software Environments

<table>
<thead>
<tr>
<th>Modality</th>
<th>Accuracy</th>
<th>Abstraction</th>
<th>Reuse</th>
<th>Decoupled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Replication</td>
<td>Manual</td>
<td>Equivalent</td>
<td>Difficult</td>
<td>√</td>
</tr>
<tr>
<td>Programmatic-Standalone</td>
<td>Approximation</td>
<td>Approximation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programmatic-Integrated</td>
<td>Approximation</td>
<td>Architecture Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance/Loading Tools</td>
<td>Approximation</td>
<td>Machine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERA</td>
<td>Equivalent</td>
<td>Approximation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPE</td>
<td>Approximation</td>
<td>Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Machines</td>
<td>Approximation</td>
<td>User/Client</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPE</td>
<td>Approximation</td>
<td>Architecture Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Emulation</td>
<td>Approximation</td>
<td>Network</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modality</th>
<th>Performance/Loading Tools</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Replication</td>
<td>√</td>
<td>One-to-One</td>
</tr>
<tr>
<td>Programmatic-Standalone</td>
<td>√</td>
<td>Thousands-to-One</td>
</tr>
<tr>
<td>Programmatic-Integrated</td>
<td>√</td>
<td>Thousands-to-One</td>
</tr>
<tr>
<td>Performance/Loading Tools</td>
<td>√</td>
<td>Thousands-to-One</td>
</tr>
<tr>
<td>ERA</td>
<td>√</td>
<td>One-to-One</td>
</tr>
<tr>
<td>SPE</td>
<td>√</td>
<td>One-to-One</td>
</tr>
<tr>
<td>Virtual Machines</td>
<td>√</td>
<td>One-to-One</td>
</tr>
<tr>
<td>SPE</td>
<td>√</td>
<td>One-to-One</td>
</tr>
<tr>
<td>Network Emulation</td>
<td>√</td>
<td>One-to-One</td>
</tr>
</tbody>
</table>
The key idea behind the integrated programmatic approach is to provide rough implementations of the interfaces of the distributed systems with which the system interacts. As such the interaction mode is passive being invoked only as required by the system under (test) investigation.

Performance and load testing tools are highly scalable allowing thousands of interactions to be hosted on a single physical system. The abstraction used, however, is generally the user or client, meaning that the interactions represented by the resulting representation are targeted towards those interactions which are generated by a human user or agent interacting with the system. As such the interaction mode of these approaches are predominantly active.

System-level virtual machines, like the physical replication approach, provide an interactive behavioural representation which is approximately equivalent to that of a real distributed software environment. The key benefits to using virtual machines for such purposes being an improved scalability and more straightforward reuse. Unfortunately the abstraction of the virtual machine is based on the physical machine itself and as such scalability improvements are only relatively modest when compared to the scalability of a performance testing tool.

Network emulation approaches provide scalability capabilities closer to that of performance testing and load generation tools, hundreds of communication infrastructure entities being emulated using a single physical host. The key limitation of the network emulator approach however, is the abstraction being emulated. Computer network researchers are interested in the underlying communication infrastructure which facilitates the interaction between higher (application) layer services. As such network emulators are inappropriate for the task of representing higher-level, application-layer interactions. Especially because lower level interactions do not have the same need to carefully consider the content of request and response message payloads, just the network headers of the packets being transmitted.

Table 2.1 highlights the key capabilities and limitations of the current approaches to providing interactive representations of distributed software environments. Specifically it highlights the lack of an approach which is simultaneously highly-scalable (thousands-to-one), supports model reuse, is decoupled from implementation details of systems under investigation, provides an approximate level of accuracy, is based on the interaction (application-)level of abstraction and is capable of server-class interaction behaviours.
2.3 Interaction Modelling and Specification

As stated earlier, interaction modelling and emulation is the approach we propose to providing interactive representations of enterprise environments thereby enabling analysis of the run-time properties of enterprise software systems. A key element of this approach concerns the interaction models and resulting specifications which, in a sense, describe the observable communication behaviour of individual enterprise software systems, thereby enabling emulation of this behaviour through execution or interpretation of the interaction specifications.

In this section we review the existing work on modelling and specifying interaction behaviour of distributed software systems. The majority of this work falls under the category of protocol modelling and specification as this is the most well-developed area of interaction modelling and specification we uncovered. We segregate the existing work into subsections covering firstly automata based models in Section 2.3.1, moving onto the process calculi family in Section 2.3.2, active objects are discussed in Section 2.3.3, while formal specification languages are considered in Section 2.3.4, finally, grammar based approaches are discussed in Section 2.3.5. A summary discussion in Section 2.3.6 brings the treatment of existing work on interaction modelling and specification to a close.

2.3.1 Automata

Automata based formalisms have long been used in the design and verification of communication protocols. Work in this field can be traced back to the late 1970’s early 1980’s where first Merlin [Mer79] and followed by Bochmann [BS80] investigated and surveyed existing work on formal methods in communication protocol design, specification and verification, all based on automata.

In 1983 Brand and Zafiropulo described a finite-state formalism capable of describing some communication protocols [BZ83]. Their intention was to create a model of protocols which could be used to reason about and ensure generally desirable properties of those protocols. In particular, they want to be able to know when a protocol is well-formed, i.e. it specifies a response only to events which can actually occur.

Late on in the 1980’s (1989), Lynch and Tuttle provided an overview of input/output automata [LT89], detailing notions of composition, fairness, problem specification and numerous proof techniques for the formalism. The key goal of Lynch and Tuttle is again in reasoning about and proving properties of complex distributed systems and algorithms.
More recently (2001) Alfaro and Henzinger introduced the *interface automata* formalism \cite{dAH01}, an optimistic extension of traditional pessimistic automata approaches to capturing component interfaces. The use of an optimistic approach to component composition in contrast to the pessimistic approaches used in \cite{LT89} is based on the observation that components “are usually designed under assumptions about the environment”. This can be contrasted with the pessimistic approach which assumes the environment is free to act as it pleases leading to the necessity of large automata capable of handling almost arbitrary input from the environment.

Modelling protocols using automata based formalisms have some significant benefits. These models are readily amenable to static analysis allowing researchers to prove that modelled protocols are free of certain undesirable properties such as deadlock and livelock. A significant benefit, from our perspective, is that automata based formalisms are that they are relatively straightforward to synthesise into scalable executions with well understood resource requirements.

Unfortunately, automata based models of interaction typically lack explicit representation of the communication channels which carry and deliver messages. Channels are instead treated as implicit and as such cannot be manipulated directly by the automata based model.

### 2.3.2 Process Calculi Based

Process calculi such as Milner’s CCS \cite{Mil89} and $\pi$-calculus \cite{Mil99} are formalisms, designed for, and adept in, modelling communicating, concurrent and distributed computing systems. In contrast to automata based formalisms, communication channels are explicitly treated by process calculi. System transience is also well supported by process calculi, particularly the $\pi$-calculus which allows for process creation and termination, as well as for the movement of channels between processes. Process calculi have also been explicitly used for protocol specification; Session types \cite{THK94, HVK98, GH99} are an extension of the $\pi$-calculus which support specification the temporal properties of potentially long running client-server interactions.

Session types are introduced in the literature primarily by Takeuchi and Honda in their papers entitled “An Interaction-Based Language and its Typing System” \cite{THK94} and more primarily “Language Primitives and Type Discipline for Structured Communication-Based Programming” \cite{HVK98}. These two papers lay the foundation for a collection of further work on session types.

Gay and Hole \cite{GH99} define a subtyping relation for session types based on the work of Takeuchi and Honda in \cite{THK94,HVK98}. Gay and Hole demonstrate how the subtyping relation can strengthen the application of session types to specification of client-server protocols by a POP3 case study. In the POP3 example, an initial server specification is extended with an additional input command. It is shown that the existence of an additional command does not break the original client which does not make use of the new command.
The work on session types has not been purely theoretical. Neubauer and Thiemann have described a Haskell encoding [NT04] and Dezani-Ciancaglini and colleagues have incorporated session types into the distributed object-oriented language: $L_{doos}$ [DCYAD05 DCMYD06]. Furthermore Hu and colleagues have worked on incorporating session types into Java [HYH08], furthering knowledge of encoding session types in object oriented languages initiated by Dezani-Ciancaglini et al.

2.3.3 Objects, Components and Service Protocols

Interaction protocols are not only present in distributed systems connected through computer networks. In object oriented programming languages (OOPLs), for instance, an object must be instantiated before it may be used in the traditional manner, which in itself can be viewed as a kind of protocol. A protocol which must be followed in order for the object entity to behave as expected. Nierstrasz investigates this issue in [Nie95] introducing a type framework based on a notion of regular types for active objects. The idea being that these active objects can help deal with the dynamic issues of concurrent OOPLs. Such as, for example, non-uniform service availability and protocol conformance. This type framework is used to check that clients of the objects use said object in a safe - protocol compliant - manner.

Yellin and Strom have consider similar issues from the component based perspective [YS97]. Unlike Nierstrasz’s type framework work, Yellin and Strom consider bi-directional protocols, where a given participant in the interaction may act as a server, client, or a hybrid.

Ten year later, Broy, Krüger and Mesinger establish a formal foundation of service orientation [BKM07] based on earlier work which developed a theory of distributed systems [BS01b]. The primary intention of the work is to describe a formal model of services based on partial behaviour. Services and components are described in terms of functions over communication histories represented with steams or timed data flows. The resulting model is capable of capturing the syntactical and semantic notions of component and service interfaces.

2.3.4 Formal Specification Languages

The focus of our modelling and specification is on being able to represent an interactions in a generic way that may be directly turned into some executable representation. To a certain extent, formal specification languages such as Estelle [BD87], Language Of Temporal Ordering Specification (LOTOS) [BB87] and Specification and Description Language (SDL) [sdl99] can provide a means of facilitating this specification and subsequent translation. These languages along with their associated tools, have found some use in the telecommunications domain aiding the design, verification and implementation of lower-level communication protocols.
Designers of enterprise system (application-layer) protocols, however, have not embraced these tools and techniques. Message ordering for protocols such as LDAP [Ser06], HTTP [FGM+99], and SNMP [HPW02], remain informal published primarily through RFCs.

### 2.3.5 Grammars

Declarative event patterns (DEPs) [WV04] allow specification of protocols which may require context-free grammars for recognition. This is the case for protocol specifications where the current state alone is not enough to define the future interaction behaviour. DEPs are used by Walker and colleagues to help implement protocols, allowing high-level description (in the form of DEPs) and annotation of actions to be taken when those patterns are matched, implemented using Aspects. A limitation of this work is that the DEPs do not fully consider the dynamic and concurrent aspects of enterprise software system interactions.

### 2.3.6 Discussion

Our review of existing work dealing with modelling and specification of distributed software system interactions uncovered a plethora of material dealing with protocol modelling and specification in general. We failed to find, however, any existing work dealing specifically with enterprise or application-layer system interactions. Furthermore, we could not locate any treatment of modelling and specification of the temporal properties of application-layer protocols used for interactions between enterprise software systems. Instead, the temporal properties of these protocols tend to be specified in Request for Comments (RFC’s) where the message payload may be defined formally (typically using ASN.1 [Ste90]) but the temporal properties are provided informally via text description. LDAP [Ser06], HTTP [FGM+99] and SNMP [HPW02], for example, are all defined in this fashion.

It is worth noting that the chief interest in the protocol modelling field is in ensuring correctness and validity of modelled protocols; i.e they hold certain desirable properties such as being free of deadlock and livelock. Consequently the aims of the models in this field are typically focused on being able to defined accurate models of a protocol design and subsequently decide whether such properties hold. Our goal on the other hand is to use the protocol models and specifications as a basis for providing interactive behaviour which approximate the specification.
Summary

This chapter has discussed the existing work we have found that relates to production of interactive representations of distributed software environments which enable observation and analysis of the run-time properties of software systems intended to operate in such environments. We investigated the current state of the art in testing distributed software systems and found strong motivation for our work: although there is significant work on testing the functional properties of distributed software systems, the current tools and approaches facilitating analysis of the extra(or non)-functional run-time properties, are somewhat lacking. We also investigated the existing tools and approaches known to be used in practise to produce interactive representations of distributed software environments enabling observation and analysis of the run-time properties of the software systems of interest. Table 2.1 highlights the key capabilities and limitations of these tools and approaches and illustrates the lack of an approach which is simultaneously highly-scalable (thousands-to-one), supports model reuse, is decoupled from implementation details of systems under investigation, provides an approximate level of accuracy, is based on the interaction (application-)level of abstraction and is capable of server-class interaction behaviours. Finally, we investigated the existing work on interaction modelling and specification uncovering a substantial amount of material concerning protocol modelling and specification. This review, however, did not reveal any existing work dealing specifically with the temporal properties of enterprise or application-layer system interactions. Instead, the temporal properties of these protocols are usually specified in RFC’s where the message payload may be defined formally, but the temporal properties are described informally through text description.
Chapter 3

Mathematical Preliminaries

Emulate-able models and specifications of enterprise software system interactions are a significant contribution made in this thesis. By emulate-able we mean to say that specifications defined in terms of the models can be executed or interpreted by the emulator and thereby forms a basis for interaction between the emulation environment and other external software systems. For rigour, precision and clarity, these models and specifications necessitate a certain level of formalisation. This chapter provides a central location detailing the formal mathematical concepts used throughout the thesis. This centralisation enables brevity when describing the interaction models, and allows the discussion in these areas to focus the models themselves, rather than become sidetracked in descriptions of the underlying mathematical concepts used in their definition. Readers of the thesis are also able to use this single chapter as a reference when mathematical concepts or conventions are encountered within the body of the work.

3.1 Symbols

Throughout this dissertation we adhere to, as much as is possible, a number of conventions in selecting and using symbols to represent different formal concepts. Where possible, we prefer to recycle symbols which have existing associations of similar conceptual meaning. Using $\Delta$ and $\delta$, for example, to represent change in our conceptual model in Section 4.3.3. In other cases we first look to use the first alphabetic character of the name of the concept. If however, this character has been previously assigned, we move onto the Greek alphabet, preferring symbols which have similar shape or phonetic to the first alphabetic character of the name of the concept. Exhausting these options we typically move onto Greek or English alphabet symbols which, to the best of our knowledge, lack existing strong associations in the software engineering and computer science domains.

Generally symbols and function names, which have straightforward relations to previously defined symbols and function names, will keep these previous symbols or functions, and be assigned a superscript or subscript which suggests the context of the symbol or function.
3.2 Sets

We use standard symbolism when dealing with sets: \( \emptyset \) for the empty set, \( \in \) for membership, \( \cup \) for union, \( \cap \) for intersection, \( \setminus \) for complements and \( \times \) for Cartesian(cross) product. We stick to the unambiguous symbols for subsets using \( \subseteq \) for subsets and \( \subsetneq \) for proper (or strict) subsets. Elements listed between curly braces: \{a, b, c\} for example, denote a set containing those elements \( a, b, \) and \( c \).

Additionally, a convention is followed when denoting a universe, sets consisting of elements of a universe and individual elements themselves throughout this thesis. In general we use a calligraphic capital character to denote the set of all elements of a set, i.e. the universe of some concept. A non-calligraphic capital is used to represent a specific set of those elements, i.e. a subset of the universe. Finally, we use plain non-capitalised characters to denote single elements of the associated universe. To illustrate the general case, suppose \( A \) (calligraphic) is used to denote the set of all elements of some conceptual entity, i.e. the universe of that concept, then \( A \) will be used to denote a set which contains only elements from that universe \( A \), i.e. \( A \subseteq A \), and \( a \) will denote a single element of the universe \( A \), i.e. \( a \in A \).

3.2.1 Power Sets

A power set of \( \mathcal{X} \), is the set of all subsets of set \( \mathcal{X} \), including the empty set. Throughout this thesis we will use the notation \( 2^\mathcal{X} \) to denote the power set of \( \mathcal{X} \). We use this notation rather than that using the \( P \) as we reserve \( P \) to refer to protocols, a fundamental concept in the interaction modelling introduced later in this dissertation.

3.2.2 Set Builder Notation

When defining sets we often use set builder notation. To provide an example, \( \{ x : x \in \mathcal{X} \} \) can be read as the set of elements \( x \) which are members of the set \( \mathcal{X} \). We allow for multiple guards on set membership to be separated by either logical conjunctions, or commas which represent the “and” logical conjunction. \( \{ x : x \in \mathcal{X}, x \notin \mathcal{Y} \} \) for example, represents the set of elements \( x \) which are members of \( \mathcal{X} \) but are not members of \( \mathcal{Y} \).

3.2.3 Natural Numbers

We use the set of natural numbers throughout this thesis, especially when dealing with sequences. To resolve ambiguities in what is and is not considered a natural number we use \( \mathbb{N} \) to refer to the set of all positive integers (not including zero), and \( \mathbb{N}^+ \) to denote the set of non-negative integers (including zero).
3.2.4 Booleans

A number of our definitions rely on the ability to differentiate between true and false values. To this end we present the set of all Booleans \( \mathbb{B} \), as:

\[ \mathbb{B} = \{ \text{false}, \text{true} \} \]

Where, \text{false}, denotes a false value and, \text{true}, a true value.

3.3 Sequences

Occasionally we find it necessary to provide orders over a collection of elements. When we do so we use the following notation:

\[ [a_1, a_2, \ldots, a_i : a \in \mathcal{A}, i \leq \mathbb{N}] \]

Where \([\ ]\) denotes the empty sequence and \(a_1\) denotes the first element, \(a_2\) denotes the second element, etc, of a sequence. When referring to a sequence using a single symbol we either place a hat over that symbol or append \(s\) as the suffix, for example:

\[ \hat{a} = as = [a_1, a_2, a_3] \]

Sequences are often in the domain and/or co-domain of the functions we define throughout this thesis. Rather than use the full sequence notation, which can be somewhat cumbersome, we use \([\mathcal{A}]\) as a more concise way to express:

\[ [\ ] \cup \{[a_1, a_2, \ldots, a_i : a \in \mathcal{A}, i \leq \mathbb{N}] \} \]

3.4 Functions

Mathematical functions are used throughout this dissertation to describe relationships between different modelling concepts. Most functions we define will be given a signature consisting of the functions name, domain and co-domain. In the cases where a full definition is necessary and is available, the function definition will follow the signature. We will use the term operation as a synonym for function throughout wherever it improves textual flow.

Function signatures appear in the form:

\[ \text{func-name} : D \rightarrow D' \]

Where func-name represents the function’s name, and \(D\) denotes the domain and \(D'\) the co-domain. When necessary, the specific kind of a function: injective, surjective of bijective, will appear with the function’s definition.

Oftentimes a function’s co-domain may include some value to indicate error. We reserve \(\varepsilon\) to serve as the universal error symbol for this thesis and, when it is a member of a functions co-domain, place it towards the left side of the co-domains declaration. This
is so that the right, non-erroneous results appear towards the right of the declaration. A convention borrowed from the Haskell [OSG08] ‘Either’ data type convention.

Function definitions are not always provided, but when they are, they will typically follow the signature and appears in the form:

\[ \text{func-name}(d) = d' \]

In providing such definitions we often use cases to delineate between alternative behaviour:

\[ \text{func-name}(d) = \begin{cases} d' & \text{if } d \in \mathcal{D} \\ \chi & \text{otherwise} \end{cases} \]

The preceding definition of \( \text{func-name} \) results in \( d' \) if \( d \) is an element of the set \( \mathcal{D} \), and \( \chi \) otherwise. Cases are resolved in top-down order. The topmost case with a matching predicate is the result of the function. We use otherwise to cover all situations where none of the predicates of the other cases are met.

In more complex definitions we use local symbolic assignments to aid the presentation and comprehension. These assignments follow the main definition and are preceded by the where term. The following definition, for example, assigns the \( d' \) symbol the value of 1:

\[ \text{func-name}(d) = d' \quad \text{where} \quad d' = 1 \]

**Functions on Sequences**

It can be cumbersome to deal with sequences directly in function definitions. To alleviate this issue and to demonstrate typical function definition we provide two higher-order functions for dealing with sequences: \texttt{map} and \texttt{fold-err}. These operations, or equivalents thereof, are commonly provided as standard in contemporary functional programming languages. Informally, \texttt{map} applies a given function to every element of a sequence, returning a sequence where each element has had the function applied. The \texttt{fold-err} function is similar to \texttt{map} except that it reduces a sequence of elements to either a single value, or error if that application of the function should fail at any point while reducing the sequence. Both functions are defined formally in Definition 3.1.
Definition 3.1 (Functions on Sequences):

\[ \text{map} : (a \rightarrow b) \times [a] \rightarrow [b] \]
\[ \text{map}(f, []) = f(a) \]
\[ \text{map}(f, [a_1, a_2, \ldots, a_i]) = [f(a_1), f(a_2), \ldots, f(a_i)] \]

\[ \text{fold-err} : (a \times b \rightarrow e \cup a) \times a \times [b] \rightarrow e \cup a \]
\[ \text{fold-err}(f, a, []) = a \]
\[ \text{fold-err}(f, a, [b_1, b_2, \ldots, b_i]) = \begin{cases} 
\text{fold-err}(f, f(a, b_1), [b_2, \ldots, b_i]) & \text{if } f(a, b_1) \neq e \\
\epsilon & \text{otherwise}
\end{cases} \]

3.5 Generative Grammar Notation

In certain situations we find it convenient to formalise concepts through a generative grammar notation [Cho57, Bac59]. We follow the general convention of generative grammar notation using \textit{synProduceSymb} for productions and \mid to separate alternative productions. To illustrate, consider the following:

\[ A \rightarrow B \mid C \]

This states that \( A \) can be produced either through \( B \), or alternatively through \( C \). Production alternatives may appear on the one line, as above, but more commonly will appear on consecutively over multiple lines.
Chapter 4

Enterprise Software Systems, Environments and Observation

A brief introduction to enterprise software systems, their operating environments and the challenge of analysing the run-time properties of software systems operating in such environments was provided in Chapter 1. We now expand on that introduction, detailing the key concepts of enterprise software systems and environments, essentially building the foundations upon which later interaction modelling and specification will be based. The result is a conceptual model of enterprise software systems and their operating environments with abstractions that encapsulate the fundamental elements of these environments and their relationships with one another. We then use this conceptual model to precisely state the core problem considered by this thesis: production of interactive closure environments for open software systems which enable observation and analysis of that system’s run-time properties.

This chapter begins with a detailed informal description of enterprise software systems and the environments in which they operate in Section 4.1. We use this informal understanding to subsequently develop a conceptual model in Section 4.2. This model is then used in Section 4.3 to formally define the problem of enabling observation and analysis of run-time properties of open (enterprise) software systems. We conclude the chapter in Section 4.4 with a summary of the most significant content and contributions.

4.1 Enterprise Software Systems and Environments

As stated in the introduction, enterprise software systems can be defined, most freely, as software systems which exist in order to addresses some enterprise problem. Typically, enterprise problems arise out of the need to conduct essential business activities, possibly straightforward in small organisations, in larger scale corporations. Tasks such as identity management, supply chain management, customer relations and asset management can be substantially more difficult when a business has a complex hierarchy of personnel, organised into various departments, and physically located at different sites across the globe. Enterprise software systems are intended, and actively used, to help manage these
kinds of critical business activities for large enterprises. Large businesses rely on these enterprise software systems to help manage critical aspects of their day-to-day activity, supporting operations which are can be of particularly high business value. A point evidenced by the amount of money organisations are willing to spend on this software; sales and service contracts measured in the millions-of-dollars being not the exception, but the rule.

These enterprise software systems are commonly used by large organisations, such as multi-national financial institutions and automotive companies, to facilitate numerous aspects of their operations. Such an organisation may, for example, use an “Identity and Access Management Suite” to administer account privileges on the thousands of machines and services distributed throughout the organisation. This organisation may also use some “Automated Software Delivery” system to ensure that each of their computer assets have access to site-licensed software and also have the most recent security patches applied.

The typical characteristics of enterprise software system operating environments is due largely to the kinds of problems they address and the businesses likely to have such problems: large, potentially multi-national, organisations with complex internal structures, and complex external relationships with a multitude of suppliers and customers. Such organisations tend to have operating environments which, to a certain extent, mimic their own structure: large-scale, distributed, and heterogeneous environments. The operating environment of a large financial institution, for example, may have tens-of-thousands of physical machines scattered across the globe, running any of a number of different operating systems and individual software stack configurations, and communicating with one another using a variety of different application-layer protocols. The behaviour of the individual systems in this environment depends not only on their initial configuration and state, but on the aggregate of past events occurring over some period of time with others in the environment, as well as on certain unobservable events occurring within the system itself, perhaps triggered by local users or internal processes.

### 4.2 Conceptual Model

The environment in Figure 4.1 illustrates each of the key entities included in our conceptual model though a diagrammatic depiction of a relatively small enterprise software environment. Detailed description and discussion of each of these conceptual elements is the subject of the rest of this section, it is instructive however to have a high-level understanding of these elements and how they relate to one another before getting into their underlying details.

There are five hosts each of which represents some network addressable machine connected to the enterprise’s communication infrastructure, and executing some collection of communication processes. There are three different kinds of communication process all depicted by the ellipse shape: clients, services and peers. These processes represent the different kinds of software systems being run on enterprise hosts which can interact with other systems in the environment. Client processes are depicted as being contained...
within their hosts as they are not directly contactable by other processes in the environment. Service and peer processes, on the other hand, are presented as straddling the border of their hosts as they can be directly contacted by clients and peers in the environment, respectively. Communication processes can interact with one another by first establishing channels with one another over the communication infrastructure, depicted by the double ended arrows, and then transmitting messages to one another over those channels, represented in the figure by the squares with arrows depicting delivery direction. Channels connecting client and server process must be initiated by the client and is represented diagrammatically by having the solid arrowhead pointing away from the initiator (client). Channels connecting peers on the other hand can be established by either party and therefore we make no distinction in the diagram. Finally, each channel has a certain communication protocol which must be followed by each party in order to ensure coherent interaction. These different protocols are represented diagrammatically through the different stroke patterns on each channel.

The small scale enterprise environment presented in Figure 4.1 can be related back to two of the typical characteristics of enterprise software environments: distribution, scale and heterogeneity. Distribution is depicted simply through the hosts being connected to one another through the underlying communication infrastructure. We note that there is no restriction on the geographical location of these hosts, they could be scattered across the globe. Scale is depicted in two key dimensions: the number of hosts, and the number of communication processes active in the environment. Unfortunately, it is impractical to depict large-scale environments in the diagram so instead we remind the reader that in deployment these environments typically contain thousands or even tens-of-thousands of hosts, each of which may be simultaneously executing dozens of communication pro-
Heterogeneity is present mainly in the different kinds of protocols being used to communicate between the hosts. There is also however, heterogeneity that is not immediately evident in the diagram which should be highlighted. The various hosts of the environment are likely to be different kinds of devices, such as, everyday workstations, laptops, tablets, server machines, mainframes, and increasingly mobile devices, for example. These various devices may run any of a number of different operating systems as suited to the device, including but not limited to, some version of Windows, Mac OS X, a Linux distribution, UNIX, etc.

The remainder of this section elaborates the conceptual model we have outlined. Section 4.2.1 discusses hosts and their communication processes in detail, Section 4.2.2 describes channels, protocols and messages, and finally, Section 4.2.3 discusses the communication infrastructure.

4.2.1 Hosts

Hosts encapsulate the various physical, or in some cases virtual, devices present in an enterprise environment. As already established, these hosts can vary considerably from one another, being any of a range of different kinds of devices, running different operating systems and having different available hardware. They do however, share two key properties:

**Addressable** Each host has some addressable identity on the enterprise environment’s network. In practice this is typically an IP address paired with a hostname.

**Execution Environment** Each host executes a collection of communication processes which interact with other such processes being executed on other hosts in the environment.

In addition to these key properties, hosts also have some internal state and processes which warrant mentioning, but only indirectly affect interaction between systems and as such we omit them on the basis that they are not crucial for our own purposes.

Definition 4.1 provides the signatures of the operations which may be invoked on conceptual hosts. The address and communication processes of a host can be retrieved through $\text{addr}$ and $\text{endpoints}$ respectively:

**Definition 4.1 (Operations on Hosts):**

\[
\text{addr} : \mathcal{H} \rightarrow \mathcal{A}^G \\
\text{endpoints} : \mathcal{H} \rightarrow \mathcal{P}
\]

Where $\mathcal{H}$ denotes the set of all hosts, $\mathcal{A}^G$ denotes the set of all global addresses, i.e. IP addresses and hostnames, and $\mathcal{P}$ denotes the set of all communication processes. We use $\mathcal{P}$ to denote the set of all communication processes due to the conceptual similarity with processes in the $\pi$-calculus [Mil99].
Communication Processes and Endpoints

Communication processes are the central conceptual entities which initiate and carry out interaction between distributed enterprise software systems. It is these processes which offer services, establish channels, and transmit and receive messages from these channels, thereby interacting with other systems in the environment.

In practise, an enterprise software system may include numerous communication processes, even running on different hosts. This discussion is however, postponed until Section 4.3 where it is more pertinent. Here we focus our attention on the communication processes, or as we will more commonly refer to them from here on, endpoints. These endpoints can essentially be thought of as partially independent application-layer network processes. By partially independent we mean that their behaviour may be influenced by the activities of other endpoints in the environment, but in general, are not entirely determined by those activities.

We arrange endpoints into three distinct categories: servers, clients and peers. Regardless of type each endpoint is responsible for:

- Establishing new channels either through passively accepting connection requests, actively making connection requests, or both.
- Processing message receptions, and generating messages for transmission, on channels which it establishes.
- Closing and discarding disused communication channels.

More specifically the different types of endpoints can be described as such:

**Services** The passive communication processes. These endpoints listen on a particular port, or network access point, allowing clients to establish new connections. A service will also listen to each established channel, processing requests and transmitting the results of those requests in the form of messages. Examples of service endpoints include LDAP directory servers and HTTP web servers.

**Clients** The active communication processes which seek out and initiate connections with services. A client will typically transmit a series of requests on its established channels retrieving information which is relevant to its purpose. Examples of client endpoints include LDAP directory clients and Web Browsers.

**Peers** Capable of simultaneously acting as both server and client, accepting channel requests initiated by other peers, actively seeking out and initiating channels of its own as necessary. Perhaps the most common example of peer endpoints, although uncommon in enterprise environments, are BitTorrent peers.

We provide the signatures of functions which operate on the various endpoints in Definition 4.2. Services and peers can passively accept new channel requests through the accept operation, while clients and peers may actively establish new channels through the connect operation.
4.2 Conceptual Model

Definition 4.2 (Operations on Endpoints):

accept : \( \Pi^S \cup \Pi^P \rightarrow \epsilon \cup \Pi^S \cup \Pi^P \)  
connect : \( \mathcal{A}^G \times \mathcal{A}^P \times \Pi^C \cup \Pi^P \rightarrow \epsilon \cup \Pi^C \cup \Pi^P \)

Where the set of all endpoint services, clients and peers are denoted \( \Pi^S \), \( \Pi^C \) and \( \Pi^P \) respectively and \( \Pi^S \cup \Pi^C \cup \Pi^P \subseteq \Pi \). We denote the set of all port addresses as \( \mathcal{A}^P \). Although technically, these signatures allow mutation between endpoint types, it does not generally occur in practice and is merely allowed in our model for the sake of simplicity. Whenever we invoke accept on a service for instance, either an error or a service is the implicit result, never a peer.

4.2.2 Channels

Interaction between endpoints is a fundamental aspect of enterprise software environments. The majority of this interaction occurs on what we model as communication channels which is the main subject of this subsection. We also describe the other key elements of endpoint interaction, namely messages and protocols which directly relate to the channel concept. These details contribute to the conceptual model we have been developing in this Section. As stated earlier, the enterprise software systems we are interested in typically operate at or above the application-layer of the OSI reference model. Our model of enterprise environment communication channels reflects this fact, abstracting away from the lower level details enabling communication which we dub the communication infrastructure.

Communication channels facilitate interaction between the semi-independent endpoints executing on the various hosts of an enterprise software environment. These channels allow information to be passed between the endpoints, regardless of physical locality, through message exchange. We model our channels as being dyadic; connecting two and only two endpoints. This being the most prevalent paradigm in enterprise environments. If necessary, alternative paradigms such as multicast can be modelled atop of this dyadic foundation.

An unavoidable consequence of distributed environments, such as enterprise environments, is asynchronicity. It is futile to attempt an entirely reliable global clock. It follows that the two endpoints connected to one another via a channel, are unable to have a shared consistent view of that channel; the destination endpoint for a message transmitted by the other endpoint on the channel, cannot know that message is on the channel until it arrives at destination end. Furthermore, the sender of the message cannot know when the message has been received by the destination endpoint without some acknowledgement message, which again the sender cannot know when is received at the other end.

To handle the inherent asynchronicity of distributed environments we model communication channels as being shared by the two associated endpoints by reference. Each endpoint then, in a sense, has a reference to this shared channel and it is this reference which each endpoint uses to communicate with the other. We rely on the underlying
communication infrastructure to ensure that the relationship between the two channel references and the channel itself are properly maintained. By taking this approach we ensure that both endpoints associated with a channel are able to have a locally consistent view of a channel, even though a globally consistent view is not generally feasible.

**Protocols**

In order for two enterprise endpoints to be able to interact with one another they must both agree upon and faithfully implement some predetermined communication *protocol*. It is this protocol which governs the allowable (temporal) sequence of messages which are valid to be exchanged throughout interactions occurring on channels and sets in place the general expectations of both communication parties. The protocols we focus on in our conceptual model are the application-layer protocols such as:

**LDAP** The lightweight directory access protocol used to query and modify hierarchical data structures, commonly referred to as directories. In enterprise environments these directories are often used as a kind of phone book, storing contact-type information of employees, clients and suppliers.

**SNMP** The simple network management protocol is used to monitor manage devices in enterprise environments.

**CORBA** The common object request broker architecture is an implementation language agnostic standard for normalising and transmitting objects between enterprise endpoints.

**SOAP** The simple object access protocol is an XML based protocol that enables exchange of structured information used extensively in the by Web Service based software systems.

**JMS** The Java message service is used by Java Message Oriented Middleware for transmitting messages between Java clients.

**HTTP** The hypertext transfer protocol used for data communication on the web is also widely used by enterprise endpoints to deliver web services to one another. It is often used, for instance, as the protocol to deliver SOAP messages between clients and web services.

**Messages**

Messages are the finite, but perhaps variable, sized units of information which are transmitted along and delivered by channels to and between enterprise endpoints. We use this message based paradigm in our conceptual model as it is the most prevalent in enterprise environments. Additionally however, it is possible to simulate stream based interactions atop a message based foundation; media streaming services, for example, stream video and audio over packet (message) based communication infrastructure. It is worth noting that messages typically have some on-the-wire encoded representation when being
transported on channels. So that we can deal with messages of different encodings in a uniform manner we assume that each encoding has some means of being mapped both to and from our conceptual message model.

Operations

Channel (reference) use is supported by three operations, the signatures of which are provided in Definition 4.3: (i) receive which consumes a message from a channel that was sent by the remote endpoint connected at the other end, (ii) transmit which sends a message to the other remote node connected to the channel, (iii) close which closes a channel meaning no further messages shall be transmitted nor received using it. For the sake of flexibility we make only a single assumption regarding implementation of these operations. Namely that there are no phantom messages. The only messages which can be received on a channel, are those which have been previously transmitted by the remote endpoint connected to the other end of the channel. This does not however, mean that messages must be delivered uncorrupted. Only that they do not appear out of the ether. Beyond this assumption channels are free to operate according to any desired paradigm. In particular, a channel does not necessarily guarantee ordered or uncorrupted delivery of messages. In most cases however, because these channels operate at the application-layer, communications are usually guaranteed to be free of corruption and delivered in order.

Definition 4.3 (Operations on Channels):

\[
\begin{align*}
\text{receive} & : \mathcal{C} \rightarrow \epsilon \cup \emptyset \cup \mathcal{M} \\
\text{transmit} & : \mathcal{C} \times \mathcal{M} \rightarrow \epsilon \cup \emptyset \\
\text{close} & : \mathcal{C} \rightarrow \epsilon \cup \emptyset
\end{align*}
\]

Where \(\mathcal{C}\) denotes the set of all communication channel references and \(\mathcal{M}\) the set of all messages.

The receive operation takes as input a channel reference and returns either some error if something goes wrong, an empty set indicating that there is not currently any message present, or a successfully received message. The use of an empty set indicate the lack of a message on the channel allows receive to be implemented in a non-blocking manner so that the caller need not wait for a message to arrive, allowing other computational tasks to be performed in the meantime.

