An Integrated Methodology for Creating Composed Web/Grid Services

A thesis submitted in accordance with the requirements of
the University of Stirling for the degree of Doctor of Philosophy

by

Koon Leai Larry Tan

Department of Computing Science & Mathematics
University of Stirling, Scotland, UK

September 2010

Declaration

I hereby declare that the work presented in this thesis is my own original work unless otherwise indicated in the text, and that it has not been submitted for any other degree or award. Where work presented in this thesis appears in publications of which I am a named author, references are indicated in the text.

In particular, I note the following contributions from others:

- The original CRESS, MUSTARD, and CHIVE tools were used as the basis for the development of this thesis, though I have significantly extended these. The original developer was my supervisor Kenneth J. Turner. CRESS has been extended by me with grid services as a new domain, with new notation to support a wider range of compositions, and with new code generation to support the extended notation as well as WS-BPEL 2.0 as a new target language. Tool integration was extended by me to include the newly developed tools CLOVE and MINT for verification and implementation validation. MUSTARD was further developed by me to validate partner services, its notation being extended to support expressions such as variables. I have developed the integration with MINT for implementation validation. The CHIVE interface was lightly extended to include the ability to invoke formal verification.
- The translation of MUSTARD tests for use in MINT presented in section 6.3.4 was developed by my supervisor Kenneth J. Turner, though the MINT interpreter is my own work. The automated annotation of LOTOS specifications presented in section 5.4.2.2 was partly developed by my supervisor Kenneth J. Turner. My contribution was in extending this to annotate values for base types and to automatically generate information for dynamic potential annotations.
- The composite lender service and its partner specification used in one case study and in the thesis were originally developed by my supervisor Kenneth J. Turner, though I extended and applied these in various ways to demonstrate the extensions made to the CRESS capabilities.
- The RAFMC (regular alternation-free mu-calculus) verification patterns were defined by Radu

Mateescu of the VASY team who developed the CADP toolkit. I used these verification patterns and adapted its single predicate actions for multiple predicate use as property templates in the CLOVE tool which I developed. I also designed the CLOVE verification language myself, using basic ideas from the RAFMC work.

Koon Leai Larry Tan

September 2010

Abstract

This thesis presents an approach to design, specify, validate, verify, implement, and evaluate composed web/grid services. Web and grid services can be composed to create new services with complex behaviours. The BPEL (Business Process Execution Language) standard was created to enable the orchestration of web services, but there have also been investigation of its use for grid services. BPEL specifies the implementation of service composition but has no formal semantics; implementations are in practice checked by testing. Formal methods are used in general to define an abstract model of system behaviour that allows simulation and reasoning about properties. The approach can detect and reduce potentially costly errors at design time.

CRESS (Communication Representation Employing Systematic Specification) is a domain-independent, graphical, abstract notation, and integrated toolset for developing composite web service. The original version of CRESS had automated support for formal specification in LOTOS (Language Of Temporal Ordering Specification), executing formal validation with MUSTARD (Multiple-Use Scenario Testing and Refusal Description), and implementing in BPEL4WS as the early version of BPEL standard. This thesis work has extended CRESS and its integrated tools to design, specify, validate, verify, implement, and evaluate composed web/grid services. The work has extended the CRESS notation to support a wider range of service compositions, and has applied it to grid services as a new domain. The thesis presents two new tools, CLOVE (CRESS Language-Oriented Verification Environment) and MINT (MUSTARD Interpreter), to respectively support formal verification and implementation testing. New work has also extended CRESS to automate implementation of composed services using the more recent BPEL standard WS-BPEL 2.0.

Acknowledgements

I am grateful to the Department Computing Science & Mathematics, the GEODE and DAMES projects, an Overseas Research Studentship, and a Thomas and Margaret Roddan Trust bursary for the financial support over my course of study. This support has enabled the author to develop quite a collection of 'herbs' and 'spices'! Many thanks to the researchers in the GEODE and DAMES projects (Stirling and NeSC Glasgow) whom I have had opportunity to collaborate with; this has inspired the application of the thesis to the domain of grid service composition. Thanks to Hubert Garavel, Frédéric Lang, Wendelin Serwe, and Radu Mateescu of the CADP team for their technical advice on verification.

I would also like to thank all the academic, secretarial and technical staff within the Department who have supported me in many ways.

To all the PhD students in the Department for their network support, where there is invaluable exchange of views, knowledge, and laughter. To my family and friends at home and in Stirling who are a constant support and sanctuary for me, sharing my ranting and laughter.

Many thanks to my supervisors Prof. Kenneth J. Turner and Prof. Evan Magill for their technical advice and constantly presenting opportunities to expose my research. Many thanks to Prof. Kenneth J. Turner once more for guidance, kindness, encouragement, concerns and so much more that I'm grateful for and always fondly remind myself of: rounding out the "herbal" formula, the CRESS hydra, catch twenty-twos, the 'woman who swallowed the fly', red ink, etc. which have helped to shape the thesis presented here.

List of Publications

Based on the work presented in this thesis, the following technical reports and papers have been published, listed in chronological order.

1. K. L. L. Tan and K. J. Turner. Orchestrating grid services using BPEL and Globus Toolkit 4. In M. Merabti, R. Pereira, C. Oliver, and O. Abuelma'atti, editors, *Proc. 7th PGNet Symposium*, pages 31–36. School of Computing, Liverpool John Moores University, Liverpool, UK, June 2006.

2. P. S. Lambert, K. L. L. Tan, K. J. Turner, V. Gayle, K. Prandy, and R. O. Sinnott. Development of a grid enabled occupational data environment. In R. Procter, editor, *Proc. 2nd. International Conference on e-Social Science*, pages 1–12, Manchester, UK, June 2006. National Centre for e-Social Science.

3. K. L. L. Tan, V. Gayle, P. S. Lambert, R. O. Sinnott, and K. J. Turner. GEODE – Sharing occupational data through the grid. In S. J. Cox, editor, *Proc. 5th. UK e-Science All Hands Meeting*, pages 534–541. National e-Science Centre, Edinburgh, Sept. 2006.

4. P. S. Lambert, K. L. L. Tan, K. J. Turner, V. Gayle, R. Sinnott, and K. Prandy. Data curation standards and the messy world of social science occupational information resources. In *Proc. 2nd. International Digital Curation Conference*, pages 2.1–2.8, Glasgow, Nov. 2006.

5. K. J. Turner and K. L. L. Tan. Graphical composition of grid services. In D. Buchs and N. Guelfi, editors, *Rapid Introduction of Software Engineering Techniques*, number 4401 in Lecture Notes in Computer Science, pages 1–17, Berlin, Germany, May 2007. Springer.

6. K. J. Turner and K. L. L. Tan. A rigorous approach to orchestrating grid services. *Computer Networks*, 51(15):4421–4441, Oct. 2007.

7. K. L. L. Tan and K. J. Turner. Automated analysis and implementation of composed grid services. In D. Dranidis and I. Sakellariou, editors, *Proc. 3rd South-East European Workshop on Formal Methods*, pages 51–64. South-East European Reseach Centre, Thessaloniki, Greece, Nov. 2007.

8. K. J. Turner and K. L. L. Tan. A rigorous methodology for composing services. In M. Alpuente, B. Cook, and C. Joubert, editors, *Proc. 14th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2009)*, number 5825 in Lecture Notes in Computer Science, Eindhoven, The Netherlands, Nov. 2009.

9. K. L. L. Tan, P. S. Lambert, K. J. Turner, J. Blum, A. Bowes, D. N. F. Bell, V. Gayle, S. B. Jones, M. Maxwell, R. O. Sinnott, and G. Warner. Enabling quantitative data analysis through e-infrastructure. In *Social Science Computer Review*, volume 27, pages 539–552. SAGE Publications, Nov. 2009.

10. K. L. L. Tan. Case studies using CRESS to develop web and grid services. Technical Report CSM-183, Department of Computing Science and Mathematics, University of Stirling, UK, Dec. 2009.

Contents

Declaration i

Abstract iii

Acknowledgements iv

List of Publications v

Contents xiii

List of Figures xv

List of Tables xvii

List of Abbreviations xviii

1 Introduction 1

1.1 Motivation 1

1.2 Context 2

1.2.1 Services And Their Composition 2

1.2.2 Design-Time Analysis of Service Compositions 2

1.2.3 Implementation and Testing of Service Compositions 2

1.3 Objectives 2

1.3.1 Scope and Assumptions 3

1.4 Achievements 4

1.4.1 Supported Analysis 4

1.5 Thesis Structure 5

2 Background 6

2.1 Services 6

2.1.1 Service-Oriented Architecture Concept 6

2.1.2 Web Service Concepts 7

2.1.3 Grid Service Concepts 9

2.2 Service Description . . . 11
2.2.1 Data Definition . . . 11
2.2.1.1 XML Schema . . . 11
2.2.2 Service Interface . . . 12
2.2.2.1 WSDL . . . 12
2.2.3 Service Resource . . . 15
2.2.3.1 WSRF Specification . . . 15
2.2.4 Service Composition . . . 16
2.2.4.1 BPEL . . . 16
2.2.4.2 BPMN . . . 17
2.2.4.3 WS-CDL . . . 18
2.2.5 Evaluation . . . 18
2.3 Formalisation . . . 18
2.3.1 Introduction . . . 18
2.3.2 Formal Methods . . . 19
2.3.2.1 Petri Nets . . . 19
2.3.2.2 Automata . . . 20
2.3.2.3 Process Algebra . . . 20
2.3.2.4 Evaluation . . . 21
2.3.3 Techniques . . . 21
2.3.3.1 Validation . . . 21
2.3.3.2 Verification . . . 22
2.3.3.3 Evaluation . . . 22
2.3.4 LOTOS . . . 23
2.3.4.1 Concepts and Language Overview . . . 23
2.3.4.2 Data Type . . . 24
2.3.4.3 Behavioural Expression . . . 24
2.3.4.4 Example Specification . . . 25
2.3.5 Topo/Lola . . . 27
2.3.6 CADP . . . 28
2.3.7 Evaluation . . . 28
2.4 Implementation . . . 28
2.4.1 Web Services . . . 28
2.4.1.1 AXIS . . . 29
2.4.2 Grid Services . . . 29
2.4.2.1 Globus Toolkit . . . 29
2.4.3 Service Orchestration . . . 30
2.4.3.1 ActiveBPEL . . . 30
2.4.3.2 Oracle BPEL Process Manager . . . 31

2.4.3.3 Apache ODE . . . 31
2.4.4 Evaluation . . . 31
2.5 Implementation Validation and Performance Evaluation . . . 32
2.6 Service Composition Methodologies . . . 33
2.6.1 Formalising Composed Services . . . 33
2.6.2 Implementing Composed Services . . . 38
2.6.3 Comparison With Related Work . . . 40
2.7 CRESS . . . 40
2.7.1 CRESS Framework . . . 43
2.8 Summary . . . 44

3 An Integrated Methodology 46
3.1 Goals . . . 46
3.2 Methodology Overview . . . 47
3.3 Development Lifecycle of Composed Web/Grid Services . . . 49
3.3.1 Design . . . 50
3.3.1.1 Describing Service Diagrams . . . 50
3.3.1.2 Describing Service Configuration . . . 50
3.3.2 Specification . . . 51
3.3.2.1 Specification of Partner Services . . . 52
3.3.3 Formal Analysis . . . 53
3.3.3.1 Formal Validation . . . 53
3.3.3.2 Formal Verification . . . 54
3.3.4 Implementation . . . 55
3.3.5 Implementation Validation . . . 56
3.4 Evaluation . . . 57

4 Describing Composed Web/Grid Services 61
4.1 Introduction . . . 61
4.2 The Original CRESS Notation . . . 62
4.2.1 Rule Box . . . 64
4.2.2 Service Behaviour Description . . . 66
4.2.2.1 Features . . . 68
4.2.3 Service Configuration . . . 69
4.2.4 CRESS Diagram Editors . . . 70
4.2.4.1 CHIVE Graphical Editor . . . 71
4.2.5 Checking Service Description . . . 73
4.3 Extensions To The CRESS Notation . . . 74
4.3.1 Service Diagrams . . . 75
4.3.1.1 CRESS Notation For Dynamic Partners . . . 75

4.3.1.2 Type Ownership . . . 77
4.3.2 Service Configuration . . . 78
4.4 Extensions To CRESS Framework . . . 79
4.5 Evaluation . . . 80