The transmit operation takes as input a pairing of a message to send and a channel on which to send it. The result being either some error or the empty set indicating a successful transmission attempt. The return of the empty set rather than an error does not necessarily guarantee that the delivery of the message will be successful, nor that the message will be intact (uncorrupted) through transmission. Merely that delivery is successfully being attempted. Specific channels and channel operation implementations are thereby free to make stronger guarantees regarding delivery reliability and ordering as required.
4.2.3 Communication Infrastructure

The endpoints of enterprise software systems typically operate at or above the application-layer of the OSI networking reference model. At this layer the underlying details of network communication tends to be hidden from the endpoints themselves. Accordingly our conceptual model abstracts over the underlying communication infrastructure encapsulating things such as the physical wired and wireless network interfaces, routers, switches, cabling, etc. This communication infrastructure facilitates interaction between our conceptual endpoints by silently uniting the channel references held by endpoints as consistent underlying communication channels capable of delivering messages to each reference. This simplification allows us to focus on interaction behaviour of endpoints and segregates our work from similar efforts carried out in the well established field of computer networking [ns, CR10, Fal99]. Partitioning in this manner allows concerns such as communications network capacity to be studied by experts in that domain and allows more straightforward integration of results obtained in that domain with our own work.

4.3 Observing Run-Time Properties of Open Software Systems

The preceding section of this chapter introduced our conceptual model of enterprise software environments. In this section we use this foundation to precisely define the core of the problem considered in this dissertation: production of interactive closure environments for open software systems which enable observation and analysis of that system’s run-time properties. To do this we firstly define our notion of open software systems which are essentially distributed software systems, like those typical in enterprise environments, which depend on other external software systems to operate at run-time. We then define the notion of closure environments which are essentially environments consisting of interactive endpoints required so that certain open software systems are able to operate as intended. It is through these so called closure environments that it is possible to observe and analyse the run-time properties of (open) enterprise software systems.

4.3.1 Open Software Systems

We have already introduced the high-level diagrammatic representation of our conceptual model of enterprise software environments in Section 4.2, specifically Figure 4.1. Here however, it will be useful to have an even higher level diagrammatic representation of these environments.

Figure 4.2 presents an enterprise software environment consisting of 14 individual endpoints. At this level of diagrammatic abstraction we omit everything besides the channels and endpoints, represented by arrows and circles respectively. We also refrain
from assigning types to the endpoints as this information, for the purposes here, is unnecessary. The arrowheads of the channel have the same meaning as before, solid arrowheads pointing away from the initiator of the channel. Although this is not a particularly important concept at this level of abstraction, it does not clutter the diagram and is congruent with the channel representation of the previous diagram. We use the enterprise software environment presented in Figure 4.2 throughout this section to help describe the problem of enabling observation and analysis of the run-time properties of enterprise software systems.

Figure 4.2: High-Level Depiction of Endpoints in an Enterprise Software Environment

Software Systems

Software systems operating in enterprise software environments can be defined as a collection, or set of, endpoints, perhaps being executed on a range of different hosts. Typically each software system having some specific purpose to fulfil within the environment as a whole. In order to achieve this purpose a software system must interact with the other endpoints in the environment, establishing channels for communication and exchanging messages on those channels, designed ultimately to fulfil the specific purpose of that software system. An individual software system essentially coordinates, to a certain extent, the behaviour and actions of a collection of endpoints in its environment.
To illustrate, we use the environment of Figure 4.2 as a base, and highlight an individual (enterprise) software system in an environment in Figure 4.3. This system consists of two endpoints, π₂ and π₄ contained within the ellipse labelled s. There is internal interaction occurring between the endpoints as well as interaction with the external endpoints π₁, π₃ and π₆. It is plain to see from this figure that the behaviour of the software system is dependent on its environment: communications received from π₁, for example, may influence the behaviour of π₂, which may subsequently trigger some action in the other endpoint within the system, π₄. This dependence and influence of external endpoints is the key criterion for the openness property of (enterprise) software systems.

We may now formally define this openness property and do so with the aid of a couple of helper operations in Definition 4.4. The hosts operation takes either a channel or software system and returns the set of addresses of hosts directly associated with that channel or software system. Calling hosts, for example, with the software system s in Figure 4.3 will result in the global addresses of the hosts which are executing π₂ and/or π₄. Similarly, the chans operation returns the channel reference directly associated with the software system provided. Again using the system from Figure 4.3 as an example, the result of chans when passed the system s would be the channels connecting π₂ with π₁ and π₄, as well as those connection π₄ with π₂, π₃ and π₆. Finally, the is-open? operation check to see whether the software system provided is, or is not, open. We define is-open? by checking whether a channel exists, which is directly associated with the software system provided, that is directly associated with some host that is not within the hosts directly associated with the system itself. In the case of the software system in Figure 4.3 assuming that any endpoint π₁, π₃, or π₆ is being executed on a different host from π₂ and π₃, then the result of is-open? is true for s.
4.3 Observing Run-Time Properties of Open Software Systems

Definition 4.4 (Operations on Software Systems):

hosts : \(\mathcal{C} \cup \mathcal{S} \rightarrow 2^{\mathcal{C}}\)  
chans : \(\mathcal{S} \rightarrow \mathcal{C}\)

\[\text{is-open?} : \mathcal{S} \rightarrow \mathbb{B}\]

\[
\text{is-open?}(s) = \exists c \in \text{chans}(s) : \text{hosts}(c) \not\subset \text{hosts}(s)
\]

Where \(\mathcal{S}\) denotes the set of all software systems and \(\mathbb{B}\) denotes the set of Boolean values, i.e. true and false.

4.3.2 Closure Environments

Consider the task of analysing the run-time properties of an open enterprise software system. This particular system may be deployed into and intended to operate within a wide variety of different environments. It is expected that this system will be capable of handling and operating in these various environments. In particular it is meant to be capable of operating in environments of enterprise scale, environments which may consist of tens-of-thousands of endpoints. Given that it is prohibitively expensive to physically construct and configure such an environment merely to assess the run-time properties of an individual system, how is one meant to enable observation and subsequently analysis of this system’s run-time properties in environments of realistic scale?

Some run-time interactive representation of the system’s external environment is required. Without such an environment, the open software system cannot operate as it was intended. This environment must be interactive in the sense that it is capable of communicating with the system during that systems execution, so that the run-time properties may be observed and analysed. What is needed is some interactive closure environment, which can be used in lieu of a real enterprise environment, closing the environment for the open system and thereby enabling observation and analysis of that software system.

Figure 4.4 illustrates the endpoints required to form a closure environment for system \(s\). The closure environment consists of those endpoints residing within the dotted line blob. Provided that the endpoints \(\pi_1, \pi_3\) and \(\pi_6\) are present in the environment, the system \(s\) is able to operate as intended enabling observation and analysis of its run-time properties. Notice that the channel which previously connected endpoints \(\pi_1\) and \(\pi_5\) has been discarded. This is necessary for the closure environment to actually result in a closed environment when combined with the open software system.

We provide the set of all closure environments of open software systems in Definition 4.5. A closure environment, as stated in the definition, is not actually an environment but a software system. This is because there is no formal distinction between the concepts, making one would only distract from the more important issues. The set of all closure environments is denoted \(\text{ce}\) and is defined as those software systems (environments) which when combined (placed in union) with some open software system results in a closed (not open) system.
Observing Run-Time Properties of Open Software Systems

4.3

Figure 4.4: Closure Environment for an Open Software System

Definition 4.5 (Closure Environments of Open Software Systems):

\[ \{ce : (ce, s) \in S^2, \text{is-open}(s), \neg \text{is-open}(s \cup ce) \} \]

4.3.3 Dynamic Systems and Environments

The conceptual model we have developed thus far is predominantly static. Although many of the operations we have introduced are in fact dynamic, we are yet to consider how the invocation of these operations affects enterprise environments and software systems. As we have stated previously, run-time interactivity is a necessity for an environment which intends to enable observation and analysis of the run-time properties of enterprise software systems. To restate using the conceptual model, the closure environment for an open software system must be interactive at run-time. This means we must find a way to accommodate the invocation of the dynamic operations we have defined and how that changes the environment overall. To this end we now present a model of change or transition for our conceptual model.

We begin by providing the signatures of transition or change operations for dynamic endpoints and systems in Definition 4.6. We omit an operation for transitions of environments because, as stated earlier, environments are formally indistinct from software systems. A transition of an endpoint or system represents that entity changing from one (current) state into another (future) state. This will typically involve invocation of the operations we provided earlier in Definitions 4.2 and 4.3, establishing new channels, receiving and processing requests on these channels as well as generating and transmitting messages, etc. We leave these transition functions undefined for the sake of later flexibility. We could, for instance, define system transition as being some relation to the
transitions of its underlying endpoints. For the moment however, this places unnecessary restrictions on what constitutes valid dynamic behaviour in our model and so we leave both transition functions undefined.

**Definition 4.6 (Dynamic Endpoints and Systems):**

\[
\Delta^\pi : \Pi \to \Pi \quad \Delta^s : S \to S
\]

We use \(\Delta\) to represent change and transitions in general, \(\Delta^\pi\) to denote the set of all endpoint transitions, and \(\Delta^s\) to denote the set of all software system transitions.

Incorporating change into our conceptual model means we must also consider how this affects closure environments. We provide an operation which defines what does and does not constitute a closure environment for dynamic software systems in Definition 4.7. The \(\text{is-closure}\) operation takes two pairs as input, each of which being a pairing between a system transition function and a software system. The first pair is what may be the dynamic closure environment and the second is the dynamic software system which the first pair intends to close. We define \(\text{is-closure}\) recursively, checking that the current static closure environment and software system are not open under union, and invoking \(\text{is-closure}\) recursively with the next static closure environment and software system to ensure that the software system remains closed after future transitions. We note that in practice a dynamic closure environment may only need to close an open software system for a reduced subset of possible transitions; developers analysing the run-time properties of an open software system may only be interested in behaviour when performing certain key functions. As such closure environments which close an open software systems for all possible futures, as Defined in 4.7, are not typically required in practice. Regardless this definition expresses the theoretical essence of closure environments and therefore the core problem investigated in the dissertation. As such we do not consider these practical concerns in our formal model.

**Definition 4.7 (Closure Environments of Dynamic Software Systems):**

\[
\text{is-closure} : (\Delta^s, S) \times (\Delta^s, S) \to \mathcal{B}
\]

\[
\text{is-closure?}(((\delta, ce), (\delta', s))) = \neg \text{is-open?}((s \cup ce) \land \text{is-closure?}((\delta, \delta(ce)), (\delta', \delta'(s))))
\]
4.4 Summary

This chapter has provided a comprehensive treatment of what we consider to be enterprise software systems and environments. We have developed a formal conceptual model of these environments breaking them down into entities such as hosts, endpoints (communication processes), channels, messages and protocols. We defined the key operations on these conceptual entities and thereby formalised their relations to one another. This conceptual model is fundamental to the dissertation and will be used extensively in the coming chapters, especially those concerning the modelling framework and subsequent emulate-able model stacks. In this chapter we have also put this conceptual model to use, it underpins the rigorous formal description of the core underlying problem investigated throughout this thesis: providing interactive closure environments for open (enterprise) software systems, thereby enabling observation and analysis of open (enterprise) software systems run-time properties.
Chapter 5

Modelling and Emulation

At this point we have discussed thoroughly the context and problem investigated in this thesis. We went as far developing a formal conceptual model of this context, enterprise software systems and environments, so that we could formally and without ambiguity, state the core of the problem, producing interactive closure environments which enable open enterprise software systems to operate as intended and thereby enable observation and analysis of the run-time properties of such systems. Furthermore, we have surveyed the existing academic literature as well as the popular tools and approaches adopted in industry, which are either directly, or more tenuously, related to the topic. We found that there is no such published research, approach or tool capable of producing closure environments which are simultaneously: highly-scalable, well-suited to the representation of complex server-class interaction behaviours and sufficiently decoupled from the open software system under analysis.

In this chapter we detail our approach to providing such a closure environment: interaction modelling and emulation. Firstly in Section 5.1 we elaborate upon this approach in general discussing the core rationale and predicted benefits of the emulation approach in contrast with the current work and methods. We then go on to describe a modelling framework, in Section 5.2 which provides the fundamental structure and foundation upon which a rich variety of emulate-able models and specifications can, and will in Chapters 6 and 7, be constructed.

5.1 Interaction Modelling and Emulation

Emulation, in the context of software development and computer science, generally refers to the ability of a piece of software (or in some cases hardware) to behave or mimic another piece of software, hardware device, or even an entire platform. The term emulation is often conflated with simulation. Anecdotally, these terms have typically been treated as synonyms in many of our discussions with academic colleagues as well as industry practitioners. To the best of our knowledge there is no universally accepted distinction. In order to avoid ambiguities, we use the term emulation for systems which are able to interact with others at run-time, and reserve simulation for systems which do not primar-
ily interact with others run-time. To use the terminology from Chapter 4, simulations are *closed* whereas emulations are *open*. This ability to interact with the system of interest in real-time is, according to our definitions of the terms, the fundamental difference between simulation and emulation software.\footnote{It is worth noting that this perspective on the emulation and simulation terms is not universally shared. The earliest use of the term emulation which we found dates back to 1965 where it was used to refer to a specific technique developed at IBM [Tuc56]. This, however, seems an excessively restrictive, the definition we adopt is more flexible whilst still unambiguously delineating between the terms.}

The ability of an emulator to mimic the behaviour of another system can be very useful. Developers building software for embedded devices can, for example, use an emulator in place of the real physical embedded device for much of the development activities. Deployment and testing of embedded software on a physical device is arduous and time consuming compared to conducting the same tasks using an emulator. Emulators can also be used to execute defunct software; DOSBox [DOS] provides a means to execute old DOS games on current hardware, enabling joyful reminiscence of childhood memories. Finally, as mentioned in the Section 2.2.6, network emulators such as ns-3 [ns-] can be configured to behave as complex computer networks with certain extra-functional characteristics, a useful tool for computer network researchers.

Interaction modelling and emulation is the approach we propose to providing interactive representations of (enterprise software) closure environments for open (enterprise) software system, thereby enabling the observation and analysis of open (enterprise) software system’s run-time properties. The approach consists of two key elements: (i) *interaction models* which describe the interactive behaviour of individual enterprise software systems and (ii) an *emulator* which provides a means to “execute” or “interpret” these models and thereby interact with a real enterprise software system under investigation. More specifically, we aim for (our) emulated closure environments to be interactive approximations of real environments, sacrificing behavioural *accuracy* for the sake of emulator *scalability* and consequently being better able to represent the typical extra-functional *characteristics* of these environments, rather than being too precise with the runtime, functional behaviour of the emulated endpoints.

We present a high-level diagrammatic representation of our emulation approach in Figure 5.1. This is a revised version of Figure 1.1 which was used earlier in Chapter 1 to provide a general overview of the interaction modelling and emulation approach. Figure 5.1 uses the more precise terminology and diagrammatic conventions introduced in Chapter 4.

The most important difference between these figures is the change from the somewhat vaguely labelled “enterprise software systems” situated in the deployment environment, to the more precise concept of endpoints. Consequently, the interactions we model are focused on the messages exchanged on an endpoint’s channels. This is the key purpose of the *interaction model*, specifying an *approximation* of interactive behaviour of endpoint channels in such a way that the resulting specification can be used as a basis for the runtime behaviour of an emulator. We do not illustrate the hosts of these endpoints as, to a large extent, this is immaterial to the approach.
One of the endpoints in Figure 5.1 is a different type than the others, indicated by the dash-dot pattern. This does not mean that it is a different class of endpoint, like the other two it is a server, only that it has a different type of behaviour and, as illustrated by the dash-dot pattern on its channel, interacts with other systems using a different communication protocol. The fact that this different endpoint as well as the other two are in fact servers can be inferred from the arrowheads of their channels. Following the convention used in Figure 4.2, the solid arrowheads point away from the client endpoint and towards the server. We include this different type of endpoint in the figure to highlight the heterogeneity of our approach; different kinds of endpoints and protocols can be specified as interaction models and subsequently emulated.

As already established, the interaction models form emulate-able behavioural approximations of real deployment environment endpoints. These models, being emulate-able, are passed to an emulator, which has some internal engine (not pictured) capable of concurrently interpreting and/or executing the supplied models. The emulator, supplied with interaction models for each endpoint type in the real deployment environment, can be configured so that it appears to be, and behaves similarly to, the real deployment (closure) environment. Once the emulator is launched with such a configuration, the system under investigation is able to commence its own operations; interacting with emulated endpoints as necessary, and in the same manner as it would real endpoints in a real deployment environment. Meanwhile, as these operations are taking place, the run-time properties of the system under investigation can be observed, measured and recorded for analysis and assessment.
5.1.1 Behavioural Approximation

*Behavioural approximation* is a key concept of our interaction modelling and emulation approach. This behavioural approximation refers to the difference between the observable behaviour of an emulated endpoint and that of the real endpoint on which its behaviour is based. Behavioural approximation can be contrasted with that of behavioural *equivalence*. Informally, a behaviourally equivalent closure environment is one where there is no observable difference between its behaviour and that of the real deployment environment. The closer an emulated endpoint’s behaviour is to that of the real system, i.e. the closer to equivalence, then the better the behavioural approximation. Striving to achieve behavioural equivalence using interaction modelling and emulation, despite being a desirable characteristic in general, is likely to prove futile. This will essentially requiring interaction specifiers to replicate implementations of an endpoint’s behavioural functions. Moreover, contemporary system level virtual machine technology can already be used to supply closure environments which are accurate enough to be considered behavioural equivalences for the majority of practical purposes. Furthermore, there is a computational cost associated with approaching behavioural equivalence; the more accurate an interaction model, then the more complex it is to emulate which has a negative effect on scalability of the emulator. Thus there is a trade-off between modelling power and the potential scalability of an emulator which is demonstrated and investigated by experimentation in Chapter 10.

In practice, the requisite degree of accuracy for an emulated endpoint’s behaviour largely depends on the specific scenario in which the emulator is being used to close the environment. Some scenarios, for example, require modifications to an endpoint to persist so that responses to later requests will reflect that change. While in other scenarios, it may be possible to simply fake the modification, making no change to the endpoint yet reporting success. If the update is not referenced by future interactions, then this non-persistent behaviour may be sufficient. These modelling “shortcuts” tend to reduce the human effort required to specify endpoint behaviour, and ultimately results in better run-time performance and scalability for the emulator.

We refine the aim of our research to provide, via interaction modelling and emulation, run-time interactive closure environments which constitute some behavioural approximation of the real closure environment on which they are based. An approximation which is accurate enough to enable ongoing interactions between an emulator and an external open software system under investigation thereby enabling observation of that system’s run-time properties.
5.1 Interaction Modelling and Emulation

5.1.2 Expected Benefits and Limitations

There are numerous advantages to emulation as an approach to providing interactive closure environments. The heterogeneity of enterprise software environments can, to a certain extent, be naturally accommodated. Any type of real endpoint whose behavioural approximation can be expressed by an interaction model, can by definition be emulated.

Emulation is an intrinsically scalable approach. A single emulator can execute many distinct models simultaneously, appearing to be a large scale closure environment whilst in reality, it is a single piece of software running on a single physical host. The degree to which an emulator can scale depends on the complexity and expressive power of the underlying interaction models. As demonstrated in Chapter 10, an emulator executing models which use little expressive power can be scaled more efficiently than an emulator which is executing more complex and expressive models. These simpler, less accurate behavioural approximations are typically cheaper to store in memory and also swift to evaluate. Finally, the scalability of an emulator may be further improved by exploiting similarity and equivalences between the interaction models. If, for example, multiple instances of a certain endpoint type are being emulated, the equivalent internal states across these endpoints can be stored once and simply referenced by each endpoint, rather than having each endpoint store its own copy.

The fundamental driving force of this research is better enabling run-time observation and analysis of enterprise software systems. Although we focus on the enabling rather than conducting of the observation and analysis, emulation is well positioned to support these kinds of activities. It is straightforward for example, to implement a comprehensive logging module which faithfully and accurately records all interaction between external and emulated endpoints. Data collected by such a module would be incredibly valuable in post-emulation diagnosis and analysis. Furthermore, such logging data may be translated back into some simple, record-and-replay based interaction model, which when emulated repeats the emulation side behaviour of the previous emulation exactly. The ability to accurately repeat past behaviour is particularly useful when attempting to diagnose unexpected behaviours.

Reliance upon specification of interaction models is the most apparent limitation of the interaction modelling and emulation approach. Specifying new models can be a tedious, time consuming activity. Thankfully, once defined an interaction model specification may be shared publicly, within a team, or an organisation. We expect that over time, interaction model and specification libraries may evolve containing many different kinds emulate-able interaction models and specification. Existence of such a library is likely to reduce the amount of effort required to adopt emulation. Meanwhile, alternatives to manual specification are being sought by our colleagues. Specifically, colleagues of ours are currently investigating automated derivation of partial interaction models and specifications using traces of network communications. If successfully, this will further reduce obstructions to adoption of the interaction modelling and emulation approach.
5.1.3 Discussion

Our take on emulation is somewhat unique in the area. The distributed nature of the enterprise environment suggests a similarity with computer network emulators. The needs, however, of an enterprise software environment emulator and a network emulator, are significantly different. Network emulators are typically used to investigate non-functional attributes of communication infrastructure and protocols, such as, round trip time of packets and reliability of delivery [CS03, CR10], for instance. Accordingly, the models supplied to a network emulator focus on characterising extra-functional attributes and specific infrastructure configurations. Models supplied to our enterprise software environment emulator on the other hand, focus upon (at least for the moment) approximate functional or behavioural characterisation. Specification of extra-functional attributes for an enterprise software environment emulator is an auxiliary feature, and can only be of significant use once functional approximation is sufficient to close the systems real environment that interaction is enabled between the two.

Embedded device emulators, such as the iOS Simulator provided as a part of Apple’s Xcode development environment [xco], as well as other interpreter backed emulators, despite generally lacking any need for distribution, are quite similar in need to our enterprise software environment emulator. The central purpose of these emulators, including our enterprise software environment emulator, is functional; providing interactive representations of emulated endpoint’s behaviour, the emulated endpoint appearing to be (from the perspective of another software system), and indeed behaving as if it were the real endpoint. This similarity in purpose is mirrored by a similarity in approach. Interpreter backed emulators essentially map the primitive operations and values used by the emulated endpoint onto some equivalent abstraction executable on the host system. Similarly, our enterprise software environment emulator involves mapping the behaviour of our emulation targets (individual enterprise endpoints) onto an abstraction (node model) which is executable on the host system (the emulator). Despite the considerable difference in the level of abstraction between these emulators, the approach itself is effectively identical.

There is however, in addition to the lack of distribution, another significant difference between interpreter based emulators and our enterprise software environment emulator. Interpreter based emulators have the advantage of emulating formally defined targets, meaning that its mapping can be performed automatically. Modelling for an enterprise software environment emulator on the other hand, must (at least for the moment) be performed manually, by a human typically relying on multiple, potentially unreliable and/or inconsistent, information sources such as innate domain and system knowledge, network traces of previous interactions or just plain documentation.

No existing work, tool or approach aimed at producing interactive representations of (enterprise software operating) closure environments for open (enterprise) software systems is simultaneously highly-scalable, well-suited to the representation of complex server-class interactions and adequately decoupled from any particular system under investigation. Interaction modelling and emulation is an approach to providing approx-
5.2 Interaction Modelling Framework

Endpoints are complex communicating processes whose run-time behaviour is determined not only by its implementation, but also by a non-trivial combination of initial configuration (starting state), interactions with other endpoints, as well as intermittent interactions with human users. Emulation requires models of these endpoints, models which express some approximation of a real endpoints behaviour. Constructing these models, despite the approximation, is a non-trivial activity which increases in difficulty as models are required to express increasingly accurate behavioural approximations.

This section introduces a modelling framework which facilitates construction and specification of approximate endpoint behaviour models. This framework provides a guideline for model development, defining a standard structure and organisation for interaction models. It also provides definitions for common operations and a basis for developing future extensions. Furthermore, readers may find it a useful guide to the contents and overall organisation of Chapters 6 and 7 (appearing later in this dissertation).

We present our modelling framework diagrammatically in Figure 5.2. This framework consists of 10 elements organised into 2 main groups: (i) the principals, which provide definitions of the fundamental concepts ubiquitous throughout the framework, all of which were discussed at length in Chapter 4 where we introduced our conceptual model. The (ii) interaction elements, on the other hand are, with the exception of the protocol element, new. These elements support top-down or, more accurately, outside-in modelling and specification of endpoint interactions. Collectively, the five modelling principals message, service, channel, endpoint and host, constitute the base units of this, and potentially other similar, interaction modelling frameworks. Meanwhile, the interaction facet of our framework is layered and consists of the protocol, behavioural and data store entities. An interaction stack represents a specific grouping of interaction models, one from each layer, which collectively form a kind of mould. Continuing with this analogy, the material to be poured into the mould is defined by the interaction specification. Essentially it is this specification which will be interpreted and/or executed by the emulator.
A feature of our framework is its focus on modelling endpoint *interactions*. We do this because it is these endpoints and, more specifically, their run-time interaction behaviour which are the principal requisite for closing dynamic open software systems. Any endpoint behaviour unrelated to interaction is considered to be, at least for the moment, superfluous. Another feature of our framework is its external-to-internal layered organisation. This structure allows interaction models to be built up step-wise, beginning with the externally observable aspects of interaction and providing increasing levels of internal unobservable detail at each layer.
5.2 Interaction Modelling Framework

5.2.1 Principals

The interaction modelling framework defines 5 principal elements. These elements represent the fundamental abstractions introduced by our conceptual model and are likely essential, in some form or another, for any modelling of the behaviour and characteristics of enterprise software systems and environments. They appear ubiquitously throughout our interaction focused modelling framework and would, at the very least, be of some use in alternative modelling frameworks. The principal elements: channel, service, endpoint and host have been established in Chapter 4 to an extent which is sufficient for our framework. As such we will refer the reader to Chapter 4 for treatment of those concepts. We do however, augment the previous understanding of messages, so that interaction models are able to reason about and construct them. We shall also augment our understanding of protocols, presenting them as a layer of the interaction framework which helps ensure modelled endpoint interactions are conformant with whichever application-layer protocol is appropriate.

Messages

We have introduced the message abstraction as the unit of interaction in Chapter 4 and assigned \( M \) to refer to the set of all such messages. This level of definition was sufficient for the purposes of that chapter. Our interaction modelling framework, however, requires more detail so that it may reason about and construct messages of its own.

The revised definition of messages is provided in Definition 5.1 and is the cross product of the set of all value sequences and the set of all message shapes. We also define two operations on messages: provided a message \( \text{msg-shape} \) returns the shape of that message, \( \text{msg-vals} \) on the other hand, returns the value sequence of the message, i.e. the message’s content in the form of a sequence.

Definition 5.1 (Messages):

\[
M = \big[ V \big] \times \zeta
\]

\[
\text{msg-shape} : M \rightarrow \zeta \quad \text{msg-vals} : M \rightarrow \big[ V \big]
\]

\[
\text{msg-shape}(\langle vs, \sigma \rangle) = \sigma \quad \text{msg-vals}(\langle vs, \sigma \rangle) = vs
\]

Where \( \zeta \) denotes the set of all message shapes and \( V \) the set of all values.

Message shape is used to encapsulate the underlying intent or purpose of a message. Generally, but not necessarily, the shape of a message corresponds to a request for or result of a particular operation. Considering HTTP for example, a web browser may send a message with the shape GET to a server, requesting a specific web page, to which the server may respond with a message of shape OK, indicating that the request was successful and the response message contains the page requested. If on the other hand, the page was not found, the shape of the response message may be Not-Found. In previous work \cite{HSHV09, HSHV10} we used the term type to refer to this same concept. Type
however, is somewhat loaded as a term, with precise formal meanings in other contexts, which is why we now use the term *shape*.

The shape of a message can usually be deduced from its content. In some cases the message shape is even explicitly stated within the content. It is useful however, in some circumstances, to have message shape stated distinct from the underlying content, especially when modelling protocols and in message dispatching.

We introduce the concept of *value* to encapsulate message *content*. Specifically, the content of a message is represented as a value sequence. We use a sequence explicitly as in practise the content of a message will always have some kind of (internal) order, even if this ordering happens to be arbitrary.

### 5.2.2 Interaction Layers

The layers of our interaction model enable a top-down, or more accurately, an external-internal approach to modelling and specifying enterprise endpoint interactions. The interaction model consists of three layers which beginning with the highest-level, externally observable temporal properties of interaction, defined by the protocol model, proceeds to add increasing levels of internal detail and logic through the behaviour and data store layers, respectively. In a nutshell, the protocol model is intended to define the temporal properties of interaction, i.e. the high-level *what* and *when* of interaction behaviour. The behaviour model defines interaction logic, i.e. the *how* of interaction. Finally, the data store model defines what the behaviour model may refer to in defining the how, i.e. the *with* of interaction.

**Definition 5.2 (Operations on Interaction Modelled Endpoints):**

\[
\begin{align*}
\text{protocol} &: (\emptyset \cup \mathcal{C}) \times \Pi^I \to \epsilon \cup \mathcal{P} \\
\text{set-protocol} &: \mathcal{C} \times \mathcal{P} \times \Pi^I \to \Pi^I \\
\text{behaviour} &: \Pi^I \to \Delta_{\Pi^I}^T \\
\text{data-store} &: \Pi^I \to \mathcal{D}
\end{align*}
\]

Where \( \Pi^I \) denotes the set of all endpoints modelled in our interaction framework and is a subset of the set of all endpoints \( \Pi \). We use \( \Delta_{\Pi^I}^T \) to denote the set of all transitions for the endpoints modelled within our framework and \( \mathcal{D} \) to denote the set of all data stores.

Definition 5.2 provides signatures for the base operations for endpoints modelled in our interaction framework. More detailed treatment of the key underlying concepts: protocols, endpoint behaviour and data stores will follow. For the moment, however, we present the endpoint-level operations so that the reader has a bird’s-eye view of the interaction model itself. We use protocol and set-protocol to retrieve and update an endpoints protocol respectively. The protocol operation will retrieve the base, or initial, protocol of the endpoint if supplied the empty set as the first argument. Otherwise, if it is also provided a channel reference, it will retrieve the protocol associated with that channel and endpoint, returning error if it cannot be resolved. The set-protocol operation on the other hand only updates the protocol associated with a channel and an endpoint. For the sake of simplicity we assume endpoints have some base protocol which will remain
invariant throughout interaction which is why we do not define a mechanism for altering this underlying base protocol. The behaviour operation retrieves the transition function of the supplied endpoint, while the data-store operation retrieves the endpoints data store. For the moment we do not provide a mechanism to alter endpoint data stores as this is unnecessary in the simpler rudimentary interaction stack.

Protocol

In order for two enterprise endpoints to be able to interact with one another they must both agree on and faithfully implement some predetermined communication protocol. It is this protocol which governs the allowable temporal sequence of messages which are valid to be exchanged throughout interactions occurring on established channels and sets in place the general expectations of both communication parties.

It follows that in order for an emulator to approximate the interaction behaviour of these endpoints, some notion of communication protocols is necessary. This will ensure that an emulated interaction, based in part on an accurate model of the expected communication protocol, can be protocol conformant. A reasonable baseline for approximated interaction behaviour as an emulator will at the very least transmit messages of valid shape throughout its interactions. Additionally, an emulator may use this protocol model to detect and report conformance issues exhibited by external systems thereby facilitating observation and analysis of those systems.

Definition 5.3 (Protocol Model Operations):

\[
\text{next} : \{?,?,!\} \times \zeta \times \mathcal{P} \rightarrow \epsilon \cup \mathcal{P}
\]

\[
\text{valid} : \{?,?,!\} \times \zeta \times \mathcal{P} \rightarrow \{\chi, \checkmark\}
\]

\[
\text{valid}(o, \sigma, p) = \begin{cases} 
\checkmark & \text{if } \text{next}(o, \sigma, p) \in \mathcal{P} \\
\chi & \text{otherwise}
\end{cases}
\]

\[
\text{valid-outputs} : \mathcal{P} \rightarrow 2^\zeta
\]

\[
\text{valid-outputs}(p) = \{\sigma : \sigma \in \zeta, \text{valid}(!, \sigma, p) = \checkmark\}
\]

\[
\text{finished} : \mathcal{P} \rightarrow \{\chi, \checkmark\}
\]

\[
\text{finished}(p) = \begin{cases} 
\checkmark & \text{if } \exists (o, \sigma) : \text{valid}(o, \sigma, p) = \checkmark, o \in \{?,?,!\}, \sigma \in \zeta \\
\chi & \text{otherwise}
\end{cases}
\]

Where \(\mathcal{P}\) denotes the set of all protocol models, message origin is denoted ? for input and ! for output, and \(\chi, \checkmark\) denote invalidity and validity, as well as finished and not finished, respectively.
Definition 5.3 provides the operations of our protocol model. This definition provides signatures of our four protocol model operations: valid, next, valid-outputs and finished, it also provides the full definitions for the valid, valid-outputs and finished operations. Protocols are modelled from the perspective of the endpoint being specified and we delineate between receptions and transmissions, from the endpoint’s perspective, using ? and !, respectively.

Protocol models change over the course of an interaction as messages are exchanged over the channel. This ability for a protocol to change is supported through the next operation, which provided an origin, message shape and protocol model, determines the following protocol model, provided supplied input constitutes a conformant interaction, otherwise returning error $e$ as the result. Concrete protocol models can be fully defined simply by implementing this operation as the three others can be derived from this single operation. In practise however, it may be more efficient to provide custom implementations of these other operations.

Protocol conformance can be checked directly by the valid operation. When provided with an origin, message shape and protocol model, this operation determines whether it is, or is not conformant, denoted by $\checkmark$ and $\chi$, respectively. This can be done by simply invoking the next operation and checking whether the result is another protocol model, indicating conformance, or error, indicating non-conformance.

So that a protocol model may be used as a basis for determining interaction behaviour, we define the valid-outputs operation. This operation calculates the set of message shapes which constitute conformant outputs, i.e. valid transmissions, for a given protocol model. Provided a protocol model, this operation is simply defined as the set of message shapes which when used as an output, are conformant for that protocol, i.e are valid. An interaction model equipped with this information knows what shapes of messages it can transmit throughout the interaction, a fact exploited by our rudimentary interaction stack provided in Chapter 6.

Finally, so that an emulator may know when it is acceptable to close and discard a channel, we define the finished operation. This operation calculates whether there are any more protocol conformant messages which could be received or transmitted. A result of $\checkmark$ indicates that the interaction, according to the protocol, is in fact finished, while $\chi$ indicates that there are still possible interactions to take place. A channel with a protocol that is not finished can be closed regardless and does not necessarily indicate some fault or failure.
5.2 Interaction Modelling Framework

**Behaviour**

The protocol layer of our interaction modelling framework is used to specify the externally visible temporal interaction properties for an enterprise endpoint. Specifically, the *when* of interaction can be expressed at the protocol layer. The data store layer models the *with* of interaction, containing the data or information necessary for interaction with external software systems. The middle layer is the *behaviour* layer and it is at this level that the *how* of interaction is defined. This layer uses surrounding layers and principals to support specification of behavioural approximations of real enterprise endpoints.

The behaviour layer of our interaction modelling framework is essentially synonymous with the endpoint transition function from Chapter 4. An interaction, or endpoint, modelled in our framework requires some transition function which decides how the modelled endpoint will behave currently and in the future. Definition 5.4 provides the details:

**Definition 5.4 (Transitions for Interaction Modelled Endpoints):**

\[
\Delta_\tau^I : \Pi^I \rightarrow \Pi^I
\]

Where \(\Delta_\tau^I\) denotes the set of all transition functions for endpoints modelled in the interaction framework.

The essence of the behaviour layer is defining how to handle messages receptions, how to construct messages for transmission, and how all these activities relate to the overarching protocol model and underlying interaction data store. There are many potential methods for achieving the goals of this layer. For example, behaviour can be based on transmission of protocol conformant messages based on skeletons containing placeholder fields which are filled in at run-time. This is the essence of the behaviour layer used in the rudimentary stack described in Chapter 6. Alternatively, behaviour can be modelled by handler functions responsible for processing certain message shapes, which is illustrated by the expressive stack described in Chapter 7. In order to maintain flexibility and to allow such different approaches to behaviour we do not further restrict operations at this layer.

**Data Stores**

We now introduce the bottom and final layer of our interaction modelling framework, the data store layer. The purpose of the data store layer is to define the *with* of interaction. It holds the information necessary to actually behave and interact like a real (enterprise) endpoint. This layer stores all the data necessary, in an appropriate structure, for the behaviour layer to calculate and construct the responses to requests. It may, for example, mimic the interface and behaviour of an SQL database, or alternatively, a key-value lookup map, otherwise whichever data store and structure is appropriate for the corresponding behaviour layer. For the sake of flexibility we do not restrict the operations on data stores other than to note that, at the very least, there must be some way to query and, in some cases, modify the stores. We use \(\mathcal{D}\) to refer to the set of all data stores:
Definition 5.5 (Data Stores):

\[ \mathcal{D} \]

Where \( \mathcal{D} \) denotes the set of all interaction model data stores.

5.2.3 Stacks and Specifications

Protocol, behaviour and data store layers are typically developed in conjunction with one another, aiming at a specific approach or level of approximation for endpoint interaction behaviour. Two such stacks are defined in this thesis, a basic rudimentary stack given in Chapter 6 and more powerful expressive model stack described in Chapter 7. A stack is a protocol, behaviour, and data store model, intended for operation with one another, they are essentially a convenient abstraction which collects together compatible, concrete interaction layers.

Interaction specifications define a specific instance of a some interaction stack. These specifications are used to define endpoints of a particular kind, such as, an LDAP Directory server, or a BitTorrent peer, for example. A specification essentially provides the endpoint specific values required to instantiate a model which is emulate-able. In object-oriented terms, one can think of an interaction stack as being some concrete class, while a specification is the specific parameter values that will be passed to the constructor of the class on instantiation. These specification instances, when provided to an emulator, can be executed and interacted with at run-time by other software systems, thereby providing a closure environment with approximated observable behaviour.

5.3 Server-Class Endpoints

The interaction modelling framework presented in Section 5.2 is flexible enough to accommodate all three endpoint classes: servers, clients and peers. In this work however, we focus on modelling and emulating server class endpoints and so in this section we refine our framework so that server class endpoints are explicitly handled. As it turns out, this does not require much additional work. The protocol and data store layers may remain as they are. The only significant change we make is the addition of the request handler function \( \text{handle-rq} \) which is the fundamental basis of interaction behaviour of server class endpoints; an emulated server endpoint should at the very minimum, be capable of handling incoming requests.

The signature of the handle-rq operation is provided in Definition 5.6. An emulator, upon receiving a message (request) on a channel associated with a server-class endpoint being emulated, is expected to invoke the corresponding handle-rq operation passing in the request message along with the channel on which it arrived. The handle-rq operation will process this request, perhaps referring to the protocol model associated with the channel as well as the endpoint’s data store in doing so. In the majority of cases the request processing will result in a subsequent response or sequence of response messages being transmitted along the channel to the endpoint which made the request. If
5.4 Summary

This chapter has covered the details of our approach to providing closure environments for open (enterprise) software systems and also laid out a framework in which interaction models of differing expressive power can be accommodated. The interaction modelling and emulation approach was covered in detail, describing the rationale, expected benefits and inherent limitations. The interaction modelling framework which was introduced provides a layered, interaction focused infrastructure for defining interaction models of enterprise environment endpoints to various degrees of accuracy. By combining concrete models at each layer interaction stacks can be assembled which are then subsequently used as a basis for specifying interaction behaviour of various types of enterprise endpoints. The idea being that these specifications can be provided to an emulator for instantiation and execution, thereby providing closure environment approximations through emulation. The interaction modelling framework we have defined provides a foundation for two of the major contributions of this dissertation which follow in Chapters 6 and 7 in which we describe two different interaction stacks capable of providing different degrees of behavioural approximations. Finally, we refined the general interaction modelling framework so that server-class endpoints, the key endpoint class targeted in our work, can be explicitly and more easily considered in the coming chapters.
We now present our rudimentary interaction stack demonstrating the framework introduced previously in Chapter 5. The aim of the rudimentary stack is simplistic models of server-class endpoint interactions which are straightforward to specify and highly-scalable, yet are accurate enough that emulated interactions can be both protocol conformant and transmissions with value sequences which are, to a limited extent, congruent with corresponding requests and shape of the message being transmitted. Briefly, protocol conformance means that emulated transmissions are always a valid shape, in the temporal sense, i.e. the responses to a request are always messages of correct message shape. Again briefly, congruence of transmission value sequences means that the content of transmissions have the expected structure and values implied by the message shape, and furthermore, that symmetric values across corresponding requests and responses are preserved, i.e. the value of request message identifiers are present and have the same value in corresponding responses. This rudimentary stack can be thought of as a highly-scalable but somewhat superficial model of server-class interactions whose observable behaviour is accurate enough to close environments for a variety of open software systems but incapable of any deeper, or realistic, processing of requests.

This chapter is structured following the outside-in (top-down) layers of the interaction modelling framework. Section 6.1 presents a rudimentary automata based protocol model, after which Section 6.2 describes the rudimentary behaviour based on placeholder values and message skeletons, and finally, Section 6.3 describes the rudimentary data store. Throughout the chapter we use an LDAP Directory Server as an example enterprise server-class endpoint whose interactions we specify using the rudimentary stack. This illustrates both the rudimentary model stack itself as well as the general process behind specification of endpoint interactions.
6.1 Protocol

The protocol layer of the interaction modelling framework, as introduced in the previous chapter, is used to characterise the high-level observable communication properties of enterprise endpoints and, more specifically, their communication channels. By observable, we mean that the behaviour is able to be witnessed from an external perspective. As stated earlier, the protocol layer is intended to define the what and when of interaction between enterprise endpoints by constraining when certain message shapes are valid for reception and transmission.

Simplicity is a key aim of the rudimentary interaction stack. The rationale being that simple models generally lead to straightforward specification of basic concepts and relationships. Furthermore, simple models are generally expressed and executed efficiently which, in turn, implies emulator scalability. To this end, we use a deterministic finite state automata [Reg09, Ch. 3.4] to as the basis of our rudimentary protocol model. Such automata being relatively straightforward to synthesise into efficient executable structures and widely understood in the software development community.

6.1.1 Deterministic Finite State Automata

Automata based formalisms have long been used in academia for the design and verification of communication protocols [BZ83, Dan80, LT89]. We base our rudimentary protocol model on a particular member of the automata family: deterministic finite state automata (DFA) [Reg09, Ch. 3.4]. DFA’s are capable of recognising regular languages [Cho57], meaning it is of relatively low expressive power requiring nominal computational resources for storage and execution. We expect this minimalism to ultimately result in highly-scalable emulation, DFA’s being particularly straightforward to synthesise into memory and time efficient executions.

It should be noted that the goals of our protocol modelling differ significantly from those of the traditional research [BZ83, dAH01, LT89] in this area. In the past, automata have been used in this area primarily as a means to ensure valid protocol designs. Automata based models of protocols are amenable to static analysis which enables formal proofs that modelled protocols are free of undesirable states such as deadlock or livelock. The protocols we intend to model however, are designed and implemented externally; any design defect is outside of our control. In practice, protocol design defects can make it through to implementation. Such defects if present in practise should also be present in our models so that the capability of external system implementations to handle these anomalies gracefully can be analysed.

The ability to interact in a manner which is valid with respect to the temporal rules of the communication channel is crucial for an emulation; an interaction where an endpoint is
Protocol

6.1

consistently transmitting non-conformant messages is not likely to enable any interesting observations or analysis of an external communication partner.

Informally, a deterministic finite automaton consists of two fundamental concepts: state and transition between states. In our rudimentary protocol model, we use transitions to represent the transmission and reception of particular message shapes. In general, each state in a protocol has some number of transitions to other states in the protocol, this is how valid sequences of message shape transmissions and receptions are specified. As discussed in Chapter 5, the protocol modelling is done from the perspective of a single target endpoint intended for emulation, so that an emulator can use this information to help guide interactions with external systems.

Definition 6.1 (DFA based Protocol Model):

\[
P_R \subseteq \mathcal{P}
\]

\[
P_R = \{(Q, \Sigma, \delta_R, q, F) :
Q \subseteq Q, \epsilon \notin Q,
\Sigma \subseteq \{?,!\} \times \zeta,
\delta_R : Q \times \Sigma \rightarrow \epsilon \cup Q,
q \in Q,
F \subseteq Q\}
\]

Where \( P_R \) represents the set of all rudimentary protocol models and \( Q \) is the set of all rudimentary protocol model states. As usual \( \epsilon \) denotes error, and in this case specifically protocol conformance error. \( \Sigma \) is used to denote the alphabet of the automata, and \( \delta_R \) denotes it’s transition function. The current state of the protocol model is denoted \( q \) while \( F \) refers to the set of final states of the protocol.

Definition 6.1 gives the formal definition of the rudimentary protocol model. Our definition does not strictly follow conventional DFA definitions. Specifically, rather than having some fixed initial state for the automata, we have the protocol model maintain a current state to track which messages shapes are valid for transmission and reception throughout an emulation. This is a more convenient formulation for our purposes as it means the emulator need not track and maintain protocol state separately, rather it can be maintained by the protocol model directly. The initial protocol model is represented simply as a protocol model which has its current state set to the anointed initial protocol state. The set of all rudimentary protocol models is defined to be the set of quintuples where each tuple consists of a set of rudimentary protocol model states \( Q \), an alphabet \( \Sigma \), a transition function \( \delta_R \), a current state \( q \) and a set of final states \( F \). We use \( Q \) to denote the set of all rudimentary protocol model states. The set of states for a specific rudimentary protocol model must be a subset of these states. We reserve \( \epsilon \) as a special state which represents the occurrence of a protocol conformance error. The current state of a protocol is denoted by \( q \), and must be an element of the set of protocol states. The alphabet of a rudimentary protocol is denoted by \( \Sigma \) and is restricted to the set constructed by the cross-product of the set of origins \( \mathcal{O} \) and the set of all message shapes \( \zeta \). Each alphabet element corresponds to either the reception or transmission of a message of the associated message shape. The set of final states for a rudimentary protocol model is represented...
by $F$ and must be a subset of the set of protocol states $Q$. Final states are used to indicate the end of interaction on a channel. An emulator can use this information to determine when it is OK to invoke close and discard a previously active channel.

The valid sequence of message receptions and transmissions for a protocol is determined by the transition function $\delta^R_p$. The domain of this function is a pairing between a rudimentary protocol model state and an element from the alphabet of the protocol model. The result of the transition function is either a $e$, indicating a protocol conformance error, or if it is not an erroneous interaction, the next state of the rudimentary protocol model. This function is total meaning that every pairing of protocol model state with an element from the alphabet of the protocol model will either result in error or the next state of the protocol.

### 6.1.2 Next Operation

With the underlying foundation of the rudimentary protocol model provided in Definition 6.1 we are now in a position to define the next operations for the rudimentary protocol model as required by the protocol layer of the interaction modelling framework. As stated when we discussed the protocol layer earlier, Section 5.2.2, the valid, valid-outputs and finished operations can all be defined generically and so we refer the reader to Definition 5.3 for those operations.

Calculating the next protocol model for a rudimentary protocol model is essentially nothing more than a convenient protected wrapper around the underlying automata’s transition function and is provided in Definition 6.2. Essentially $next^R$ checks that invoking the transition function with the current state of the protocol, along with origin and message shape provided, results in a protocol model state, rather than an error: $\delta^R_p(q,(o,\sigma)) \in \mathcal{P}_R$. If all is well, the current state of the protocol model is set to this next state and combined with the other protocol model elements as the result. Otherwise, error is returned indicating a protocol conformance error.

**Definition 6.2 (Rudimentary Protocol Model - Next):**

\[
next^R : \{?,!\} \times \zeta \times \mathcal{P}_R \rightarrow e \cup \mathcal{P}_R
\]

\[
next^R(o,\sigma,(Q,\Sigma,\delta^R_p,q,F)) = \begin{cases} 
(Q,\Sigma,\delta^R_p,\delta^R_p(q,(o,\sigma)),F) & \text{if } \delta^R_p(q,(o,\sigma)) \in \mathcal{P}_R \\
\epsilon & \text{otherwise}
\end{cases}
\]
6.1.3 Sample Specification

The ubiquity and simplicity of finite state automata makes specification of enterprise endpoint protocols using the rudimentary protocol model provided in Definition 6.1 relatively straightforward. The high-level steps can be summarised as follows:

1. Identify and ascribe shapes to the various messages exchanged on the channel. The underlying operation or intent of the message usually provides sufficient information. Consider, for instance, HTTP, each different request method can be assigned a different shape. One for HEAD, GET, POST, etc. Similarly, the different status codes of responses may assigned different shapes depending on their value.

2. Identify the different states the protocol. These serve as the points at which messages of different shapes may be transmitted or received. Nominate a starting (current) state from the set of protocol states and a set of states from the set of protocol states to represent the final state set.

3. Define the transition function for the protocol. Which message shapes are valid for reception and transmission at each state, and which subsequent state do these actions lead to. All other actions returning error.

We now present a specification of the protocol for an LDAP [Ser06] directory server using the rudimentary protocol model. This specification does not completely define LDAP for directory servers, it is however, sufficient for demonstration of the specification process and the rudimentary protocol model.

Message Shape Specification

We begin with step 1, giving the set of LDAP message shapes we consider in Definition 6.3. This definition was straightforward to determine. Each of the message shapes correspond directly to a protocol operation of LDAP. The terms used for each LDAP message shape in Definition 6.3 are self-descriptive; the beginning of the term suggests the underlying intent of the message, while the suffix of the term Rq or Res indicates whether the message is a request or a response. LDAP Add functionality, for example, is supported by AddRq and AddRes shapes. The AddRq represents a message sent by a client requesting that the directory server add the entry described by the contents of the message. The directory server receiving this request must respond with a corresponding AddRes detailing whether or not the add was successful. Similarly LDAP Bind, Delete, and Modify functions are supported by message shapes beginning with terms Bind, Del, and Mod. Search and Unbind messages do not have singular request, response pairs. An LDAP Search request, SearchRq, has two shapes of responses, SearchEntry and SearchDone. An LDAP Unbind request, UnbindRq, does not have a corresponding response as it is used by the client to indicate the end of interaction on the channel. It is important to note that although the Rq and Res suffixes indicate whether a message is a request or a response, it does not in itself define the origin of the message. An origin must still be assigned when it is used as input to the protocol model.
Definition 6.3 (LDAP Message Shapes):

\[ \xi_{\text{LDAP}} = \{ \text{AddRes, AddRq, BindRes, BindRq, DelRes, DelRq,} \]
\[ \text{ModRes, ModRq, SearchDone, SearchEntry, SearchRq,} \]
\[ \text{UnbindRq} \} \]

State Specification

The next step is identifying the protocol states for an LDAP directory server, assigning a start state and set of final states. We proceed by assigning a state for every supported LDAP directory operation. Our LDAP directory begins in the Bound state, as authentication through a BindRq is optional. The set of final states for LDAP contains the single Unbound, this indicates that no further operations should be requested by the client of the associated channel and the channel may be closed. The naming of the bind and unbind operations is somewhat misleading in the sense that the underlying intent of each operation are not in fact the opposite of one another as their naming suggests. Binding is simply an authentication mechanism whereby credentials can be issued which allow different levels of access to parts of the underlying directory. An unbind issued by an LDAP client does not “logout” or in anyway alter the credentials supplied by any previous bind request. An unbind is really a notification that the client is done interacting and the channel should now be closed at the server side.

Definition 6.4 gives the details of the set of LDAP server states. We use \( Q_{\text{LDAP}} \) to denote the set of LDAP directory server protocol states, \( q_{\text{LDAP}} \) to denote current, i.e. the start state and \( F_{\text{LDAP}} \) to represent the set of final states for an LDAP server.

Definition 6.4 (LDAP Directory States Specification):

\[ Q_{\text{LDAP}} = \{ \text{Adding, Binding, Bound, Deleting, Modifying, Searching, Unbound} \} \]

\[ q_{\text{LDAP}} = \text{Bound} \]

\[ F_{\text{LDAP}} = \{ \text{Unbound} \} \]
State Transition Function Specification

With the states and input alphabet for the LDAP directory server protocol specified, we are onto the final step: providing a definition of the corresponding state transition function. This definition will specify which LDAP message shapes are valid for reception and transmission at any given point during an interaction.

We provide this state transition function by means of a state transition table which can be found in Table 6.1. This table defines for each current state, indicated by the row, what potential input pair can lead to a next state, indicated by the column. So that the state transition function for the defined LDAP directory server protocol can be total (as required), we assert that for any pairing of state and input which is undefined in Table 6.1 to result in protocol conformance error $\epsilon$.

To help illustrate the behaviour pattern defined in this table, we consider some brief examples: An LDAP directory server which has a channel with a client, that is currently in the Bound, can move into the Adding if it receives an AddRq from that client. Once this happens, the protocol state becomes Adding. From here, the directory server may transmit an AddRes, causing the state to move back to Bound. Or alternatively, the channel may become Unbound if the server receives an UnbindRq before responding to the add. The majority of modelled LDAP operations proceed in a similar manner. Searching however, has a different pattern. Whilst in the Searching state, the directory may transmit a SearchEntry, leading to the state remaining as Searching. This pattern allows a directory server to respond with arbitrarily many SearchEntry messages, until finally it is done, indicated by the transmission of a SearchDone message. This pattern is also used by the non-modelled LDAP Extended operation, which allows an arbitrarily many number of intermediate response messages to be transmitted before a final extended response.
Table 6.1: LDAP Directory Server Protocol Transition Table
6.1.4 Emulation

The rudimentary protocol model is straightforward to encode as a compact highly scalable executable structure. The heart of the model, the transition function, can be represented as a map which relates every valid combination of state, origin and message shape with the next state of the protocol. All other combinations being invalid and indicating the occurrence of a protocol conformance error. As this transition function is fixed (will not change) it can be freely shared by all channels interacting in accordance with the protocol. Through this sharing, the only additional memory resources required for a protocol encoding, when the transition function is already known, is a reference to that channels current state in the model. Furthermore, assuming the map is implemented using a hash table, time performance for the protocol model will be approximately constant $O(1)$. This degree of time efficiency is also a boon to scalability; protocol operations can be completed very quickly leaving computational resources free for other concerns.

6.2 Behaviour

We now present the behaviour layer of the rudimentary interaction stack. The key aim of this rudimentary behaviour layer is to provide a simple mechanism for specifying server-class endpoint interactions which is capable of producing temporally valid interactions with responses that are somewhat consistent with regard to their content. To accomplish this we focus the modelling on the messages exchanged during interactions, preferring to ignore the underlying semantics of real request processing. To this end, we base response message content on skeletons which contain the bulk of the message's value sequence and prepared a priori. These skeletons are then fleshed out at run-time using information harvested from corresponding requests, thereby preserving symmetric values. The protocol model of (the previous) Section 6.1 is used to ensure that messages transmitted throughout an interaction are temporally valid, i.e. that the order of transmissions produced by an emulator, according to their shape, is valid. The end result is emulated endpoint behaviour which responds with messages of the right shape at the right time, but whose content does not significantly vary with differing parameter values of requests.

6.2.1 Placeholder Values and Message Skeletons

Transmission in the rudimentary behaviour model is based on prepackaged, structurally valid, partially-complete messages, which we term message skeletons. It seems reasonable to assume that real messages exchanged between enterprise endpoints would be available for use as a basis of these skeletons. The core of this idea is that each skeleton is essentially a message which has gaps in its value sequence, which must be filled prior to transmission. These gaps representing values which cannot be known (filled) until run-time.
6.2 Behaviour

This skeleton based approach allows the rudimentary behaviour model to be largely ignorant of the details concerning the creation of structurally sound messages. Furthermore, the computational complexity of producing a complete message, ready for transmission, depends solely on the mechanism used to fill the gaps, which as demonstrated by the by the rudimentary behaviour model presented here, is straightforward to achieve in linear time: $O(n)$, where $n$ is the size of value sequence of the skeleton.

Placeholder Values

We model the gaps in a message skeleton as members of a special subset of values which we denote $\mathcal{V}_r$ provided in Definition 6.5. We refer to elements this set as placeholder values. The key responsibility of a placeholder value is to indicate how it may be resolved, which is synonymous with filling the gap in the value sequence of a message. That is not to say that a placeholder value must fully encapsulate the logic behind how it may be resolved. Generally a placeholder value serves firstly, to identify the appropriate resolution function which can be used to resolve the value, and secondly, to provide values for any parameters which may be required by that function. By having placeholder values indicate their resolution function we have some flexibility in the rudimentary behaviour model; additional placeholder values can be incorporated by providing the corresponding resolution functions.

Definition 6.5 (Placeholder Values):

$$\mathcal{V}_r \subseteq \mathcal{V}$$

Although there is essentially no limit to the different kinds of placeholder values, our rudimentary behaviour model requires only one. This is the request-offset based placeholder provided in Definition 6.6. This placeholder represents a value which can be resolved simply through a lookup on the corresponding requests content. The request-offset offset based placeholder is essentially nothing more than a wrapped non-negative integer, the value of which indicates an offset from the beginning of a value sequence of the corresponding request. The value located at this offset in the content of the request, is the value appropriate for filling the corresponding gap in the message skeleton. We use offset based access rather than indexing by the $n$th element as the means for sequence lookup as it is the more common idiom in current widely-used programming languages; an offset of 0 indicating the first element of the value sequence. This is how symmetric values can be preserved across requests and responses. A message identifier of a response, for example, may be located at a specific offset in the requests content.

Definition 6.6 (Request Offset Based Placeholder Value):

$$\text{request-offset} \times \mathbb{N}^+ \in \mathcal{V}_r$$
The resolution of request offset based placeholder values is straightforward and is provided formally in Definition 6.7. The function resolve-request-offset takes as input, a value sequence paired with a request offset based placeholder value and yields either an error $\epsilon$ or the resolved value. The value sequence passed to the function is treated as the content of the request and is in the attempt to resolve the value. In the case that the value sequence is empty, then there is no reasonable way for a request offset based placeholder value to be resolved, and therefore error is returned. Otherwise, in the case where the value sequence of the message is non-empty, the value located at the appropriate offset value is returned, unless the offset is outside of the valid range of the value sequence, in which case the error is the result.

Definition 6.7 (Resolve Request Offset Based Placeholder Value):

\[
\text{resolve-request-offset} : [V] \times \{\text{request-offset} \times \mathbb{N}^+\} \to \epsilon \cup \mathbb{V}
\]

\[
\text{resolve-request-offset}([], \text{request-offset}, n) = \epsilon
\]

\[
\text{resolve-request-offset}([v_1, v_2, \ldots, v_i], \text{request-offset}, n) = \begin{cases} 
  v_{n+1} & \text{if } n < i \\
  \epsilon & \text{otherwise}
\end{cases}
\]

Skeletons

Message skeletons are no different formally than other messages. The placeholder values contained as part of the content of message skeletons are values themselves, and as a result there is no need to model skeletons any differently from our existing definition of messages, see Definition 5.1. It is, however, an expectation that content of request and response messages do not contain any placeholder values, i.e. they are gap-less.

The construction of valid response messages is essentially the main goal of a server-class endpoint behaviour model. With the function for resolving request offset based placeholders defined, we can move directly to achieving this goal in the rudimentary behaviour model. Definition 6.8 formally defines a function msg-from-skel which is capable of constructing complete response messages using a value sequence that represents the values of the corresponding request message and a message skeleton.
The msg-from-skel function is essentially a wrapper around the resolve-request-offset function provided earlier in Definition 6.7. The result of msg-from-skel is either an error $e$ indicating failure to resolve a placeholder, or a complete message from the set of messages $\mathcal{M}$, gap-less and ready for transmission. In the case that the message skeleton has an empty value sequence, then there is no need to try and resolve any values and the message skeleton is simply returned, $([], \sigma)$, apparently being a valid message in itself.

In the more typical case, where the value sequence of a message skeleton is non-empty, the placeholder resolution function resolve-request-offset is applied to any request offset based placeholder values present in the sequence, any other kind of value is not a gap in the sequence and is thus passed through unmodified. If any of the placeholder values could not be resolved, error is returned $e$, otherwise the value sequence with the fully resolved placeholders is returned paired with the message shape of the message skeleton: $(\mathcal{V}s', \sigma)$.

**Definition 6.8 (Message Construction from a Skeleton):**

$$
\text{msg-from-skel} : [\mathcal{V}] \times \mathcal{M} \rightarrow \epsilon \cup \mathcal{M}
$$

$$
\text{msg-from-skel}(\mathcal{V}s, ([], \sigma)) = ([], \sigma)
$$

$$
\text{msg-from-skel}(\mathcal{V}s, ([\mathcal{V}s_1, \mathcal{V}s_2, \ldots, \mathcal{V}s_i], \sigma)) = \begin{cases} 
(\mathcal{V}s', \sigma) & \text{if } \forall \mathcal{V}s_j \in \mathcal{V}s', \mathcal{V}s_j \in \mathcal{V} \\
\epsilon & \text{otherwise}
\end{cases}
$$

where $\mathcal{V}s' = \{ [\mathcal{V}s'_1, \mathcal{V}s'_2, \ldots, \mathcal{V}s'_i] : j \in \mathbb{N} \land j \leq i \\
(\mathcal{V}s_j \notin \{\text{request-offset} \times \mathbb{N}^*\} \land \mathcal{V}s'_j = \mathcal{V}s_j) \\
\lor (\mathcal{V}s_j \in \{\text{request-offset} \times \mathbb{N}^*\} \land \\
\mathcal{V}s'_j = \text{resolve-request-offset}(\mathcal{V}s, \mathcal{V}s_j)) \}$

In practice we have found that the request-offset based placeholder value to be sufficient across a wide variety of cases. There are however, additional placeholders which may provide further utility to this rudimentary behaviour model. A data-store key based placeholder, for instance, may allow values to be sourced at run-time from an interaction model’s data store, based on a lookup of value which is associated with some provided key. Incorporating additional kinds of placeholder values requires definition of the placeholders themselves as well as their resolution mechanism. It also involves modifying the msg-from-skel function so that it invokes the additional resolution mechanism as appropriate. In the case that many different kinds of placeholders are desired, it may be advantageous to abstract out the resolution invocation of the msg-from-skel operation and into a specialised dispatcher operation. This dispatcher could handle invocation of the appropriate resolution mechanisms for any placeholder value $\mathcal{V}^p$. 

---

Rudimentary Interaction Stack
Illustrative Example

To help illustrate the concept of placeholder values and message skeletons we provide an example which demonstrates how a transmittable response message can be constructed from a request and a message skeleton.

Definition 6.9 shows the value sequence for a sample LDAP bind request, typically the first message transmitted in LDAP interactions. This sequence begins with an associated value containing the message ID. The message ID is used to identify requests and in particular to which request message the servers responses refer. It is this message ID field which is the focus of this illustrative example. The second value represents the protocol operation specified by the message. This is a series of nested associations making up the bind request operation where the version is set to 3, the name is set to dc=swin,dc=com, the authentication is simple and has the value sequence ["user", "passwd"]. We use braces to group associated values so that the association boundaries are clear.

Definition 6.9 (Sample Value Sequence for an LDAP Bind Request):

\[
\begin{align*}
\{(messageID : 1), (protocolOp : (bindRequest : \{(version : 3), (name : "dc=swin,dc=com"), (simple : ["user","passwd"]))\})\}
\end{align*}
\]

In LDAP a bind request issued by a client necessitates a bind response from the server. Definition 6.10 illustrates a message skeleton for a bind response indicating a successful bind. The structure is very similar to that of the request in Definition 6.9, the key difference however, is the presence of the placeholder as the first value of the sequence. This request-offset placeholder indicates that this value of the bind response message skeleton can be resolved by taking the value located at offset 0 in the request message. The other values in the message skeleton have been filled ahead of time. Even the matchedDN value which happens to match the name value of the bind request has been filled without reference to the request; the fact the values match is coincidental.

With the request and message skeleton value sequences specified, we are now able to construct the value sequence for the response. Definition 6.11 illustrates the result of resolving the placeholder in the skeleton of Definition 6.10 using the request of Definition 6.9. The resulting value sequence is the same as the skeleton except for the first field which has been replaced with the message ID field from the request.
Definition 6.10 (Sample Value Sequence for an LDAP Bind Response Skeleton):

```plaintext
[(request-offset, 0), (protocolOp : (bindResponse : [(resultCode : 0),
    (matchedDN : "dc=swin,dc=com"),
    (diagnosticMessage : "")]),
    )
]
```

Definition 6.11 (Sample Value Sequence for an LDAP Bind Response):

```plaintext
[(messageID : 1), (protocolOp : (bindResponse : [(resultCode : 0),
    (matchedDN : "dc=swin,dc=com"),
    (diagnosticMessage : "")]),
    )
]
```
6.2.2 Request Handling

Request handling is the core behaviour of server-class endpoints. These classes of endpoints are predominately passive, waiting for messages to arrive on established channels, performing some processing on that message and, in most cases but not all, responding with one of more messages of its own. Our interaction modelling framework encapsulates this activity in the handle-rq operation. We provide the formal definition of handle-rq, the request handling function for the rudimentary interaction stack in Definition 6.12.

**Definition 6.12 (Rudimentary Handle Request Operation):**

\[
\text{handle-rq}^R : \mathcal{M} \times \mathcal{C} \times \Pi^R \rightarrow e \cup \Pi^R
\]

\[
\text{handle-rq}^R(m, c, \pi) = \begin{cases} 
\text{set-protocol}(c, p', \pi) & \text{if } \{p, p'\} \subseteq \mathcal{P}^R, \text{ } rs = [\ ], \\
\text{set-protocol}(c, p'', \pi) & \text{if } \{p, p', p''\} \subseteq \mathcal{P}^R, \\
rs = [r_1, r_2, \ldots, r_i] \\
\text{fold-err}(\text{transmit-c}, r_1, [r_2, \ldots, r_i]) & \text{if } rs = [r_1, r_2, \ldots, r_i] \neq e \\
\epsilon & \text{otherwise}
\end{cases}
\]

where \( p = \text{protocol}(c, \pi) \)

\( p' = \text{next}(!, \text{msg-shape}(m), p) \)

\( rs = \text{map(} \text{msg-from-skel,} \\
\text{zip-skels(} \text{skel-seq(valid-outputs(} p', \text{data-store(} \pi))) \text{)}) \)

\( p'' = \begin{cases} 
\text{fold-err}(\text{next-out}, r_1, [r_2, \ldots, r_i]) & \text{if } rs = [r_1, r_2, \ldots, r_i] \\
\epsilon & \text{otherwise}
\end{cases} \)

\( \text{zip-skels(} [\ ] \text{)} = [\ ] \)

\( \text{zip-skels(} [s_1, s_2, \ldots, s_i] \text{)} = [(\text{msg-vals}(m), s_1), (\text{msg-vals}(m), s_2), \ldots, (\text{msg-vals}(m), s_i)] \)

\( \text{transmit-c}(e, r) = e \)

\( \text{transmit-c}(\emptyset, r) = \begin{cases} 
\text{transmit}(r, c) & \text{if } r \in \mathcal{M} \\
\epsilon & \text{otherwise}
\end{cases} \)

\( \text{next-out}(e, r) = e \)

\( \text{next-out}(p'', r) = \begin{cases} 
\text{next}(!, \text{msg-shape}(r), p'') & \text{if } r \in \mathcal{M} \\
\epsilon & \text{otherwise}
\end{cases} \)

Where \( p \) and \( p' \) denote the protocol of channel \( c \) before and after reception, respectively. The response to the request is represented as a sequence and denoted \( rs \) and the protocol model of the channel after transmission of this response sequence is denoted \( p'' \). The three local functions \( \text{zip-skels}, \text{transmit-c} \) and \( \text{next-out} \), serve as wrappers which facilitate invocation of higher-order functions on sequences \( \text{map} \) and \( \text{fold-err} \).
The rudimentary request handler operation handle-rq\(^R\) takes as input a triple consisting of a message \(m\), i.e. the request, a channel \(c\), on which the message was received, and a rudimentary server-class endpoint \(\pi\), which is associated with the channel. The result of this operation is either error indicating failure somewhere, or the rudimentary endpoint after successful request handling. The definition of handle-rq\(^R\) is presented in three main cases: (i) the topmost covering the situation where the provided request is protocol conformant however there are no responses to be transmitted, (ii) the middle handles the case where the request is protocol conformant and there is a response sequence which has been successfully transmitted, (iii) the bottom case catches all other situations, i.e. all erroneous eventualities. There are also four local symbolic assignments and three local functions to aid presentation.

Informally, the handle-rq\(^R\) operation checks that the received request is protocol conformant and, if this is so, attempts to generate a sequence of responses by firstly locating an appropriate sequence of message skeletons in the endpoint’s data store and, subsequently, transforming those into messages ready for transmission. These transformed messages are then transmitted on the channel provided and the protocol model is updated.

Both successful cases of the handle-rq\(^R\) operation result in a call to set-protocol. If the request is protocol valid: \(\{p, p'\} \subseteq \mathcal{P}^R\), but the response sequence is empty: \(rs = [\ ]\), then the protocol model is updated for the endpoint as the protocol model after the request: \(p'\). If there are responses to be transmitted: \(rs = [r_1, r_2, \ldots, r_i]\), which maintains protocol conformant across the operation: \(\{p, p'\} \subseteq \mathcal{P}^R\), and is successfully transmitted on the channel: \(\text{fold-err}(\text{transmit-c}, r_1, [r_2, \ldots, r_i]) \neq e\), then the protocol model is updated for the endpoint as the protocol model after the transmissions are made: \(p''\). All other cases, as already mentioned, are erroneous.

In defining handle-rq\(^R\) we make use of the higher-order functions on sequences that were provided in Chapter 3, Definition 3.1. We could not, however, invoke these functions directly due to domain restrictions. The local functions zip-skels, transmit-c and next-out overcome this limitation enabling the use of map and fold-err by providing convenient local wrappers preparing input to these functions so that the domain restrictions are met. We use zip-skels to pair each response skeleton of a sequence with the request provided. Transmission is facilitated by transmit-c, which wraps transmit making it appropriate for fold-err. Finally, next-out serves a similar purpose, wrapping the next function on protocols for fold-err, all messages being outbound.

We also use a function, skel-seq, that is yet to be defined in our definition of handle-rq\(^R\). Details on this function will be given in Section 6.3, as it operates on rudimentary data stores. For the moment however, we note that provided a set of message shapes and a rudimentary data store, handle-rq\(^R\) returns a, possible empty, sequence of message skeletons. It is these message skeletons which are used to derive the ultimate responses by invocation of msg-from-skel.
6.2.3 Emulation

The rudimentary behaviour layer may be encoded by translating each of the defined operations as a subroutine. This may take the form of a procedure in a procedural-based host language, or a method on an object in an object-oriented language, or as a pure function in a functional language. In the emulator we describe in Chapter 8, we use a functional host language as this provides a straightforward translation from the operation definition provided here and throughout the thesis.

Regarding the computational resource requirements of the resulting behaviour layer encoding, time is of more interest than memory. The memory requirements for the encodings depend on the chosen host language, but in general the requirements are nominal as each subroutine need only be stored once. The time requirements however, will effect the performance and thereby the scalability of the emulator.

There are three operations to consider when analysing the time requirements of the rudimentary behaviour layer: resolve-request-offset, msg-from-skel, and handle-rqR. The resolve-request-offset operation essentially does no more than extract a value from a sequence. Assuming this sequence is index-able, this operation can be performed in constant time $O(1)$. The msg-from-skel operation is less efficient as it needs to traverse an entire value sequence, once, before completion. It therefore takes linear time $O(n)$, where $n$ represents the length of the value sequence being traversed. This is likely to be acceptable given that messages represent base units of communication these sequences are unlikely to be prohibitively long.

The handle-rqR operation invokes a mix of constant and linear time operations. In the worst case linear operations are nested four times: every element of the sequence of response message skeletons is transformed into a message through msg-from-skel, which in turn is ultimately traversed for transmission as well as to calculate the updated protocol model. Therefore handle-rqR, and consequently any server-class endpoint specification based on the rudimentary interaction stack, has a worst case polynomial time complexity of $O(n^4)$, where $n$ denotes the number of response messages.