5 Formalising Composed Web/Grid Services 82
5.1 Introduction . . . 82
5.2 Automatic Formalisation . . . 83
5.2.1 Original Translation Strategy . . . 86
5.2.1.1 Data Type Translation . . . 86
5.2.1.2 Partner, Port and Operation Translation . . . 89
5.2.1.3 Behavioural Translation . . . 90
5.2.1.4 Partner Service And Phantom Partner Specification . . . 92
5.2.2 Extended Translation Strategy . . . 93
5.2.2.1 Dynamic Partners and Endpoints . . . 93
5.2.2.2 Type Ownership . . . 94
5.2.3 Automatic Specification . . . 94
5.3 Rigorous Validation using MUSTARD . . . 95
5.3.1 Original MUSTARD Overview . . . 95
5.3.1.1 Original MUSTARD Notation . . . 96
5.3.1.2 Lola Validation Technique . . . 97
5.3.2 Extended MUSTARD . . . 98
5.3.3 Tool Design Overview . . . 99
5.3.3.1 MUSTARD Translation Strategy to LOTOS . . . 99
5.3.3.2 Diagnostic Support . . . 101
5.3.3.3 Tool Integration . . . 101
5.3.4 Examples . . . 102
5.4 Formal Verification using CLOVE . . . 106
5.4.1 CLOVE Notation . . . 107
5.4.1.1 CLOVE Property Verification Syntax . . . 107
5.4.1.2 CLOVE Verification Patterns . . . 108
5.4.1.3 CLOVE Data Type Range Syntax . . . 110
5.4.2 Tool Support . . . 112
5.4.2.1 Tool Design Overview . . . 112
5.4.2.2 Annotation Strategy for LOTOS Specification . . . 113
5.4.2.3 Abstract Base Type Implementation for CADP . . . 114
5.4.2.4 Approach to Constraining Infinite Types . . . 116
5.4.2.5 Automated Implementation of Data Enumeration . . . 117
5.4.2.6 Translation Strategy for CLOVE Properties to μ-calculus . . . 118
5.4.2.7 Compositional Verification . . . 120

5.4.2.8 Diagnostic Support . . . 121
5.4.3 Examples . . . 122
5.4.3.1 CLOVE Data Value Enumeration . . . 122
5.4.3.2 CLOVE Properties . . . 123
5.4.4 Tool Integration . . . 125
5.5 Evaluation . . . 126

6 Implementing Composed Web/Grid Services 128
6.1 Automatic Implementation . . . 129
6.1.1 Original Translation Strategy . . . 130
6.1.1.1 Web Service Deployment Plan . . . 130
6.1.1.2 Service Interfaces (WSDL) . . . 131
6.1.1.3 Common Definitions . . . 133
6.1.1.4 Process Behaviour . . . 136
6.1.1.5 Inclusion of Web Service Partner Implementation . . . 138
6.1.2 Extended Translation Strategy . . . 139
6.1.2.1 Dynamic Partner Definition . . . 139
6.1.2.2 Dynamic Partner Binding . . . 140
6.1.2.3 Type Ownership . . . 141
6.1.2.4 Translation Strategy for WS-BPEL 2.0 . . . 141
6.1.2.5 Inclusion of Grid Partner Service Implementation . . . 142
6.2 Compatibility . . . 143
6.2.1 Interworking of ActiveBPEL and GT4 . . . 144
6.2.2 SOAP-Level Message Harmonisation . . . 144
6.3 Validation using MINT . . . 145
6.3.1 MINT Notation . . . 145
6.3.2 Tool Support . . . 147
6.3.2.1 Tool Design . . . 147
6.3.2.2 Reset Feature . . . 152
6.3.2.3 Diagnostics . . . 154
6.3.2.4 Performance Evaluation . . . 155
6.3.3 Examples . . . 156
6.3.3.1 Acceptance Test . . . 156
6.3.3.2 Refusal Test . . . 157
6.3.4 Tool Integration . . . 157
6.4 Evaluation . . . 158

7 Case Studies 162
7.1 Introduction . . . 162
7.2 Development Of Composed Web Services . . . 162

7.2.1 Service Diagrams . . . 165
7.2.1.1 Lender . . . 165
7.2.1.2 Supplier . . . 167
7.2.1.3 Broker . . . 169
7.2.1.4 Web Service Configuration . . . 170
7.2.2 Partner Specification . . . 171
7.2.3 Partner Implementation . . . 172
7.2.4 MUSTARD Scenarios . . . 173
7.2.4.1 Approver Scenarios . . . 173
7.2.4.2 Assessor Scenarios . . . 174
7.2.4.3 Lender Scenarios . . . 175
7.2.4.4 Dealer1 Scenarios . . . 176
7.2.4.5 Supplier Scenarios . . . 177
7.2.4.6 Broker Scenarios . . . 177
7.2.5 CLOVE Properties . . . 178
7.2.5.1 Lender Properties . . . 178
7.2.5.2 Supplier Properties . . . 179
7.2.5.3 Broker Properties . . . 179
7.2.6 Formal Specification . . . 180
7.2.7 Formal Analysis . . . 182
7.2.7.1 Formal Validation . . . 182
7.2.7.2 Formal Verification . . . 184
7.2.8 Implementation . . . 187
7.2.9 Implementation Validation . . . 188
7.2.9.1 Functionality Validation . . . 188
7.2.9.2 Performance Evaluation . . . 190
7.2.10 Evaluation Compared To Other Approaches . . . 191
7.2.10.1 Supported Composition Activities . . . 191
7.2.10.2 Analysis and Tool Support . . . 192
7.3 Development of Allocator Composed Grid Service . . . 195
7.3.1 Service Diagrams . . . 196
7.3.1.1 Allocator . . . 196
7.3.1.2 Service Configuration . . . 196
7.3.2 Partner Specification . . . 197
7.3.3 Partner Implementation . . . 198
7.3.3.1 Factory Service . . . 198
7.3.3.2 Mapper Service . . . 200
7.3.4 CLOVE Properties . . . 200
7.3.5 MUSTARD Scenarios . . . 201

7.3.6 Formal Specification . . . 202
7.3.7 Formal Analysis . . . 203
7.3.7.1 Formal Verification . . . 203
7.3.7.2 Formal Validation . . . 204
7.3.8 Implementation . . . 204
7.3.9 Implementation Validation . . . 205
7.3.9.1 Allocator Implementation Validation . . . 205
7.3.9.2 Performance Evaluation . . . 205
7.3.10 Evaluation Compared To Other Approaches . . . 206
7.3.10.1 Supported Composition Activities . . . 207
7.3.10.2 Analysis and Tool Support . . . 207
7.4 Evaluation . . . 209

8 Conclusions **210**
8.1 Thesis Summary . . . 210
8.2 Evaluation . . . 211
8.2.1 Strengths and Weaknesses . . . 211
8.2.2 Future Work . . . 212
8.2.3 Concluding Remarks . . . 213

A CRESS to LOTOS Translation **214**

B CRESS to WS-BPEL 2.0 Translation **217**

Bibliography **220**

List of Figures

2.1 Service Discovery and Consuming Services 15
2.2 Desired Approach For Creating Composed Web/Grid Services 41
2.3 CRESS Goals Compatible With Proposed Approach 41
2.4 CRESS Framework [104] 44
2.5 CRESS Modules Dependency [104] 45

3.1 Integrated Methodology Development Lifecycle For Composed Web/Grid Services 47
3.2 Partner Specification Using Generated Interface 53
3.3 Include Partner Specification Directly 53

4.1 CRESS Feature Diagram Example 65
4.2 CRESS Web Service Domain Configuration Diagram 70
4.3 CRESS Diagram Translation Approach 71
4.4 CHIVE Window 72
4.5 CHIVE Preferences Domain Configuration (web service) 73
4.6 Dynamic Partner Example 76
4.7 Type Ownership Example 77

5.1 CRESS Translation Strategy 86
5.2 MUSTARD Architecture 99
5.3 Levels of Automated Validation Procedure 102
5.4 CLOVE's Generic Framework 112
5.5 Levels of Automated Verification Procedure 125

6.1 Web Service Deployment Plan 131
6.2 Include Partner Service Implementation 139
6.3 Grid Service Partner Deployment Plan 143
6.4 MINT Approach to Automated Implementation Validation 149
6.5 Integration of Implementation Validation Tools and Execution Approaches . . . 158

7.1 Composite Web Service Development 165
7.2 Lender CRESS Diagram 166

7.3 Supplier CRESS Diagram 168
7.4 Broker CRESS Diagram 170
7.5 Web Service Configuration Diagram 171
7.6 Automated Implementation Process for Lender, Approver and Assessor 188
7.7 Allocator CRESS Diagram 197
7.8 Grid Service Configuration 197
7.9 Implementation of Factory and Mapper 205

List of Tables

2.1 A Subset of LOTOS Behavioural Expressions . . . 25
2.2 Subset of CRESS Supported Domains and Target Languages . . . 43
2.3 CRESS Tools Relevant to Web/Grid Services [104] . . . 44

4.1 CRESS Supported Data Types For Web Services . . . 66
4.2 CRESS Node Activity Syntax . . . 67
4.3 CRESS Deployment Options . . . 70
4.4 CRESS Service Configuration . . . 79

5.1 CRESS Data Types . . . 87
5.2 Original MUSTARD Notation . . . 96
5.3 Extensions to MUSTARD Notation . . . 99
5.4 MUSTARD Translation Strategy for LOTOS . . . 100
5.5 CLOVE Verification Constructs . . . 108
5.6 CLOVE Supported Patterns . . . 110
5.7 CLOVE Data Enumeration Syntax . . . 111
5.8 CADP Pragmas Description . . . 115
5.9 CLOVE Translation to μ-calculus . . . 118
5.10 CLOVE's Adaptation of RAFMC Response Pattern '*S responds to P*' . . . 118
5.11 CLOVE's Adaption of RAFMC Universality Pattern '*P is true*' . . . 119
5.12 CLOVE's Adaption of RAFMC Absence Pattern '*P is false*' . . . 119
5.13 CLOVE's Adaption of RAFMC Precedence Pattern '*S precedes P*' . . . 119
5.14 CLOVE's Adaption of RAFMC Existence Pattern '*P occurs*' . . . 120

6.1 Representation of MUSTARD Combinators in MINT . . . 148
6.2 MINT Service Configuration File Properties . . . 150
6.3 MINT Pseudo-Algorithm . . . 152

7.1 Specification of Web Services (no. of lines) . . . 182
7.2 Generated LOTOS Tests (no. of lines) . . . 183
7.3 Generated Verification Code (no. of lines) for Lender, Supplier and Broker . . . 186
7.4 Code Generation Summary of Implemented Web Services . . . 188

7.5 Code Generation Summary for Implementation Validation (no. of lines) 189
7.6 Support of BPEL Constructs 192
7.7 Automated Tool Support of Web Service Composition Approaches 192
7.8 Specification Summary of Allocator, Factory, Mapper, and Resource 203
7.9 Support of Composition Constructs 207
7.10 Automated Tool Support of Grid Service Composition Approaches 208

A.1 CRESS Assignment and Guards in LOTOS 214
A.2 CRESS Activities in LOTOS 215
A.3 CRESS Event and Compensation in LOTOS 216

B.1 CRESS Assignment and Guards in WS-BPEL 2.0 217
B.2 CRESS Activities in WS-BPEL 2.0 218
B.3 CRESS Event and Compensation in WS-BPEL 2.0 219

List of Abbreviations

AXIS Apache eXtensible Interaction System

CHIVE CRESS Home-grown Interactive Visual Editor

CLOVE CRESS Language-Oriented Verification Environment

CRESS Communication Representation Employing Systematic Specification

LOTOS Language of Temporal Ordering Specification

MINT MUSTARD Interpreter

MUSTARD Multiple-Use Scenario Testing and Refusal Description

BPEL Business Process Execution Language

CADP Construction and Analysis of Distributed Processes

FDT Formal Description Technique

GS Grid Services

JNDI Java Naming and Directory Interface

OGSA Open Grid Service Architecture

OGSI Open Grid Service Infrastructure

RAFMC Regular Alternation-Free μ-calculus

SOA Service-Oriented Architecture

SOAP Simple Object Access Protocol

WS Web Services

WSDD Web Service Deployment Descriptor

WSDL Web Services Description Language

WSRF Web Services Resource Framework

Chapter 1

Introduction

The claim of this thesis is that an integrated approach with highly automated formalisation, implementation and analysis can be applied to improve development of composite web/grid services. This thesis aims to show that, through an integrated methodology, the development of composite services can benefit from systematic, rigorous and highly automated development techniques and tools.

1.1 Motivation

Service-oriented computing allows web and grid services to be composed to create new services. There has been significant development in the technologies that enable creation of web/grid services such as the BPEL (Business Process Execution Language) standard [4, 6] that is widely supported [3, 77, 112]. BPEL is a language to describe the orchestration or flow of web services, but can also be used for grid services [92]. Formal methods enable specification, rigorous analysis and reasoning about systems. These methods are useful to detect errors in abstract models. Although the advantages of formal methods are being acknowledged, BPEL however has no formal semantics in its specification [25, 75]. As a form of analysis, service implementations can be tested with tools such as those developed for web services. The thesis aims to provide an integrated methodology to develop web/grid service compositions, with automated support for specification, implementation and analysis.

1.2 Context

1.2.1 Services And Their Composition

The service-oriented computing paradigm conceptualises functionality as services. Services are autonomous by nature, and can be combined to build new services – an activity generally known as composition. Web/grid services follow this paradigm, and allow service creation via composition. Languages have been defined to describe service compositions; their implementations perform the actual enactment of composite services. The BPEL standard is one of the most widely adopted approaches to orchestrate services [101]. Service composition is in attractive business proposition, but results in service behaviour becoming increasingly complex.

1.2.2 Design-Time Analysis of Service Compositions

As services become more complex, for example by means of composition, it is important to ensure their correct behaviour. Analysing services at design time has the advantages of detecting and correcting errors prior to development, thereby reducing the cost of correcting errors discovered during later development [101]. Formal methods can be used to rigorously analyse the abstraction of service behaviour, validated and verify its properties [25].

1.2.3 Implementation and Testing of Service Compositions

There has been significant development in the technologies that enable creation of composite web/grid services, and also the activity of composing services in general. The BPEL specification is the result of standardisation of a language for composing web services. There is no formal semantics in BPEL, and implementations are usually analysed by testing [75].

1.3 Objectives

The goal of this work was to develop an integrated methodology for rigorous development of composite web/grid services, with highly automated support for specification, validation, verification, implementation, and evaluation, and with abstraction of the underlying code implementations and analysis techniques.

Building on the original CRESS toolset, the thesis work has aimed to achieve the goal in the following ways:

- provide an abstract but accessible notation for describing web and grid service compositions
- automate the formal specification of web/grid service compositions
- allow user to specify scenarios in a high-level way, and automate their validation
- allow user to specify properties in a high-level way, and automate their verification
- automate implementation and deployment of web/grid service compositions through support for the WS-BPEL 2.0 standard for composed services, through automated deployment and execution using ActiveBPEL, and through support for partner web and grid services through automated deployment and execution using the AXIS tool and Globus Toolkit
- automate testing and performance evaluation of composite services and service partners, reusing the same validation scenarios

1.3.1 Scope and Assumptions

The integrated methodology supports the description, specification, analysis and implementation of the service behaviour. Factors outside service functionality (e.g. quality of service) are not considered and are therefore not defined. Resource constraints, quality of service such as networking issues and failures, and timing aspects (e.g. real-time constraints) are not supported in this work. The formalisation supports the specification, validation and verification of functional behaviour only, and does not support performance aspects of the formal model. Deadlock freedom, for example, means that the service behaviour is free from deadlock with respect only to the service functionality, not considering factors such as resource limitations. It is assumed that there are no resource constraints with regards to executing the analysis, such as memory to hold state space. The efficiency of the analysis execution such as speed and time are not considered as part of the thesis goal, but the focus is on the types of analysis that are

automated with abstraction of underlying tools. Implementation validation can detect defects with regard to externally observed behaviour expected of the target service. A specific timing aspect is supported in implementation validation where it is possible to specify and check service response timeouts. Performance evaluation supports the analysis of target services from an external perspective, executing sequential or concurrent runs of the tests specified for implementation validation. Performance issues internal to the target service cannot be specifically evaluated.

1.4 Achievements

The thesis work has developed an integrated methodology where developers have a rigorous highly automated and high-level approach to design, specify, analyse, implement, and test web/grid service compositions. The CRESS notation has been extended with capabilities to describe grid service composition, and with features for realising practical service compositions. The scope of formal validation has been extended to service partners for more thorough analysis. Formal verification of composite service behaviour has automated support, abstracting the underlying techniques whilst exploiting their advantages. Well-known verification properties can be readily specified through pattern templates. Implementations of web/grid service compositions are automatically generated for deployment, supporting the latest WS-BPEL 2.0 standard. The same validation tests are reused for implementation testing, with added support for performance analysis. The integration of this work into the CRESS environment has resulted in a rigorous development methodology for composed web/grid services. The value of the approach has been demonstrated through case studies.

1.4.1 Supported Analysis

Validation can be performed on composed and partner services, where user specifies the scenarios with success or refuse assertions, which are are automatically validated. Trace diagnostics are provided when a scenario does not pass validation.

The label transition system is automatically generated from the specification for verification. Deadlock and livelock freedom are two properties is fully automated. Templates are

provided for user to specify well-known (response, safety, and liveness) properties, which will then be verified. Counterexample diagnostics are provided when compromising properties are detected in behaviour of composed service.

Scenarios for specification are used to validate (black-box approach) service implementations. Trace diagnostics are provided when a scenario does not pass validation. Service timeout can be imposed for invocation. Repeated and simultaneous execution of multiple instances of scenario is supported. System level errors are observed and reported such as unable to establish connection.

1.5 Thesis Structure

Chapter 2 provides the background and evaluation of standards, technologies and approaches to the formalisation and implementation of web/grid service compositions. Chapter 3 presents the overall application of the methodology, with prescribed steps from design through to post-implementation. Chapters 4 to 6 present the various aspects of the methodology following the flow of service development: design, specification and analysis, implementation and testing. Chapter 4 presents in detail the CRESS high-level notation and framework which is the foundation of the methodology. It also explains the extensions developed by the thesis work, which in turn support other aspects of the methodology. The formalisation aspect of the methodology is covered in Chapter 5, outlining the original automated support for service specification and rigorous validation that has been extended through new work. This is followed by the extensions and new tool developments for the thesis, which widen the scope of validation and offer entirely new support for automated verification. Chapter 6 discusses the methodology's approach to automate implementation, along with the thesis extensions to support new implementation standards, and a newly developed tool that analyses implementations. Chapter 7 demonstrates the methodology in practice on a series of four case studies, illustrating how each aspect of the methodology benefits and supports rigorous development. Chapter 8 concludes with an evaluation and a summary of the thesis.

Chapter 2

Background

This chapter evaluates the state-of-the-art spanning the various aspects involved in the development of composed web/grid services. The concepts of service-oriented computing, specifically web and grid services, are first presented followed by a discussion of service description standards, formal methods, and service implementation technologies. This is then followed by an evaluation of service composition techniques and methodologies with regard to formalising and/or implementing composed services. Finally, the chapter covers the basis of the thesis objectives and work for realising a rigorous development methodology for creating composed web/grid services.

2.1 Services

2.1.1 Service-Oriented Architecture Concept

Service-Oriented Architecture, also known as SOA, is briefly a software integration concept with a paradigm of loosely-coupled heterogeneous components where there is autonomous and interoperable functionality known as services by which applications are developed.

SOA defines interfaces in terms of protocols and functionality (operation). In contrast to traditional application programming interfaces (APIs), SOA-defined interfaces are language and platform independent. This is a key characteristic of SOA which enables choice of implementation technologies and therefore features a loosely-coupled environment. An endpoint is the entry point to an implementation of an SOA interface, where the act of service consumption

(utilising functionality) may take place concretely. SOA achieves a heterogeneous form of distributed computing with interoperability among services of diverse implementations, even for the same interface definition. This way, features and advantages of the chosen implementation technologies can be exploited.

The SOA paradigm for building applications is 'service orchestration' whereby services are combined in a logical manner as units of the application. Orchestrating services implies communication with these services and the use of their defined interfaces (functionality) and protocols. In a loosely-coupled environment, the endpoints to service functionality are dynamic at run-time and at the level of communication protocol. Therefore the actual behaviour of an application which is defined by service orchestration may differ when it dynamically binds to different service functionality. The developed application can itself define a service interface.

Service discovery is a principle that underpins the SOA loosely-coupled characteristic, allowing functionality to be bound. Service discovery provides the search mechanisms to find suitable functionality for use in service orchestration. Service discovery may apply at different stages of service development. Discovery used in a conventional manner is the search for suitable functionality. The interfaces are then used in describing the orchestrated flow of services. Yet another level of use is akin to a brokerage framework whereby the service orchestration itself uses discovery to dynamically select functionality based on given criteria to achieve better results according to specified requirements.

SOA has achieved significant attention and uptake by industries and organisations [64], largely due to its capability for heterogeneous, dynamic interoperability. Several technologies have been developed that are based on the SOA concept. Web and grid services are probably the most popular of these.

2.1.2 Web Service Concepts

Web services use a distributed computing technology that is based on XML message exchange among services and their consumers. Web services are a specific form of SOA, defined using open standards and standard Internet protocols. The open standards are based on XML to support information portability and platform independence.

Service interfaces are defined using the XML language WSDL (Web Services Description Language [118]). WSDL describes functionality through ports, messages, data types, protocols, and bindings. Services that provide the actual functionality based on a WSDL description therefore *implement* the WSDL service interface, implying that the services understand the protocols defined by the WSDL. Clients use the information provided in the WSDL service interface to engage with endpoints that provide the actual functionality. Clients communicate with the services using the protocols described in the WSDL.

Information exchange between services and clients, i.e. service requests and responses, is achieved using the SOAP (Simple Object Access Protocol [119]) XML-based standard, and is possible with other protocols if described in the WSDL. The SOAP specification defines syntax for messages, specifies the encoding and serialization rules for data exchange, and gives conventions for representing RPCs (Remote Procedure Calls). Service invocations are represented as XML documents (SOAP messages), sent to the destination service to be processed and returned as values. Essentially, the invocation details (e.g. operation parameters, operation name) are embedded in the SOAP message to be sent to the target. Likewise in serialisation/deserialisation of data, a typical library often contain components to handle SOAP-related activities such as the construction of equivalent SOAP message service invocations. This maps the information to actual implementation code. For example, the AXIS SOAP engine [5] and libraries support hosting and development of web services using the Java and C/C++ language. The transportation of SOAP messages are automatically handled, and activities such as the serialisation/deserialisation of data have automatically generated stubs code from AXIS in the implementation language.

Although being transport protocol-agnostic, it is common that web services use the standard Internet protocols such as HTTP (HyperText Transfer Protocol) and the Internet protocol suite TCP/IP (named after Transmission Control Protocol / Internet Protocol) to transport the SOAP messages to their destinations. This is typically done with the advantage that these transport protocols are globally used. They usually do not have network configuration issues like a firewall blocking specific ports as these well-known ports are usually open.

As interaction is oriented towards the level of SOAP message exchange, the complexity of services and clients is not restricted and can be heterogeneous as long as the chosen imple-

mentation technologies can support SOAP to exchange information. Therefore web services are independent from implementation platforms and languages. This allows a free choice of preferred technology for service and client implementation, supporting heterogeneous interoperability. SOAP message exchange supports a concept of loosely-coupled environment, where service functionality and hence orchestration, are bound when required through the exchange of SOAP messages between clients and endpoints.

In a loosely-coupled environment there may be the need to search for web services that have suitable functionality. Service discovery is a means to achieve this. As XML documents, WSDL service interfaces are portable and may be published for discovery in many ways such as in web URLs or file storage. UDDI (Universal Description Discovery and Integration [74]) is a registry and service discovery mechanism that is used for service registration and querying service catalogues. UDDI is often used to publish web service listings comprising their descriptions and interfaces, by which potential clients may access and then engage the services.

2.1.3 Grid Service Concepts

Grid services are one of the distributed computing technologies, based on the web service architecture. This means that grid services have the (SOA) characteristics of web services. The grid computing paradigm is analogous to that of the electrical grid. Just as access to electrical power can be achieved with standard interfaces (e.g. sockets) regardless of source (e.g. power stations), so computing resources should be made accessible via uniform interfaces with abstraction from their implementation and other complexities [29]. In addition to inheriting the features of web services, grid service computing defines the concept of virtualisation where the goal is to define uniform access via standard protocols to specific groups of resources. Grid services or computing have been widely used across many domains such as physics, astronomy, medicine, etc. It is also used for inter-disciplinary research in social sciences [59].