Theoretically, it is possible to improve this worst case time complexity by streamlining the msg-from-skel function. Rather than traversing the whole value sequence of the skeleton, a shorter sequence containing only placeholder values and insertion location of that value within the corresponding skeleton, may be traversed instead. Use of this shortened sequence may lead to a slight improvement in time performance. We expect, however, that in practise this optimisation to will not be necessary. It should be easier to improve time performance through a combination of concurrent execution and sensible restrictions on the size of response skeleton sequences.

Concurrent execution is straightforward as emulated endpoints do not have dependencies on one another and, therefore, handle-rqR operations can be safely run in parallel thereby improving the total throughput of the emulator, although run-time performance of individual endpoints will remain unaffected. Furthermore, by keeping the size of response skeleton sequences small, the power of four has only a negligible effect on run-time performance. If necessary, run-time performance may be improved further by
implementing a transmit operation which can transmit a sequence of messages in bulk. We expect though, that the time complexity of this rudimentary behaviour layer is likely acceptable for most practical scenarios.

6.3 Data Store

The data store layer of the interaction modelling framework handles the with of endpoint interactions. In the case of the rudimentary data store layer, this means providing information which helps resolve ambiguity between valid response message choices; the protocol model can decide what set of message shapes is valid for transmission throughout and interaction by invoking valid-outputs, but there may be many valid output message shapes at any given point. In practise, an emulator must have some mechanism to select which of these valid output message shapes it will transmit throughout an interaction. The selection, or choice resolution mechanism, provided by the rudimentary interaction stack is provided in the rudimentary data store and is based on providing an ordering over protocol conformance output message skeletons. This is the purpose of the skel-seq operation that was briefly mentioned in Section 6.2.2 and defined fully in this section.

We provide the rudimentary data store in Definition 6.13. The set of all rudimentary data stores is a subset of the set of all data stores and is defined as the power set of all choice resolution mappings. The rudimentary data store resolves output choice ambiguity by using these choice resolution mappings which are essentially associations between message shape sets and skeleton sequences.

**Definition 6.13 (Rudimentary Data Store):**

\[
\mathcal{D}^R \subseteq \mathcal{D}
\]

\[
\mathcal{D}^R = 2^\Theta
\]

\[
\Theta = 2^\mathcal{F} \times \{[\ ]\} \cup \{[s_1, s_2, \ldots, s_n] : s_i \in \mathcal{M}, i \in \mathbb{N}\}
\]

Where \(\Theta\) denotes the set of all choice resolution mappings.

The set of all choice resolution mappings is defined as the power set of the set of all message shapes, in product with the set of all, possibly empty, message skeleton sequences. Each choice resolution mapping then, is a set of message shapes paired with a non-empty sequence of message skeletons. The idea being that for any reachable collection of valid output message shapes there is some ordering over message skeletons of corresponding shape. By reachable we mean that the collection of valid output message shapes is the result of reaching a certain protocol model through a series of protocol conformant transitions. This message skeleton ordering is subsequently used to resolve output choice.
Definition 6.14 presents the formal definition of skel-seq. It is quite straightforward and in the most typical case the skeleton sequence \(ss\), from the rudimentary data store associated with the message skeleton sequence provided, is returned. In the case that such an association cannot be found, the empty sequence is returned as a default value.

**Definition 6.14 (Skeleton Sequence Operation):**

\[
\text{skel-seq} : 2^\mathcal{L} \times D^R \rightarrow \{[\ ]\} \cup \{[s_1, s_2, \ldots, s_n] : s_i \in \mathcal{M}, i \in \mathbb{N}\}
\]

\[
\text{skel-seq}(\sigma, d) = \begin{cases} 
ss & \text{if } \exists ss : (\sigma, ss) \in D^R \\
[\ ] & \text{otherwise}
\end{cases}
\]

### 6.3.1 Sample Specification

To help illustrate the rudimentary data store model we present in Table 6.2 a sample collection of choice resolution mapping for an LDAP directory server. This table contains choice resolution mappings for each potential set of valid output message shapes according to the rudimentary LDAP protocol model defined earlier in Section 6.1.3. The message shape sets \(2^\mathcal{L}_{\text{LDAP}}\) are given in the left column and are associated with the message skeleton sequences \(\mathcal{M}_{\text{LDAP}}\) provided in the right column. Message skeletons have been treated in depth earlier, see Section 6.2.1, and so we omit full specifications for each of the message skeletons here.

<table>
<thead>
<tr>
<th>(2^\mathcal{L}_{\text{LDAP}})</th>
<th>(\mathcal{M}_{\text{LDAP}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>{AddRes}</td>
<td>{AddRes}</td>
</tr>
<tr>
<td>{BindRes}</td>
<td>{BindRes}</td>
</tr>
<tr>
<td>{DelRes}</td>
<td>{DelRes}</td>
</tr>
<tr>
<td>{ModRes}</td>
<td>{ModRes}</td>
</tr>
<tr>
<td>{SearchDone, SearchEntry}</td>
<td>{SearchEntry, SearchEntry, SearchDone}</td>
</tr>
</tbody>
</table>

Table 6.2: Sample Choice Resolution Map for an LDAP Server

All except one of the mappings in Table 6.2 are sequences with a single element. The majority of LDAP request operations are followed by a single response message. Whist performing the search operation however, there is some ambiguity, valid response messages may have the shape of either an SearchDone or SearchEntry message. The rudimentary data store provided by in Table 6.2 resolves this ambiguity by associating the set of LDAP message shapes consisting of the SearchDone and SearchEntry with an ordering, i.e. sequence of specific LDAP message skeletons: \([\text{SearchEntry}, \text{SearchEntry}, \text{SearchDone}]\). Consequently, an emulator executing this LDAP specification will know what sequence of message skeletons to transmit even when there is multiple valid output message shapes.
6.3.2 Emulation

The rudimentary data store can be efficiently encoded. Time efficiency can be achieved by storing the choice resolution mappings of as an associative map. By using an associative map, backed by some hash table, the skel-seq operation can be carried out in approximately constant time $O(1)$. The memory use of the rudimentary data store, at worst, increases linearly with the number of endpoints being emulated. Fortunately, the rudimentary data store is read only and therefore can be easily shared among endpoints of the same kind, thereby increasing scalability. The rudimentary LDAP data store, presented in Table 6.2 for instance, can be shared simultaneously by numerous emulated LDAP endpoints. This leads to emulated endpoints which, to a certain extent, appear to be behaviourally equivalent to one another, but is highly-scalable, a key goal of the interaction modelling and emulation approach.

6.4 Discussion

In this chapter we have detailed a rudimentary interaction stack which can be used to specify approximate representations of server-class enterprise endpoint interactions. The main purpose of this interaction stack is to provide a simplistic, yet practical, model of server-class endpoint interactions. Simple models, i.e. those of low expressive power, being inherently more scalable than more complex, expressive counterparts. To achieve this the rudimentary stack is focused, as much as is possible, upon the messages exchanged in the interactions, abstracting away from real request processing semantics. The result is a stack which interacts in a protocol conformant manner, and to a certain extent, transmits messages whose internal content is consistent.

We have based this rudimentary stack on a protocol model defined by a deterministic automaton, a behaviour model defined in terms of message skeletons and placeholder value resolution, and a data store which handles output choice resolution and request content tracking, see Sections 6.1, 6.2 and 6.3 respectively. Interaction specifications were provided along the way to help illuminate the underlying concepts and their relationships to one another and real enterprise endpoints. We have also discussed the emulation concerns at each layer as well as the potential scalability in terms of memory and time consumption. An emulator executing specifications based on this model stack can receive requests in constant time and generate output messages in polynomial time depending on the number of elements in that messages value sequence. It can also scale in terms of memory consumption, which grows linearly with the addition of emulated endpoints.
In constructing this model stack we have dismissed models which may be considered even more rudimentary. One such option, for example, was to ensure that transmitted messages were merely of the right shape for the channel, disregarding protocol (temporal) expectations. Another was to not bother about placeholder values and message skeletons, just sending a message known to be of correct shape at the correct time. Both of these approaches however, as well as others we conceived, were too rudimentary, resulting in interactions with real endpoints which cannot be maintained beyond the first transmission.

Although the rudimentary interaction stack is practically viable and highly-scalable, is not without its limitations. In particular, the DFA based protocol model is ill-equipped to model concurrency which is common in interactions between enterprise endpoints. Furthermore, the lack of ability to model some finer approximation of request processing, causes trouble when interacting with some external systems. Especially those systems which make some attempt to verify their previous, thought to be persistent, operations. These limitations motivate the development of the expressive interaction stack detailed in the following Chapter[7]. This expressive stack introduces more powerful mechanisms for specifying enterprise endpoint protocols, behaviour and underlying data stores.
Chapter 7

Expressive Interaction Stack

The rudimentary interaction stack described in Chapter 6 provides a means to specify enterprise server-class endpoint interactions at a somewhat superficial level; specifications being based on message skeletons constructed a priori and having certain details filled in at run-time using values sourced from corresponding requests. Although we anticipate this rudimentary stack to enable highly-scalable emulations, the lack of more meaningful request processing limits the range of scenarios in which it is appropriate. In particular, scenarios where external endpoints attempt to validate previous modification requests with follow up queries, cannot be handled. Furthermore, the automata based protocol model of the rudimentary interaction stack is not well suited to specifying the asynchronicity and parallelism exhibited in certain enterprise endpoint communications, such as LDAP [Ser06].

We detail in this chapter an expressive interaction stack which aims to address some of the key limitations of the rudimentary stack. The chapter is structured in the same way as Chapter 6, each expressive stack layer being presented in its own section, proceeding top-down. We begin with a description of the expressive protocol model that is well suited to specification of parallelism and subservient parallelism exhibited in certain enterprise endpoint interactions in Section 7.1. This expressive protocol model is based on our previous work on modelling enterprise endpoint protocols and trace conformance [HSHV10]. We then move onto discussing the expressive behaviour and data store layers in Sections 7.2 and 7.3 respectively. In combination these layers provide facilities capable of specifying deeper, more meaningful, request processing and response generation. Furthermore, the expressive data store ensures that modifications triggered by external endpoints are persisted and the responses to subsequent queries reflect these modifications. Overall this leads to an interaction behaviour which is more consistent with that exhibited by real enterprise endpoints; a closure environment which is a more accurate behavioural approximation. We again conclude the chapter with a discussion in Section 7.4 where we describe the limitations and trade-offs of the expressive interaction stack in comparison with its rudimentary counterpart.
7.1 Parallelism in Enterprise Protocols

Enterprise endpoint protocols specified using the rudimentary interaction stack are limited to those which can be expressed using deterministic finite automata (DFA). While automata are well suited to expressing arbitrarily complex sequential patterns, they are less suitable when expressing concurrency. Concurrent computation is different enough from its sequential counterpart that it warrants its own collection of formalisms including Carl Adam Petri’s petri-nets [Pet81], Milner’s \( \pi \)-calculus [Mil99] and calculus of communicating systems (CCS) [Mil89], Hoare’s communicating sequential processes (CSP) [Hoa85], and the actor model [Agh85]. Some enterprise endpoint protocols, such as LDAP [Ser06], for instance, exhibit concurrent behaviours and as such the DFA based protocol model of the rudimentary interaction stack is not well suited to their accurate specification.

So that an enterprise software environment emulator is capable of exhibiting these concurrent behaviour patterns and thereby more accurately mimicking the behaviour of a real endpoint, we introduce an expressive protocol model in this section. Similarly to the rudimentary protocol model, this expressive protocol model is based on transmission and reception of message shapes. The expressive model, however, is extended so that it may succinctly model \textit{parallelism} and, in particular, \textit{subservient} parallelism in enterprise endpoint protocols. Informally, standard parallelism refers to the ability for a protocol to be the product of two or more protocols operating simultaneously, while \textit{subservient} parallelism refers to a protocol having one or more \textit{child} protocols which are running in parallel but may be terminated by an action in the \textit{parent} protocol. Specifically, the expressive protocol model introduces protocol \textit{product} for standard parallelism as well as the concepts of run-time \textit{extension} and \textit{contraction} of protocols, which together enable concise specification of subservient parallelism exhibited by enterprise endpoint protocols.

We present the expressive protocol model as a series of four subsections. The first Subsection 7.1.1 presents the abstract syntax of the protocol model and is followed, in Subsection 7.1.2, by two sample specifications to illustrate the use of the syntax, as well as its expressive power. We then move onto the evaluation rules of the model, in Subsection 7.1.3 these rules formally define how protocol specifications defined using the abstract syntax can check conformance of incoming requests, as well as ensure conformance of outgoing transmissions.
7.1 Abstract Syntax

The abstract syntax of our expressive protocol model is given in Definition 7.1. At the
top-level, an expressive protocol model specification \( S^x_p \) is defined as either a declaration
\( D^x_p \) or an expressive protocol model \( P^x \). A declaration contains a non-empty sequence
of variables which bind a (locally defined) protocol specification \( P^x \) to a variable
\( v^x \) from the set of expressive protocol model variables \( V^x_p \). We require equality and
inequality to be defined for variable names, i.e. over the set of elements in \( V^x_p \).

Definition 7.1 (Expressive Protocol Model - Abstract Syntax):

\[
\begin{align*}
S^x_p & \rightarrow D^x_p \quad \text{Specification} \\
& \mid P^x \\
D^x_p & \rightarrow v^x = P^x \text{ and } D^x_p \quad \text{Multi Declaration} \\
& \mid v^x = P^x \text{ in } P^x \quad \text{Single Declaration} \\
P^x & \rightarrow P^x \ast P^x \quad \text{Product} \\
& \mid P^x \sim P^x \quad \text{Extension} \\
& \mid I^x_p \quad \text{Interaction} \\
& \mid v^x_p \quad v^x_p \in V^x_p \quad \text{Variable} \\
& \mid 0 \quad \text{Inaction} \\
I^x_p & \rightarrow I^x_p + I^x_p \quad \text{Choice} \\
& \mid \sigma . P^x \quad \sigma \in \Sigma \quad \text{Standard Interaction} \\
& \mid \sigma > P^x \quad \sigma \in \Sigma \quad \text{Contractive Interaction}
\end{align*}
\]

An expressive protocol model \( P^x \) may specify the composition (or product) of two
protocols, denoted by \( P^x \ast P^x \), an extension of a protocol with another protocol, denoted
by \( P^x \sim P^x \), an interaction, denoted by \( I^x_p \), a variable denoted by \( v^x_p \), or inaction, denoted
by \( 0 \). An interaction \( I^x_p \) is either a choice between two interactions, denoted by \( I^x_p + I^x_p \),
a standard interaction, denoted by \( \sigma . P^x \), or a contractive interaction, denoted by \( \sigma > P^x \).
The definition of \( \Sigma \) is the same as it appeared in the rudimentary protocol model, see
Definition 6.1, namely the set of origins paired with the set of all message shapes: \( \{?,!\} \times \zeta \), where ? denotes input origin, ! denotes output origin, and \( \zeta \) denotes the set of all message shapes.

In expressive protocol model specifications, (i) extension takes precedence over com-
position (product), and (ii) protocol extensions, product compositions, and choice are
evaluated from left to right (e.g., \( P^x_1 \sim P^x_2 \sim P^x_3 \) is an extension of \( P^x_1 \sim P^x_2 \) with \( P^x_3 \)).
Brackets may be used to group expressions in order to enhance readability and/or over-
come the default precedence rules.

The underlying idea of the expressive protocol model, like that of its rudimentary
counterpart, is to specify the protocol conformant (valid) messages throughout an inter-
action on a channel implementing the protocol. This is specified in the expressive proto-
col model using the base syntactical elements \textit{standard interaction} and \textit{contractive interac-
tion. Each standard and contractive interaction represent a valid (protocol conformant) reception or transmission and the resulting (next) protocol after that messages reception or transmission whichever the case may be. Both the standard and contractive interactions consist of this protocol model alphabet element and a protocol model continuation (appearing to the right of the symbol). It is this continuation which is used to determine the valid transmissions and receptions after a successful match against the corresponding alphabet element.

The specific protocol composition element used, combined with (i) the continuation of a matching base interaction and (ii) the type of this interaction (standard or contractive), fully determines the set of valid (protocol conformant) interactions for the next interaction. The main difference between the two forms of base interaction is that in case of a contractive interaction, all extensions to a currently active base protocol are to be terminated, whereas in the standard form, extension remain intact.

The syntactical elements product, extension, and choice allow the expression of various compositions of protocol models and interactions. Choice between interactions allows a protocol model to define multiple valid transmissions and/or receptions. If one of the interactions in a choice matches a given message, then this message is considered to be valid, and the protocol model specification uses the continuation of the matching interaction as the basis for testing the next interaction. All other choices are discarded. Unsuccessful matches in all of the choices result in a failure.

The expressive protocol model defines two types of protocol level compositions: product composition and extension composition. The product composition of two expressive protocol models allows for two protocols to be combined in such a way that message traces can be interleaved. Therefore, a protocol model alphabet element $\sigma$ that is valid for either $P_1$ or $P_2$ in $P_1 \cdot P_2$, is considered to be valid for the product composition, and the corresponding matching protocol model will evolve, leaving the other protocol specification unchanged. If $\sigma$ is valid for both, $P_1$ and $P_2$ at the same time, only one of the protocol specifications evolves, but not the other. As a result, a successful match in one protocol model does not change the valid interactions of the other.

A variable declaration $v_p^x = P^x$ assigns the expressive protocol model $P^x$ to the variable $v_p^x$. All variables defined for a given protocol model specification are visible “globally” and, as a consequence, all (locally defined) protocol specifications can refer to these variables. Thus, protocols may refer to one another by name and recur as required.

The reader may note that the expressive protocol model does not allow for nested declarations; declarations may only appear at the specification level, not within the protocol level itself. We are yet to encounter an enterprise protocol whose specification would significantly benefit from such expressiveness. Hence, in order to avoid the additional complexity they introduce, we omit nested declarations from the model.
Parallelism in Enterprise Protocols

Expressive protocol models $P^*$ can be interpreted as binary trees with the syntactical elements standard interaction, contractive interaction, inaction, and variable as leaf nodes, and product, extension, and choice as non-leaf (branching) nodes, respectively. A depth-first traversal of the tree allows us to check whether an origin-message shape pair $\sigma \in \Sigma$ is valid in $P^*$ or not. A successful match of $\sigma$ at one of the leaf nodes will trigger a rewriting of this tree, the resulting tree being used to check protocol conformance of the future messages of an interaction. The formal semantics of protocol evaluation provided in Section 7.1.3 is based on this interpretation.

Extension and Contraction

The extension of a protocol specification with another protocol specification is the key aspect of our model. It allows a base protocol to be extended by another protocol which, similarly to the product composition, interleaves the valid message traces of base and extending protocol, respectively. However, the difference is that the two protocols are not entirely independent. A contractive interaction in the base protocol terminates any extensions; the interactions defined by its extensions become invalid.

Protocol extension is particularly useful in specifying temporary extensions to a base set of interactions. This allows some additional messages to be deemed valid, but also allows the base protocol to terminate the extensions by contraction as necessary. As an example, consider the search functionality of an LDAP server. Searching is orthogonal to the underlying base functionality of an LDAP server. However, any outstanding search operations must be immediately terminated if either an unbind or bind request is received by the server. The searching operation is subservient to the base protocol and can be terminated by contraction as necessary. Details of this appear in the following Section 7.1.2.

7.1.2 Sample Specifications

We now demonstrate the expressive power of the proposed protocol model by specifying both, the LDAP and BitTorrent application-layer protocols.

LDAP Directory Server

The Lightweight Directory Access Protocol LDAP [Ser06] is a communication protocol widely used in enterprise environments, in particular to help manage access to an enterprise’s computational resources. It is a typical client-server class protocol describing requests which a client may issue to a server, and what responses it can expect. Briefly, an LDAP directory server is a server-class endpoint which allows a client to search and modify a persistent directory (tree) structure stored on the endpoint’s host.

---

1 During this traversal, variable nodes are replaced by their respective protocol models.
The regular interaction generally begins with the client establishing a connection with an LDAP server and transmitting a bind request with some authentication details. The LDAP server then issues a bind response to the client which indicates either the success or failure of the corresponding bind operation. This step however, may be omitted as most LDAP servers allow restricted access to anonymous users.

After an optional bind operation, a client can issue a number of different requests. These requests fall into two categories: (i) those which either query or modify the servers data, and (ii) those which serve some administrative purpose for the session. The majority of the administrative and data modifying requests a client can make result in just a single response from the server, indicating either success or failure of the request. However, as these requests may take some time to process, an LDAP server does not block until the response is generated, rather it can immediately and asynchronously accept further requests.

A search request is different from the majority of LDAP operations as it can result in zero or more search result entries matching the search criteria of the request. After these zero or more result entries have been transmitted, search completion is indicated by the search result done message. To complicate matters, an LDAP server has the ability to service multiple search requests simultaneously in a single LDAP session. These searches may complete in an arbitrary order and may even terminate before completion, either due to a request by the client or the server.

An LDAP session is usually closed by the client issuing an unbind request. A server, however, may also close the session after transmitting a disconnect notification. Any outstanding searches will be terminated when the session is closed. This will also happen if a client rebinds to the server, possibly using different authentication credentials.

\[
\text{Base} = (?, \UnbindRq) + 0 + (?, \BindRq) \triangleright (?!, \BindRes).\text{Base} + (?, \SearchRq).\text{Base} \sim \text{Search} + (?, \ModRq).\text{Base} \sim (?!, \ModRes).0 + (?, \AddRq).\text{Base} \sim (?!, \AddRes).0 + (?, \DelRq).\text{Base} \sim (?!, \DelRes).0
\]

and

\[
\text{Search} = (?!, \SearchEntry).\text{Search} + (?!, \SearchDone).0
\]

in \text{Base}

Figure 7.1: Expressive Protocol Specification for an LDAP Directory Server
Parallelism in Enterprise Protocols

Figure 7.1 illustrates the specification of the LDAP server protocol using the proposed protocol model. To enhance readability, all occurrences of protocol variables are underlined. We specify the basic protocol functionality in \textit{Base}, the functionality of searching in \textit{Search}, and extend \textit{Base} with \textit{Search} whenever a new search request is received. Similarly, in order to enable the non-blocking of an LDAP server in the context of processing administrative and data modifying requests, the \textit{Base} protocol is extended with protocol specifications encoding the appropriate responses. Finally, contractive interactions are used to terminate any pending operations when an UnbindRq or BindRq request is received.

BitTorrent

BitTorrent [Coh03] is a peer-to-peer (P2P) application-layer protocol supporting the scalable distribution of large files or directories to a large number of interconnected hosts. To achieve this, each file intended for distribution is split into a number of smaller pieces. A peer interested in downloading the file(s) joins an appropriate BitTorrent swarm and proceeds to (i) retrieve pieces from other peers which it does not currently have, and (ii) makes available to other peers pieces which it has already obtained. Eventually, a peer will succeed in downloading all individual pieces of the requested data-file and can merge them into a full copy of the original file or directory.

The BitTorrent peer protocol defines the majority of the interaction required to exchange pieces between individual peers. Figure 7.2 defines a specification of this peer protocol, assuming that the initial handshake and optional bitfield message exchanges have already taken place. This specification defines a well-behaved BitTorrent peer; a peer whose transmissions are congruent with its view of channel states.

Interaction behaviour of a BitTorrent peer is defined as the product of three protocols: \textit{Base}, a local channel state (initially $C_L I_L$), and a remote channel state (initially $C_R I_R$). The use of a product enables the \textit{Base}, local and remote state protocols to operate independently of one another; interactions matching in one model do not effect the interactions of either of the others.

\textit{Base} specifies interactions which do not modify the state of either the local or the remote channel including reception or transmission of keep-alive and have messages. These messages have no effect upon the validity of subsequent transmissions and receptions.

A BitTorrent channel has four state variables which affect valid transmissions. These state variables can be further categorised into those describing local and remote peer state respectively. Specifically, whether the local or remote peer is currently being choked by the other peer, or is interested in the other peer. Figure 7.2 uses a variable naming convention to help clarify which state is which. Choked and unchoked is represented by $\hat{C}$ and $\hat{C}$, respectively. Similarly, interested and not-interested is represented by $I$ and $\hat{I}$. Local and remote are denoted by the $L$ and $R$ subscripts. The combination of two states forms the logical conjunction of the two states. $\hat{C}_L I_L$, for example, represents the local channel state of being unchoked and interested.
Parallelism in Enterprise Protocols

7.1

a :- keep-alive
s :- cancel
u :- unchoke
h :- have
p :- piece
c :- choke
r :- request
n :- not-interested
i :- interested

\[ \text{Base} = (?, a). \text{Base} + (!, a). \text{Base} + (?, h). \text{Base} + (!, h). \text{Base} \]

and

\[ \text{C}_L I_L = (? , u). \text{C}_L I_L + (! , n). \text{C}_L I_L \]

and

\[ \text{C}_L \hat{I}_L = (!, r). (\text{C}_L I_L \sim (? , p). 0 \oplus (!, s). 0) \oplus (?, c) \triangleright \text{C}_L I_L + (!, n) \triangleright \text{C}_L \hat{I}_L \]

and

\[ \text{C}_L I_L = (!, i). \text{C}_L I_L \]

and

\[ \text{C}_L \hat{I}_L = (? , c). \text{C}_L I_L + (! , i). \text{C}_L I_L \]

and

\[ \text{C}_R I_R = (!, u). \text{C}_R I_R + (? , n). \text{C}_R I_R \]

and

\[ \text{C}_R \hat{I}_R = (? , r). (\text{C}_R I_R \sim (! , p). 0 \oplus (? , s). 0) \oplus (!, c) \triangleright \text{C}_R I_R + (?, n) \triangleright \text{C}_R \hat{I}_R \]

and

\[ \text{C}_R I_R = (? , i). \text{C}_R I_R \]

and

\[ \text{C}_R \hat{I}_R = (!, c). \text{C}_R \hat{I}_R + (? , i). \text{C}_R I_R \]

in

\[ \text{Base} \ast \text{C}_L \hat{I}_L \ast \text{C}_R \hat{I}_R \]

Figure 7.2: Expressive Protocol Specification of the BitTorrent Peer Wire Protocol
The BitTorrent peer protocol defined in Figure 7.2 is well-behaved. A peer will only send transmissions which will result in a meaningful change of state. When a remote peer is choked, it may send an unchoke, but it will not re-send a choke as this is already the state of the channel. Similarly, when the peer is interested, it will not re-send interested, only not-interested. Furthermore, a peer adhering to the protocol specification in Figure 7.2 will only send requests for pieces when it is unchoked by a remote peer and has registered its interest. Moreover, cancel messages may only be sent after a piece request has been made.

Product composition of the proposed protocol model enables the succinct specification of the BitTorrent peer protocol. Without product composition, 16 unique and somewhat lengthy declarations are required to define the valid interactions for the various channel states. The protocol product operator allows us to effectively halve the number of declarations required, leading to a simpler specification than would be possible otherwise.

7.1.3 Evaluation

The expressive protocol model changes throughout the course of an interaction through the process of message origin and shape evaluation. We now formally define evaluation for the expressive protocol models which, although not formally, can be thought of as the semantics of the protocol model. The relation between the expressive protocol model and the protocol layer of the interaction modelling framework is also formally defined and discussed.

Closed Specifications

As detailed in Definition 7.1, the abstract syntax of the expressive protocol model allows for the declaration of variables $v^p_x$ in protocol specifications $S^x_p$ and their use (calling) in protocol models $P^x$. For every variable used in a protocol model $P^x$, we expect that a corresponding declaration occurs in its overarching specification $S^x_p$. Undefined variables used in a protocol model must be avoided so that evaluation can proceed safely without the risk of encountering errors caused by missing variable bindings.

In order to avoid problems caused by undefined variables in evaluation we define the notion of bound and free variables of a protocol specification in Definition 7.2. Combined, the concepts of bound and free variables allow us to define when an expressive protocol model specification is closed, as provided in Definition 7.3, meaning that bindings exist for each and every variable used in the protocol model.

Definition 7.2 provides $\text{bv}$ which determines the variables bound by an expressive protocol model declaration. This is simply the variable being declared and if it is a nested declaration, any variables bound within it as well. The variables bound by a declaration are used to determine ensure that the free variables are calculated correctly given the specification level scope of variables which is desired.
Definition 7.2 (Bound and Free Variables in Expressive Protocol Model Specifications):

\[ \text{bv} : D^x_p \to 2^{V^x_p} \]
\[ \text{bv}(v^x_p = P^x \text{ and } D^x_p) = \{v^x_p\} \cup \text{bv}(D^x_p) \]
\[ \text{bv}(v^x_p = P^x_1 \text{ in } P^x_2) = \{v^x_p\} \]

\[ \text{fv}_d : D^x_p \to 2^{V^x_p} \]
\[ \text{fv}_d(d) = \text{fv}_d(d) \quad d \in D^x_p \]
\[ \text{fv}_s(p) = \text{fv}_s(p) \quad p \in S^x_p \]

\[ \text{fv}_d : D^x_p \to 2^{V^x_p} \]
\[ \text{fv}_d(v^x_p = P^x \text{ and } D^x_p) = ((\text{fv}_p(P^x) \cup \text{fv}_d(D^x_p)) \setminus \text{bv}(D^x_p)) \setminus \{v^x_p\} \]
\[ \text{fv}_d(v^x_p = P^x_1 \text{ in } P^x_2) = (\text{fv}_p(P^x_1) \cup \text{fv}_p(P^x_2)) \setminus \{v^x_p\} \]

\[ \text{fv}_p : P^x \to 2^{V^x_p} \]
\[ \text{fv}_p(P^x_1 \ast P^x_2) = \text{fv}_p(P^x_1) \cup \text{fv}_p(P^x_2) \]
\[ \text{fv}_p(P^x_1 \sim P^x_2) = \text{fv}_p(P^x_1) \cup \text{fv}_p(P^x_2) \]
\[ \text{fv}_p(v^x_p) = \{v^x_p\} \]
\[ \text{fv}_p(I^x_p) = \text{fv}_p(I^x_p) \]
\[ \text{fv}_p(0) = \emptyset \]

\[ \text{fv}_i : I^x_p \to 2^{V^x_p} \]
\[ \text{fv}_i(I^x_{p_1} \ast I^x_{p_2}) = \text{fv}_i(I^x_{p_1}) \cup \text{fv}_i(I^x_{p_2}) \]
\[ \text{fv}_i(\sigma.P^x) = \text{fv}_p(P^x) \]
\[ \text{fv}_i(\sigma.P^x) = \text{fv}_p(P^x) \]
The free variables of an expressive protocol model specification determines the set of variables in the specification which do not have a corresponding binding and is provided in Definition 7.2. The free variables of a specification $fv_s$ is straightforward, merely wrapping around the free variable function of a declaration $fv_d$ or protocol model $fv_p$, whichever is appropriate.

Free variables of declarations are more complicated. In particular, to ensure the global scope for variables, the free variables of a nested declaration must take into account the free and bound variables of the nested declaration. Otherwise the protocol being assigned to the variable $v^x_p = P^x$ cannot not use any variables subsequently declared within the nested declaration $D^x_p$. Each of the nested and binding declarations remove the variable being declared $v^x_p$ from the set of free variables being returned. The binding declaration removing this variable from the union of free variables contained in the protocol being declared $P^x_1$ and the protocol in which it is bound $P^x_2$. The nested declaration removes any variables bound in the nested declaration $bv(D^x_p)$ from the union of the free variables of the protocol and the free variables of the inner declaration before removing the declared variable.

Determining the free variables of a protocol $fv_p$, similarly to determining the free variables of a specification, mostly wraps around the appropriate free variable functions. The free variables of protocols in product or extension being simply the union of the free variables of each left and right protocol. The occurrence of a variable in the protocol model however, results in that variable being returned as free, this being the source of all free variables in the expressive protocol model. Free variables of an interaction is again a simple wrapping function collecting any encountered variables in choices as well as standard and contractive interactions. The free variables of inaction is the empty set.

In order for expressive protocol models to be evaluated safely, i.e. ensure that there are no variables which lack a binding, we provide in Definition 7.3 the set of closed expressive protocol model specifications. This set simply being the set of specifications which do not contain any free variables $fv_s(s) = \emptyset$. A specification which has bindings for each of its variables is safe to be used in evaluation as whenever a variable is encountered it can be substituted by the protocol model to which that particular variable is bound.

**Definition 7.3 (Closed Expressive Protocol Model Specifications):**

\[
\{ s : s \in S^x_p, fv_s(s) = \emptyset \}
\]
Context

The notion of closed protocol model specifications ensures that all variables mentioned in a protocol have corresponding protocol model bindings. We now introduce the concept of expressive protocol model context so that variables can be substituted for their corresponding protocol during evaluation. A context essentially stores all the associations between variables and protocol model associations made in the declarations of a specification. When variables are encountered during evaluation this context is used to find the appropriate protocol model for substitution.

Definition 7.4 provides the formal details of the evaluation context for the expressive protocol model. An evaluation context is simply a set of pairings between variables and expressive protocol models. We also provide in Definition 7.4 the lookup-in-context function which when provided with a variable and a context, yields the corresponding protocol from the context. We leave the result of lookup-in-context for missing bindings undefined as we only ever invoke this operation on contexts for closed protocol model specifications, which guarantee that a binding will exist. Construction of evaluation contexts is straightforward, achieved simply through set union, and as such it does not warrant formal definition. The context is essential in defining the evaluation of the expressive protocol model; providing a means to store variable associations appearing in declarations and a means to later retrieve those associated protocols when variables are encountered in evaluation.

Definition 7.4 (Expressive Protocol Model Context):

\[ \Xi = 2^{V_P \times P_x} \]

\[ \text{lookup-in-context} : V_P \times \Xi \rightarrow P_x \]

\[ \text{lookup-in-context}(v_p^x, \xi) = p : \exists (v_p^x, p) \in \xi \]

Where \( \Xi \) denotes the set of all expressive protocol model contexts.

Evaluate-able Protocol Model

With closed specifications and evaluation contexts defined we are now able to present the evaluate-able expressive protocol model. The evaluate-able protocol model pairs an evaluation context with an expressive protocol model enabling evolution when supplied with message origins and shapes. It is this evaluate-able protocol model which implements the next operation and is therefore compatible with the protocol layer of our interaction modelling framework described earlier in Section 5.2.

We provide, in Definition 7.5, the formalisation of the evaluate-able expressive protocol models. The set of all such evaluate-able protocol models is defined as a subset of the set of all protocols \( P_x^\Xi \subset P \). More specifically, an evaluate-able protocol model is a a pairing of a context and an expressive protocol model \( \Xi \times P_x^\Xi \).

Expressive Interaction Stack
Definition 7.5 (Evaluate-able Expressive Protocol Model):

\[ P^\xi_\Xi \subset \mathcal{P} \]
\[ P^\xi_\Xi = \Xi \times P^\pi \]
\[ \text{mk-eval} : \{ s : s \in S^x_p, \text{fv}_s(s) = \emptyset \} \rightarrow P^\xi_\Xi \]
\[ \text{mk-eval}(d) = \text{mk-eval}_d(\emptyset, d), \quad d \in D^x_p \]
\[ \text{mk-eval}(p) = (\emptyset, p), \quad p \in S^x_p \]
\[ \text{mk-eval}_d : P^x_\Xi \rightarrow P^x_\Xi \]
\[ \text{mk-eval}_d(\xi, v^\xi_p = P^\pi x \textbf{ and } D^x_p) = \text{mk-eval}_d(\xi \cup (v^\xi_p, P^\pi), D^x_p) \]
\[ \text{mk-eval}_d(\xi, v^\xi_p = P^\pi_1 \textbf{ in } P^\pi_2) = (\xi \cup (v^\xi_p, P^\pi_1), P^\pi_2) \]

Where \( P^\xi_\Xi \) denotes the set of all evaluate-able expressive protocol models.
Definition 7.5 also provides two operations for instantiating evaluate-able protocol models, mk-eval and mk-eval$_d$. Provided a closed specification, mk-eval creates an evaluate-able protocol model by either calling the underlying mk-eval$_d$ operation if the specification happens to be a declaration, or by simply returning the empty set paired with the protocol model if the specification is a protocol model. Provided an evaluate-able protocol model, mk-eval$_d$ builds up the context adding variable bindings encountered through union of the binding with the existing context. For nested declarations, mk-eval$_d$ is called recursively up until the point where the final (binding) declaration is encountered. At this point the final variable binding is added to the context (again through set union) and paired with the protocol model in which the variables are bound. Although the domain of mk-eval$_d$ is the set of all evaluate-able protocol models, it is only defined for declarations; there is no other way to alter an evaluation context besides declaration. The ultimate result of the mk-eval operation is an evaluate-able expressive protocol model which, due to its context, is capable of evaluation when presented with origin, message shape pairs.

Evaluation

The core of our expressive protocol model evaluation semantics are rules for evaluation of interactions, testing the validity of a reception or transmission against whichever collection of interactions happens to be valid at that moment in time.

Annotations

Due to the different semantics of standard and contractive interactions, we need to annotate evaluation results so that higher level elements know how they must evolve. Annotations encode whether a message matched in a standard interaction, denoted by $\bullet$, or a contractive interaction, denoted by $\triangleright$. A failed match does not annotate a result, rather it serves to indicate that no valid interaction has been found, and as usual, is denoted by $\epsilon$.

The semantics of our expressive protocol model evaluation are defined in terms of a number of evaluation rules, provided in Definition 7.6. The functions eval and eval$^I$ contained within this definition take as input an origin-message shape pair $\Sigma$, paired with either an evaluate-able $P^x_\Sigma$ or a plain expressive protocol model $P^x$, depending on whether evaluation is occurring at the protocol or interaction level. The interaction level operation, eval$^I$, has no need of the context during evaluation, and consequently, it operates on expressive protocol models directly rather than dealing with the evaluate-able augmentation. The result of these functions is either failure to find a match $\epsilon$, or what we term an annotated protocol continuation consisting of an annotation $\{\bullet, \triangleright\}$ paired with either an evaluate-able protocol model $P^x_\Sigma$ or a plain expressive protocol model $P^x$ depending on the function. For the sake of readability, we omit the superscript and subscript from some of the expressive protocol model symbols when defining the evaluation rules, i.e $P_L$ should be read as $P^x_{\Sigma L}$, $I_R$ as $I^x_{PR}$, etc.
Definition 7.6 (Expressive Protocol Model Evaluation):

\[ \text{eval} : \Sigma \times P^\times \rightarrow \epsilon \cup (\{\bullet, \triangleright\} \times P^\times) \]

\[ \text{eval}(\sigma, (\xi, P_L \cdot P_R)) = \begin{cases} 
(a_L, (\xi, P_L')) & \text{if } \text{eval}(\sigma, (\xi, P_L')) \neq \epsilon \\
(a_R, (\xi, P_L') \cdot p_R) & \text{if } \text{eval}(\sigma, (\xi, P_R)) \neq \epsilon \\
e & \text{otherwise} 
\end{cases} \quad (R1) \]

\[ \text{eval}(\sigma, (\xi, P_L \cdot P_R)) = \begin{cases} 
(\bullet, (\xi, p_L' \cdot P_R)) & \text{if } \text{eval}(\sigma, (\xi, P_L')) \neq \epsilon \land a_L = \bullet \\
(\triangleright, (\xi, p_L')) & \text{if } \text{eval}(\sigma, (\xi, P_L')) \neq \epsilon \land a_L = \triangleright \\
(a_R, P_L \cdot p_R') & \text{if } \text{eval}(\sigma, (\xi, P_R)) \neq \epsilon \\
e & \text{otherwise} 
\end{cases} \quad (R2) \]

where \((a_L, (\xi, P_L')) = \text{eval}(\sigma, (\xi, P_L))\)

\[(a_R, (\xi, P_L') \cdot P_R) = \text{eval}(\sigma, (\xi, P_R))\]

\[ \text{eval}(\sigma, (\xi, v^x_p)) = \text{eval}(\sigma, (\xi, \text{lookup-in-context}(v^x_p, \xi))) \quad (R3) \]

\[ \text{eval}(\sigma, (\xi, I^x_p)) = \begin{cases} 
(a_I, (\xi, P_I')) & \text{if } \text{eval}(\sigma, I^x_p) \neq \epsilon \\
e & \text{otherwise} 
\end{cases} \quad (R4) \]

where \((a_I, (\xi, P_I')) = \text{eval}(\sigma, I^x_p)\)

\[ \text{eval}(\sigma, (\xi, 0)) = \epsilon \quad (R5) \]

\[ \text{eval}^I : \Sigma \times P^\times \rightarrow \epsilon \cup (\{\bullet, \triangleright\} \times P^\times) \]

\[ \text{eval}^I(\sigma, I_L + I_R) = \begin{cases} 
\text{eval}^I(\sigma, I_L) & \text{if } \text{eval}^I(\sigma, I_L) \neq \epsilon \\
\text{eval}^I(\sigma, I_R) & \text{otherwise} 
\end{cases} \quad (R6) \]

\[ \text{eval}^I(\sigma, \sigma_1. P) = \begin{cases} 
(\bullet, P) & \text{if } \sigma = \sigma_1 \\
\epsilon & \text{otherwise} 
\end{cases} \quad (R7) \]

\[ \text{eval}^I(\sigma, \sigma_1 \triangleright P) = \begin{cases} 
(\triangleright, P) & \text{if } \sigma = \sigma_1 \\
\epsilon & \text{otherwise} 
\end{cases} \quad (R8) \]
**Interaction Evaluation**  Rules (R6) to (R8) (relating to the function evalI), given in Definition 7.6, provides the meaning of evaluation in protocol mode interactions. We have to consider the three syntactic categories choice, standard interaction, and contractive interaction.

The rules for standard interaction and contractive interaction define the fundamental building blocks of the expressive protocol models evaluation semantics. These rules are analogous to the transitions of the DFA protocol based model from Chapter 6 and essentially check whether the origin-message shape pair submitted for evaluation are valid in the protocol model, and if so, directions for what the next state of the protocol model should be.

The rule for standard interaction (R7) evaluates an origin-message shape pair σ, in the standard interaction σI,P. If σ is equal to σI, then σ is considered to be a conformant reception or transmission in the given context, and the rule generates the annotated protocol continuation (p, P) indicating a successful match at a standard interaction. If σ and σI differ, then a failure is signalled e. Similarly, a successful match in a contractive interaction (R8) will generate (>, P) if the origin-message shape pair σ matches the expected message σI. In any other case failure is again returned.

Finally, in the case of interaction choice (R6), a successful match is obtained if at least one of the left or right branches signals a positive match; the left being tested first and then the right. The first left-most interaction to find a successful match will have its annotated protocol continuation returned as the result, the other choices being discarded.

**Protocol Evaluation**  The meaning of evaluation for expressive protocol models are given in Definition 7.6 by the rules (R1) through to (R5), covering the corresponding five syntactic categories. Unlike the rules for interaction evaluation, protocol evaluation requires the context, see Definition 7.4, so that any variables encountered during evaluation can be resolved.

The rules for the syntactic categories inaction and interaction are straightforward. The 0 protocol specification considers every possible origin-message shape pair as invalid and, as a consequence, evaluation always result in a failure e (R5). Evaluation for interactions at the protocol level simply wraps around the evalI operation (R4). In doing so, the context is withheld from the underlying interaction evaluation operation as it is not required. Upon a successful match in the interaction evalI(σ, Iσ) ≠ e, the context is reunited with the protocol continuation and paired with the annotation to form the evaluation result (aI, (ξ, P'))).

(R3) defines evaluation for variables. In essence, a variable is looked up in the context ξ and the bound protocol model is used to proceed with the evaluation process. Due to the fact that evaluation is only defined on evaluate-able and therefore closed protocol models, no error can occur when calling lookup-in-context making it safe to call directly.

Of more interest are the rules for product composition and protocol extension, respectively. Similar to the rule for choice, evaluation of a product composition succeeds if the origin-message shape pair σ is valid in either of its two branches. If this is the case, the continuation protocol is defined as the product composition of the continuation of
the matching branch with the original protocol model of the non-matching branch (the topmost two cases of \((R1)\)). The annotation of the successful match is preserved in this process. The remaining case deals with failed matches in both branches, resulting in failure \(\epsilon\).

The evaluation rule for protocol extension is similar to the one for product composition. The main difference is that the two types of successful matches • and \(\triangleright\) in the left branch (i.e. the base protocol) are treated differently. In the first case, a standard interaction match, the continuation protocol generated by the successful match is re-extended with the existing extension in the right branch (i.e. the extension protocol). In the second case however, a contractive match, the extension protocol is discarded, and only the continuation protocol generated by the successful match is used. This is how subservient (child) protocols can be terminated prematurely by their master (parent) protocol.

Treating these two cases differently is the main reason why the message evaluation rules for interaction and protocol specifications generate annotated protocol continuations; we need to be able to distinguish between the continuation generated by \(\sigma.P\) and the one generated by \(\sigma\triangleright P\).

Evaluation in both compound protocols and compound (choice) interactions has a left bias; the first matching interaction towards the left will always be taken as the result. Any interactions which would also match occurring to the right of the successful match are essentially ignored. Although if those interactions appear to the right in a compound protocol they may get another chance to match in future.

Next

Definition 7.6 provides the core of the semantics for the expressive protocol model. We can now build atop of the provided evaluation rules to fit the expressive protocol model within the interaction modelling framework. To this end, we provide the definition of next for our expressive protocol model in Definition \[7.7\]

**Definition 7.7 (Next for an Expressive Protocol Model):**

\[
\text{next}_x : \Sigma \times P^x_\Sigma \rightarrow \epsilon \cup P^x_\Sigma
\]

\[
\text{next}_x(\sigma, p) = \begin{cases} 
p' & \text{if } (a, p') \in \{•, \triangleright\} \times P^x_\Sigma, \ (a, p') = \text{eval}(\sigma, p) \\
\epsilon & \text{otherwise}
\end{cases}
\]

As can be seen in Definition \[7.7\], the \(\text{next}_x\) operation is a straightforward wrapper around the expressive protocol model evaluation function eval. In the case that a successful match is made, then the annotation is stripped from the result (there is no need for it by any caller), and the evaluate-able protocol model \(p'\) is returned. In any other case, the origin-message shape pair is not protocol conformant and therefore no next protocol is known, leading to the error as the result.
7.14 Emulation

Considering implementation, the abstract syntax and evaluation rules provided in this section provide suitable foundations for input to standard compiler-compiler tools such as lex and yacc [Joh75]. The result however, is significantly more complex than its rudimentary counterpart. As such the expressive protocol model is not as straightforward to store and emulate. Unlike the DFA based model, the size of an individual protocol model is unbounded. Extensions and contractions mean that an expressive protocol model is capable of growing and shrinking at run-time, making it challenging to place any hard limits on memory use. Furthermore, naive execution requires walking the protocol model until a matching interaction is found, which in the worst case (no valid interactions), takes linear time $O(n)$ to complete, where $n$ is the number of valid interactions in the protocol model at the time.

Despite these concerns, we have found in practise that the expressive protocol model scales adequately enough for the majority of practical cases. The expressive behaviour and data store layers play a more significant role in terms of scalability. Regardless, scalability of the expressive protocol model can be improved if necessary. Run-time performance may be aided through an ancillary map with constant retrieval time containing all the valid interactions thereby speeding up evaluation. Memory use may also be made more efficient by sharing equivalent protocol aspects, such as interactions, across many emulated endpoints. Such optimisation strategies, although unnecessary for the moment, may become useful in future work as the upper limits of emulator scalability are reached.

7.2 Behaviour

Lack of more sophisticated request handling is a significant limitation of the rudimentary interaction stack. Although responses in the rudimentary stack are protocol conformant and have a certain degree of message content consistency, the facile focus on message exchange and disregard for any deeper endpoint behavioural semantics limits the range of scenarios in which rudimentary endpoints specifications are of use. In particular, being able to interact as though an external system’s modifications have been persisted, is necessary and cannot be achieved using the rudimentary model of endpoint behaviour.

To enable a richer, more sophisticated variety of emulated endpoint interactions, we detail in this section, the expressive behaviour layer. Unlike it’s rudimentary counterpart, the expressive behaviour layer supports more realistic processing of requests through specification of handler functions which manage processing for requests of specific message shapes. Combined with a persistent underlying value store, these handlers allow emulated interactions to reflect modifications made by external systems in future transmissions; modifications can be independently verified by external endpoints.
7.2.1 Shape Specific Handlers

The core idea behind request processing in the expressive behaviour layer is associating specific handler functions with specific request message shapes. Enterprise endpoints specifiers using the expressive interaction stack are required to provide definitions for each handler of each different kind of message shape which may be received during emulation.

Definition 7.8 provides the signature for shape specific handler operations, named handle-shape. These handlers are functions which map from pairings of value sequences, being the contents of the request message, and expressive data stores, yielding either an error, or the possibly empty sequence of response messages paired with the updated expressive data store. This definition allows a significant degree of flexibility in individual handler specification. Any function matching the interface being supported by the expressive behaviour model. This flexibility allows specification writers to model interaction behaviour up to whichever degree of accuracy is appropriate for their particular scenario. Furthermore, the inclusion of the expressive data store in the co-domain means that modification requests can be persisted by an emulated endpoint and independently verified subsequent queries made by external software systems.

Definition 7.8 (Shape Specific Handlers):

\[ \text{handle-shape} : \mathcal{V} \times \mathcal{D}^x \rightarrow \varepsilon \cup (\mathcal{M} \times \mathcal{D}^x) \]

Where \( \mathcal{D}^x \) denotes the set of all expressive interaction stack data stores.

Sample Specification

To help illustrate the handler concept we provide, in Definition 7.9, the specification for an LDAP Add Request handler. This handler represents an approximation of a real LDAP server’s add request handling. It does not deal with proper handling of schemas. It does however, ensure that the entry being added does not already exist and that a parent entry does exist. Provided these conditions are met, the add request handler provided in Definition 7.9 persists the addition of the entry; subsequent search operations will be able to locate the added entry.

The add request handler takes as input a sequence of values and an expressive LDAP directory endpoint data store. It produces either an error or a message sequence, paired with the updated LDAP directory data store as the result.

We specify the add request handler in four cases. The first three cases represent behaviour for valid, although perhaps unsuccessful, add requests. The bottommost case returns error as the add request supplied was invalid. This bottommost case covers any situation wherever the structure of the supplied message does not conform to expectations. If, for example, the first value of the sequence is not an associated value with a label of messageID.
Definition 7.9 (LDAP Add Request Handler):

\[
\text{hdlAddRq} \in \text{handle-shape} \\
\text{hdlAddRq} : [V] \times D_{\text{LDAP}}^x \to \epsilon \cup ([M] \times D_{\text{LDAP}}^x)
\]

\[
\text{hdlAddRq}(\text{vs}, d) = \begin{cases} 
(\text{vs}_\text{v}, \text{AddRes}), d' & \text{if } \neg \text{entry-exists} \land \text{parent-exists} \\
(\text{vs}_\text{x}, \text{AddRes}), d & \text{if } \text{entry-exists} \\
(\text{vs}'_\text{x}, \text{AddRes}), d & \text{if } \neg \text{parent-exists} \\
\epsilon & \text{otherwise}
\end{cases}
\]

where \( \text{vs} = [(\text{messageID} : \text{msg-id}), (\text{protocolOp} : (\text{addRequest} : [ \\
(\text{entry} : \text{entry-dn}), (\text{attributes} : \text{entry-attrs}))]))] \)

\( \text{vs}_\text{v} = [(\text{messageID} : \text{msg-id}), (\text{protocolOp} : (\text{addResponse} : [ \\
(\text{resultCode} : 0), (\text{matchedDN} :")", \\
(\text{diagnosticMessage} : "")]))] \)

\( \text{vs}_\text{x} = [(\text{messageID} : \text{msg-id}), (\text{protocolOp} : (\text{addResponse} : [ \\
(\text{resultCode} : 68), (\text{matchedDN} :")", \\
(\text{diagnosticMessage} : "Entry Already Exists")]))] \)

\( \text{vs}'_\text{x} = [(\text{messageID} : \text{msg-id}), (\text{protocolOp} : (\text{addResponse} : [ \\
(\text{resultCode} : 16), (\text{matchedDN} :")", \\
(\text{diagnosticMessage} : "Missing Parent")]))] \)

\( \text{entry-exists} = \exists \text{entry-dn} : (\text{entry-dn}, \text{attrs}) \in d \)

\( \text{parent-exists} = \exists \text{parent-of(} \text{entry-dn} \text{)} : (\text{parent-of(} \text{entry-dn} \text{)}, \text{attrs}) \in d \)

\( d' = (\text{entry-dn}, \text{entry-attrs}) \cup d \)

Where \( D_{\text{LDAP}}^x \) denotes the set of expressive data stores for LDAP endpoints. We model these LDAP data stores sets of pairings between distinguished names (DN) and attribute sequences. The parent-of operation is a function which is able to calculate the parent DN of an entry when supplied its DN.
All three non-erroneous cases return a single message wrapped as a sequence, paired with the potentially modified, LDAP directory data store. The result of a valid add request is always a single add response. By valid we mean to say that the bind request is structurally well formed (according to our expectations). The result of a valid add request may actually indicate the failure of the add operation which are in fact evidenced by the second and third listed cases. The topmost case meanwhile, covers the valid and successful application of the add operation.

We use a number of symbolic assignments in Definition 7.9 to aid the specification. We deconstruct the value sequence of the add vs so that the parameters can be used in specifying the handler. We use msg-id to refer to the request’s message identifier, entry-dn to refer to the DN of the entry being added and, entry-attrs for its attribute sequence. For valid add requests we assign three different value sequences: vs for the successful add, vs_chi for the unsuccessful add due to the entry already existing, and vs_chi for unsuccessful add due to a missing parent entry. We use entry-exists and parent-exists to denote whether the entry being added and its parent exists in the LDAP data store. Finally, d0 is used to refer to the LDAP data store after successful addition of the entry.

Whether the entry being added currently exists in the data store, as well as its parent, is checked by looking for entries in the data store with matching distinguished names (DN). We assume that the expressive LDAP data store d is a set of pairings between DNs and attribute sequences. We also assume the existence of the parent-of operation which, when supplied a DN, is able to calculate the parent DN. In practise, the parent-of is straightforward to implement, it essentially being a split of the DN string, as such we omit full formal definition here.

The updated LDAP data store d0 is defined as the union of the entry supplied as part of the request with the original data store passed in as the second argument. In the case that the entry does not already exist, and a parent does exist then this updated data store is returned as part of the result, thereby persisting modifications. If on the other hand, the entry already exists or a parent entry cannot be found, the data store remains unmodified so that it can maintain consistency.

7.2.2 Request Handling

Shape specific message (request) handlers are the fundamental underpinning which enables rich specification of server-class endpoint behaviour in the expressive interaction stack. With the shape specific handlers concept fully defined it is now relatively straightforward to define the handle-rq operation for the behaviour layer of the expressive interaction stack and is presented in Definition 7.10.

The expressive request handler is structurally very similar to the rudimentary presented earlier in Definition 6.12. It is organised into three cases, the topmost dealing with requests which do not trigger and responses, the middle case dealing with requests which trigger one or more responses and the bottommost case catching all other situations, returning error. We again use a number of local symbolic assignments and a couple of local functions: the transmit-c and next-out operations being exactly the same as those...
Definition 7.10 (Expressive Handle Request Operation):

\[
\text{handle-rq}^X : \mathcal{M} \times \mathcal{C} \times \Pi^x \rightarrow e \cup \Pi^x
\]

\[
\text{handle-rq}^X(m, c, \pi) = \begin{cases} 
\text{set-protocol}(c, p', \pi') & \text{if } \{p, p'\} \subseteq P^x, \, rs = [], \\
\text{set-protocol}(c, p'', \pi') & \text{if } \{p, p', p''\} \subseteq P^x, \\
rs = [r_1, r_2, \ldots, r_i] & \text{fold-err}(\text{transmit-c}, r_1, \\
& [r_2, \ldots, r_i]) \neq e & \text{otherwise}
\end{cases}
\]

where \( p = \text{protocol}(c, \pi) \)

\[
\text{hdl} = \text{handler}(\text{msg-shape}(m), \pi) \\
p' = \text{next}(?, \text{msg-shape}(m), p) \\
(rs, d') = \text{hdl}(\text{msg-vals}(m), \text{data-store}(\pi)) \\
np'' = \text{fold-err}(\text{next-out}, r_1, [r_2, \ldots, r_i]) \\
\pi' = \text{set-data-store}(d', \pi)
\]

\[
\text{transmit-c}(e, r) = e \\
\text{transmit-c}(?, r) = \begin{cases} 
\text{transmit}(r, c) & \text{if } r \in \mathcal{M} \\
e & \text{otherwise}
\end{cases}
\]

\[
\text{next-out}(e, r) = e \\
\text{next-out}(p'', r) = \begin{cases} 
\text{next}(!, \text{msg-shape}(r), p'') & \text{if } r \in \mathcal{M} \\
e & \text{otherwise}
\end{cases}
\]

Where \( p \) and \( p' \) denote the protocol of channel \( c \) before and after reception, respectively. The response to the request is represented as a sequence and denoted \( rs \) and the protocol model of the channel after transmission of this response sequence is denoted \( p'' \). The two local functions \( \text{transmit-c} \) and \( \text{next-out} \), serve as wrappers which facilitate invocation of the \( \text{fold-err} \) higher-order function on sequences. We use \( \text{hdl} \) to denote the shape specific handler function that will be invoked and \( d' \) to denote the endpoints data store after the request has been handled. Finally, \( \pi' \) denotes the expressive endpoint after successful request processing.
used previously in Definition 6.12. We use $p$, $p'$ and $p''$ to denote the protocol model before reception of the request, after reception, and after transmission of the calculated response sequence.

The chief difference between \texttt{handle-rq} and \texttt{handle-rq} is in the way requests are processed. The expressive behaviour layer handles requests by dispatching them to the appropriate, specific handler function, specified for processing requests of that shape. This shape specific handler function is denoted in Definition 7.10 $\texttt{hdl}$ and is located by invoking the handler operation, passing in the message shape of the current request and the corresponding endpoint. The request is then dispatched along with the current state of the data store to this shape specific handler function which if successful will result in a pair consisting of the, possible empty, response sequence, and possibly updated, expressive data store.

### 7.2.3 Emulation

Similarly to emulation of its rudimentary counterpart, emulation of the expressive behaviour layer can be achieved by encoding the defined operations as procedures, functions or subroutines as appropriate in some host programming language. There is however, additional work involved in encoding the expressive behaviour layer due to the presence of shape specific request \textit{handlers}. Each of these handlers need to be encoded as some executable routine which may be invoked by the emulator at run-time. Perhaps the most straightforward approach to achieving this is to encode each handler using the same host language as the emulator itself. This makes dispatching requests as simple as calling the routine associated with the message shape of the request which has been received. More sophisticated approaches however, may allow handlers to be encoded in whichever host language is most appropriate, and presented to the emulator as external local libraries or even remote libraries provided through web services or RPC.

Unlike the rudimentary behaviour layer, we cannot state with much certainty, the potential scalability of the expressive behaviour layer as it depends primarily on the memory and time efficiency of the specific shape handlers which are used. There are however, like the rudimentary behaviour layer, certain costs associated with validating protocol conformance of receptions and transmissions, and with transmitting the response messages themselves. These costs will be the same as those incurred in the rudimentary stack. Therefore, differences in scalability between the stacks will come down to the time efficiency of the shape specific handlers and the memory needs of the endpoint data stores which in the expressive stack persist and change over the course of an emulation.
The purpose of the data store layer of the expressive interaction stack, like its rudimentary counterpart, is to provide storage and retrieval facilities to the behaviour layer so that interaction between an emulator and other external systems may occur. As we have stated previously, this is the with of endpoint interaction specification; the behaviour layer may take the content of the data store into consideration when handling requests from external endpoints. The key difference between the expressive and rudimentary data store layers lies in the content and persistency of their data stores. Whereas the primary concern of the rudimentary data store was retrieval mechanisms for message skeleton sequences, helping to resolve output choice ambiguity, the primary concern of the expressive data store is storage and retrieval of endpoint specific information necessary for the behaviour layer to handle requests and persist modifications across an emulation.

Definition 7.11 provides the formal definition of the set of all expressive data stores. These data stores contain any and all information required by the various shape specific request handlers which is not present in the value sequences of the requests themselves. Unfortunately, it is not possible to discuss any general operations on these data stores as these are entirely dependent on the requirements of the endpoint being modelled. We can however, discuss the structure of the expressive data store for a specific endpoint and do so in the following Section 7.3.1.

Definition 7.11 (Expressive Data Store):

\[ D^x \subseteq D \]

Where \( D^x \) denotes the set of all expressive interaction stack data stores.

7.3.1 LDAP Directory Data Stores

We used an expressive LDAP directory data store previously in Definition 7.9 to specify the add request handler of an expressive LDAP endpoint. Now, in Definition 7.12, we formally define the LDAP directory expressive data stores, used to define the add request handler, to illustrate the structure of expressive data stores that can be used in practise. We define expressive data stores of LDAP directory endpoints as sets of pairings between distinguished names and attribute sequences. This is essentially a map where distinguished names play the part of the key, and the attribute sequences are the values. This may seem counter-intuitive, as LDAP directories are typically thought of as hierarchical tree based structures however, the nesting is achieved in our model by through operations on the distinguished names of LDAP entries. In practise, each distinguished name essentially acts as a reference not only to entry itself, but to its position within the directory hierarchy.
**Definition 7.12 (LDAP Directory Expressive Data Store):**

\[ \mathcal{D}_{\text{LDAP}}^x \subseteq \mathcal{D}^x \]
\[ \mathcal{D}_{\text{LDAP}}^x = 2^{\mathcal{DN} \times [\text{ATTR}]} \]

Where \( \mathcal{D}_{\text{LDAP}}^x \) denotes the set of all LDAP directory expressive data stores, \( \mathcal{DN} \) denotes the set of all LDAP distinguished names and, \( \text{ATTR} \) denotes the set of all LDAP attributes.

### 7.3.2 Emulation

The data store layer of the expressive interaction stack can be emulated in much the same way as its rudimentary counterpart. In terms of the operation time efficiency, provided a good implementation of the fundamental data structures are available, i.e. binary trees, hash maps, etc. is available, each expressive data store operation may be completed efficiently, or as efficiently as theoretically possible using the underlying data structures. Meaning that the scalability of an emulator using the expressive data store layer, in terms of its execution speed, is likely to be satisfactory no matter the endpoint systems being emulated simultaneously, so long as the data store is mapped onto an appropriate underlying structure.

Unfortunately memory scalability is more difficult to gauge. Depending on the endpoint being emulated, the data store supporting interaction behaviour may need to be quite large. If, for example, an LDAP directory containing one million entries was intended to be emulated, then the corresponding data store may occupy quite a lot of main memory. This problem is clearly exacerbated when many such systems are required to be emulated simultaneously. Much of the time however, such large data stores are unnecessary. Even so, efficient storage of such large stores across emulated endpoints is an area for potential future work.
7.4 Discussion

In this chapter we have presented an expressive interaction stack capable of modelling rich enterprise endpoint interactions. Each layer of this stack helps to address limitations in their rudimentary interaction stack counterpart. The expressive protocol model described in Section 7.1 introduces concurrency primitives so that the temporally asynchronous nature of many enterprise endpoint protocols can be accurately represented. The expressive behaviour layer in Section 7.2 introduces message shape specific request handlers so that, to a certain extent, more realistic request processing can be modelled and subsequently emulated. Finally, the expressive data store layer described in Section 7.3 introduces persistency so that modifications requested by external systems can in fact be applied and thereby reflected in subsequent interactions. Collectively these expressive layers substantially improve the range of enterprise endpoint interactions which may be modelled and consequently improves the range of scenarios in which emulation is a viable approach to providing run-time interactive closure environments for open enterprise software systems.

This increased expressivity, however, has its price. In most cases specification of enterprise endpoint interactions using the expressive stack requires more human effort than specification using the rudimentary stack. Although interaction specifiers are free to provide as much or as little detail deemed necessary within the handlers and generators themselves, a request handler is still necessary for each expected request message shape. In addition to the heightened specification effort, the computational resources required to execute expressive interaction stack specifications has, most likely, increased. Time performance of the expressive behaviour layer is not merely bound to simple lookup operations as is the case in the rudimentary stack. Rather it is bound to the time performance of the request handlers which are essentially outside of the emulator’s control. Similarly, the memory requirements for an emulator using the expressive interaction stack are bound to the size of the data stores of the emulated endpoints. As a consequence of the decrease in time and memory performance, the scalability of an emulator using the expressive interaction stack is unlikely to be as high as is possible using the rudimentary stack. In practise however, and as evidenced by the experiment results detailed in Chapter 10, this reduction in scalability is largely negligible; an emulator employing the expressive interaction stack can still typically scale to the extent that many thousands of endpoint interactions are emulated simultaneously.
Chapter 8

Emulator Architecture and Implementation

As a proof of concept we describe in this chapter an architecture and reference implementation of an enterprise software environment emulator capable of providing interactive behaviour characteristic of enterprise software environments. The resulting implementation is able to interact with external software systems, in real(istic)-time, based on specifications defined using either the rudimentary or expressive model stacks described in Chapters 6 and 7, respectively. Although currently, only these two model stacks are supported by our emulator implementation, the architecture on which it is based allows for additional stacks to be incorporated as they become available.

8.1 Architecture

We present in Figure 8.1 a foundational architecture for an enterprise software environment emulator. This architecture is foundational in that it does not provide all the auxiliary components which may be desired in an emulator, such as, run-time monitoring and visualisation components. It does, however, highlight the fundamental software elements of such an emulator as well as their relations to one another. The external systems (ES) appear in the left of the diagram and represent the real world software systems which the emulator is intended to interact with and thereby enable their observation. The emulator itself is segregated into two principal components: the network interface which as its name implies is responsible for the network communication with the external systems, and the engine which is responsible for the observable behaviour of the emulated endpoints. Each endpoint in the engine is initialised according to an interaction specification which has been defined using one of the model stacks supported by the engine. The configuration module, as its name suggests, is responsible for the initialisation and configuration of the emulator.
Figure 8.1: Architecture for an Enterprise Software Environment Emulator
This architecture treats the external software systems (ESs) as closed systems. We expect that these systems may be configured so that they are directed to interact with the emulated endpoints, beyond this however, we make no assumption regarding their control. As shown in Figure 8.1 we place no restriction on the number of these external systems and it is indeed possible for the emulator to interact with many external systems simultaneously.

8.1.1 Network Interface

An enterprise software environment emulator needs to be able to actually interact with distributed systems connected to the emulator by some computer network. This is the purpose of the network interface component of our architecture. This component allows communication with external (distributed) software systems in a manner which is native to those external systems. By native we mean that communication between the emulator and the external systems must occur in the manner expected by those external systems; the messages must be encoded on-the-wire according to the native formatting requirements of the external system. Consider, for instance, the situation where the external system to be observed is an LDAP client. This LDAP client expects to interact with an LDAP server by establishing a network socket with that server, over which it will send and receive ASN.1 encoded LDAP messages. The network interface facilitates this behaviour by acting as a kind of two-way translator between the native messages transmitted over the communication infrastructure and the abstractions and primitives which can be interpreted by the emulator.

There are two key components of the network interface: the native services which allow external systems to establish native communication channels for communication with the emulator, and the conduits which associate network channels with emulated channels as well as translate between the native message encodings required by the network (native) channels and the message primitives understood by the emulator’s engine.

Each native service is bound to a specific IP address and port number at which it listens for incoming connection requests that are initiated by external systems. This allows an external system to establish a network communication channel with the emulator in the same manner that it would a real environment service. Upon receiving such a request the native service notifies the corresponding service of the engine that a new channel has been established passing in all relevant details. It also triggers the construction of an appropriate conduit which will handle the communication of actual messages between the external system and the emulator.

Each conduit is responsible for the exchange and transmission of messages on a single native channel and the corresponding channel abstraction provided by the engine. Upon receiving a native message from an external system it must decode that message into the message structure which can be interpreted by the engine. Once this is complete the conduit passes this message on to the corresponding engine channel for further processing. Likewise, when a message is received from the conduit’s engine channel, the conduit must encode the message into the native format and transmit it to the external
system using the native channel. The network interface conduits are dyadic; linking one and only one external system to one and only one engine channel.

The operation of an LDAP conduit is depicted in Figure 8.2. The native format of messages exchanged between LDAP clients and servers is an ASN.1 encoding. The LDAP conduit must be able to decode these native messages into LDAP messages which the engine is capable of reasoning about. These LDAP messages are freely transmitted between the conduit and the underlying engine channel with which the conduit is associated. LDAP messages received from the engine channel are encoded by the conduit as appropriate for transmission on the ASN.1 channel and transmitted to the external software system using the native ASN.1 channel.

![LDAP Conduit Example](image)

Figure 8.2: LDAP Conduit Example

### 8.1.2 Engine

The engine provides the core logic behind the emulator’s interaction behaviour. It is this component which emulates endpoint interactions based on specifications defined using some model stack supported by the emulator, such as, for example, the rudimentary or expressive stacks described earlier in this work. The engine’s role is essentially to interpret these specifications, perhaps many thousands of them simultaneously, providing the runtime interactive behaviour of the emulator. This ability of the engine to interpret many thousands of interaction specifications simultaneously is key to enabling representation of large scale enterprise software environments using a single physical machine as a host.

The endpoints of the engine correspond to the conceptual endpoints described throughout this dissertation and is comprised of five kinds of entities. (i) An endpoint can have zero or one (indicated by the question mark superscript) services, which represents a point at which channels can be established by some other, external endpoint. (ii) An endpoint is able to have zero or more channels which represent the communication mechanism that carries messages between emulated endpoints and external systems. (iii) Each service and channel associated with an endpoint must in turn, be associated with a protocol. The protocol of a channel will change over the course of an interaction, as denoted by the double ended arrow, whereas the protocol of service will remain unchanged throughout an emulation. (iv) The behaviour of an endpoint represents the way in which requests are handled and what responses are generated. (v) Finally, the data store of an endpoint is used by the behaviour entity to help populate contents of response messages with valid data, and perhaps store changes to that data. Essentially, each engine endpoint encapsulates and maintains the interaction behaviour of a particular enterprise software
environment endpoint, throughout an emulation.

Although the endpoint entity is responsible for the engine’s observable behaviour during an emulation, this behaviour is ultimately based on an underlying specification which has been defined using some interaction model stack which is supported by the engine. As described earlier in Chapter 8.1 Section 5.2.3 protocol, behaviour and data store layers of our interaction modelling framework are typically developed in conjunction with one another, aiming at a specific approach or level of approximation for endpoint interaction behaviour. A stack is a protocol, behaviour, and data store model, intended for operation with one another, they are essentially a convenient abstraction to collect together compatible, concrete interaction layers. Specifications define a particular instance of a stack. These specifications are used to define endpoints of a particular kind, such as, for example, an LDAP Directory server, or a BitTorrent peer. A specification essentially provides the bulk of the endpoint specific values required to instantiate a model which is emulate-able, i.e. a specification is used to instantiate each of the engine’s endpoints.

Each endpoint service provided by the engine is associated with a native service in the network interface. It is the responsibility of this native service to notify the endpoint service whenever a new connection is established by some external software system. When called, an endpoint service creates a fresh channel on which the messages being exchanged with the external software system will be carried. This channel will have its protocol model set to the initial state which is held by the service.

Endpoint channels are the abstractions on which the emulated interactions actually occur. These channels carry messages back and forth between the network interface and the engine, invoking the appropriate behaviour whilst also maintaining the current “state” of the protocol model.

The behaviour component of the engine endpoints corresponds to the behaviour layer of the corresponding model stack. This behaviour describes how to process messages arriving on channels and what sequence of messages to generate in response. It uses the current state of the channel’s protocol model along with the specification’s data store to produce the individual response messages and, in the case of an expressive model stack specification, a modified data store.

The data store component of a specification corresponds to the data store layer of the corresponding model stack. This data store maintains the data to which the specification’s behaviour may refer in deciding how to handle requests and generate messages. In the case of a rudimentary specification this is fixed, in the expressive, however, this data store may change over time depending on previous interactions.
8.1.3 Configuration

The configuration component, as its name suggests, facilitates the initialisation and configuration of the emulator. It encapsulates all the necessary configuration for the emulator for both the network interface and engine components. This includes which interaction specifications should be emulated by the engine and how many instances of them should be provided. It also includes what native services to provide in the network interface and to which IP address and port number these services should be bound.

8.2 Kaluta

Kaluta is an enterprise software environment emulator we have constructed as a proof of concept of the emulation concept in general and the architecture described in the previous section. Kaluta supports interaction specifications defined using either the rudimentary or expressive model stacks and therefore can, and will in Chapter 10, be used to investigate the difference in computational resource requirements of these two model stacks. The network interface and engine components of Kaluta are separate programs, written in different general purpose languages, which communicate with one another using the Apache Thrift based remote procedure call framework [thr].