OGSA (Open Grid Service Architecture [30]) governs the concept and defines capabilities of grid computing. OGSA has specified requirements that underpin a grid computing environment such as virtualisation of resources, management services, resource discovery, stateful services, and virtual organisations. Virtualisation of resources, such as CPU and data, means their

access is via standard interfaces and abstracts away from implementation complexity. This enables interoperability in a heterogenous environment, thereby supporting resource sharing across organisations. Resources have management services such as supporting the monitoring and control of resources, for example reservation of CPUs for a job execution. Resource discovery is required to support a dynamic grid environment, comprising dynamic resource availability and resource capability that match client requirements. OGSA specifies that services be stateful in that state is kept across invocations, for example in long-running job execution where it may be necessary to make changes during its lifecycle. Virtual organisations, underpinned by other capabilities of the Grid, enable entities from different organisations to participate in specific collaborations. This uses federated authentication and authorisation to resources, governed by policies set by resource owners. Shibboleth [40] is an infrastructural middleware to enable trust and security mechanisms.

OGSI (Open Grid Service Infrastructure [97]) was developed by the OGF (Open Grid Forum) to provide the infrastructure layer for OGSA. OGSI is a collection of specifications comprising a specific extension of WSDL service interfaces known as GWSDL (Grid WSDL), in anticipation for WSDL 2.0 standard [124], to define a grid service and also underpin the objectives set by OGSA. In particular, these specifications consist of the service interfaces for virtualised access to resources such as CPU, data access and transfer, job execution submission, data aggregation, resource discovery etc. It also provides mechanisms to define service data (for stateful services) and management functionality (e.g. lifecycle management), port type extensions which grid resources will can build upon and utilise. Globus Toolkit 3 is a well-known grid service development toolkit that implements the OGSI specifications. OGSI has a few major drawbacks, particularly the incompatibility between GWSDL and the WSDL standard, but also the implementation between grid and web services. Both drawbacks imply there is no direct interoperability. The OGSI approach to stateful services is to instantiate a service instance with service data, resulting in duplicates of service functionality only differing in state information. The use of the underlying protocols was adapted, and hence is not directly compatible with the standards. For example, orchestration OGSI services using BPEL would require extensions [86].

WSRF (Web Service Resource Framework [39]) standard was developed by the OASIS

(Organization for the Advancement of Structured Information Standards), replacing OGSI as a more harmonised specification for both web service and grid services. WSRF is also a collection of standards comprising service interfaces that implement OGSA; however, instead of using GWSDL, the service interface definitions are defined using the WSDL standard. WSRF decouples the *stateful* service aspect, separating service data from service implementation and thereby having only instances of state instead of services with state. As a result, WSRF grid services are compatible with web services, implying compatibility with technologies that are web service-compatible such as web service orchestration, which then can be readily exploited. GT4 (Globus Toolkit 4) is also one of the well-known grid service development toolkits that implement the WSRF specifications.

2.2 Service Description

2.2.1 Data Definition

Data is central in information exchange, and therefore implies the need for data definition whereby the structures, types, and values can be understood. There are several data definition languages available. The DTD (Document Type Definition [115]) and the XML Schema [122] are languages for defining XML data structures. The XML Schema is used to define application data and even many XML standards, including those for web and grid services.

2.2.1.1 XML Schema

XML Schema, also known as XSD (XML Schema Document [122]), is a W3C (World Wide Wide Consortium) standard XML schema language used to describe data schemas, and that has been used to create new languages. XSD underpins the structural and validity aspects of information that is used in the representation of XML data. XSD has a set of primitive data type definitions and constructs to create new data structures. The complex data structures that are typical may be nested, contain value restrictions, be constrained with occurrences (e.g. size), have mandatory requirements, and use specified namespaces. Collectively these structural definitions and rule descriptions constitute the XSD schema that is used to validate the conformance of XML documents. There are validators developed for the specific purpose

of validating XML documents, which is often an implicit and embedded part of the process of using the data itself.

The following is an example of an XSD that described the structure of a complex data type:

```
<complexType name="proposal" >                          0
  <sequence>                                            1
    <element name="name" type="xsd:string"/>            2
    <element name="address" type="xsd:string"/>         3
    <element name="amount" type="xsd:integer"/>         4
  </sequence>                                           5
</complexType>                                          6
```

XSD schemas are widely used, and some of the schemas that were created have become standards or new technologies. For example XSLT (Extensible Stylesheet Language Transformations [117]) is an XML language for transforming XML documents into desired formats. XSLT language and syntax constructs are defined using XSD, and the XML data being transformed is usually validated by XSD also. XSD is used to define SOAP (Simple Object Access Protocol [119]) and WSDL (Web Service Description Language [118]), which are well-known standards used to define data communication protocol and service interfaces for web services. XSD is also directly involved in the development of application web services. Their data structures are defined and also operationally checked in the validation of message exchange. Data structures defined in XSD can be fully translated into implementation code which contains access functionality and (un)marshalling information for on-the-wire transfer to carry out message exchange in XML. XSD is also used in a similar fashion in grid computing, specifically the WSRF (Web Service Resource Framework) specifications which collectively underpin grid service implementation.

2.2.2 Service Interface

2.2.2.1 WSDL

Adequate information such as addresses, protocols, operation names, etc. must be available in order to have the preliminary means to invoke a service (also known as consuming it). This information constitutes the service interface which is published externally for potential clients to use as directives that is the construction of valid messages to exchange. WSDL is a W3C standard language [118] for defining interface to web services. WSDL in turn uses other XML

standards, particularly SOAP, as the message exchange protocol, and WS-Addressing [121] to specify service endpoints. WSDL was also used in the WSRF specification, and in grid service interfaces.

WSDL is fundamentally used for describing the syntactic bindings of the implementing service. There are variations supporting semantic information such as ontological annotations to describe properties and capabilities of web services in unambiguous, computer-interpretable form. An example is the Semantic Web Services, which uses the OWL-S (Ontology Web Language – Semantics [120, 19]) to describe concepts and properties. A WSDL document typically contains descriptions of service ports, service operation signatures, messages, data types for parameters, communication grammar (e.g. particular SOAP style), service namespaces, and addresses of implementing endpoints. A WSDL document may also import other WSDL documents. The following XML shows the high-level schema of WSDL:

<**definitions**>
  <**types**>
    definition of types in XSD language
  </**types**>
  <**message**>
    defines messages, with logical groups of data, used by operations
  </**message**>
  <**portType**>
    defines a port containing operations
  </**portType**>
  <**binding**>
    defines protocols (e.g. HTTP, SOAP style) for a service
  </**binding**>
  <**service**>
    defines the service location(s)
  </**service**>
</**definitions**>

A WSDL document is defined from the root element <definitions> where, apart from the fundamental declaration of XML namespace prefixes, the namespace of the service (targetNamespace attribute) is declared and used as the qualified name in conjunction with operations and messages. Imports of other WSDL files are then specified if necessary. After this is the definition of data types using the XSD language, enclosed within the WSDL <types> element. These data types have qualified names comprising the defining schema's namespace and the name of the data type. XSD schemas defined separately may also be imported at this point.

The <message> element defines a logical group of parts and their types which will be used in the definition of operation signatures that follow. The <portType> element specifies an endpoint to a range of operations defined with <operation>. Operations may have inputs, outputs, and faults which make reference to the message elements for their content type. The <binding> element defines for the associated <portType> its transport protocol and SOAP style for operations. Finally the <service> instantiates port bindings with given names, and designates endpoint addresses for ports.

Figure 2.1 briefly illustrates a typical scenario for consuming a service. This may involve discovery, obtaining a service interface, code generation, and then invocation. Practically, the discovery of a service and its interfaces are prior to consuming the service for the first time. A discovery process for the service can use any means of publication, for example UDDI registries purposed for discovering web services, or evaluation of functional compatibility and service. This is followed by obtaining the service interface (WSDL) of the service that is to be used, usually by a file download process. The WSDL files contain the service communication directives, specifically how SOAP messages are constructed to invoke services. Potential service clients are usually implemented as applications, which act on behalf of the actual user. With regard to clients that are explicitly programmed, it is possible to manually construct the SOAP messages either by writing the XML code or even writing a program that does it; however, it is often the case that there are tools to translate WSDL directives into implementation code to handle these needs, thereby allowing developers to focus on developing the actual functionality of applications. These tools usually support code translation/generation from WSDL into code stubs in supported target implementation languages. Applications can make service invocations using these code stubs.

Most of the service orchestration technologies that are used to implement composition of services do not require the code stub translation process. Instead, they have a framework to import the WSDL for direct use into the enacting environment. This will create the SOAP messages directly according to the WSDL descriptions.

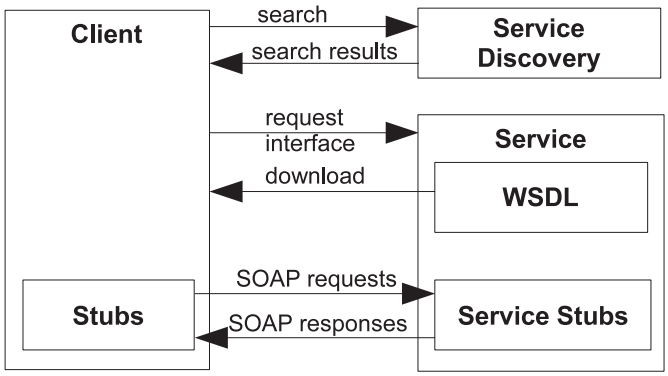

Figure 2.1: Service Discovery and Consuming Services

2.2.3 Service Resource

2.2.3.1 WSRF Specification

The WSRF specification is a framework and collection of service interfaces that implement OGSA for the development of grid services. With its development, the difference between grid and web service implementation has become less distinct. Services developed using WSRF development kits are sometimes known as WSRF services, unlike the predecessor OGSI services which are known as grid services.

WSRF introduces WS-Resource [39] to represent the composition of a resource (state) and a web service. Service operations are associated with resources identified by WS-Resource endpoints. Endpoints are referenced using WS-Addressing, which generally contain the service address and the resource key which identifies a unique resource.

WSRF comprises the following component specifications: WS-ResourceProperties, WS-ResourceLifeTime, WS-BaseFaults, and WS-ServiceGroup. Each of these specifications is a WSDL service interface with defined XSD types. WS-ResourceProperties specifies the functionality by which a set of typed values configured as properties of the WS-Resource may be accessed. WS-ResourceLifeTime specifies the functionality to manage the lifetime of a WS-Resource, such as setting termination time or destroying the resource. WS-BaseFaults serves as an extensible mechanism to describe rich service (SOAP) faults. WS-ServiceGroup defines the means by which web services and WS-Resources can be aggregated for specific a domain (virtual organisation), supporting the discovery and querying of registered service groups. WSRF uses the WS-Notification standard within its specifications, which is a service interface speci-

fication enabling notification that pushes information to subscribers. For example, notification can be triggered from subscribed WS-Resources when properties (WS-ResourceProperties) are updated, destroyed or have lifetime renewed (WS-ResourceLifeTime). Notification is also used for new member registration in a virtual organisation (WS-ServiceGroup). GT4 is one of the popular toolkits available for supporting WSRF services.

2.2.4 Service Composition

One of the notable characteristics of the service-oriented architecture paradigm is that existing services can be combined together in a logical fashion, resulting in a new service which is added to the pool of existing services as a peer service. Orchestration and choreography are terms affiliated with such composition. Although these terms are generally used interchangeably, they do have distinct orientation. Orchestration is a centralised approach where there is execution control of activities. Choreography defines the collective message exchange among interacting peers – there is no centralised coordination. As with the rise of service-oriented technology, in particular for web services and combining them, standards have been developed for describing composite behaviour.

2.2.4.1 BPEL

BPEL (Business Process Execution Language) is one of the most popular standards for orchestrating services. The standard is a multi-organisation effort that was developed to describe composition of web services based on the WSDL and SOAP standards. BPEL is the successor to IBM's WSFL [61] and Microsoft's XLANG [93], which are web service orchestration technologies. It inherits the features of directed graphs from WSFL and blocked structure from XLANG, as well as other characteristics and features of both predecessors. BPEL can be used for modelling web service *composition* and web service *choreography*, which it refers to as executable process and abstract process respectively.

BPEL models tasks as invocations of web services, where input and output are specified by messages, and whose addresses are identified by URIs (Uniform Resource Identifiers) of WSDL port types. This implies that the WSDL definitions of relevant Web Services are referenced in BPEL specifications. SOAP is used as the communication protocol for message

exchange in service invocations. BPEL provides a range of control constructs to model the flow of activities in a business process which is the orchestrating service, including receiving requests, invoking partner service operations, concurrency, replies, conditional switches, etc. Data handling is expressed via variables that are often instantiated based on message types defined in WSDL. Manipulation of data values is normally performed via XPath [116] expressions which are a form of query syntax for XML data. Process instances can be related to message sequences using correlation sets. BPEL also supports handling of events and faults as well as compensation of transactions. This is the executable process model, akin to the term orchestration. An executable process specification can be deployed and be executed.

The BPEL standard can also be used to describe an abstract process. An abstract process is a business protocol, specifying the black-box message exchange behaviour between different parties without revealing the internal behaviour of any party, akin to choreography. The abstract process is not executable as it only specifies the contracts of the interacting parties.

BPEL4WS (BPEL for Web Service [4]) was the initial standard that was produced from this combined effort. OASIS took over the updates to BPEL, renaming it as WS-BPEL. Generally, BPEL and WS-BPEL refers to this standard. The standard was further developed by the OASIS as the WS-BPEL 2.0 standard [6].

2.2.4.2 BPMN

BPMN (Business Process Modeling Notation [14]) is an OMG (Open Management Group) standardised graphical notation intended for human readability in the description of business processes. This uses a flow-chart format which appeals to business people, with the objective of understanding the business model. Although it is a notation to describe business processes in general, BPMN provides a mapping for its notation to BPEL4WS in recognition of its popular uptake. However, the notation covers other attributes not described in BPEL4WS. There has been uptake of BPMN by BPEL implementation vendors who provide it as the graphical notation in their application design tools.

2.2.4.3 WS-CDL

WS-CDL (Web Services Choreography Description Language [123]) is an XML standard that describes collaborations of participants by defining from a global viewpoint their common and complementary observable behaviour. It also describes where information exchanges occur and when the jointly agreed ordering rules need to be satisfied. In short, it defines the contract by which a collaboration (business process) is achieved. A WS-CDL document therefore specifies 'what must happen', not 'how to make it happen': it is not executable. As its nature is choreography, the WS-CDL specification is independent from any business process implementation language.

2.2.5 Evaluation

There is a comprehensive set of (XML) standard specifications that underpins web and grid services. Amongst the service orchestration languages, the BPEL (now referred to as WS-BPEL 2.0) is most suitable as my methodology's target language for implementing composed web/grid services. This is needed to achieve the thesis objective of automated implementation. BPEL was chosen because it is an open standard, compatible with the web and grid (WSRF) service standards, and has significant adoption in terms of BPEL engine providers and users.

2.3 Formalisation

2.3.1 Introduction

Formal methods are mathematically-based techniques used to model or specify systems at abstract level and then rigorously analyse them to detect potential errors as early as possible. There is a prejudice that formal methods requires highly-trained mathematicians, useful only for safety critical systems, used only by formal methods people, cannot be applied to large scale systems, unacceptable to users, unnecessary and increases the cost of development, but is not [43, 12]. Formal methods have been used in industry [1] and its applicability certainly can be improved [111]. Formal methods have been used in the design of hardware such as disk[34], where it is critical product for manufacturers and businesses to reduce operational

risks and costliness of bug fixes – especially in the later stages of design and development. Formal methods have also been applied to software to analyse its functionality and to establish confidence in its quality, sometime even with techniques to evaluate performance. Formal methods are applied in the design of (distributed) computer systems, including web services [85, 31, 44] where with regards to BPEL it is acknowledge that the specification will benefit from formalism [75].

2.3.2 Formal Methods

There are different types of formal specification languages, notations, techniques and tools. Formal techniques define abstract models of system behaviour and analyse them, which produces feedback on the quality and the level of assurance of the system. Simulation, validation, automated theorem proving, and model-checking are well-known techniques for analysis. Simulation is usually an interactive analysis where the formal behaviour of the system is observed. Validation in formal methods refers to testing of the specification which can be easily related to testing in software engineering [11]. Formal validation can be performed with controlled simulation of the specification. Automated theorem proving is a technique that produces a formal proof of some theorem (e.g. system property), given a description of the system, a set of axioms, and a set of inference rules. Model checking verifies if a model with states satisfies a specified logical formula (e.g. temporal logic) thereby asserting a property.

2.3.2.1 Petri Nets

Petri nets [83] are a modelling approach to describing concurrent and distributed systems, supporting asynchrony, non-determinism and also true concurrency semantics. They have a graphical notation for directed bipartite graphs comprising places, transitions and directed arcs. They have a mathematical semantics for the execution and also analysis of models. Execution of a model involves tokens being fired upon fulfilled conditions, transiting between places and following transitions along directed arcs. The standard petri net notation does not support data type modeling. Petri-nets are a potential candidate for modelling composed web/grid services as they are suitable for describing concurrent and distributed systems.

2.3.2.2 Automata

Automata are abstract machines with (finite) states and transitions depicting the flow between states given the fulfilment of conditions (transitions) triggered by inputs. State machines can therefore model the behaviour of systems. UML (Unified Modeling Language) Statecharts [41] are one such example. State machines can be created from other formal languages, e.g. LOTOS, to labelled transition systems, and used in analysis such as model checking. Abstract state machines, a more generalised form of state machine, support the notion of non-determinism and data structures, for example in labelled transition systems. State machines can be composed to model composite behaviour of components, which suits the nature of service composition.

2.3.2.3 Process Algebra

Process algebras and calculi are languages used to model concurrent systems with the concept of high-level description of communicating processes or agents. Agents can be composed together with special operators which describe a complex behaviour; the possible interactions are defined by these composition operators. The synchronisation of agent interaction points communicates information between the agents, e.g. by value passing. This semantics is very applicable to the notion of composing web and grid services. CCS (Calculus of Communicating Systems [46]), CSP (Communicating Sequential Processes [69]), and LOTOS (Language Of Temporal Ordering Specification [50]) are examples of process algebras which are similar but do have distinctive differences [26]. LOTOS developed into an international standard FDT (Formal Description Technique) which has been used to specify distributed systems and protocols. Full LOTOS includes the rigorous specification of data types which CCS and CSP do not support.

π-calculus [70] is a specific process algebra that supports the concept of mobile processes where the control rights of a process can be transferred. However this concept is not used from the view of the practical realisation of web/grid services – BPEL does not move processes, although there is ability to pass endpoint references that bind to different partners. But in doing so, there is no concept of control transfer – just dynamic binding.

2.3.2.4 Evaluation

Considering Petri nets, they offer a more primitive notation than process algebras [26]. For instance, process algebras have the concept of using channels for communication, which enables reasoning about processes. This makes process algebras, which have compositional semantics for agents (distributed processes constrained in compositions), more suitable for modelling web and grid service compositions which have the notion of service endpoints.

Process algebra notation can express composition of behaviour in a concise and compact manner in comparison to using state machines, although in many approaches the analysis of specifications in process algebras actually uses state machines; however this is usually automatic. In addition, the semantics of process algebras is easier to interpret at high level, where the behaviour of communicating agents and their interaction points are depicted more clearly. LOTOS, specifically Full LOTOS, was found to be the most suitable approach for modelling web and grid services, thanks to: compositional semantics with support for modeling data types; its status as an international standard; and comprehensive tools to perform analysis. There are work done in formalising web service compositions using LOTOS [25, 17, 101].

2.3.3 Techniques

2.3.3.1 Validation

Validation, in the context of formal methods, is the testing of a system specification. Validation is performed by controlled simulation of the formal model and checking if it behaves in accordance with the test cases. Validation can be quickly executed, checking if the behaviour realises its functionality as required by the user, and also analysing feature interactions [113]. This is because simulation of the model is constrained by the tests. Validation therefore produces results and diagnostics in a short time. It can be applied to specifications that may have infinite state space, such as having infinite range of data values, as it deals with only specific values. Validation, however, only analyses the specification to an extent as large as that determined by the set of test cases or scenarios. Analyses such as deadlock freedom, livelock freedom, and general properties across the entire system are difficult, if not impractical to perform. Formal validation can be related to testing in software engineering context where implementation are

checked [11] if they are 'doing the right thing'.

2.3.3.2 Verification

Verification, in context of formal methods, is the proving of system properties in general. Model checking and theorem proving are the well-known formal verification techniques. Model checking, in simple terms, verifies if a system model satisfies desired properties which are specified using logic formulas. As an example in process algebra, LOTOS can be model checked (via a labelled transition system) using the μ-calculus temporal logic property language [66]. Properties across the entire behaviour such as deadlock freedom, safety, and liveness can be specified and verified. Model checking faces a problem of state space explosion in the case of finite state space as a result of composition. However there are techniques that reduce [37] or avoid (symbolic model checking [68]) this problem. Even so, model checking in finite state spaces can be practical, cost effective, and can detect errors even in constrained spaces [18].

Another verification technique is theorem proving, with an overall concept of inference from a constructed theory, and automated reasoning to obtain proofs. Generally there is automated deduction of theorems by the prover inference engine from existing axioms and theorems. Theorem proving has great capacity in verification, but is very costly and most times requires interaction or guidance (strategy, tactics) from tool users to achieve verification [62]. This may not be considered worthwhile unless it is of high importance. In comparison, many of the model checking techniques are cost-effective when verifying critical systems and protocols.

2.3.3.3 Evaluation

The well-known techniques have their strengths and weaknesses. With regard to web and grid services, formal analysis requires practicality and effectiveness. Validation and verification by model checking have these attributes – validation is fast and relevant to testing software, and model checking verification is more push-button, simpler, practical, and cost-effective in comparison to theorem proving. The combined use of validation and verification for composed web and grid services is able to offer formal analysis in a practical and effective way.

2.3.4 LOTOS

2.3.4.1 Concepts and Language Overview

LOTOS (Language Of Temporal Ordering Specification [50]) is a process algebra which is based on the CCS (Calculus of Communicating Systems [46]) and CSP (Communicating Sequential Processes [69]) for describing the dynamic behaviour of systems. LOTOS allows the modeling of concurrency (interleaving), nondeterminism, synchronous (and asynchronous) communication. LOTOS is based on the idea that a system can be specified by defining the temporal relation and interactions among the processes that constitute the behaviour of a system. These points of interaction are known as (event) gates in LOTOS, and their communication is known as synchronisation. LOTOS has powerful and expressive parallel composition operators which describe interleaving, multi-way, and full synchronisation.

There are variations of the LOTOS language, of which Full LOTOS is used as the specification language by the thesis. Full LOTOS consists of two parts: abstract data type modelling which is based on the ACT ONE [21] an algebraic approach, and behaviour specification using process algebra.

A Full LOTOS specification has the syntactic structure of data types from a library, overall specification behaviour, local type definitions, and local process definitions. The data type library is a convenient way to include common data types already defined, thereby reusing their specifications. The overall behaviour specifies the behaviour of the entire system, which is a behavioural expression that typically describes the composition of processes. Usually this composition may constrain independent processes, thereby specifying the overall behaviour expected of the system. After the overall behaviour comes the specification of local data types which are visible to all processes (though a process may define its own data types). Processes defines parts of behaviour which will finally be composed as the overall behaviour. A process usually specifies at least one gate (otherwise there is no communication at all), and may have input and exit parameters if required in its behaviour. A process behaviour is called using its name, gates, and input parameters.

2.3.4.2 Data Type

Abstract data types capture the data operations and values which may be used in behaviour, enabling behaviour to be influenced by data. A data type is usually defined by its sorts ('types'), operations, and algebraic equations which describe the operations. Operations can be defined to be parameterless, or infix/prefix with parameters. Equations are rewritten/expanded to transform data values. The following is an example of a Natural data type, with operations and equations which model an infinite value range of natural numbers.

```
Type Natural Is                                          1
  Sorts                                                  2
    Nat                                                  3
  Opns                                                   4
    0: -> Nat                                            5
    succ: Nat -> Nat                                     6
    _+_: Nat,Nat -> Nat                                  7
  Eqns                                                   8
    ForAll m,n: Nat                                      9
      OfSort Nat                                         10
        m + 0 = m;                                       11
        m + succ(n) = succ(m) + n;                       12
EndType
```

The value 0 is represented by the parameterless operation '0' which is already its canonical form as there are no equations that rewrite it. The 'succ' operation represents the successor value, e.g. succ(0) which is in canonical form corresponds to 1. The infix _+_ operation is for adding two naturals, and has the equations at line 11-12 for rewriting. For instance, the following expression is rewritten as:

```
0 + succ(succ(0))
-> succ(0) + succ(0)
-> succ(succ(0)) + 0
-> succ(succ(0))
```

Formal data types may be specified, meaning that they are not actual types but are templates used for instantiation, where the formal sorts, operations, and equations are actualised. Data types can be used as the basis of defining new data types, where the new data types have access to their sorts, operations and properties (equations).