The network interface component of Kaluta is written in the Java programming language [GJSB05]. We selected Java as the implementation language of the network interface primarily due to the large number network communication libraries available on the platform. The availability of such libraries is critical to ensure interaction can be supported for a wide variety of different kinds of external systems.

Thrift remote procedure call (RPC) was used for the inter-component communication between our Java based network interface and Haskell based engine. The Thrift interface definition language was used to define the specific message structures, shapes and service interface for each interaction specification. This thrift definition is subsequently used to generate the boilerplate inter-process communication code for the Java network interface and the Haskell engine. Essentially, the Haskell engine provides Thrift RPC services which act as the front-end to the emulated interaction specifications. These RPC services are invoked by the native service and conduits of the network interface to “interact” with the emulated specifications through the engine’s RPC interface.

Configuration of Kaluta is achieved using a comma separated value (CSV) file which lists the interaction specification configurations to be used in an emulation. Each record of the CSV file containing the IP address, port number, model stack, endpoint type and data source of a single interaction to be emulated. This configuration file is subsequently read by both the network interface and engine as the basis for initialising the relevant structures. Interested readers can find a sample configuration file in Appendix A, Section A.1, Listing A.1.
Engine

Kaluta’s engine component is written in the Haskell programming language [OSG08], a pure functional programming language. We selected Haskell as the implementation language for our engine because the functional flavor of the language allowed the interaction specification encodings to be similar to their formal definitions. Haskell also provides language level light-weight threads with reasonable performance characteristics. This allowed us to use a simple execution model for the specification; one light-weight thread for every specification is much simpler than the complexities inherent in event-based execution models. Furthermore, Haskell has good facilities for the construction of embedded domain specific languages, known in the Haskell community as combinator libraries. This feature may be quite valuable in the future as it can enable encoding of interaction specifications using a domain-specific vocabulary, rather than by encoding specification using the host language directly.

The engine is where the majority of the formal models used in this thesis are encoded. As mentioned, Haskell makes it possible for these encodings to be similar to their formal definitions, which we now demonstrate through a series of examples.

Firstly we present, in Listing 8.1 the encoding of protocol model operations defined earlier in Chapter 5, Definition 5.3. Message origin is defined by the data type Origin and like Definition 5.3, assigned the symbols ? and !, for input and output, respectively. Error is encoded as ProtocolErr which has a String, a pairing of an Origin and message shape (ms), and a protocol (p) which can be used for diagnosis purposes. The protocol model operations are encoded in the ProtocolModel type class. The Haskell encodings of these operations are curried rather than taking Cartesian products. This representation is more idiomatic of Haskell code, is more flexible than the alternative, and does not cause any significant divergence from the formal definition. The union result of the next function is encoded using the Either type provided by the Haskell standard library, known as the Prelude. As is convention, the left type of the Either type is the error, while the right is the non-erroneous, i.e. correct type.
Listing 8.1: Haskell Encoded Protocol Model

```haskell
data Origin = Input | Output deriving (Eq, Ord)

instance Show Origin where
  show Input = "?"
  show Output = "!"

data ProtocolErr p ms =
  ProtocolErr String (Origin, ms) p deriving (Eq, Show)

class ProtocolModel p ms | p -> ms where
  next :: Origin -> ms -> p -> Either (ProtocolErr p ms) p

isValid :: Origin -> ms -> p -> Bool
isValid origin msg_shape prot =
  case next origin msg_shape prot of
    Right _ -> True
    otherwise -> False

validOutputs :: p -> Set.Set ms

finished :: p -> Bool
```

The valid operation is encoded as the `isValid` function and it’s parameters, \( o, \sigma, p \), map to origin, msg-shape and prot, respectively. Validity \( \checkmark \) and invalidity \( \chi \) are encoded by the built-in `Bool` type and represented as `True` and `False`, respectively. The Haskell encoding looks for the next protocol in a case statement and if the result is a `Right`, indicating success, returns `True` to indicate this validity. Otherwise, `False` is returned to indicate invalidity.

The protocol model operations, `finished` and `validOutputs` are given type signatures as functions `finished` and `validOutputs` respectively, but are not assigned default implementations. The default implementations provided by Definition 5.3 were impractically slow and have been omitted so that concrete protocol models are able to implement these functions in a more performant manner.

We now move onto the Haskell encoding of the rudimentary protocol model detailed earlier in Section 6.1 of Chapter 6. The structures are provided in Listing 8.2 encode Definition 6.1, while the operations are given in Listing 8.3 and encode Definition 6.2, as well as implementations of the `finished` and `validOutputs` operations which were omitted from Listing 8.1. Before going into details it is worth mentioning that the encoding presented is not very efficient. In practise this encoding is translated into a more efficient, map based, structure prior to operation. The presented encoding is, however, quite similar to the corresponding formal definitions which is the reason we describe it, rather than introduce new concepts required to describe the efficient map based version.
Listing 8.2: Haskell Encoding of Rudimentary Protocol Model Structures

```haskell
data AlphabetElement ms = AlphabetElement { 
    elemOrigin :: Origin , 
    elemMsgShape :: ms } deriving (Eq)

data DFAProtocolMdl ms s t = 
    DFAProtocolMdl { 
        stateSet :: Set.Set st , 
        alphabet :: Set.Set ms , 
        transFunc :: st -> AlphabetElement ms 
            -> Either (ProtocolErr 
                (DFAProtocolMdl ms st ) 
                ms ) st , 
        currState :: st , 
        finalStateSet :: Set.Set st }
```

The Haskell encoding of the structures of the rudimentary protocol model are presented in Listing 8.2 and can be interpreted as a encoding of Definition 6.1. The AlphabetElement data type encapsulates elements of the DFA’s alphabet $\Sigma$ as a record containing an Origin and a message shape ms accessed by functions elemOrigin and elemMsgShape, respectively. Rudimentary protocol models $P^R$ are encoded by the DFAProtocolMdl data type which, like AlphabetElement, is a record. The state set $Q$, alphabet $\Sigma$, transition function $\delta^R$, current state $q$ and final states $F$ of a protocol model are accessed by the functions stateSet, alphabet, transFunc, initState, currState, finalStateSet, respectively. The Either data type is again used for union of results, left being erroneous, and right being non-erroneous, i.e. valid or correct (right). Set.Set maps to the efficient set implementation provided by Haskell’s Data.Set library the details of which are not particularly important for our description of the encoding.
The Haskell encoding of the operations of the rudimentary protocol model are presented in Listing 8.3 and encode Definition 6.2 as well as implementations of the finished and valid-outputs operations which were omitted earlier. The encoding is implemented as an instance of the ProtocolModel type class defined in Listing 8.1. The next operation binds the origin \( o \), message shape \( \sigma \) and protocol model \( p \) parameters to \( o \), \( \sigma \), and \( p \), respectively. Within the protocol binding the DFAProtocolMdl constructor is unwrapped to bind the transition function \( \delta_p^{\sigma} \) and current state \( q \) to \( trans \) and \( currState \), respectively. The transition function is then applied in the case statement \( \text{trans st (AlphabetElement o ms)} \) and if successful returns the protocol model with the updated state wrapped in the Right constructor to indicate success: \( \text{Right } \$ p \{ \text{currState } = \text{st}' \} \). Otherwise, if an error is returned, it is also used as the result of the next operation.
The encodings of valid-outputs and finished are quite similar to the default definitions formalised in Definition 5.3. Both use Haskell’s list comprehension syntax to mimic the set builder notation used in the corresponding definitions. To give an example,

```
[ ms | ms ← Set.elems ms_set, isValid Output ms p ]
```

roughly equates to:

\[ \{ \sigma : \sigma ∈ \zeta, \text{valid}(!, \sigma, p) \} \]

except that it generates a list rather than a set. To address this the `Set.fromList` function is invoked with the generated list creating a set which is, in the practical sense, equivalent to the corresponding set builder form, i.e. unordered and containing only unique elements. This concludes our description of Kaluta, however, the interested reader can find further code listings in Appendix A.

### 8.3 Practical and Technical Considerations

In the process of constructing our enterprise software environment emulator we encountered a number of technical issues. These issues mostly arise out of attempting to emulate many thousands of endpoints. We take a moment here to discuss the issues and practical considerations we encountered along the way so that others wishing to build an enterprise software environment emulator may know how to address them should they arise.

**Network Address Appropriation**

The external systems with which we wish to our emulator to interact expect to operate in an environment containing many individual endpoints with their own network identity, i.e. unique host names and/or IP addresses. Therefore the emulator host machine must appear to in fact be multiple network identities from the perspective of an external system. There are a number of ways in which this could be achieved. A router, for example, may be placed in-between the emulator machine and any external system configured to direct any packets destined for an emulated endpoint to the emulator machine. Alternatively, some form of network address translation may be performed on the emulator machine through manipulation of the operating systems IP tables. We opted however, for the most straightforward approach we encountered, Ethernet aliasing.

Ethernet aliasing is a technique supported by various Unix-like operating systems which allows a single physical Ethernet device to be associated with numerous IP addresses. We wrote a simple Python script to manage the capture and release of IP addresses using this technique. This script simply wrapped around the `ifconfig` command capturing or releasing a range of IP addresses as directed by its command line arguments.
Scaling

Scaling a single physical host to handle tens-of-thousands of unique network identities is a complicated technical task. In order to allow such scaling on the Linux operating system we had to modify two limits set in the kernel. Firstly, the number of network sockets which may be open at a given point in time is tied to the file descriptor limit which is typically set to 1028. If this limit is not increased the underlying select() call is likely to complain and if using Java an IOException will be raised with some message along the lines of “No buffer space available (maximum connections reached?)”. It is straightforward to increase this limit using the ulimit command. By increasing this limit on our Kaluta host we were able to avoid future instances of this error.

The second, and far more difficult to diagnose, limit we came across related to the size of the kernel’s ARP cache. The Linux ARP cache has a default hard limit of 1024 which due to us capturing tens-of-thousands of IP addresses on the emulator machine on the same sub-net, quickly filled. This error raises a similar exception to the case of the reaching the file descriptor limit. A Java IOException with the message: “No buffer space available.”. Note the similarity of the messages in both instances. This similarity led us to believe for quite some time that we were actually stuck on the same file descriptor limit as we had previously addressed. We incorrectly assumed that the slight difference in the error message had something to do with different Java VM implementations and we were somehow not increasing the file descriptor limit properly or we were and it was being overridden by some other mechanism when the emulation was run. After much frustration experimenting with changing various networking limits set by the Linux kernel we eventually came across the increasing ARP cache solution [Jon09]. To increase the size of the ARP cache size we edited the /etc/sysctl.conf file setting increased values for:

- net.ipv4.neigh.default.gc_thresh1
- net.ipv4.neigh.default.gc_thresh2
- net.ipv4.neigh.default.gc_thresh3

It should be noted that Kaluta was primarily developed on machines running the Mac OS X operating system. We did not run into the insidious ARP cache issues when running the Kaluta on these machines. We think that the ARP cache issue was not present on these machines because OS X is based on a BSD operating system. Therefore we suggest that BSD based operating systems are the preferred operating system for highly-scalable enterprise software environment emulator hosts.

---

1Matthew B. Jones has our eternal gratitude for posting this information.
8.4 Discussion

In this chapter we have described an architecture and reference implementation of an enterprise software environment emulator capable of providing interactive behaviour characteristic of enterprise software environments. The architecture was presented in Section 8.1 where we outlined the fundamental entities and relationships critical for enterprise software environment emulators. Researchers and practitioners wishing to construct an emulator of their own will find this architecture to be a useful blueprint. We discussed Kaluta, our own reference implementation of an enterprise software environment emulator, in Section 8.2 which provides a means to emulate endpoint interactions by interpreting interaction specifications defined using either the rudimentary or expressive model stacks described in Chapters 6 and 7 respectively. Finally, in Section 8.3 we outlined the most significant practical and technical issues which need to be considered when developing highly-scalable enterprise software environment emulators.
Chapter 9

Practical Application by Case Study

There are compelling opportunities to apply this research on interaction modelling and emulation to problems encountered in the enterprise software industry. In particular, enterprise environment emulators can be of significant assistance to quality assurance teams when analysing the run-time, extra-functional properties of enterprise software systems. In this chapter we discuss the practical application of interaction modelling and emulation, driving discussion with an industry case study which is based on work carried out in collaboration with CA Technologies. This case study serves not only to illustrate the process of practical application, but also verifies the practical utility of interaction modelling and emulation; showing that an emulated environment is indeed capable of interacting with a real enterprise software system, taking the place of a real deployment environment for observation and analysis of its run-time purposes.

The chapter begins with an introduction of the particular industry situation which precipitated the use of interaction modelling and emulation in Section 9.1. This is followed by a general procedure for practical application in Section 9.2, a procedure which can be followed by practitioners when looking to apply interaction modelling and emulation in industry. This procedure is applied in Section 9.3 to the industry situation described earlier in Section 9.1, giving a concrete example of how this procedure can be applied in practice.

9.1 An Opportunity for Practical Application

In this section we describe the context and events occurring at CA Technologies which essentially led to practical application of interaction modelling and emulation in industry. This can be read as an example set of initial circumstances in which an enterprise environment emulator can be of significant benefit; an opportunity for practical application.
9.1 An Opportunity for Practical Application

9.1.1 CA Identity Manager

CA Identity Manager (IM) [Gar06] is an enterprise-grade identity management suite capable of managing access and permissions on all of the various systems within an enterprise environment. It is typically purchased by and deployed into large corporations, such as banks, who use it to manage the digital identities of personnel as well as to control access of their vast and distributed computational resources and services.

To further illustrate IM’s role within an organisation we describe some of the key operations which it can, and is commonly used to, perform:

**Endpoint Acquisition** Acquire an endpoint so that IM may subsequently control its access and user permissions.

**Endpoint Exploration** Once an endpoint is acquired it must be explored in order to retrieve the information required in subsequent management. This essentially collates the manageable identity and access objects on the endpoint so that it can be subsequently modified as appropriate.

**Add an Account** Once an endpoint is acquired and explored, accounts for employees may be added to an endpoint. This is typically invoked when a new employee is added to staff so that they may have the suitable level of access to the endpoint.

**Modifying an Existing Account** An employee’s identity information is likely to change numerous times over their course of employment. IM facilitates these changes by allowing modification to their attributes in a central location, and the subsequently applying these modifications on the appropriate managed endpoints.

Naturally, in order to perform these, as well as other, operations, IM is required to interact with the many different endpoints within its deployment environment. Each of the key operations we outlined above, for instance, requires some form of interaction with endpoints in its environment. In this sense and in the terminology of our modelling in Chapter 4.3, IM is essentially an open (enterprise) software system. A software system which, in order to fulfil its role within an organisation, is required to interact with other semi-independent, external endpoints.

9.1.2 Situation

A potential customer inquires after the run-time scalability of IM. Acting responsibly, this customer requires some assurance regarding IM’s scalability, preferably an assurance based on sound empirical evidence. Only once they are confident IM will scale as necessary and operate effectively in their environment, will a purchase be made.

We are informed that this potential customer’s environment contains 10,000 LDAP directory server endpoints. This is too many endpoints to represent using the system-level virtualisation approach, which the IM quality assurance team would normally use, and furthermore, is too many endpoints to be handled by any of the conventional approaches known to the team.
9.1.3 Approach

Fortunately, the research arm of CA Technologies, CA Labs, have been involved in a project which investigates an approach to providing interactive representations of such large-scale distributed environments. An approach based on interaction modelling and subsequent emulation. Moreover, they have access to a prototype emulator which may just be able to approximate the interaction behaviour and characteristics of this client’s environment, using a single physical host.

The IM quality assurance team decide to make use of this prototype so that they may investigate and report on IM’s run-time properties. Specifically, on IM’s run-time properties when operating in an environment of similar behaviour and characteristics to that described by the potential customer. It is hoped that the run-time properties of IM observed while interacting with the approximated analogue of the client’s environment will be relevant and resemble those which the client can expect upon deployment.

9.2 General Procedure for Practical Application

Although we have argued for the practical applicability of interaction modelling and emulation throughout this thesis, we are yet to provide any guidance as to how practical application may be achieved. As such, it may be yet unclear to the practically minded reader just how an enterprise environment emulator may be of real-world use. In this section, we address this issue by describing a general procedure for the use of interaction modelling and emulation in what we term practical situations. We expect that this procedure will be of considerable utility to industry practitioners wishing to use the emulation approach in a variety of different specific situations.

9.2.1 Practical Situation

We begin with a more precise description of the term practical situation. A practical situation is an set of circumstances occurring in industry where:

- An open software system is required to interact with some interactive approximation of a specific real closure environment, which henceforth will be referred to as the target.

- The characteristics of the target environment are known. These characteristics include cover the general information regarding the hosts and endpoints required in the target environment. The host addresses, endpoint types and classes which are executed on the various hosts, as well as the port numbers of any server-class endpoints.

- The range and necessary accuracy of interactive behaviour required of each different endpoint in the target environment is, to a certain extent, known. There needs to be some notion of just what endpoint behaviours are necessary and how accurate these behaviours need to be with respect to a real instance thereof.
General Procedure for Practical Application

9.2.2 General Procedure

With the practical situation defined we can now move onto describing the general procedure itself. The procedure defined is closely related to the emulator architecture defined earlier. We refer the reader to Chapter 8.1 for more discussion of these entities, Figure 8.1 may be particularly illustrative.

Provided a target environment, a practical situation, and a collection of model stacks and corresponding specifications provided by the emulator, the general procedure is as follows:

1. For each unique endpoint type in the target environment, where no appropriate specification exists:
   Where an appropriate endpoint specification is one in which the requisite endpoint operations are included, and the model stack on which it is based is capable of expressing the behaviour of the endpoint to a sufficient degree of accuracy.
   
   (a) Select or Construct an Appropriate Interaction Model Stack
       Typically, the simplest, or in other words least-powerful, model stack capable of achieving the requisite level of behavioural accuracy should be selected; endpoint specification using simpler model stacks generally requiring less human effort than more powerful alternatives.
       
       If no such (appropriate) model stack is provided by the emulator, one must be constructed. This can be achieved by combining some existing protocol, behaviour, and/or data store layers, or where no appropriate layer or layers exists, creating new protocol, behaviour, and/or data store models for the model stack. For a detailed account of the procedure for constructing new model stacks we refer the reader to Chapters 6 and 7, where we describe the rudimentary and expressive model stacks, respectively.

   (b) Specify Endpoint Layers Using Selected Stack
       i. Protocol Model
       ii. Behaviour Model
       iii. Data Model
       
       The emulator will provide some means to specify each layer of the selected model stack. This may take the form of some separate specification language which is parsed and interpreted by the emulator upon instantiation. Or alternatively, it may take the form of some embedded domain specific language leading to specification using a subset of the emulators host language. Otherwise, some kind of framework may be provided which is allows endpoint specification through extension.
       
       When specifying each layer consider the range of interaction behaviours necessary in the target environment. Avoid wasting time providing specifications of behaviours which are unnecessary to the practical situation in hand.
(c) Define Requisite Network Abstractions

The emulator, like the case of interaction specification, will provide a facility for encoding network interface abstractions enabling communication between external software systems and emulated endpoints. Generally, this will be in the form of a framework which allows definition of the services and conduits through extension of that framework. Where possible, existing network communication libraries can, and are recommended to, be used to help define these network abstractions which follow:

i. Service (if server-class endpoint)
   A. Bind the network service to corresponding engine (emulated) service so that the engine is notified when new channels are established.
   B. Bind to the corresponding conduit which will be created upon new channels being established thereby enabling further interaction between the external system and the engine.

ii. Conduit
   A. Define an operation which can decode network messages into the message abstractions understood by the engine.
   B. Define an operation which can encode engine messages as the native messages ready for transmission and will be understood by the external system.

It should be noted that this step is not always necessary. In particular when there exists an inappropriate specification for the endpoint type in question, i.e. corresponding type but unsuitable model stack, the existing network abstractions for that endpoint type can be largely be reused; the decoding and encoding operations being essentially the same regardless of model stack. The bindings between the network interface and then engine may, however, need to be modified accordingly.

2. Configure Emulator

This is the selection, enumeration and parametrisation of endpoint specifications. These values are closely related to the characteristics of the target environment.

3. Run Emulator with Open Software System

Execute the emulator with the configuration defined as well as the open software system. So long as there are no technical issues, i.e. the emulator host and open software system are connected to one another through the network, then the open software system should be able to operate similarly to how it would in a real environment. The run-time properties of this open software system may be observed and analysed as it operates in this approximated target environment.
By following this procedure an industry practitioner is able to use interaction modelling and emulation as appropriate in practise. With continued use and collaboration, the collection of available endpoint specifications will grow with time. The bigger this collection of endpoint specifications grows, the less effort it takes to apply emulation in practise; so long as appropriate endpoint specifications are available for all endpoints in a target environment, the emulator need only be configured and simply run.

9.3 An Application of the General Procedure

To illustrate application of the general procedure introduced by the previous Section 9.2, we now describe its application to the industry situation involving CA Identity Manager (IM) introduced earlier in Section 9.1. This application of the general procedure is split into two subsections, mirroring those of Section 9.2. The first Subsection 9.3.1 states the IM situation using the terminology and requirements of the practical situation defined in the general procedure. The second Subsection 9.3.2 follows the general procedure as it is applied to the specific IM situation. Throughout this section we presume that the emulator available to the IM quality assurance team is the one we described in Chapter 8. Consequently endpoint interactions may be specified and subsequently emulated using either the rudimentary or expressive interaction stacks, which were defined earlier in Chapters 6 and 7 respectively.

9.3.1 Situation

To summarise the earlier description, a practical situation is a set of circumstances occurring in industry where: (i) an open software system is required to interact with some interactive approximation of a specific real closure environment, known as the target environment. (ii) The characteristics of the target environment are known as well as (iii) the range and necessary accuracy of interactive behaviour required of each different endpoint in the target environment. For full details we refer the reader to Section 9.2.1.

The open software system (i) in question is CA Identity Manager, described in detail in Section 9.1.1, which is required to interact with a specific potential customer’s environment, which is the target.

Target Environment Characteristics

The characteristics of the target environment (ii) in question have been, to a certain extent, supplied by the customer. The target environment consists of 10,000 LDAP directory server endpoints. There is not any information available regarding the host addresses nor the port numbers of the directory servers. As such we assume that each LDAP directory is run on a separate host and uses the standard port number 389. So that the target environment characteristics can be complete, we simply allocate the IPv4 addresses 10.0.1.1 through 10.0.40.250 as the host addresses for the directory servers. These IP addresses are allocated such that 250 hosts are placed on each class C subnet, i.e. there are 250 hosts
on addresses 10.0.1.1 through to 10.0.1.250, and 500 hosts on addresses 10.0.1.1 through to 10.0.2.250. We use 250 as the quantum as it is nicely divisible into 1000 and 10,000.

**Range and Accuracy of Endpoint Interactive Behaviour**

We finally consider the range and accuracy of behaviours (iii) required of the endpoints. Given that the focus of this investigation is IM’s extra(non)-functional properties, it seems excessive to mimic the full range of behaviours exhibited by the customers environment. We use this insight to reduce the workload, focusing investigation on the key IM operations, described earlier in Section 9.1.1, and providing behavioural approximations of environments which close IM’s environment as it carries out those key operations only.

The approach we take to understanding the requisite range and accuracy of endpoint behaviour is based on analysis of network traces which capture the interaction between IM and a real LDAP directory server endpoint whilst performing the key IM operations. Using this trace it was straightforward to determine: the subset of LDAP message shapes exchanged and also the range of LDAP requests which need to be handled, and also, the necessity for persistency; the *modify account* operation verifies the modification with a follow up query.

### 9.3.2 Procedure

With the practical situation described using the relevant lexicon, we are now able to move onto the application of the procedure itself. The presentation of this application follows the same format as the presentation of the procedure in Section 9.2.2.

1. For the LDAP directory server endpoint.
   
   We find that there is not any suitable existing LDAP directory server specification available and consequently we must define such a suitable specification. There are not any other endpoint types in the target environment so we need only specify this endpoint specification.

   (a) Select Expressive Interaction Model Stack

   The accuracy of the LDAP endpoint in the target environment indicates that persistency is required to facilitate one of the IM operations. Given that we have the rudimentary and expressive stacks to select from, the expressive stack is the only suitable option as the rudimentary does not support persistency. Alternatively, if it was decided that the modify operation was not necessary, the rudimentary stack could be used. This would decrease the specification effort and increase scalability, unfortunately though, this is not currently the case and therefore the expressive stack must be used.

   (b) Specify Endpoint Layers Using Selected Stack

   The Haskell engine of the emulator in use facilitates layer encoding through type classes which require definitions of certain functions for emulation. The type classes relevant here are the ExpressiveSpecification and ServerClass type classes which require definitions for the functions:
We will not describe the init and start functions in detail as their purposes are simply administrative, allowing instances of the endpoint to be initialised and started respectively. The other three functions, however, directly relate to the specification of the expressive model stack layers and as such will be discussed.

i. Protocol Model
The emulator’s engine provides a facility to define expressive protocol models using a concrete implementation of the abstract syntax provided earlier in definition 7.1. Following this syntax the LDAP protocol model is specified essentially as it appears in Figure 7.1. This is written in a separate file and is bound to the LDAP endpoint type of the engine through the protocol function.

ii. Behaviour Model
The behaviour model is tied to the handle-map function and involves writing definitions of each supported request handler in Haskell. These definitions are included in Appendix A Section A.2.1 Listing A.2. The handle-map associates the LDAP message shapes with the handler functions as:

\{(AddRq, handleAddRq), (BindRq, handleBindRq), (ModRq, handleModRq), (SearchRq, handleSearchRq), (UnbindRq, handleUnbindRq)\}.

iii. Data Model
The expressive LDAP directory data store is encoded as the Haskell map provided by the “Data.Map” module. At the top level this map encodes distinguished names (DNs) as plain strings to serve as keys which are associated with entries. Each entry is also a map where attribute names serve as keys being associated with sets of attribute values. Both attribute names and values are left as strings for the sake of simplicity although other types of values may be added as necessary in the future. The Haskell encoding of this data store is also provided in Appendix A Section A.2.2 this time in Listing A.3.

As expressive data stores may be quite large, there is the option to define a function to initialise them from file. The standard interchange format for LDAP directories is the text based LDIF format. An existing Haskell library for LDIF is used to parse the supplied LDIF file and used to create a corresponding expressive data store.
(c) Define Requisite Network Abstractions
The emulator being used allows network abstractions to be defined through
implementation of the corresponding Java interfaces. Binding between the
LDAP network abstractions and the corresponding engine abstractions is ac-

ieved through Apache Thrift RPC [thr] where the engine provides a service
interface which is invoked by the LDAP network interface abstractions as re-
quired.

i. LDAPService
Each new LDAP channel results in the construction of a corresponding
LDAPConduit for communication between the external system and the
emulated LDAP endpoint. This service also notifies the engine of the new
channel.

ii. LDAPConduit
Utilises an existing Java library for LDAP which facilitates reception, de-
coding, encoding and transmission of LDAP messages on ASN.1 chan-
nels.

Each time a request arrives, it is decoded into the emulator’s message ab-

straction and forwarded on to the engine. When the response sequence is
received back from the engine it is encoded and transmitted to the external
system using the underlying ASN.1 channel.

The listings for the LDAPService and LDAPConduit classes appear in Ap-
pendix A Section A.3 as Listings A.4 and A.5 respectively.

2. Configure Emulator
Configuration is achieved using CSV file format, each entry (line) denoting the nec-
essary parameters for initialising a single endpoint. Each entry reads from right to
left: host-addr,endpoint-type,port-num,data-src

Where host-addr indicates the IPv4 number for a host, i.e. “10.0.1.1”, endpoint-
type is “ldap-expr” which points to the expressive LDAP server specification, port-
num is 389, and data-src points to an LDIF file used as to form the initial state
of the LDAP directory data stores. A small Python script is used to generate this
configuration file to avoid having to type it out manually.

3. Run Emulator with IM
The emulator is now ready to close IM’s environment and be used in experiments
regarding the extra-functional, and specifically, the run-time scalability of IM.
9.3.3 IM Scalability

The result of applying the general procedure for practical application to the situation involving CA Identity Manager is an emulator which may be easily configured to mimic the behaviour and characteristics of the potential customer’s environment consisting of 10,000 LDAP directory server endpoints. This emulator was subsequently used by the IM development team to investigate the scalability and other run-time properties of IM whilst it operates in such an environment. The results of these experiments have been written up in [VHSH12]. We can however, state that the scalability and other run-time properties of IM were observed over multiple runs as it operated in an emulated environment consisting of 10,000 LDAP directory server endpoints. The ability to reuse configurations and specifications made repetition of experiments straightforward and consequently it was straightforward to increase confidence in the observations through repetition. Furthermore, it was straightforward to investigate the effect different data store sizes has on IM’s run-time properties by providing various LDIF files as a configuration option to the emulator. Altogether, through experiments enabled by the enterprise environment emulator, the IM team was able to gather sufficient empirical evidence that IM can indeed cope with the characteristics of the potential customer’s environment.

In addition to verifying IM’s ability to handle this potential customer’s environment, this study has also verified the practical utility of interaction modelling and emulation: an emulated environment, based on interaction specifications, is able to interact with a real enterprise grade software system, in this case CA Technologies Identity Manager. Meaning that the behaviour of the closure environment provided by an emulator is accurate enough that real enterprise software systems may interact with them in the same manner that they would endpoints in a real environment.
9.4 Summary

In this chapter we have described the practical application of interaction modelling and emulation using a case study involving a real enterprise software system to drive the discussion. We began in Section 9.1 with a description of a situation in which CA Technologies found themselves where an enterprise software environment emulator could be of significant aid. We then moved onto the general procedure for practical application in Section 9.2, in which we described the steps involved in applying interaction modelling and emulation to practical situations. We then demonstrated this general procedure in Section 9.3 by applying it to CA Technologies specific situation enabling the IM team to thoroughly investigate scalability and other run-time properties in an emulated representation of a potential customer's environment. Additionally, this case study involving IM has verified the practical utility of interaction modelling and specification; an emulated environment, based on interaction specifications, is able to interact with real enterprise grade software systems.
We conducted a series of experiments to ascertain the scalability of our enterprise software environment emulator as well to develop an understanding of the differences in computational costs when considering the rudimentary and expressive interaction stacks. We were particularly interested in how many interaction specifications can be emulated at once and how the choice of model stack, either rudimentary or expressive, affects emulator scalability. The results of these experiments show that the interaction modelling and emulation approach can indeed scale, allowing tens of thousands of endpoints to be emulated simultaneously on a single physical host.

The emulator scalability discussion is presented in three major sections. We begin in Section 10.1 with a description of the procedure that was followed throughout the scalability experiments. Section 10.2 presents the results of these experiments primarily through graphical representation and discussion of evident trends. The collated data on which these graphs are based can be found in Appendix B. Finally, Section 10.3 provides a summary discussion of emulator scalability thereby concluding the chapter.

10.1 Procedure

Two LDAP directory server interaction specifications were defined. One specified in the rudimentary interaction stack, the other with the expressive stack. For the sake of simplicity, we specified only a subset of the full set of LDAP directory server operations. The operations specified, although incomplete, cover the most essential of LDAP activities: add, bind and unbind, delete, modify, and search.

The expressive specification has a persistent underlying data store which is initialised as a directory with a hierarchy three levels deep and 100 entries in total. The add, delete and modify requests all result in persistent changes to this data store, so long as the request is valid. The bind operation checks the provided credentials against those found in the data store returning success or failure as appropriate. Finally, the search operation returns those entries in the data store which match the provided search criteria.
The rudimentary specification, on the other hand, has no persistent data store. The requests are assumed to be valid and the responses reflect this assumption. Whereas with the expressive stack, the number of search result entries returned vary depending upon the contents of the search request, as well as the contents of the data store, the search results of the rudimentary stack must be fixed, as such regardless of search query, the rudimentary LDAP server specification will return a constant two search result entries.

10.1.1 Workload

To exercise the emulator we defined a workload to be carried out against each emulated LDAP server. This workload was programmed using Java [GJSB05] and consisted of a sequence of LDAP requests which fully explore the emulated endpoint’s capabilities:

1. Simple authentication bind with plain text user name and password. For the expressive specifications, the credentials provided by the request are validated against the contents of the corresponding data store. For the rudimentary model, however, the credentials are ignored as they are assumed to be valid.

2. A whole directory search where every entry of the directory is returned. The result being 100 entries returned by the expressive specification and 2 entries returned by the rudimentary.

3. Add a single entry. This entry has a valid parent entry already which already exists in the expressive specification’s data store. This same entry is added to each emulated endpoint.

4. A sub-tree search which selects the branch of the directory in which the previously added entry was placed. This simulates an LDAP client verifying that a previous add has in fact been successful. The expressive LDAP server specification returns 12 entries (11 original entries and the newly added entry) while the rudimentary emulated endpoint again returns the 2 entries as with any search performed upon it.

5. Change the value of attribute of an existing entry through a modify request. The value to which it is changed being unique to each emulated endpoint and thereby causing the data stores of each emulated endpoint to become different from one another over an emulation.

6. A single-level search, selecting the entry which has had its password updated by the previous modification.

7. Delete one of the original entries from the directory.
This workload was encapsulated as a *Runnable* Java class allowing subsequent execution using an *ExecutorService*. This design makes it relatively straightforward to run multiple workloads simultaneously, thereby simulating multiple concurrent connections being made to the emulation environment. Throughout our experiments we have used eight threads meaning that there are up to eight active connections with our emulator at a given point in time. We decided to use eight threads as there were eight CPU cores available on the machine hosting the emulator. Additionally, we wanted to avoid excessive context switching overhead on the machine executing the workload, which had only two CPU cores.\footnote{Some preliminary testing where up to 32 simultaneous connections are active in the emulator has been performed. No significant issues have been observed during these preliminary tests. A more thorough investigation is, however, necessary before we are able to state anything more conclusive and is left as future work.}

### 10.1.2 Measurements

While the workload was run against our emulator, we measured resource consumption of the emulator as well as its timing characteristics. The time taken to complete each individual task was recorded directly by the workload script. Resource consumption of the emulator was regularly polled by another script, calling the UNIX command `ps` to retrieve the memory and CPU usage and `ifconfig` to retrieve the network traffic statistics. As a result, for every experiment run extensive data regarding emulator timing characteristics as well as memory, CPU and network usage was collected.

### 10.1.3 Experiments

We ran the workload against our emulator whilst it emulated 1, 10, 20, 55, 150, 400, 1000, 3000, 8000 and 10,000 LDAP endpoints, for both the rudimentary and expressive specifications; an approximately exponential growth rate for the number of emulated endpoints. The experiments were repeated to increase confidence in the measurements: 10 times when emulating 1 endpoint, 9 times when emulating 10 endpoints, 8 times when emulating 20 endpoints, etc, until emulating 400 endpoints, from which point each experiment was repeated 6 times as 6 is the minimum number of data points needed to use a Student t-test to calculate a confidence interval \[\text{Box87}\]. We selected 10,000 as the maximum number of emulated endpoints as it is this scale of environment which has been our explicit scalability goal throughout the dissertation. This is also this scale of environment which, according to discussions with enterprise software quality assurance teams, observation and analysis of run-time properties becomes particularly impractical and troublesome. Although we have run the emulator with as many as 20,000 endpoints without encountering any obvious scalability issues, we have not yet had the opportunity to systematically investigate the behaviour of our emulator beyond the 10,000 emulated endpoint limit. This more thorough investigation is left as future work.
This range of experiments provides sufficient data for considering the scalability of emulation when up to 10,000 individual, but initially identical, endpoints are emulated using either the rudimentary or expressive model stack. As the workload is carried out the initially equivalent emulated endpoints diverge, becoming increasingly dissimilar. The protocol models of emulated endpoints are frequently being updated and unique modifications change the data store of each expressive endpoint. Nevertheless, this initial equivalence is bound to have some effect on the emulator scalability and run-time performance. Identical endpoints can be efficiently stored, being held in memory once and referenced many times, and thereby improving scalability. At run-time, however, as the endpoints diverge local copies of data must be made so that modifications on one endpoint do not affect others. This copying will, in turn, take CPU cycles and therefore adversely affect run-time performance. The extent to which emulated endpoint similarity affects emulation is an interesting nontrivial issue. It is not however, currently essential to the problem and as such we leave it for future work.

10.1.4 Environment

The environment in which our experiments were run consisted of two physical hosts directly connected to one another’s 1Gbps LAN card with CAT-5 cable. The more powerful of these two machines was used to run the emulator whilst the other was used to execute the workload. The emulator machine had 8 Intel Xeon E5440 2.83GHz CPU’s and 24GB of main memory available. The workload machine had 2 Intel Pentium 4 3.00GHz CPU’s and 2GB of main memory. Both machines ran Linux Ubuntu 11.04 as the host operating system, Linux kernel version 2.6.38, and Sun Java Virtual Machine version 6.26.

10.2 Results

The data collected from the experiments are presented in this section, and in general, summarised using box-plots with the number of emulated endpoints on the horizontal axis using a log scale, and the variable of interest being plotted on the vertical axis linear scale unless stated otherwise. All box-plots represent five-number summaries: the lower and upper most vertical points representing the minimum and maximum observations, the box itself representing the half of the data which is closest to the median and the line creating a cross-section of the box represents the median itself. The most significant numerical data used to generate these plots can be found in Appendix B. The numerical data is presented separately so that we are able to focus our attention on the discussion and analysis of more general relationships and trends here. The general trends and relationships being more pertinent to our analysis of interaction modelling and emulation than any of the specific numerical data which was collected.
10.2 Results

10.2.1 Memory Consumption

For every experiment run, the memory use of both the engine and network interface was polled every second. Of most interest to us here is peak memory usage, i.e. the largest amount of memory used by each emulator component during the experiments. The peak memory used by the engine being particularly interesting given that it is chiefly this component where differences in memory requirements of the two model stacks can be observed.

Network Interface  We begin by considering the peak memory used by the network interface. Memory use of the network interface will not shed any light on the relationship between interaction stack selection and memory use. It is responsible for transcoding between the native messages of the network channel and messages understood by the engine. It is not significantly effected by model stack selection. Nevertheless, memory use of the network interface is a real practical consideration. The encoding and decoding process it provides is a necessity so that emulators are able interact with a real enterprise software system. Figure 10.1 provides a summary of the network interface’s peak memory usage that was observed during our experiments. The interested reader may find the numerical data presented in this Figure in Appendix B Tables B.1 and B.2.

![Figure 10.1: Peak Memory - Network Interface](image)

There is little difference to be observed in the memory use between 1 and 1000 emulated endpoints, regardless of model stack. As we move up to 8000 and 10,000, however, it appears that the peak memory used, especially by the rudimentary network interface begins to climb and vary more significantly. The eventual increase in peak memory use makes sense as there is an increase in the number of server sockets required to open up native communication channels. The difference between the median rudimentary and
expressive peak memory usage towards the higher number of endpoints, the rudiment-
tary stack having a higher memory usage, we believe is due to the speed with which
each workload is being executed. The workload for the rudimentary model stack is com-
pleted more quickly than the expressive model stack. This means that the workload script
when interacting with rudimentary LDAP server specifications is acquiring new sockets
more quickly than the in the expressive case and is therefore leaving less time for Java
to garbage collect the now inactive communication structures, resulting in the observed
larger peak memory use in the rudimentary case.

**Engine**  The peak memory used by the emulator *engine* during our experiments is more
interesting as it can be used to analyse the relationship between the different model stacks
and the computational resource use, specifically main memory. Primarily we are inter-
ested in the impact of the persistent data store used by the expressive specifications on the
memory consumed by the engine. These persistent stores are initially identical across all
emulated endpoints but steadily diverge as entries are added, removed, and in particular
modified in a unique fashion by the workload.

![Figure 10.2: Peak Memory - Engine Log-Log Scale](image_url)

Figure 10.2 summarises the peak memory usage observed to be used by the engine
across all our experiments. This figure uses a log-log scale to illustrate the approximate
linear growth of the peak memory, regardless of model stack, as the number of emulated
endpoints increases. It is possible to decrease this growth rate by more effectively shar-
ing data between the emulated endpoints. Given that in the worst case the peak memory
used by the engine when emulating 10,000 expressive endpoints, each of which containing
100 entries, was 950MB, we expect this linear growth rate to be acceptable for the
majority of usage scenarios. As mentioned earlier, we intend to test this hypothesis more
thoroughly in future work, investigating the effects of dissimilarity between endpoint specifications on scalability.

The log-log scale of Figure 10.2 also highlights the fact that the peak memory used by the expressive engine is larger than that of the rudimentary engine by an approximately constant factor of between two and three. Selecting to use the expressive LDAP specification rather than the rudimentary, at least in specifications we defined, will lead to an emulation which consumes up to three times as much main memory as would have been the case if the rudimentary specification was selected.

The difference in memory usage of the engine between the rudimentary and expressive model stacks is perhaps made clearer by the plot in Figure 10.3. This plot represents the same data as that in Figure 10.2 however, the vertical axis is changed to linear while the horizontal remains logarithmic. This plot suggests that the peak memory of the expressive engine is increasing linearly (due to the scale this linearity appears as exponential growth in the vertical axis).

Figure 10.3: Peak Memory - Engine

Figure 10.3 also suggests that the peak memory use of the expressive model stack may be beginning to diverge from its rudimentary counterpart towards the higher end of the number of emulated endpoints spectrum. Unfortunately we cannot say with certainty whether this is truly happening as 10,000 endpoints is the upper limit on the data we have collected. Further experiments where more than 10,000 endpoints are emulated are necessary to be able to determine whether this divergence is indeed occurring and is something we intend to investigate in future work.

Although peak memory use of an emulator gives us an idea of just how much memory is necessary under different emulation configurations, it does not give us any idea of the emulator’s memory consumption during an emulation. To illustrate memory con-
sumption during emulation we present in Figure 10.4 the memory use of the network interface and engine during one of our experiments where 10,000 expressive LDAP servers were emulated.

![Figure 10.4: Memory Use - 10,000 Expressive LDAP Servers](image)

Figure 10.4 shows a consistent pattern in the engine of linear memory growth followed by a sharp fall likely corresponding to the Haskell garbage collector being invoked. Over the course of this emulation, the memory not freed by the engine garbage collector is gradually increasing. This is likely due to the fact that the data store of the expressive emulated endpoints are becoming increasingly dissimilar. Each endpoint has a unique modification applied to its data store and consequently the endpoint data stores cannot be freely shared by reference. This gradual memory growth in the engine means that over the course of an emulation there is a decreasing amount of memory which can be freed. More of it being used to store the increasingly dissimilar data stores. The minimum required memory of the engine, when emulating expressive interaction stack specifications, slowly but steadily increases over the course of emulation.

The memory consumption of the network interface over time is not as straightforward as that of the engine. The network interface tends to accumulate memory more quickly and then gradually release memory over time. The network interface also seems to oscillate more frequently and less predictably than the engine performing garbage collection at irregular intervals. Finally, the plot indicates that the peak memory use of the network interface occurs right at the end of the emulation, likely after the last of the interaction had taken place. This suggests that the peak memory use of the network interface may not be a reliable measure of its typical memory consumption, as the peak essentially occurs after the emulation has finished due to some quirk in the Java memory allocation mechanism.
We conjecture that the additional complexity exhibited by the network interface’s memory use, as compared with that of the engine, is likely due to the sophistication of the Java virtual machine memory allocation and garbage collection routines. Java is much more popular and widely used than Haskell, and consequently has had significantly more time and effort invested in honing memory allocation and garbage collection performance.

10.2.2 CPU

Given that the expressive model stack provides significantly more modelling power than the rudimentary, we expect the emulation of expressive specifications to be more CPU intensive than their rudimentary counterparts. To verify whether this is indeed the case and also to get some idea of just how much more computationally expensive the expressive model stack is compared with the rudimentary, we investigate the CPU usage of both model stacks in our experiments.

In comparing the different model stacks we are predominantly interested in the CPU usage of the engine, rather than the network interface. This is because the behaviour of the network interface is largely independent from the particular model stack used; the encoding and decoding of LDAP messages is the same regardless of which model stack was used to specify the interaction. It should be noted however, that the CPU usage of the network interface when using an expressive model stack will be significantly more than when using a rudimentary stack. This is simply because the expressive interaction stack results in far more transmissions, ultimately leading to more encoding work for the network interface to perform.

We present in Figure 10.5 a direct comparison of the CPU usage of the emulator’s engine summarised over our experiments. We again use the log-log scale so that it is clearer what relationship may exist between the rudimentary and expressive mode stacks CPU usage. The vertical axis contains the CPU time used by the engine in units of minutes and seconds (mm:ss). The horizontal axis as usual contains the number of emulated endpoints. As in the previous summary data plots we use a box-plot to present the minimum, maximum, inter-quartile range and median of the collected data.

Figure 10.5 suggests that the CPU usage of both the rudimentary and expressive model stacks increase linearly with the number of endpoints being emulated. In both stacks CPU use at the lower end of the number of emulated endpoints scale is measured as zero. These experiments being completed in less CPU time than we could detect using the *ps* tool. CPU use is observed only once 20 expressive endpoints are emulated or 400 rudimentary. For the experiments conducted, the expressive stack was slower than the rudimentary stack by a factor which remained constant over the range of endpoints tested. Therefore, using the expressive model stack specification we defined rather than the rudimentary, requires more CPU by some constant factor. The data collected for emulations consisting of 10,000 endpoints may suggest that this constant factor is beginning to diverge, becoming non-constant. Unfortunately for reasons already discussed we have not yet thoroughly investigated emulator behaviour beyond the 10,000 endpoint bound-
The expressive LDAP server specification used in our experiments results in much more actual interaction than the rudimentary specification. As such it is somewhat unfair to directly compare the CPU usage of these model stacks as we have done in Figure 10.5. The expressive model stack is required to make many more transmissions than the rudimentary and thereby inadvertently skews the CPU use measures; the whole directory search in the expressive specification results in 100 search entries whilst the rudimentary results in only two. In order to provide a fairer comparison of the CPU usage of the different model stacks we now take network traffic into account.

Figure 10.6 presents the average of the engine CPU time in ratio with the total processed network traffic for each different number of emulated endpoints. The total processed network traffic being the total bytes both received and transmitted during an emulation. Unfortunately, due to the CPU use in seconds being less than one for a number of the lower end number of emulated endpoints experiments, the data presented is incomplete. Nevertheless this ratio as presented in Figure 10.6 gives us an idea of just how much CPU power is required to receive and transmit messages by each model stack and how this varies with the number of endpoints being emulated. Once again, the numerical data used to produce this plot can be found in Appendix B, in this case Tables B.3 and B.4.

The ratio is presented in the familiar log-log scale and in both the expressive and rudimentary cases suggests a fairly constant CPU time required for every network byte received or transmitted as the number of emulated endpoints increases. The CPU time required for processing and generating network traffic using expressive endpoint specifications may increase slowly as the number of emulated endpoints increases. This is
likely due to some constant administration overhead which accumulates as endpoints are added. This overhead however, seems to be reasonable.

![Figure 10.6: Engine CPU Use (sec) : Network Traffic (bytes)](image)

The trend shown in Figure 10.6 is not entirely as expected. We had presumed that interaction specifications based on the rudimentary model stack would be noticeably less computationally expensive, in terms of CPU use, than those based on the expressive model stack, even when total network bytes processed were taken into account. The lack of significant difference, in this regard, between the two interaction stacks suggests that our emulator is I/O bound. Even though the expressive interaction stack requires more CPU resources than the rudimentary to handle each request, this difference is insignificant when the time taken to perform the network I/O is taken into account.

### 10.2.3 Workload Timing Characteristics

Having discussed the general resource usage of our emulator, we move onto an analysis of the workload timing characteristics of the emulator. This timing data was measured by the workload script for each experiment run so that we can now investigate the run-time performance of the emulator. By run-time performance we mean the speed with which the emulator was able to perform the workload tasks and how this varied depending on differing numbers of emulated endpoints and whether these endpoints were specified using the rudimentary or expressive model stacks.

We begin our analysis of the workload timing characteristics of our emulator by looking into the average total time taken to complete the whole workload for each endpoint. Figure 10.7 summarises this data using the familiar box-plot format. As usual the horizontal axis is presented using a log scale, the vertical axis however, in this case uses a linear scale. We can observe in this plot how the median total workload time across the...
various number of emulated endpoints is relatively stable, especially when the emulator is using the rudimentary model stack. This stability indicates that the emulator is relatively well balanced and capable of approximately the same run-time performance regardless of the number of endpoints being emulated.

Figure 10.7: Summary - Average Time (ms) to Complete Total Workload

Figure 10.7 illustrates how, as expected, the expressive model stack requires a longer length of time on average to complete the workload than the rudimentary. This can be attributed to two main reasons. Firstly, the expressive model stack results in many more messages being returned as part of the workload results and therefore takes more time to transmit over the wire. We established earlier, when considering the relationship between CPU usage and network use, that it is likely that our emulator is I/O bound. As such, by simply transmitting more messages the expressive stack is at a significant disadvantage with respect to workload performance. The extent of this disadvantage will be discussed shortly. Secondly, the expressive model stack allows for more powerful (expressive) interaction modelling than the rudimentary. Consequently emulation of expressive endpoint specifications simply takes more CPU power than that which is required to emulate rudimentary specifications. This means more CPU cycles, longer time between receiving a request and generation of the response sequence, and ultimately, longer average total workload times.

As a final note regarding Figure 10.7 for the expressive model stack, once the number of emulated endpoints exceeds 1000, the worst case for the total workload time begins to increase approximately linearly (logarithmic horizontal axis with linear vertical axis). We are yet to investigate whether this trend continues beyond the 10,000 endpoint as per the reasons already stated. As such, we leave this investigation as future work.
The workload performed against our emulator consists of a number of individual LDAP operations, the many of which result in the same number of messages being transmitted by the emulator, regardless of whether the emulated endpoints happen to be expressive or rudimentary. We now take a closer look at workload performance by analysing the median time within which each individual task of the workload was performed.

Figure 10.8 presents a histogram of the median time taken to complete each of the individual workload tasks for both rudimentary and expressive specifications, in the case that 1000 endpoints are emulated. We have selected 1000 as the case to examine here as the results appear typical (see Figure 10.7) of all experiments run with up to 1000 emulated endpoints. Specific numbers can be found in Table B.11, Appendix B, Section B.3.

The individual workload tasks can be organised into 3 categories. Firstly, there are the ancillary tasks which are necessary in enabling interaction but not really a part of the interaction itself, this includes the connect and close tasks. Unfortunately due to restrictions imposed by the library used for LDAP communication the time for the close and unbind tasks are lumped together. Secondly, there are the tasks which result in the same volume of data being transmitted by the emulator regardless of model stack, including bind, add, modify, single-entry-search, delete and unbind. Thirdly and finally, there are the tasks which result in an increased volume of traffic being produced by the expressive specification compared to that produced by the corresponding rudimentary specification. This categorisation greatly helps in our analysis of Figure 10.8.
Results

There is no significant difference in the time it takes to perform the ancillary tasks. The connect task for both rudimentary and expressive specifications has a median of one millisecond. Similarly, the unbind-and-close task only differs by 1 millisecond, the expressive model stack taking one extra millisecond to complete the task on average.

As expected, the tasks where the expressive specification generates a higher volume of network traffic than the rudimentary, take longer to complete for the expressive specifications than the rudimentary. The whole-dir-search and subtree-search tasks return many more messages in the expressive specifications than in the rudimentary which is reflected by the difference in average time these tasks took to complete when using the expressive rather than rudimentary specifications. The whole-dir-search task in particular results in 101 (100 search result entries plus 1 search result done) messages being returned by the expressive specification, whereas the rudimentary specification returns just 3 messages for the same task. A fact reflected by the significant difference in average completion time for that task. In short, more messages to transmit will necessarily increase the time a task takes to complete.

Finally, we consider the tasks which generate approximately the same volume of network traffic regardless of model stack. We can observe in Figure [10.8] that each such task (bind, add, modify, single-entry-search and delete) takes slightly longer to complete on average when using the expressive model stack rather than the rudimentary. This is as expected; the expressive model stack takes more CPU power to process requests, even those generating responses of similar volume. The add task handling in the expressive specification, for instance, like the rudimentary specification always responds with a single message. Unlike the rudimentary specification however, the expressive checks its data store for any existing entries matching the entry being added, as well for the existence of an appropriate parent entry, adding the supplied entry to the data store only if all is well. These checks, and the other instances of behavioural logic used in expressive endpoint specifications, represent a processing overhead which is simply absent from the rudimentary interaction stack. It is this processing overhead which accounts for the slight difference in run-time performance between the stacks when generating similar volumes of network traffic in responding to requests.
10.3 Discussion

The scalability experiments, as expected, showed that interaction specifications utilising the expressive model stack are more computationally expensive, both in terms of memory and time, than their rudimentary counterparts. Generally, the resource consumption of the expressive specifications was more than that of the rudimentary by some constant factor. It was still possible however, to emulate up to 10,000 expressive specifications using a single physical machine as the host. Even though the expressive model stack requires more memory and CPU resources than the rudimentary, these requirements are still low enough that up to 10,000 endpoints can emulated simultaneously using a single server class machine. This suggests that human effort is a more crucial factor concerning model stack selection than any physical resource limits. To summarise, so long as there is time available and a chance that data persistency will be required, choosing to use the expressive model stack for an interaction specification will not unduly cripple the scalability. If however, time is short and/or there is no chance that persistency will be required, a rudimentary interaction specification will suffice and in all likelihood save on the extra time and effort an expressive specification typically requires.

Investigations into the relationship between our emulator’s resource usage (CPU and main memory) and total network (I/O) traffic suggest that Kaluta I/O, rather than CPU or memory, bound: CPU time required to process (received and transmit) network traffic is relatively stable up to 10,000 endpoints and main memory at this scale is not a concern. The implication of this is that the network infrastructure connecting an emulator to the system under investigation may limit scalability if that system generates large amounts of load against the emulator. Generally, however, this is not a concern as the system under investigation is usually using a single network interface and, therefore, can only generate so much load. In any case, emulation can be manually configured to make use of multiple host machines, if network load does become an issue.

The experiments conducted thus far are not exhaustive. There is ample opportunity for future work and experimentation. In particular, it would be interesting to more thoroughly investigate the relationship between the size and content of the persistent data store and scalability. Future work should also consider using a range of different interaction specifications and enterprise endpoints so that the relative power of the rudimentary and expressive model stacks are more fully understood in the practical context.

The experiments conducted in this chapter illustrate that through interaction modelling and emulation it is indeed possible to provide interactive behavioural approximations of large scale enterprise software environments. The scalability experiments in this section explored questions regarding just how many endpoints can be emulated simultaneously using a single physical host, and how choice in model stack affects scalability. To summarise, the experiments conducted in this chapter demonstrate that through interaction modelling and emulation, up to 10,000 server-class endpoints can be represented simultaneously, using a single physical host.
Chapter 11

Conclusions and Future Work

This chapter brings the dissertation to a close. We summarise our major contributions and suggest topics for future work.

11.1 Conclusion

In this dissertation we have demonstrated that:

Interaction modelling and emulation is an approach to producing interactive approximations of enterprise software environments capable of acting as and appearing to be real environments, enabling observation and measurement of the run-time properties of enterprise software systems. This approach is highly-scalable, well-suited to representing server-class interactions, and decoupled from implementation details of systems under investigation.

Assessing the run-time properties of enterprise software systems is a task best undertaken by observing and measuring properties of interest whilst the system in question operates in a range of environments that exhibit behaviour and characteristics representative of those expected in deployment. Production of such environments is a significant engineering challenge given the scale and complexity typical of enterprise deployment environments. In order to be effective, an environment of this nature must allow the software under assessment to communicate with it at run-time in the same manner that it would a real environment and, furthermore, must perform these communications in real(istic)-time. An environment capable of achieving this level of interactivity, as well as being highly-scalable, well-suited to representing server-class interactions, and decoupled from implementation details of systems under investigation, can significantly benefit assessment of enterprise software system run-time properties.

Interaction modelling and emulation is the approach we develop for providing interactive approximations of enterprise environments, better enabling run-time observation and measurement of enterprise software systems. This approach focuses on using light-weight interaction models requiring modest computational resources for execution. By using light-weight models, we hypothesised and later demonstrated, that interaction
modelling and emulation is highly-scalable; an emulator hosted on a single physical machine can, due to the modest computational resources required for execution, simultaneously emulate many thousands of interaction specifications. By explicitly targeting server-class systems we ensured and demonstrated that our approach is well-suited to the representation of such complex reactive behaviours. Finally, our interaction modelling and emulation approach treats the external systems which interact with the emulator, those systems being observed and measured, as black-boxes and as such the approach provides an interactive environment which is decoupled from implementation details the of any particular system under investigation.

Early on in our work it became apparent that different scenarios in which an enterprise environment emulator would be of use, required interaction models with differing levels of expressive power. So that a rich variety of such emulate-able models could be supported by an enterprise environment emulator, we defined an interaction modelling framework providing the fundamental structures and foundations upon which a rich variety of models and specifications can be built. The main result of this is the three level layered interaction model consisting of, from top to bottom, a (i) protocol layer defining the valid temporal orders of messages on channels, a (ii) behaviour layer defining how to handle requests and generate valid responses, and a (iii) data store layer containing the data used by the behaviour layer in processing requests and generating responses. We demonstrated the utility and flexibility of this framework through the rudimentary and expressive model stacks, which themselves, constitute two of the key contributions made in the dissertation.

The rudimentary interaction stack provides a shallow, yet practical, model of interactions between enterprise software systems. The idea being that a shallow, simple models require only low expressive power, which are easy to scale and therefore will enable highly-scalable emulation. To achieve this, the rudimentary stack is focused, as much as is possible, upon the messages exchanged in the interactions, abstracting away from any of the semantics involved in a real system’s request processing. The protocol layer of this stack is a deterministic automaton, defining what shape of message is valid for transmission and reception throughout interactions. The behaviour model is defined in terms of message skeletons and placeholder value resolution which uses the request as a source of missing values. Finally, the data store is used to store the message skeleton sequences which resolve output choice. The result is a highly-scalable, easy to use, interaction stack whose emulated behaviour is protocol conformant and which produces transmissions that have content consistent with values provided in corresponding requests.

The simplicity of the rudimentary interaction stack limits the scenarios in which it may be applied. To overcome the most significant of these limitations we developed the more powerful expressive interaction stack. The deterministic automaton used to model protocols in the rudimentary stack is ill-equipped to model concurrency in protocols, a property exhibited in some important enterprise software system interactions, such as LDAP. The protocol model of the expressive stack overcomes this limitation by presenting a model well-suited to specifying the parallelism and subservient parallelism exhibi-
ited in some enterprise software system interactions. In particular this model uses the concepts of protocol product, extension and contraction, introduced in some of our earlier work [HSHV10]. The lack of more semantically accurate model of request processing in the rudimentary interaction stack, combined with the lack of a persistent underlying data store, made it impossible to properly model interactions which make some attempt to independently verify previous modification operations. The expressive interaction stack overcomes this issue through the introduction of request handlers and a persistent data store shared across channels of a service. The handlers allow requests with certain message shapes to do some processing on the request, making reference to the persistent data store, generating a sequence of response messages and perhaps modifying the data store. In this way modifications can actually be carried out on the underlying data store and subsequent queries will return messages consistent with the modification’s success. Overall the expressive interaction stack is significantly more powerful than the rudimentary. This additional expressive power does, however, come at a cost. Generally more human effort is required to produce expressive rather than rudimentary specifications. Consequently both stacks have their uses: The rudimentary stack is well-suited to simple testing scenarios which do not require data persistency. The expressive stack, on the other hand, is suitable when a testing scenario requires data persistency and more realistic request processing.

As a proof of concept we described an architecture and implementation of an enterprise software environment emulator capable of providing interactive behaviour characteristic of enterprise software environments. The resulting implementation bases its behaviour on interaction specifications defined using either the rudimentary or expressive model stacks. The architecture serves as a guideline for others who intend to develop an enterprise software environment emulator, and the prototype emulator provides a concrete example of such an implementation.

The presented case study provides more practically inclined readers with a concrete example of how exactly, given a real testing scenario and an enterprise software environment emulator, one can produce an environment which enables observation and measurement of the enterprise software system under test. This case study describes how we have used our enterprise environment emulator to provide an interactive environment used to enable assessment of the run-time properties of CA Identity Manager [Gar06]. This demonstrates the practical utility of the approach showing that an emulation of a real enterprise software system’s environment can in fact interact with that software system and thereby enable its observation and measurement. Unfortunately, due to the nature of our agreement with CA Technologies, we are not able to discuss the results of the assessment in detail. Regardless, the case study demonstrates the general process behind using interaction modelling and emulation to produce interactive environments suitable for assisting in assessment of the run-time properties of enterprise software systems.
We conducted a series of experiments to ascertain the potential scalability of our enterprise software environment emulator as well to develop an understanding of the differences in computational costs when considering the rudimentary and expressive interaction stacks. The results of these experiments show that the interaction modelling and emulation approach can indeed scale, allowing up to ten-thousand endpoints to be emulated simultaneously on a single physical host. As expected, the scalability study showed that interaction specifications utilising the expressive model stack are more computationally expensive, both in terms of memory and time, than their rudimentary counterparts. Regardless, we were still able to emulate up to 10,000 expressive specifications using a single physical host machine. This suggests that human effort is a more important factor concerning model stack selection than computational resources. To summarise, so long as there is time available and a chance that data persistency will be required, choosing to use the expressive model stack for an interaction specification will not unduly affect scalability. If however, time is short and/or it is unlikely that data persistency will be required, a rudimentary interaction specification will suffice and probably save on the extra time and effort an expressive specification typically requires.

To reiterate, through enterprise endpoint interaction specification, experimentation and case study, this dissertation has demonstrated that:

Interaction modelling and emulation is an approach to producing interactive approximations of enterprise software environments capable of acting as and appearing to be real environments, enabling observation and measurement of the run-time properties of enterprise software systems. This approach is highly-scalable, well-suited to representing server-class interactions, and decoupled from implementation details of systems under investigation.
11.2 Future Work

Emulating enterprise software environments is, to the best of our knowledge, a novel technique in software engineering. Consequently there are numerous opportunities for future work. We discuss some of these opportunities here in no particular order of importance.

Modelling Client/Active Interactions

The focus of this work has been on modelling and emulating server or reactive types of interactions. Software systems which provide some means for others to establish connections and subsequently issue multiple requests. The lack of support for modelling and emulating client, or active types of interactions is a limitation of this work and something we intend to address in the future. The principal difference between active and reactive behaviours in this context is the triggering mechanism. Reactive, server-class interactions, which we have modelled throughout this work, remain dormant until triggered by some external party. Active interactions, on the other hand, require some other means for triggering communication, perhaps manually by some human, or through some timing mechanism. Load testing tools may be a source of guidance in this matter, client interactions being their speciality.

Scheduling

The round-robin algorithm currently used for scheduling interaction in our emulator is rather elementary. Run-time performance of the emulator may be significantly improved by considering the network traffic load on the services and channels being emulated when scheduling interaction. Channels under heavy load, for instance, may be scheduled with a higher priority or more frequently than those under lighter load.

An additional scheduling aspect to consider is emulating delay in request processing. The performance of systems in real enterprise environments is not uniform. It is necessary, therefore, that enterprise software systems be capable of handling interactions with systems that are slower to react than others. With this in mind it would be valuable if an enterprise environment emulator was capable of varying the interaction performance of emulated systems and thus reflect the varied system performance exhibited in real enterprise environments. One possible way to achieve this is through modifying the scheduling frequency or priority of certain systems so that they appear to have lower run-time performance than others.
11.2 Future Work

**Visualisation**

A fundamental objective of our research is to better enable assessment of the run-time properties of enterprise software systems. A crucial aspect of this quality assurance activity is the identification and diagnosis of issues with the run-time properties of the software system in question. Appropriate visual representation of an emulated enterprise environment could significantly help quality assurance teams in these activities. More specifically, the contents of messages may be displayed as they are received and transmitted by the various emulated systems providing a kind of visual trace of the communication during an emulation or after as a kind or replay. A bird’s-eye (top-down) view may be displayed during an emulation with some visual indicators for the various types of interaction which occur. Furthermore, a more detailed view of individual emulated systems may be provided showing the contents of its data store and how it changes over the course of an emulation. These as well as other visualisations may significantly help quality assurance teams in diagnosing run-time issues of enterprise software systems.

**Efficient Persistent Data Stores**

Some enterprise software systems have much larger persistent data stores than those we have experimented with to date. This may cause scalability issues when many such systems are emulated simultaneously, the demands on main memory could become substantial. Future work may investigate the effect larger data stores have on emulator scalability and suggest approaches which more efficiently handle sizeable persistent stores. In particular, there may be opportunities to share equivalent data and data structures between emulated systems, storing references to equivalent data or data structures rather than individual separate copies. Lazy approaches to data model representation may also be of value; data being generated on-demand as required by external systems and therefore only taking up as much main memory as is necessary for the interactions being carried out.

**Network Emulator Amalgamation**

We have been careful in our work to explicitly distance enterprise software environment emulation from emulation of the underlying communication infrastructure; network emulators already exist which provide powerful capability in this area. We do, however, recognise an opportunity to tether an enterprise environment emulator with a network emulator at the front end, so that effects caused by the underlying transport can be emulated in addition to the interaction level emulations provided by our enterprise environment emulator.
Faulty and Unexpected Behaviours

The software systems in real enterprise environments are not infallible. This is a reality which software systems must adequately handle if they intend to interact with other enterprise software systems. It would be valuable if some faulty system behaviour could be mimicked by an enterprise environment emulator so that the general robustness of systems could be ascertained. Either by human trigger or stochastically, an emulated system may knowingly transmit a message, which is not conformant with the protocol model, to check whether an external system can gracefully handle such unsociable behaviour. Furthermore, arbitrary changes to persistent data stores may be injected into emulated systems either stochastically or by human hand. This is entirely reasonable behaviour to expect of semi-independent systems as the change may be triggered by interaction with some other system, or by a human operator using that system. By exhibiting these and other faulty or unexpected behaviours an enterprise environment emulator can help assessment of the general robustness of enterprise software systems.

Specification Generation using Interaction Traces

The necessity of manual specification of enterprise software system interactions is a significant impediment to more widespread adoption of our interaction modelling and emulation approach. The activity can be arduous and time consuming. In order to ease this impediment we intend to investigate ways to use existing interaction traces, logs of messages being communicated between real enterprise software systems, to generate partial and perhaps even complete interaction specifications. The ability to generate partial interaction specifications from interaction traces is likely to significantly reduce the amount of human effort required to produce such specifications. Consequently, the effect of this impediment to adoption of enterprise environment emulation may be diminished.
Bibliography


Appendix A

Listings

This appendix contains key code listings which may aid understanding of how endpoint interactions are specified in the emulator we described in Chapter 8. It is organised into sections which contain code for specific languages.

A.1 Sample Kaluta Configuration by CSV File

Listing A.1: Sample Kaluta Configuration File Containing 200, Initially Identical, Expressive LDAP Servers

<table>
<thead>
<tr>
<th>host-name, port-num, model-stack, specification, data-src</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.1.1,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.2,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.3,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.4,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.5,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.6,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.7,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.8,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.9,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.10,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.11,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.12,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.13,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.14,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.15,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.16,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.17,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.18,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.19,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.20,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.21,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.22,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.23,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.24,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.25,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
<tr>
<td>10.0.1.26,1234, expressive, ldap-server, conf/initStates/100entries.ldif</td>
</tr>
</tbody>
</table>
Sample Kaluta Configuration by CSV File

<table>
<thead>
<tr>
<th>Address</th>
<th>Port</th>
<th>Protocol</th>
<th>Conf Directory</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.1.23</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.24</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.25</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.26</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.27</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.28</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.29</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.30</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.31</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.32</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.33</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.34</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.35</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.36</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.37</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.38</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.39</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.40</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.41</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.42</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.43</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.44</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.45</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.46</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.47</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.48</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.49</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.50</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.51</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.52</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.53</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.54</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.55</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.56</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.57</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.58</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.59</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.60</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.61</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.62</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.63</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.64</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.65</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.66</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.67</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.68</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.69</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.70</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.71</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.72</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.73</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.74</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.75</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.76</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.77</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
<tr>
<td>10.0.1.78</td>
<td>389</td>
<td>ldap</td>
<td>server, conf/initStates/100 entries</td>
<td>ldif</td>
</tr>
</tbody>
</table>
Sample Kaluta Configuration by CSV File

10.0.1.79,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.80,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.81,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.82,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.83,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.84,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.85,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.86,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.87,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.88,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.89,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.90,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.91,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.92,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.93,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.94,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.95,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.96,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.97,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.98,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.99,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.100,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.101,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.102,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.103,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.104,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.105,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.106,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.107,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.108,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.109,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.110,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.111,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.112,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.113,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.114,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.115,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.116,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.117,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.118,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.119,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.120,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.121,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.122,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.123,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.124,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.125,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.126,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.127,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.128,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.129,1234,expressive,ldap-server,conf/initStates/100entries.ldif
10.0.1.130,1234,expressive,ldap-server,conf/initStates/100entries.ldif
Sample Kaluta Configuration by CSV File

10.0.1.131,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.132,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.133,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.134,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.135,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.136,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.137,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.138,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.139,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.140,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.141,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.142,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.143,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.144,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.145,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.146,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.147,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.148,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.149,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.150,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.151,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.152,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.153,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.154,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.155,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.156,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.157,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.158,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.159,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.160,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.161,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.162,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.163,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.164,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.165,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.166,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.167,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.168,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.169,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.170,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.171,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.172,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.173,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.174,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.175,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.176,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.177,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.178,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.179,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.180,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.181,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
10.0.1.182,1234,expressive,ldap-server,conf/initStates/100 entries.ldif
A.2 Haskell

A.2.1 Message Shape Handlers

Listing A.2 contains the Haskell encoding of the LDAP message handler functions for the expressive LDAP directory endpoint. A number of functions and data types are used within this listing which have not been fully defined as they serve administrative purposes rather than being fundamental to the logic of the handlers. Functions with the “f” prefix, for instance, are generated by a Thrift IDL file we defined to facilitate communication between the network interface and the engine. The listing also relies on qualified imports of the “Data.Map” and “Data.Set” modules, as “Map” and “Set” respectively.