2.3.4.3 Behavioural Expression

LOTOS processes (defined with **Process**) are named behaviour expressions, similar to procedures in a programming language, allowing a structured and modular expression of behaviour.

Behaviour Expression	Description
i	Internal event
stop	No behaviour
exit	Successful termination
g !v1 !v2	Action at gate g with offers of values v1 then v2
g ?var:type [*predicate*]	Action at g with value passing (acceptance) of a variable var of a give type, with a boolean predicate constraining value range
B1; B2	Sequential behaviour where B2 follows after B1, where B1 is an action event (not a process call)
B1 [] B2	Choice of behaviour between B1 and B2
B1 ||| B2	Interleaving behaviour of B1 and B2
B1 |[p,q]| B2	B1 and B2 synchronised at gates p and q
B1 || B2	All gates of both B1 and B2 are synchronised
hide g **in** B1	Gate g is hidden from behaviour external to B1, treating it as an internal event
B1 » B2	Successful behaviour of B1 enables B2
B1 [> B2	B1 continues (may exit) unless B2 happens (disruption)

Table 2.1: A Subset of LOTOS Behavioural Expressions

Behavioural expressions determine the possible actions that can be executed, and the possible actions that then follow. LOTOS provides a predefined set of operators to combine behaviour expressions, creating a new behaviour expression. Table 2.1 lists a subset of the LOTOS behaviour expression syntax. An action is an atomic behaviour expression. Actions comprise a gate, optional list of events (value offers and acceptance), and an optional predicate that constrains event values. The parallel composition operators '|||', '|[...]|', '||', specifically the latter two, express behaviour synchronisation where information exchange may take place.

2.3.4.4 Example Specification

The following is a LOTOS specification of a travel agent holiday booking behaviour using two agents for flight booking and hotel reservation. The Airline and Hotel are independent behaviours, composed for the holiday booking functionality. The composition for the travel agent is the logical unit behaviour of both Airline and Hotel processes interleaved '|||', with their gates synchronised with the Travel_Book process. This establishes a means of communication between the travel agent and airline, and travel agent and hotel. The 'hide' operator results in the specified flight and reserve gates becoming internal, unobservable from external view

of the travel agent. Clients need not be aware of the business partner processes inside this specification. The behaviour of Travel_Book receives a holiday plan (line 40), then makes a flight booking request using the customer's name and destination of travel (line 41); it then receives the flight booking (line 42). This is similarly done for the hotel reservation (lines 43 & 44). The travel agent then produces an itinerary with the bookings (line 45). Lines 41 and 50 are an example of value passing upon synchronisation at the gate 'flight' where the two values specified at line 41 are accepted at line 50. As seen, this specification constrains the behaviour in a sequential manner with the order of flight booking followed by hotel reservation. The behaviour expressions in Airline and Hotel processes are abstractions of what really happens, e.g. database entry for reservation. The data type used is of Text sort, which was included from a library (line 2) that was already defined by the user as a common data type to represent strings and making it available via a library. Comments in LOTOS are enclosed within '(*' and '*)'. **NoExit** (e.g. lines 1, 39) indicates that the behaviour expression should not terminate successfully at all (only stop or repeating). Conversely, an **Exit** allows a behaviour expression to terminate successfully, by specifying **exit** action.

```
Specification Travel [travel] : NoExit                                  1
Library                                                                 2
  Text                                                                  3
EndLib                                                                  4
Behaviour                                                               5
  hide flight, reserve in (                                             6
          Airline [flight]                                              7
          |||                                                           8
          Hotel [reserve]                                               9
  )                                                                     10
  |[flight,reserve]|                                                   11
  Travel_ Book[flight,reserve,travel]                                   12
Where                                                                   13
Type Flight Is Text          (* flight uses text library type *)        14
  Sorts Flight                                                          15
  Opns                                                                  16
    Flight : Text, Text -> Flight (* name and destination*)              17
  ...                                                                   18
Type RoomBooking Is Text                                                19
  Sorts RoomBooking                                                     20
  ...                                                                   21
Type Holiday Is Text                                                    22
  Sorts Holiday                                                         23
  Opns                                                                  24
    Holiday : Text,Text -> Holiday                                      25
    getDestination : Holiday -> Text                                    26
    getName : Holiday -> Text                                           27
```

```
  Eqns                                                                    28
    FORALL dest,name:Text                                                 29
    OfSort Text                                                           30
      getDestination(holiday(dest,name)) = dest;                          31
      getName(holiday(dest,name)) = name;                                 32
EndType                                                                   33
Type Itinerary Is Flight, RoomBooking                                     34
  Sorts Itinerary                                                         35
  Opns                                                                    36
    Itinerary : Flight, RoomBooking ->> Itinerary                         37
  ...                                                                     38
Process Travel_ Book [flight,reserve,travel] : NoExit :=                  39
  travel ?plan:Holiday;                                                   40
  flight !getName(plan) !getDestination(plan);                            41
  flight ?flight:Flight;                                                  42
  reserve !getName(plan);                                                 43
  reserve ?roomBook:RoomBooking;                                          44
  travel !Itinerary(flight, roomBook);                                    45
  Travel_ Book [flight,reserve,travel]                                    46
EndProc                                                                   47
                                                                          48
Process Airline [flight] : NoExit :=                                      49
  flight ?name:Text ?destination:Text;                                    50
  flight !Flight(name, destination);                                      51
  Airline [flight]                                                        52
EndProc                                                                   53
                                                                          54
Process Hotel [reserve] : NoExit :=                                       55
  reserve ?name:Text;                                                     56
  reserve !RoomBooking(name);                                             57
  Hotel [reserve]                                                         58
EndProc                                                                   59
EndSpec
```

2.3.5 Topo/Lola

Topo/Lola [81] is a set of tools can analyse data types and behaviour specified in LOTOS. Data type specification can be analysed with respect to their operations which represent abstract data values, which in turn are used by the system behaviour for value negotiation. Lola uses a rewriting technique that expands the algebraic data operation equations to their canonical form and provide step-by-step trace of the expansion, which can be used to correct specified data types. Simulation of specification can be performed interactively to observe the behaviour. Lola supports testing, known as test expansion, that allows LOTOS processes to be specifically defined as expected or unpermitted scenarios which are used to validate the specification. In addition to testing outcomes, Lola provides diagnostics of the execution of test validations which can be used to inspect successful and unsuccessful path traces of the behaviour with

respect to their tests.

2.3.6 CADP

CADP (Construction and Analysis of Distributed Processes [36]) is a comprehensive suite of tools for formal analysis, particularly supporting LOTOS specification (with undocumented support for other FDTs such as Petri nets). CADP supports a range of analysis and tools such as simulation, testing, model checking, compositional verification, state space reduction, bisimulation, performance evaluation, etc. CADP has two main components, CAESAR [38] and CAESAR.ADT [32], which translate specifications into C code that is executed to simulate the behaviour by which the model is created to be analysed. CADP has a framework and programming interface for extensibility, supporting the possibility of combining with other tools. It also provides a graphical user interface EUCALYPTUS Toolbox [33] to use the toolset. These contributes to the success of CADP, which has been used in verification of many protocols, hardware and software; see the case studies found in [110].

2.3.7 Evaluation

The formalisation of composed web services is possible using Full LOTOS, as demonstrated by various work that use the specification language to specify and analyse web service compositions [25, 17], including the approach that thesis work is based on [101]. LOTOS was evaluated to be most suitable given its semantics, expressiveness, ability to specify data types, and availability of comprehensive tool support for validation and verification. LOTOS is potentially applicable to formalise grid service compositions as they are largely based on similar service-oriented architectures, with preliminary work of this research demonstrated a development approach that includes specification and analysis of composite grid services [92].

2.4 Implementation

2.4.1 Web Services

The SOAP and WSDL standards, and other XML standards they use, are common web service specifications. There are implementations of these standards, providing an infrastructure and

framework to develop web services. Apache AXIS is a well-known, very widely used open-source implementation of these standards.

2.4.1.1 AXIS

AXIS (Apache Extensible Interaction System [5]) is fundamentally a SOAP engine and server, but also provides an infrastructure to enable web application servers (e.g. Apache Tomcat) to implement web services, and libraries for client applications to communicate in SOAP. AXIS is widely used in the development and deployment of web services. It offers open-source, stability, performance, and architectural flexibility for custom extensions. AXIS underpins many BPEL engines, and also the Globus Toolkit. AXIS provides a framework that supports application services and clients readily. The framework comprises tools that can generate an implementation of service stubs from WSDL (e.g. wsdl2java for Java implementation), service implementation code skeletons, and deployment descriptors. The service stubs provide a layer of encapsulation, taking care of underlying SOAP communication between service and client, therefore allowing focus on development of the actual functionality of service and client. For the same reasons, AXIS is also used in the implementation of web and grid services, both directly and indirectly as in the case of service compositions.

2.4.2 Grid Services

2.4.2.1 Globus Toolkit

Globus Toolkit is the de facto development tool for supporting grid services [87]. Its development has progressed from pre-web service implementations to GT3 that implements OGSI based on web service standards. GT4 is the most recent version that implements the WSRF specification. GT4 is an open-source development toolkit for developing grid services. In addition to providing a framework and build tools for service and resource development, Globus provides infrastructural services and interface providers that can be readily configured and deployed for use by application services in grid environments. Infrastructural services such as MDS (Monitoring and Discovery Service) implement the functionality commonly required in grids, as defined by OGSA. Providers are implementations of the standard interfaces defined

in the WSRF collection, such as ResourceLifetime and ResourceProperties. These provide, by deployment configuration, pluggable functionality for resources as required.

2.4.3 Service Orchestration

2.4.3.1 ActiveBPEL

ActiveBPEL is an open-source (GNU General Public License) BPEL engine [3], developed by Active Endpoints, that deploys BPEL services and executes their processes. It was developed in Java, and can be quickly deployed into any standard servlet container such as Apache Tomcat. It uses AXIS as the underlying web service engine. It defines its own framework to deploy and configure BPEL services, such as the PDD (Process Deployment Descriptor), catalogues for locating WSDL service interfaces, the file format for deployment, etc.

ActiveBPEL is purely a process execution engine and does not provide any tools for the development of the services such as describing BPEL behaviour and service interfaces. However there are service development tools available such as the ActiveVOS (previously ActiveBPEL Designer) that provide the environment for creating BPEL services, interfaces, and deployment configuration for ActiveBPEL, usually graphically.

ActiveBPEL provides an administrative console via graphical web pages (servlets) for the configuration of the BPEL engine, inspection of the details of service deployment, monitoring and diagnostics for process execution. Configurations such as setting service timeouts and resource allocation, etc. can be performed via the console. Administrators can check the deployment status containing information such deployment logs and service details (WSDL, and catalogue consolidated by engine). The console provides graphical diagnostic of process execution which is useful to inspect process flow, data values, and errors.

ActiveBPEL has a community that offers commercial support, contributing to the stability of the BPEL engine and production-level confidence. This model of product development has demonstrated itself successfully, evident from the ongoing development in the features of the engine with respect to the evolving standards, and user uptake. The recent versions of ActiveBPEL engine support the web services standards which are harmonised with those used in the WSRF specification, which means that grid services can also be orchestrated with the

engine.

2.4.3.2 Oracle BPEL Process Manager

The Oracle BPEL Process Manager [77] is a commercial BPEL engine. It is part of the Oracle SOA Suite which is geared towards rapid design, assembly, deployment, and management of business applications, comprising tools such as JDeveloper which is the service development environment.

BPEL Process Manager has its own extensions in addition to supporting the WS-BPEL standard for additional features within a BPEL service; however, this implies that using the proprietary extensions loses portability to other BPEL engines. It provides its own application server, unlike the likes of Tomcat enabled with AXIS, which are all open-source.

The Oracle JDeveloper can be use as the authoring tool to develop BPEL services, supporting phases in design through coding, debugging, optimisation, profiling and deployment.

2.4.3.3 Apache ODE

The Apache ODE (Orchestration Director Engine [94]) is a business process execution engine that supports the BPEL standard. ODE runs within Apache Tomcat like the ActiveBPEL engine. Apache ODE also provides extensions which are beyond the WS-BPEL standard. The extensions include support for REST (Representational State Transfer) web services, which are services that are not based on WSDL/SOAP protocol but rather based on an architecture that uses HTTP methods explicitly as the stateless communication protocol.

Apache ODE is a barebone distribution only supplying the execution engine supplemented with a management API which is accessed via programming. There is no user development environment for creating processes. This is suitable for BPEL vendors who can build and distribute development environments which are based on ODE, such as the Intalio Designer.

2.4.4 Evaluation

The evaluation of the technologies determined that AXIS and GT4 are the technologies most suitable for the research use to implement web and (WSRF) grid services respectively, mainly

for the reasons that they are widely-adopted and open source. Considering the research objective of high-level development, this meant that the other technologies were potential "targets" for implementation that could be supported.

The evaluation also considered the features of BPEL engines and the objective to choose a suitable BPEL engine for implementing the web and grid service compositions. ActiveBPEL and Apache ODE were favoured as they are open-source implementations, which are further based on open-source technologies (i.e. AXIS and Tomcat), and support standards in harmonisation with web and (WSRF) grid services. Their engines are also more compact compared to, for example, the Oracle Business Process Manager which provides an application server but is heavyweight. Although both are similar in comparison and potential for use, ActiveBPEL is more suitable for research than Apache ODE, as it has better support of features and control [13]. CRESS (see section 2.7), on which my research was based, can create BPEL services in the deployment format of ActiveBPEL as its service implementation support. This was exploited to the research's advantage by extending the previous work to meet the thesis objectives. However this does not rule out the use Apache ODE and others, as the thesis's methodology is not constrained to any particular deployment. For instance, CRESS could be extended to generate deployment code for Apache ODE for deployment of the same BPEL code.

2.5 Implementation Validation and Performance Evaluation

Implementation validation refers to the testing of implemented functionality. There are general technologies, such as JUnit, to define and execute tests. BUnit and soapUI are examples of technologies specific to web (SOAP) service testing. They respectively support offline simulation of BPEL orchestration (only in ActiveVOS), and web service testing.

JUnit [54] takes a programmatic (Java) approach with a framework to construct and execute tests. With regard to testing web and grid (composite) services, JUnit tests are basically service clients which requires the implementation of service interfaces and protocol (SOAP). There is not much automated support apart from tools that generate service interfaces to service stub code. Analysts or testers would have to hand-craft the tests.

BUnit (BPEL unit tests) is a functionality in ActiveVOS [2] supporting simulation of de-

fined orchestrated behaviour in BPEL, allowing specification and recording of data values; however, this is limited to the environment of ActiveVOS, which is a designer for BPEL services implemented in ActiveBPEL.

soapUI [23] is a desktop application for testing web services and REST services, with support for functional testing and load testing, and test report features. For web services, tests are oriented to the SOAP protocol, meaning that the tester defines SOAP requests and adds them to a test case or suite, and also a variety of assertions for response messages. soapUI is comprehensive for its purpose; however, the syntax to express tests is not very compact, is rather low-level, and is stored as soapUI's own XML project structure. The semantics of functional tests is limited to response-oriented assertions. There are no direct expressions of other semantics such as refusal, choices, non-determinism, and concurrency, which are potentially useful in describing validation scenarios.

In consideration of the thesis objectives, the technologies evaluated were found unsuitable as the technologies are either too general (JUnit), too proprietary, and too low-level. In addition, they do not have an aspect of formal validation for a symmetrical balance in a development methodology. These omissions, specifically in the formal analysis, are however reasonable considering their own objectives.

2.6 Service Composition Methodologies

2.6.1 Formalising Composed Services

Other authors [17, 25] have used LOTOS to specify web services. Their work advocates the use of process algebra as the initial step in the design and development of web services, specifically demonstrated using LOTOS. Their approach uses refinement and reverse engineering, which respectively encode to and abstract from BPEL implementation. They define a mapping between the formal and implementation constructs as the core of the approach. Refinement starts the development process from the abstract specification in LOTOS, where data types and system behaviour are described. The LOTOS is then re-coded as the service implementation. Reverse engineering begins from the BPEL implementation and is abstracted to LOTOS using the mapping. CADP is used as the underlying formal analysis tool for the abstraction.

Although the approach can be used in the development of web services, the possibility of its application may not be very pragmatic. The mapping that was defined has not demonstrated support for compensation and complex data structure access (although there is complex data definition); these are all typically used in practice. As such it may not be possible to apply the reverse engineering approach to obtain the abstraction. Abstraction of data types is into natural numbers, which is not sufficient to support the detailed analysis of complex types where their actual value representation is important. Finally, the analysis (verification) using CADP requires pragmas annotating the LOTOS abstract data types in order to qualify for verification, which is hand-crafted for all types. There is no automated tool support [78].

Another process algebra approach uses value-passing CCS to formalise BPEL4WS services [125]. This work defines a mapping from BPEL4WS to CCS. Bisimulation analysis is supported for iterative refinement of service composition, whereby the current version (which is less abstract) is verified to be in correspondence with the more abstract model of the previous version. However this approach does not support several of the major BPEL4WS constructs such as parallelism, fault handling, and compensation. Dynamic process interaction is also not considered in this work.

There is a process algebraic approach to specification and refinement of BPMN-defined workflow processes using CSP [114]. This work uses control flow patterns (e.g. sequence, choice, parallelism and joining) which were developed for Petri nets in separate work by another author [109], defining CSP models for workflow processes. The FDR (Failures-Divergence Refinement) model checker is used to analyse behavioural properties by which counterexamples can be obtained for compromising properties, and therefore to identify refinements to the model. The authors have stated the limitations of their current work. Exception and compensation semantics, as well as dataflow semantics, are not supported. There is no automated translation of BPMN to CSP models. The authors have considered these limitations and state their intention to address them in future work.

LTSA-WS (Labelled Transition System Analyzer for Web Services [28]) is a mature approach to describe composed web services in a BPEL-like manner. It models activities between a business process and its partners. [28] describes an elaboration of the original LTSA-WS work. It takes a synthesis approach whereby analysers and developers respectively describe

the formal behaviour as a series of MSCs (Message Sequence Charts [52]) and an implementation composition (a BPEL4WS process). These each generate a behavioural model using FSP (Finite State Processes). Validation and verification are performed by comparing the two behavioural models generated from the MSCs (Message Sequence Charts) and a BPEL4WS process, determining if the implementation contains all the specified (formal) scenarios. This approach implies the description of the same web service composition twice, once each for the MSCs and the BPEL4WS implementation. Validation is carried out in animation and simulation of the behavioural models, with interactive trace as diagnostics to adjust the relevant sequence diagrams. The verification supported in this work is trace equivalence between design and implementation, interface compatibility such as no suitable reply from a partner implementing a specific service port, and safety and progress properties using FSP property notation. This work employs a static representation that depends on the occurrence of conditional variable comparisons, in order to support the modelling of process execution paths typically affected by data values. Therefore detailed analysis, such as verifying properties involving specific data values, is not considered in the verification analysis (as stated by the authors).

WSAT [31] is used to analyse and verify composite web services, particularly focusing on asynchronous communication. Specifications can be written bottom-up or top-down, finally being analysed with Promela/SPIN [47]. For composite web services that interact asynchronously, WSAT is able to verify the concepts of synchronisability and realisability. This approach makes the assumption that service links among peers are pre-determined and established prior to interaction. This means that the locations of service partners must be statically bound. Advanced features in BPEL, such as endpoint references that dynamically determine the peer to talk to, cannot be captured in the WSAT model. A composite web service specification such as BPEL will usually contain error handling and also compensation code, therefore it is desirable to be able to model compensations in web service compositions. WSAT has not addressed these aspects, which are very commonly found in service descriptions. The scope has yet to be comprehensive enough to be of more complete practical use for creating web service specifications. The authors have indicated that extending WSAT to address these issues will be a direction in the future. A similar approach to WSAT formalisation is the translation of BPEL code into specifications [73] where there is support for the DPE (Dead Path Elimination)

of the BPEL semantics; this is used for join conditions in a flow. This approach also does not support error handling and compensation in the formalisation.

There are approaches that use Petri nets to analyse web service compositions. A theoretical framework using Petri net-based algebra has been proposed [44] with operators such as (arbitrary) sequence, alternative, iteration, parallelism, selection, etc. to model control flows of complex web service combinations. This approach, however, does not provide the implementation of the framework and has not demonstrated support for fault handling and compensation semantics. It has no pragmatic support for the kinds of analysis (e.g. compatibility) that have been suggested. Another approach [82] developed a Petri net-based technique (C-net) to model and analyse web service interactions, supporting basic and structured activities as well as interfaces. The C-net structure is analysed for service compatibility in behaviour using a policy of adding information channels to resolve incompatibility. It assumes that the services are compatible at the syntactic interface level. C-net can be transformed into BPEL code using the ActiveVOS engine, which is pragmatic for implementation. However it does not support the fault and compensation constructs of BPEL.

Another approach [78] provides mappings of control flow constructs for individual executable BPEL process to Petri nets specified in PNML (Petri Net Markup Language [10]). Analysis is automated using the WofBPEL tool [79] for detecting unreachable activities, detecting multiple simultaneously enabled activities of the same messages types, and determining for each possible state of a process the types of messages that may be consumed for the rest of the execution. Another similar work defines mapping semantics for BPEL to Petri nets, and automates this mapping [45]. A wide range of BPEL4WS constructs is supported. The authors state limitations [88] that data is abstracted to tokens, and that high-level constructs such as transition guards and variables are omitted but represented as non-deterministic choices instead of data evaluation, so the resulting model is a low-level Petri net. The automated tool support is limited only to the transformation from BPEL to Petri nets.

A variation of π-calculus (web$\pi\infty$) was also used to define unambiguous semantics for WS-BPEL 2.0 to address some open issues of BPEL [63] such as complete condition. Event-based mechanisms were proposed with emphasis on formalising the WS-BPEL error recovery framework implemented through event, fault and compensation handling. Variable and global

state handling are not supported but have been identified as future work. The focus is on addressing open issues of ambiguities in the specification documentation. The framework has no automated support in specification and analysis for compositions.

Formalisation of grid service composition has had very limited attention in contrast to web services [127]. One approach uses π-calculus to formalise grid service compositions; verification is performed in consideration of the service interactions [42]. Service behaviour is specified in π-calculus using workflow and client/server patterns. The PGSCV (Pi-calculus based Grid Service Composition Verification) algorithm automatically analyses the behaviour to establish interaction patterns; these are inserted into a grid service context that is used to verify if the composition is correct. Another approach developed a variation of π-calculus; Cpi-calculus (Conditional π-calculus) has been investigated for grid service composition [127]. It developed and proposed composition signatures for grid services to precisely model grid service compositions, particularly of their concurrency aspects. However this approach is not very pragmatic in a typical development environment as the specifications have to be hand-crafted, and likewise in the analysis which requires expert knowledge of the techniques.

Web service composition has also been studied using a performance/stochastic model-based formalism. A preliminary approach defines mappings from BPEL4WS to PEPA (Performance Evaluation Process Algebra), implemented as an automated translation tool as proof-of-concept [71]. The approach states that the BPEL and WSDL specifications give the structure of the PEPA model, but gives no information on the stochastic parameters (e.g. rates) for the activities translated. The approach extends the schema for WSDL with optional estimated "latency" attributes to be specified for each operation, and therefore translates into PEPA components with latency. BPEL activities are defaulted to a timing of 1.0 seconds. The tool invokes the PEPA workbench to calculate the throughout of the model. Fault handling is not addressed in the modelling, and factors such as communication cost and load are not considered for the analysis.

Formalisation has been applied to distributed systems in general. One approach uses UML notation as a graphical abstraction to formalise distributed systems (object and component middleware), using process algebra for the underlying specification and analysis [55]. This approach suggested a set of stereotypes as UML class diagrams and statecharts to describe syn-

chronisation (e.g. synchronous, one-way) and threading (e.g. single-threaded, multi-threaded) primitives. CORBA (Common Object Request Broker Architecture) is used as a superset of primitives to other distributed technologies such as Java RMI (Remote Method Invocation). The approach has defined process algebra semantics for these stereotypes. Users include the stereotype annotations in the design of class diagrams and statecharts. UML diagrams are translated using the defined semantics into FSP, which is then model checked using compositional reachability analysis [16] to find potential deadlocks or synchronisation flaws in the design. Property violations are presented as UML sequence diagrams. Further work added more stereotype primitives for the specification of safety and liveness properties [56]. Another approach developed a connector synthesis technique for deadlock-free COM/DCOM applications [48, 49]. This technique uses comments in component interfaces where their dynamic behaviour is specified in a CCS-like process algebra and a connector is specified to handle interactions between clients and servers. This is synthesised and deadlock analysis is performed on it, removing deadlock behaviour caused by the composition of components in the environment (but not those internal to the components). The outcome is a connector implementation containing deadlock-free routing policies that filter client requests to servers. The approach can be applied to single-threaded and single-layered (not composite) components. The authors state a major drawback is that clients will require code modification to make calls to the connector, which has become the new server component that contains the old ones. The authors have plans to address this drawback and to extend this work to support multi-threaded and multi-layer servers.