Listing A.2: Message Shape Handlers for Expressive LDAP Directory Server

```haskell
handleAddRq :: LDAPRequest -> LDAPDirDataStore ->
               Maybe (LDAPResponse, LDAPDirDataStore)
handleAddRq (LDAPRequest {f_LDAPRequest_messageID = Just msg_id, f_LDAPRequest_reqType = Just AddRqT, f_LDAPRequest_addRequest = Just (AddRequest {f_AddRequest_entry = Just entry_dn, f_AddRequest_attributes = Just attrs})))) dir_ds |
               not entry_exists && parent_exists =
               Just ([base_resp {f_LDAPResponseldapResult = Just successLDAPResult}],
               Map.insert entry_dn (foldr (\attr ->
                                           case attr of
                                           case attr of
```
Attribute { 
  f_Attribute_name = Just name,
  f_Attribute_value = Just vals} \rightarrow
Map.insert (AttrName name) vals
  \rightarrow undefined
) Map.empty attrs) dir_ds)

| entry_exists =
  Just (base Resp { 
    f_LDAPResponse_ldapResult = Just $ 
      mkLDAPResult EntryAlreadyExists ""
    "Entry already exists."}],
    dir_ds)
  }
| not parent_exists =
  Just (base Resp { 
    f_LDAPResponse_ldapResult = Just $ 
      mkLDAPResult NoSuchAttribute ""
    "Missing parent."}],
    dir_ds)
where entry_exists = Map.member entry dn dir_ds
    parent_exists = Map.member (parentOf entry_dn) dir_ds
    base Resp = mkBaseResponse msg_id AddResT

handleAddRq _ _ = Nothing

handleBindRq :: LDAPRequest \rightarrow LDAPDirDataStore \rightarrow
    Maybe ([LDAPResponse], LDAPDirDataStore)
handleBindRq (LDAPRequest { 
  f_LDAPRequest_messageID = Just msg_id,
  f_LDAPRequest_reqType = Just BindRqT,
  f_LDAPRequest_bindRequest = Just bind_rq }
} dir_ds
  — anonymous bind
| anonymous = successful bind_res
  — authenticated bind
| not anonymous &&& valid creds = successful bind_res
  — bad credentials
| not anonymous && not valid creds = bad creds res
where anonymous = all isNothing bind_rq_params

  bad creds res =
    Just ((mkBaseResponse msg_id BindResT) { 
      f_LDAPResponse_ldapResult =
        Just $ mkLDAPResult
      InvalidCredentials ""
    "Invalid bind credentials." },
    dir_ds)

  bind_rq_params = map ($ bind_rq) [
    f_BindRequest_name,
    f_BindRequest_password]
successful_bind_res =
  \[ (\{ mkBaseResponse msg_id BindResT \} \{ f_LDAPResponse_ldapResult =
     Just successLDAPResult \} ,
    dir_ds) \]

valid_creds
  | all isJust bind_rq_params =
  case bind_rq_params of
    [Just usr_dn, Just pwd] ->
      case Map.lookup usr_dn dir_ds of
        Just usr_attrs ->
          case Map.lookup (AttrName “userPassword”)
            usr_attrs of
              Just usr_pwd ->
                Set.singleton pwd == usr_pwd
                Nothing -> False
                Nothing -> False
                _ -> False
            otherwise = False

handleBindRq _ _ = Nothing

handleDelRq :: LDAPRequest -> LDAPDirDataStore ->
    \[ [[LDAPResponse], LDAPDirDataStore] \]
handleDelRq (LDAPRequest {f_LDAPRequest_messageID = Just msg_id, f_LDAPRequest_reqType = Just DelRqT, f_LDAPRequest_deleteRequest = Just (DeleteRequest {f_DeleteRequest_dn = Just del_rq_dn})) dir_ds

  | entry_exists = Just ([mk_del_res Success ""],
                        Map.delete del_rq_dn dir_ds)

  | not entry_exists = Just ([mk_del_res NoSuchObject
                             "No such object.", dir_ds)

where entry_exists = Map.member del_rq_dn dir_ds

mk_del_res res_code diagnos_msg =
  \{ mkBaseResponse msg_id DelResT \} \{ f_LDAPResponse_ldapResult =
    Just $ mkLDAPResult res_code ""
    diagnos_msg \}
handleDelRq _ _ = Nothing
handleModRq :: LDAPRequest -> LDAPDirDataStore ->
  Maybe ([LDAPResponse], LDAPDirDataStore)

handleModRq (LDAPRequest {f_LDAPRequest_messageID = Just msg_id,
  f_LDAPRequest_reqType = Just ModRqT,
  f_LDAPRequest_modifyRequest = Just (ModifyRequest {
    f_ModifyRequest_object =
      Just object,
    f_ModifyRequest_changes =
      Just changes}))
}) dir_ds

| object_exists = Just ([(mkBaseResponse msg_id ModResT) {
  f_LDAPResponse_ldapResult =
    Just successLDAPResult},
  Map.insert object obj' dir_ds])
|
| not object_exists = Just ([
  (mkBaseResponse msg_id ModResT) {
    f_LDAPResponse_ldapResult =
      Just $ mkLDAPResult NoSuchObject ""$ msg_id ModResT)
  Map.insert object obj' dir_ds])

where
  obj = dir_ds Map.! object
  obj' = foldl do_change obj changes
  object_exists = Map.member object dir_ds

  do_change obj'' (ModOperation {
    f_ModOperation_operation =
      Just mod_op_type,
    f_ModOperation_modification =
      Just Attribute {
        f_Attribute_name = Just name,
        f_Attribute_value = Just vals}
  }) = case mod_op_type of
      Add    -> Map.insertWith Set.union
        attr_name vals obj''
      DeleteOpType
        | Set.null vals -> Map.delete
          attr_name obj''
        | not $ Set.null vals ->
          Map.insertWith (Set.\()) attr_name vals obj''
      Replace
        | Set.null vals -> Map.delete
          attr_name obj''
        | not $ Set.null vals ->
Map.insert attr_name vals obj''
— catch all behaviour is to not modify
  _ -> obj''

where attr_name = AttrName name

do_change obj'' _ = obj''

handleModRq _ _ = Nothing

handleSearchRq :: LDAPRequest -> LDAPDirDataStore ->
        Maybe ([LDAPResponse], LDAPDirDataStore)

handleSearchRq (LDAPRequest {
  f_LDAPRequest_messageID = Just msg_id,
  f_LDAPRequest_reqType = Just SearchRqT,
  f_LDAPRequest_searchRequest = Just (SearchRequest {
    f_SearchRequest_baseObject =
      Just base_obj,
    f_SearchRequest_scope =
      Just scope,
    f_SearchRequest_typesOnly =
      Just types_only,
    f_SearchRequest_filter =
      Just filter_str,
    f_SearchRequest_attributes =
      Just attr_strs})
}) dir_ds

| base_obj_exists = Just (search_entry_msgs, dir_ds)
| not base_obj_exists = — no base object
Just ([base_search_res_done {
  f_LDAPResponse_ldapResult =
    Just \$ mkLDAPResult NoSuchObject "" "No_such_object.""],
  dir_ds)

where
— convert entry into a search entry response
as_resp name entry = (mkBaseResponse msg_id
          SearchResEntryT) {
  f_LDAPResponse_searchResEntry =
    Just SearchResultEntry {
      f_SearchResultEntry_objectName = Just name,
      f_SearchResultEntry_attributes = Just $ foldr (\(AttrName attr_name, attr_val) ->
        let attr_name' = Attribute (Just attr_name)
        in if types_only
          then (attr_name' Nothing :)
          else (attr_name' (Just attr_val) :)
      ) [] (Map.toList entry)\n
-- wrapped attribute strings
attrs = map AttrName attrs

-- Filter the attributes as well as values of attributes if necessary
attrs_filtered

-- attribute filtering unnecessary
| Prelude.null attrss = filtered_dir
-- return only the attributes listed
| not $ Prelude.null attrss =
  Map.map (Map.filterWithKey (const . flip elem attrss))
  filtered_dir
-- in all other cases, attribute filtering is not performed
| otherwise = filtered_dir

-- Does the base object exist?
base_object_exists = Map.member base_object dir_datasets

-- base successful search done message
base_search_res_done = (mkBaseResponse msg_id SearchResDoneT) {
  f LDAPResponse ldap_result =
    Just successLDAPResult
}

-- Apply the search filter
filtered_dir = case readFilter filter_str "" of
  Just filt -> Map.filter ('applyFilter' filt)
    scoped_dir
  Nothing -> scoped_dir

-- Scope the directory
scoped_dir = case scope of
  BaseObject -> Map.singleton base_object dir
    ds Map.! base_object
  SingleLevel -> Map.filterWithKey (const . isParentOf base_object) dir_datasets
  WholeSubtree -> Map.filterWithKey (const . isAncestorOf base_object) dir_datasets

-- the search entry messages
search_entry_msgs =
  foldr (\(name, entry) -> (as_resp name entry :))
    [base_search_res_done] (Map.toList attrss_filtered)

handleSearchRq _ = Nothing

handleUnbindRq :: LDAPRequest -> LDAPDirDataStore ->
A.2.2 Expressive LDAP Data Store

The following listing contains the data store model for the expressive LDAP directory endpoint. The same qualified imports are used as previously, i.e. “Data.Map” and “Data.Set” modules, as “Map” and “Set” respectively.

Listing A.3: Data Store Model for Expressive LDAP Directory Server

```haskell
newtype AttrName = AttrName String

type AttrVal = String

type DN = String

type Entry = Map.Map AttrName (Set.Set AttrVal)

type LDAPDirDataStore = Map.Map DN Entry

type LDAPExprDirDs = ExprDataStore LdapMsgShape LDAPRequest
                                        LDAPResponse LDAPDirDataStore
```

A.3 Java

The following listings contain the Java encodings of the network interface abstractions which allow the expressive LDAP directory server to communicate with real external LDAP clients.

A.3.1 LDAP Service

Listing A.4 contains the Java code for the LDAPService class which can be used by an emulator’s network interface to allow native LDAP clients to establish communication channels with an emulator.

Listing A.4: LDAPService

```java
package ldap;

import java.io.IOException;
import java.nio.channels.SocketChannel;

import netintf.services.BasicService;

import org.apache.thrift.transport.TTransportException;
import org.newsclub.net.unix.AFUNIXSocket;
```
A.3

Java

```java
public class LDAPService
    extends BasicService<LdapRpcConduit> {

    public LDAPService(String nodeId, int localId)
        throws IOException {
        super(nodeId, localId);
    }

    @Override
    public LdapRpcConduit accept()
        throws IOException, TTransportException {
        SocketChannel nextChan = extSrvSockChan.accept();
        if (nextChan != null)
            return new LdapRpcConduit(nextChan.socket(), nodeId, port);
        return null;
    }
}
```

A.3.2 LDAP Conduit

Listing A.5 contains the Java code for the LDAPConduit class which can be used by an emulator’s network interface to allow native LDAP clients to submit send LDAP requests to emulated LDAP directory servers.

```
package ldap;

import java.io.IOException;
import java.io.InterruptedIOException;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import netintf.services.BasicConduit;

import org.apache.thrift.TException;
import org.apache.thrift.transport.TTransportException;

import com.sun.ldap.protocol.AddRequest;
import com.sun.ldap.protocol.AddResponse;
import com.sun.ldap.protocol.BindRequest;
import com.sun.ldap.protocol.BindResponse;
import com.sun.ldap.protocol.DeleteRequest;
import com.sun.ldap.protocol.DeleteResponse;
import com.sun.ldap.protocol.LDAPAttribute;
```

Listing A.5: LDAPConduit
import com.sun.ldap.protocol.LDAPMessage;
import com.sun.ldap.protocol.LDAPModification;
import com.sun.ldap.protocol.ModifyRequest;
import com.sun.ldap.protocol.ModifyResponse;
import com.sun.ldap.protocol.ProtocolException;
import com.sun.ldap.protocol.SearchRequest;
import com.sun.ldap.protocol.SearchResultDone;
import com.sun.ldap.protocol.SearchResultEntry;
import com.sun.ldap.protocol.UnbindRequest;
import com.sun.slamd.asn1.ASN1Element;
import com.sun.slamd.asn1.ASN1Exception;
import com.sun.slamd.asn1.ASN1OctetString;
import com.sun.slamd.asn1.ASN1Reader;
import com.sun.slamd.asn1.ASN1Writer;

class LDAPConduit extends BasicConduit {
    private static final boolean debug = false;

    // readers and writers for the externally facing
    // ASN.1 channel
    private ASN1Reader asn1Reader;
    private ASN1Writer asn1Writer;

    private boolean moreWorkToDo;

    private ASN1Element currentRq;

    public LDAPConduit(Socket netSock, 
                       Socket engSock) 
                        throws IOException, TTransportException {
        super(engSock);

        // initialise the network facing readers and writers
        asn1Reader = new ASN1Reader(netSock);
        asn1Writer = new ASN1Writer(netSock);

        moreWorkToDo = true;

        currentRq = null;
    }

    public void conveyToEngine() 
                        throws IOException, TException {
        if (currentRq != null) 
            try {
                // managed to read an ASN.1 element,
                // decode it as an LDAPMessage and then
                // convert to a message
                // the engine will understand.
LDAPMessage ldapMsg =
    LDAPMessage.decode(currentRq);
// parse the native ldap request
LDAPRequest ldapRq =
    fromLDAPMessage(ldapMsg);
// transmit the parsed ldap request
ldapRq.write(engProt);
engTrans.flush();

if (ldapRq.getReqType() ==
    ldap.RequestType.UnbindRqT)
    moreWorkToDo = false;

if (debug)
    System.err.println(ldapRq.toString()
        + "---");

} catch (ProtocolException e) {
    e.printStackTrace();
}

public void conveyToNetwork()
    throws TException, IOException {
    // only read if there is data to be read
    while (hasTrafficFromEngine() && moreWorkToDo) {
        try {
            LDAPResponse response = new LDAPResponse();
            response.read(engProt);
            response.validate();

            LDAPMessage resMsg = null;
            // parse the protocol op
            ProtocolOp protOp = null;

            if (response.getResType() ==
                ResponseType.SearchResEntryT) {
                // search result entries are a special
                // case
                ldap.SearchResultEntry entry =
                    response.getSearchResEntry();

                LDAPAttribute[] ldapAttrArr =
                    new LDAPAttribute[entry
                        .getAttributes().size()];

                List<LDAPAttribute> ldapAttrList =
                    new ArrayList<LDAPAttribute>(
                        entry.getAttributes().size());

                for (int i = 0; i < ldapAttrArr.length; i++) {
                    ldapAttrList.add(ldapAttrArr[i]);
                }

                for (LDAPAttribute attr : ldapAttrList) {
                    // process each attribute
                }

            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
for (Attribute attr : 
    entry.getAttributes()) {

    ASN1OctetString[] valArr = 
        new ASN1OctetString[attr
        .getValue().size()];

    List<ASN1OctetString> valList = 
        new ArrayList<ASN1OctetString>(
            attr.getValue().size());

    for (String val : attr.getValue())
        valList.
            add(new
                ASN1OctetString(val));

    valList.toArray(valArr);

    ldapAttrList.
        add(new
            LDAPAttribute
            (attr.getName(), valArr));

    ldapAttrList.toArray(ldapAttrArr);

    protOp = new SearchResultEntry
        (entry.getObjectName(),
         ldapAttrArr);

    resMsg = new LDAPMessage
        (response.getMessageID(), protOp);

    // send the encoded ldap response
    asn1Writer.writeElement
        (resMsg.encode());

} else {
    // all other results are simply wrapped
    // ldap results
    LDAPResult result =
        response.getLdapResult();
    switch (response.getResType()) {
      case AddResT:
        protOp = new AddResponse
            (result.getResultCode()
            .getValue(),
            result.getMatchedDN(), result
            .getDiagnosticMessage());

        resMsg = new LDAPMessage
            (response.getMessageID(),
            }
// send the encoded ldap response
asn1Writer.writeElement
(resMsg.encode());
asn1Writer.flush();
break;

case BindResT:
  protOp =
    new BindResponse
      (result.getResultCode(),
       result.getValue(),
       result.getMatchedDN(), result
       .getDiagnosticMessage());
  resMsg =
    new LDAPMessage
      (response.getMessageID(),
       protOp);

// send the encoded ldap response
asn1Writer.writeElement
(resMsg.encode());
asn1Writer.flush();
break;

case DelResT:
  protOp = new DeleteResponse
    (result.getResultCode(),
     result.getValue(),
     result.getMatchedDN(), result
     .getDiagnosticMessage());
  resMsg =
    new LDAPMessage
      (response.getMessageID(),
       protOp);

// send the encoded ldap response
asn1Writer.writeElement
(resMsg.encode());
asn1Writer.flush();
break;

case ModResT:
  protOp =
    new ModifyResponse
      (result.getResultCode(),
       result.getValue(),
       result.getMatchedDN(), result
       .getDiagnosticMessage());
  resMsg = new LDAPMessage
    (response.getMessageID(),
     protOp);
// send the encoded ldap response
asn1Writer.writeElement
    (resMsg.encode());
asn1Writer.flush();
break;
case SearchResDoneT:
    protOp = new
        SearchResultDone
        (result.getResultCode()
            .getValue(),
            result.getMatchedDN(),
            result.getDiagnosticMessage());
    resMsg = new
        LDAPMessage
        (response.getMessageID(),
        protOp);
    // send the encoded ldap response
    asn1Writer.writeElement
        (resMsg.encode());
    asn1Writer.flush();
    break;
default:
    throw new IOException
        ("Cannot parse engine response:
" + response.toString());
}

if (debug)
    System.err.println(response.toString() + 
        "\n-------------");
}
}

private static LDAPRequest fromLDAPMessage
    (LDAPMessage ldapMsg) throws IOException {

    LDAPRequest ldapRq = new LDAPRequest();
    ldapRq.setMessageID(ldapMsg.getMessageID());


    // Add Request
    if (protocolOp instanceof AddRequest) {
        AddRequest addRqOp = (AddRequest) protocolOp;

        ldap.AddRequest addRq =
            new ldap.AddRequest(addRqOp.getDN(),
            addRqOp.getAttributes());
    }
convertLDAPAttributes(addRqOp.
    getAttributes ())) ;

ldapRq. setReqType(RequestType.AddRqT) ;
ldapRq. setAddRequest(addRq) ;
}

// Bind Request
else if (protocolOp instanceof BindRequest) {
    BindRequest bindRqOp = (BindRequest) protocolOp ;

    ldap. BindRequest bindRq = new
    ldap. BindRequest ( (short) bindRqOp
        . getProtocolVersion () ) ;
    bindRq. setName(bindRqOp . getBindDN () ) ;
    bindRq. setPassword(bindRqOp . getBindPassword () ) ;

    ldapRq. setReqType(RequestType.BindRqT) ;
    ldapRq. setBindRequest(bindRq) ;
}

// Delete Request
else if (protocolOp instanceof DeleteRequest) {
    DeleteRequest delRqOp = (DeleteRequest) protocolOp ;

    ldap. DeleteRequest delRq = new
    ldap. DeleteRequest(delRqOp. getDN () ) ;

    ldapRq. setReqType(RequestType.DelRqT) ;
    ldapRq. setDeleteRequest(delRq) ;
}

// Modify Request
else if (protocolOp instanceof ModifyRequest) {
    ModifyRequest modRqOp = (ModifyRequest) protocolOp ;

        getModifications () ;
    List<ModOperation> changes = new
        ArrayList<ModOperation>(
            ldapMods. length ) ;

    for ( int i = 0 ; i < ldapMods. length ; i ++) {
        changes. add
            ( new ModOperation
                ( ModOpType. findByValue ( ldapMods [ i ]
                    . getModType () ) ,
                convertLDAPAttribute ( ldapMods [ i ]
                    . getAttribute () ) ) ) ;
    }
ldap.ModifyRequest modRq = new ldap.ModifyRequest(modRqOp.getDN(), changes);

ldapRq.setReqType(RequestType.ModRqT);
ldapRq.setModifyRequest(modRq);
}

// Search Request
else if (protocolOp instanceof SearchRequest) {
    SearchRequest searchRqOp = (SearchRequest) protocolOp;

        Scope.findByValue(searchRqOp.getScope()),
        searchRqOp.getTypesOnly(), searchRqOp.getFilter().toString(),
        Arrays.asList(searchRqOp.getAttributes()));

    ldapRq.setReqType(RequestType.SearchRqT);
    ldapRq.setSearchRequest(searchRq);
}

// Unbind Request
else if (protocolOp instanceof UnbindRequest)
    ldapRq.setReqType(RequestType.UnbindRqT);

// Unknown request type
else
    throw new IOException("Cannot parse LDAP request message: "+ ldapMsg);

return ldapRq;

private static Attribute convertLDAPAttribute(LDAPAttribute ldapAttr) {
    Attribute attr = new Attribute();
    attr.setName(ldapAttr.getType());

    ASN1OctetString[] vals = ldapAttr.getValues();
    for (int j = 0; j < vals.length; j++)
        attr.addToValue(vals[j].getStringValue());

    return attr;
private static List<Attribute> convertLDAPAttributes(LDAPAttribute[] ldapAttrs) {
    List<Attribute> attrs = new ArrayList<Attribute>();

    for (int i = 0; i < ldapAttrs.length; i++) {
        attrs.add(convertLDAPAttribute(ldapAttrs[i]));
    }

    return attrs;
}

@Override
public boolean pollTrafficFromNetwork() throws IOException {
    boolean hasPolledElement = false;
    try {
        ASN1Element polledElement = asn1Reader.readElement(1);
        if (polledElement != null) {
            this.currentRq = polledElement;
            hasPolledElement = true;
        }
    } catch (ASN1Exception e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        hasPolledElement = false;
    }
    return hasPolledElement;
}

@Override
public boolean hasTrafficFromEngine() throws IOException {
    return engTrans.getSocket().getInputStream().available() > 0;
}

@Override
public boolean hasMoreWorkToDo() {
    return moreWorkToDo;
}
Appendix B

Numerical Experiment Data

In this appendix we provide the most significant data collected from the experiments described in Chapter 10. We present this data separately so that in the experiments chapter itself we are able to focus on discussion and analysis of more general relationships and trends. This being more pertinent, at least for the moment, than the specific numeric data which was collected.

B.1 Memory

Tables B.1 and B.2 contain the numeric five-number summary data for peak memory usage of the network interface when emulating rudimentary and expressive specifications, respectively. Apart from the leftmost column, i.e. the column containing the number of emulated endpoints, the units of each number is kilobytes (kB). The box-plot of this numerical data can be found in Figure 10.1.

<table>
<thead>
<tr>
<th>Endpoints</th>
<th>Minimum</th>
<th>First-Quartile</th>
<th>Median</th>
<th>Third-Quartile</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>406828</td>
<td>409320</td>
<td>797554</td>
<td>797724</td>
<td>2007420</td>
</tr>
<tr>
<td>10</td>
<td>408020</td>
<td>423492</td>
<td>780556</td>
<td>797570</td>
<td>1260364</td>
</tr>
<tr>
<td>20</td>
<td>408664</td>
<td>410552</td>
<td>629852</td>
<td>641952</td>
<td>1259224</td>
</tr>
<tr>
<td>55</td>
<td>363872</td>
<td>390894</td>
<td>697752</td>
<td>698090</td>
<td>1098324</td>
</tr>
<tr>
<td>150</td>
<td>394296</td>
<td>394296</td>
<td>659174</td>
<td>682280</td>
<td>787300</td>
</tr>
<tr>
<td>400</td>
<td>457860</td>
<td>457860</td>
<td>811768</td>
<td>815292</td>
<td>1623860</td>
</tr>
<tr>
<td>3000</td>
<td>513180</td>
<td>513180</td>
<td>1301274</td>
<td>1304308</td>
<td>1940096</td>
</tr>
<tr>
<td>8000</td>
<td>873712</td>
<td>873712</td>
<td>2055502</td>
<td>2058248</td>
<td>3536944</td>
</tr>
<tr>
<td>10000</td>
<td>456736</td>
<td>456736</td>
<td>2666732</td>
<td>3256196</td>
<td>3269092</td>
</tr>
</tbody>
</table>

Table B.1: Peak Memory - Network Interface - Rudimentary Endpoints

Tables B.3 and B.4 present more five-number summary data regarding peak memory usage, in this instance however, the engine component of the emulator is considered. The units are again kilobytes and there is a box-plot which presents this summary data, which can be found in log-log scale in Figure 10.2 and log-linear scale in Figure 10.3.
### Table B.2: Peak Memory - Network Interface - Expressive Endpoints

<table>
<thead>
<tr>
<th>Endpoints</th>
<th>Minimum</th>
<th>First-Quartile</th>
<th>Median</th>
<th>Third-Quartile</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>216392</td>
<td>250244</td>
<td>796704</td>
<td>797156</td>
<td>1262752</td>
</tr>
<tr>
<td>10</td>
<td>387392</td>
<td>419308</td>
<td>423468</td>
<td>649410</td>
<td>806816</td>
</tr>
<tr>
<td>20</td>
<td>359692</td>
<td>415920</td>
<td>418262</td>
<td>651432</td>
<td>656656</td>
</tr>
<tr>
<td>55</td>
<td>217148</td>
<td>292758</td>
<td>422504</td>
<td>482402</td>
<td>651108</td>
</tr>
<tr>
<td>150</td>
<td>23680</td>
<td>23680</td>
<td>388868</td>
<td>388876</td>
<td>823600</td>
</tr>
<tr>
<td>400</td>
<td>174284</td>
<td>174284</td>
<td>299016</td>
<td>32324</td>
<td>762240</td>
</tr>
<tr>
<td>1000</td>
<td>341476</td>
<td>341476</td>
<td>629132</td>
<td>683808</td>
<td>813708</td>
</tr>
<tr>
<td>8000</td>
<td>739368</td>
<td>739368</td>
<td>1129462</td>
<td>1318356</td>
<td>2022324</td>
</tr>
<tr>
<td>10000</td>
<td>751984</td>
<td>751984</td>
<td>1212976</td>
<td>1325924</td>
<td>2146788</td>
</tr>
</tbody>
</table>

### Table B.3: Peak Memory - Engine - Rudimentary Endpoints

<table>
<thead>
<tr>
<th>Endpoints</th>
<th>Minimum</th>
<th>First-Quartile</th>
<th>Median</th>
<th>Third-Quartile</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11764</td>
<td>11764</td>
<td>11768</td>
<td>11768</td>
<td>11768</td>
</tr>
<tr>
<td>10</td>
<td>13096</td>
<td>13096</td>
<td>13124</td>
<td>13124</td>
<td>14156</td>
</tr>
<tr>
<td>20</td>
<td>13124</td>
<td>13128</td>
<td>13184</td>
<td>13676</td>
<td>13988</td>
</tr>
<tr>
<td>55</td>
<td>15240</td>
<td>15592</td>
<td>16772</td>
<td>16774</td>
<td>17040</td>
</tr>
<tr>
<td>150</td>
<td>11716</td>
<td>11716</td>
<td>20150</td>
<td>20164</td>
<td>24228</td>
</tr>
<tr>
<td>400</td>
<td>40420</td>
<td>40420</td>
<td>43296</td>
<td>44140</td>
<td>47828</td>
</tr>
<tr>
<td>1000</td>
<td>68664</td>
<td>68664</td>
<td>94812</td>
<td>109072</td>
<td>142788</td>
</tr>
<tr>
<td>3000</td>
<td>143928</td>
<td>143928</td>
<td>335312</td>
<td>424296</td>
<td>443956</td>
</tr>
<tr>
<td>8000</td>
<td>545808</td>
<td>545808</td>
<td>573556</td>
<td>575308</td>
<td>714772</td>
</tr>
<tr>
<td>10000</td>
<td>596452</td>
<td>596452</td>
<td>649626</td>
<td>656036</td>
<td>949988</td>
</tr>
</tbody>
</table>

### Table B.4: Peak Memory - Engine - Expressive Endpoints
B.2 CPU

More five-number summary data is presented in Tables B.5 and B.6. The data in these tables summarise the CPU time use of the engine when emulating various numbers of rudimentary and expressive endpoints. The time usage is presented in the format (mm:ss), where mm are minutes and ss are seconds.

<table>
<thead>
<tr>
<th>Endpoints</th>
<th>Minimum</th>
<th>First-Quartile</th>
<th>Median</th>
<th>Third-Quartile</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
</tr>
<tr>
<td>10</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
</tr>
<tr>
<td>20</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
</tr>
<tr>
<td>55</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
</tr>
<tr>
<td>150</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
</tr>
<tr>
<td>400</td>
<td>00:02</td>
<td>00:02</td>
<td>00:02</td>
<td>00:02</td>
<td>00:02</td>
</tr>
<tr>
<td>1000</td>
<td>00:05</td>
<td>00:05</td>
<td>00:05</td>
<td>00:05</td>
<td>00:05</td>
</tr>
<tr>
<td>3000</td>
<td>00:15</td>
<td>00:16</td>
<td>00:17</td>
<td>00:18</td>
<td>00:24</td>
</tr>
<tr>
<td>8000</td>
<td>00:45</td>
<td>00:48</td>
<td>00:49</td>
<td>01:11</td>
<td>01:20</td>
</tr>
<tr>
<td>10000</td>
<td>01:02</td>
<td>01:03</td>
<td>01:10</td>
<td>01:10</td>
<td>01:11</td>
</tr>
</tbody>
</table>

Table B.5: CPU Use - Engine - Rudimentary Endpoints

<table>
<thead>
<tr>
<th>Endpoints</th>
<th>Minimum</th>
<th>First-Quartile</th>
<th>Median</th>
<th>Third-Quartile</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
</tr>
<tr>
<td>10</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
</tr>
<tr>
<td>20</td>
<td>00:01</td>
<td>00:01</td>
<td>00:01</td>
<td>00:01</td>
<td>00:01</td>
</tr>
<tr>
<td>55</td>
<td>00:03</td>
<td>00:03</td>
<td>00:03</td>
<td>00:03</td>
<td>00:03</td>
</tr>
<tr>
<td>150</td>
<td>00:10</td>
<td>00:10</td>
<td>00:10</td>
<td>00:10</td>
<td>00:10</td>
</tr>
<tr>
<td>400</td>
<td>00:28</td>
<td>00:28</td>
<td>00:28</td>
<td>00:28</td>
<td>00:28</td>
</tr>
<tr>
<td>1000</td>
<td>01:18</td>
<td>01:18</td>
<td>01:24</td>
<td>01:28</td>
<td>01:33</td>
</tr>
<tr>
<td>3000</td>
<td>04:38</td>
<td>04:38</td>
<td>06:57</td>
<td>08:20</td>
<td>08:20</td>
</tr>
</tbody>
</table>

Table B.6: CPU Use - Engine - Rudimentary Endpoints

Table B.7 presents the data relating CPU use and network throughput. CPU Time denotes the total CPU time used by the engine when emulating the various numbers of rudimentary endpoints. RX and TX denote the total network traffic received and transmitted respectively while CPU : RX + TX is the ratio of total CPU time in seconds, with total network throughput, i.e. CPU seconds required for processing a byte of network traffic. Table B.8 presents the same data except for emulation of expressive endpoint specifications. The data is plotted in Chapter 10, Figure 10.6.
### Table B.7: CPU Use : Processed Network Traffic - Rudimentary Endpoints

<table>
<thead>
<tr>
<th>Endpoints</th>
<th>CPU Time (sec)</th>
<th>RX (bytes)</th>
<th>TX (bytes)</th>
<th>CPU : RX + TX</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>12</td>
<td>4873126</td>
<td>7075128</td>
<td>1.4347583456844107 × 10^{-7}</td>
</tr>
<tr>
<td>1000</td>
<td>30</td>
<td>12158386</td>
<td>17591776</td>
<td>1.440568386052582 × 10^{-7}</td>
</tr>
<tr>
<td>3000</td>
<td>107</td>
<td>36460850</td>
<td>53743866</td>
<td>1.6945582186317495 × 10^{-7}</td>
</tr>
<tr>
<td>8000</td>
<td>342</td>
<td>97208382</td>
<td>143555032</td>
<td>2.0292594312997594 × 10^{-7}</td>
</tr>
<tr>
<td>10000</td>
<td>406</td>
<td>121593270</td>
<td>178235912</td>
<td>1.934434787605164 × 10^{-7}</td>
</tr>
</tbody>
</table>

### Table B.8: CPU Use : Processed Network Traffic - Expressive Endpoints

<table>
<thead>
<tr>
<th>Endpoints</th>
<th>CPU Time (sec)</th>
<th>RX (bytes)</th>
<th>TX (bytes)</th>
<th>CPU : RX + TX</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>8</td>
<td>566332</td>
<td>12309881</td>
<td>6.903340981458515 × 10^{-8}</td>
</tr>
<tr>
<td>55</td>
<td>21</td>
<td>1266882</td>
<td>31692976</td>
<td>7.964233341053836 × 10^{-8}</td>
</tr>
<tr>
<td>150</td>
<td>50</td>
<td>2314522</td>
<td>61662204</td>
<td>1.302557016333042 × 10^{-7}</td>
</tr>
<tr>
<td>400</td>
<td>168</td>
<td>7164410</td>
<td>197133534</td>
<td>1.1747548472636612 × 10^{-7}</td>
</tr>
<tr>
<td>1000</td>
<td>504</td>
<td>17676210</td>
<td>492717418</td>
<td>1.4106759185481054 × 10^{-7}</td>
</tr>
<tr>
<td>3000</td>
<td>2390</td>
<td>52664318</td>
<td>1478377846</td>
<td>2.2300402918790643 × 10^{-7}</td>
</tr>
<tr>
<td>8000</td>
<td>8694</td>
<td>139903564</td>
<td>3942341264</td>
<td>3.042443685594645 × 10^{-7}</td>
</tr>
<tr>
<td>10000</td>
<td>12475</td>
<td>175558352</td>
<td>4928148540</td>
<td>3.4918597302998435 × 10^{-7}</td>
</tr>
</tbody>
</table>

**Numerical Experiment Data**
We now move onto the data collected regarding the workload which was performed by our emulator. Tables B.9 and B.10 present the five-number summary data of the average times, in milliseconds (ms), taken to complete the entire workload emulating rudimentary and expressive endpoint specifications respectively. This data is presented as a box-plot in Figure 10.7 and analysed with the other results in the Section 10.2 of Chapter 10.

<table>
<thead>
<tr>
<th>Endpoints</th>
<th>Minimum</th>
<th>First-Quartile</th>
<th>Median</th>
<th>Third-Quartile</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>390</td>
<td>390</td>
<td>393.5</td>
<td>430</td>
<td>441</td>
</tr>
<tr>
<td>10</td>
<td>229</td>
<td>280</td>
<td>314</td>
<td>463</td>
<td>508</td>
</tr>
<tr>
<td>20</td>
<td>230</td>
<td>289</td>
<td>300</td>
<td>310</td>
<td>499</td>
</tr>
<tr>
<td>55</td>
<td>230</td>
<td>259</td>
<td>271</td>
<td>300</td>
<td>488</td>
</tr>
<tr>
<td>150</td>
<td>230</td>
<td>443</td>
<td>465</td>
<td>474</td>
<td>508</td>
</tr>
<tr>
<td>400</td>
<td>229</td>
<td>280</td>
<td>314</td>
<td>463</td>
<td>482</td>
</tr>
<tr>
<td>1000</td>
<td>230</td>
<td>289</td>
<td>300</td>
<td>310</td>
<td>499</td>
</tr>
<tr>
<td>3000</td>
<td>230</td>
<td>289</td>
<td>300</td>
<td>310</td>
<td>499</td>
</tr>
<tr>
<td>8000</td>
<td>230</td>
<td>289</td>
<td>300</td>
<td>310</td>
<td>499</td>
</tr>
<tr>
<td>10000</td>
<td>230</td>
<td>289</td>
<td>300</td>
<td>310</td>
<td>499</td>
</tr>
</tbody>
</table>

Table B.9: Average Time (ms) to Complete Total Workload - Rudimentary Endpoints

<table>
<thead>
<tr>
<th>Endpoints</th>
<th>Minimum</th>
<th>First-Quartile</th>
<th>Median</th>
<th>Third-Quartile</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>593</td>
<td>599</td>
<td>600.5</td>
<td>601</td>
<td>612</td>
</tr>
<tr>
<td>10</td>
<td>300</td>
<td>1271</td>
<td>1324</td>
<td>1364</td>
<td>1409</td>
</tr>
<tr>
<td>20</td>
<td>270</td>
<td>758</td>
<td>840.5</td>
<td>1321</td>
<td>1397</td>
</tr>
<tr>
<td>55</td>
<td>260</td>
<td>640</td>
<td>748</td>
<td>821.5</td>
<td>1434</td>
</tr>
<tr>
<td>150</td>
<td>299</td>
<td>630</td>
<td>750</td>
<td>800</td>
<td>1409</td>
</tr>
<tr>
<td>400</td>
<td>290</td>
<td>660</td>
<td>768</td>
<td>820</td>
<td>1421</td>
</tr>
<tr>
<td>1000</td>
<td>280</td>
<td>714</td>
<td>820</td>
<td>900</td>
<td>1454</td>
</tr>
<tr>
<td>3000</td>
<td>270</td>
<td>839</td>
<td>970</td>
<td>1280</td>
<td>2981</td>
</tr>
<tr>
<td>8000</td>
<td>300</td>
<td>1020</td>
<td>1300</td>
<td>1910</td>
<td>4410</td>
</tr>
<tr>
<td>10000</td>
<td>300</td>
<td>1120</td>
<td>1490</td>
<td>2240</td>
<td>5660</td>
</tr>
</tbody>
</table>

Table B.10: Average Time (ms) to Complete Total Workload - Expressive Endpoints

Table B.11 contains data regarding average time performance of each individual workload task when emulating 1000 endpoints. Both data for emulations of rudimentary and expressive endpoints are included. All units are milliseconds (ms). The results are plotted as a histogram in Figure 10.8 and discussed along with the other experiment results in Section 10.2 of Chapter 10.

Numerical Experiment Data
<table>
<thead>
<tr>
<th>Task</th>
<th>Rudimentary Median (ms)</th>
<th>Expressive Median (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bind</td>
<td>24</td>
<td>36</td>
</tr>
<tr>
<td>whole-dir-search</td>
<td>48</td>
<td>525</td>
</tr>
<tr>
<td>add</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>subtree-search</td>
<td>46</td>
<td>85</td>
</tr>
<tr>
<td>modify</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>single-entry-search</td>
<td>47</td>
<td>52</td>
</tr>
<tr>
<td>delete</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>unbind-and-close</td>
<td>85</td>
<td>86</td>
</tr>
</tbody>
</table>

Table B.11: Median Time to Complete Workload Tasks - 1000 Emulated Endpoints