2.6.2 Implementing Composed Services

There are several service development tools for creating BPEL services. ActiveBPEL Designer, Oracle JDeveloper, BPEL Designer, and Intalio Designer are instances of such tools; some of them are available commercially and some for community use. These tools usually support specific BPEL implementation(s). ActiveBPEL Designer creates deployments for the ActiveBPEL engine, while JDeveloper deploys on its own application servers. BPEL Designer and Intalio Designer are for Apache ODE.

JOpera [80] is a service composition tool for building new services by combining existing ones. It provides a visual composition language and also a run-time platform to execute services. JOpera claims to offer greater flexibility and expressibility than WS-BPEL. Although JOpera initially focused on web services, support for grid service composition has also been investigated. The visual composition language is not a standard like WS-BPEL. This is not so appealing especially for users who prefer a choice of implementations that support a standard.

OMII-BPEL (Open Middleware Infrastructure Institute BPEL [22]) uses BPEL to support the orchestration of scientific workflows with a multitude of service processes and long-duration process executions. This work investigated the feasibility of orchestrating grid services. It provides a designer, Sedna [112], for process development (now known as OMII-BPEL Designer), and a tailored ActiveBPEL engine to execute and monitor processes. This provided the UK e-Science community with an infrastructure for developing and executing (scientific) workflows. OMII-BPEL has provisions for WS-Security [72] to secure and authorise communication to and from the BPEL engine, achieved by using security handlers in ActiveBPEL via the AXIS handler and architecture which allow such extensibility.

Taverna [76] is a toolkit for developing bioinformatics workflows. It introduced SCUFL (Simple Conceptual Unified Flow Language) to model grid applications in a specialised workflow language. Taverna eases explicit data modelling and includes support for service discovery. Taverna has a focus on bioinformatics, although there has been application to several domains as it was developed. Although it may have its strengths, technologies that are based on standards such as BPEL are generally more applicable, and are more likely to be favoured by adopters.

There is generally no support of formalisation and analysis in development and service creation tools, although most do support static validation of implementation design. These pragmatic tools are potentially applicable and are not in conflict with regard to the thesis objectives and methodology: instead, they complement this research. The implementation backbone (engines) behind these tools, for instance Apache ODE, OMII-BPEL, and Taverna, are potential target implementations applicable to the thesis, which takes a high-level approach to service description and generating service implementations for supported target languages.

2.6.3 Comparison With Related Work

The thesis work differs with respect to the foregoing approaches in various ways, but also shares similar motivations. The thesis shares the motivation of automated tool support. The thesis employs a single, high-level, graphical design to describe entire service compositions, and the composite description is automatically translated into specification and implementation. The approach also supports grid service composition, and nested service compositions. The thesis shares the views that process algebra approaches are suitable for service composition, and the benefits of providing abstraction to formal methods and tools [17, 25, 55, 84]. LOTOS was chosen as the formal specification language as it has been successfully used to describe web service composition [101], and it allows for analysis considering data; adequate tools were already available. The thesis approach to analysis differs. High-level languages and tools are used in this thesis to specify and highly automate validation and verification, extensible and independent of the underlying techniques. Data types and values are supported in the formal validation and verification. Templates are provided to support verification for the definition of commonly specified properties [67]. The WS-BPEL 2.0 standard is supported for implementation of composed services. There is support for implementation testing that is highly automated; this reuses the same validation descriptions as defined for specification.

However it is not necessary that other methodologies and technologies be mutually exclusive with the thesis work. The implementation tools are complementary in that they provide service enactment engines which can potentially be exploited as the target underlying technologies in the methodology when turning design into implementation, taking advantage of their distinguished capabilities. OMII-BPEL, for example, provides a secured BPEL engine to enact processes enabled with security mechanisms.

2.7 CRESS

CRESS is an abstract graphical notation and tool that has a methodological framework for high-level service design, automated formalisation and analysis, and automated implementation. This approach fits nicely to the foundational part of the thesis objectives. CRESS has a web service domain where it supports the development of (composed) web services.

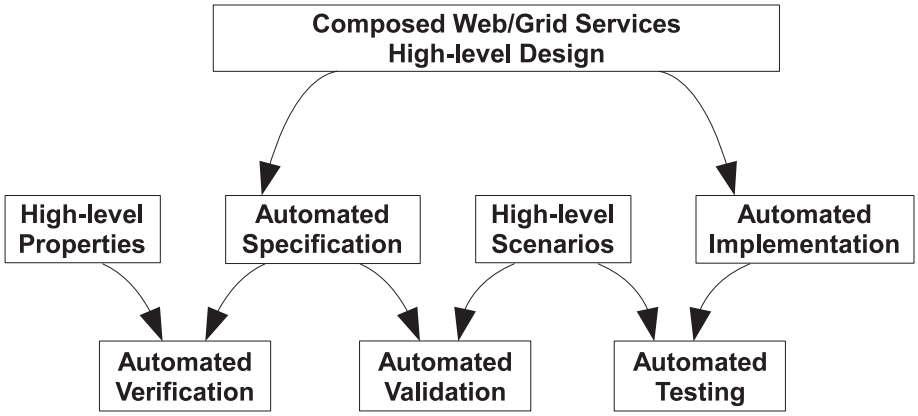

Figure 2.2: Desired Approach For Creating Composed Web/Grid Services

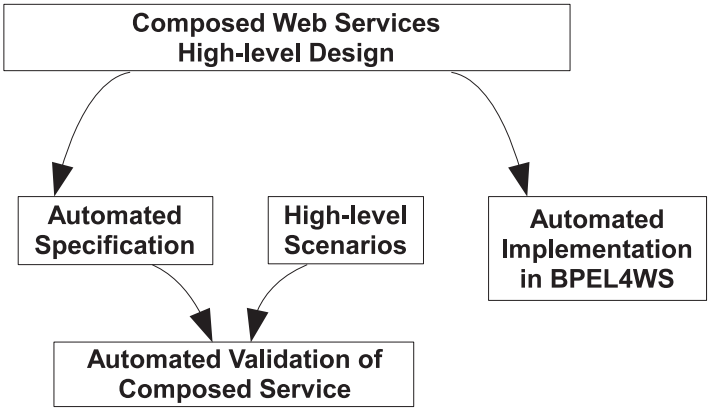

Figure 2.3: CRESS Goals Compatible With Proposed Approach

Figure 2.3 shows an overview of CRESS goals pertaining to its web service domain. These are compatible with, and are a subset of the thesis approach proposed in figure 2.2. CRESS uses a high-level approach whereby services and analyses are abstracted from actual technologies and automated in their realisation. These are therefore suitable as the basis for the approach required in the thesis work. The original CRESS framework for the web service domain supported automated specification, formal validation of composed services, and BPEL4WS as a target language [102]. Their respective strategies could be extended to support a new domain for grid services, automated specification for composed grid services, validation for web/grid partner services, and WS-BPEL 2.0 specification as target language for composed web/grid services. High-level verification property specification support, automated formal verification, and automated testing have been the new areas of development to meet the thesis required integrated methodology. CRESS's extensibility and tool integration framework were used as the basis to meet the thesis objectives.

CRESS is independent from domains and languages. The CRESS approach provides a simple high-level graphical notation to describe services and features without binding to any actual languages or implementation. The notation is context-free and is not bound to any particular application/domain. A domain can be added into the CRESS framework to enable support for service description. This implies that the context for diagram description (e.g. keywords, node syntax, etc.) are added into the CRESS framework for the new domain. Through this approach, many domains can be supported by the CRESS notation. To date CRESS has been used for creating services in the domains of IN (Intelligent Networks), IVR (Interactive Voice Response), VoIP (Voice over Internet Protocols), SIP (Session Initiation Protocol), and also WS (web services). Recently, the DS (Device Services) domain was added which enables OSGi (Open Services Gateway Initiative, now known as OSGi Alliance [95]) services to communicate with BPEL services. Actual implementations of a service are obtained by interpreting the high-level diagrams according to the domain context and translating into the actual target implementation language. Table 2.2 list a subset of the domains and their supported target languages. CRESS adopts an approach where actual implementation language support (e.g. translation strategies) can be added to the framework to achieve translation from CRESS descriptions of a given domain. Therefore it allows simple and manageable development (e.g. one-time graphical description). Multiple target implementations are supported. Target implementations do not mean only actual implementation where functionality is actually rendered, but can be virtually anything that their corresponding translations are intended for (e.g. textual documentation). Using this approach, the formal specification (as a target) of a CRESS diagram is automatically obtained. Automated formal specification has been practically applied to all services in nearly all the domains supported in CRESS, which uses languages such as SDL and LOTOS as target specification languages and has the potential to support others.

The CRESS approach therefore serves as a basis of rigorous development where formal specification is automatically generated, and in turn is the basis for formal analysis and thereafter implementation from high-level design.

Domain	BPEL	LOTOS	SDL
DS	√	√	
WS	√	√	
IN		√	√
SIP		√	√

Table 2.2: Subset of CRESS Supported Domains and Target Languages

2.7.1 CRESS Framework

As a high-level approach, the CRESS framework is mostly focused on the CRESS diagrams which are explicitly or implicitly involved in the activities executed by CRESS tools. For example, the generation of actual code (specification or implementation) involves translation from the diagrams. Performing formal validation requires a specification obtained from automated generation.

The CRESS tool framework, illustrated in figure 2.4, is extensible. The boxed area represents CRESS tools, with tools outside being externally provided. CRESS diagrams are not part of the toolset to allow freedom of choice of editors. Given a target context via the target language framework, CRESS diagrams are translated into intermediate diagram format (textual) which is then syntactically parsed and checked according to the domain's context (for example CRESS web service notation). After this the generation of the code will take place, realised via the target language realisation to the target language backend. Most of the tools in CRESS are written in the Perl scripting language for the advantages of being portable and exploiting to a large extent the language's pattern matching capability.

Table 2.3 and figure 2.5 [104] and describes the subset of the original CRESS tool framework and its relationships which support the web services domain. There are other CRESS tools are not presented here as they are intended for other domains not covered by the thesis such as VoIP, but they follow a similar framework dependency on common CRESS Perl modules (.pm files).

These CRESS modules and tools together automate the formalisation and implementation of web services, respectively in LOTOS and BPEL/WSDL. Formalisation in CRESS includes the specification and rigorous validation analysis by the tools cress_lotos and cress_validate respectively. Implementation is achieved with cress_bpel. The automated formalisation and

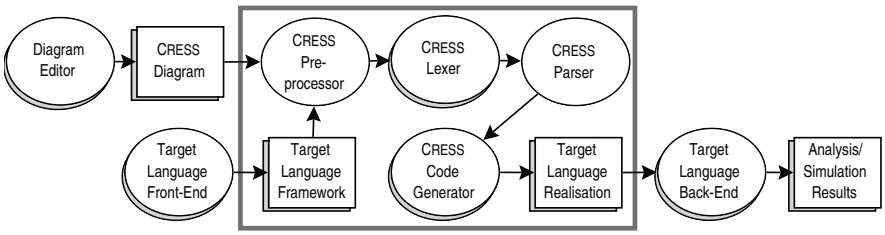

Figure 2.4: CRESS Framework [104]

Tool / Modules (.pm files)	Purpose
cress_bpel	main entry to translate CRESS diagrams to BPEL
cress_check	check CRESS diagrams based on domain (vocabulary)
cress_create	create service archives
cress_deploy	deploy service archives
cress_expand	expand macros in CRESS diagrams
cress_lotos	main entry to translate CRESS diagrams to LOTOS
cress_validate	validate CRESS diagrams
cress_bpel.pm	CRESS diagram to BPEL/WSDL translator and used by cress_bpel
cress_common.pm	CRESS common definitions
cress_lexer.pm	CRESS lexical analyser (diagram analyser) to produce the intermediate CRESS diagram format
cress_lotos.pm	CRESS diagram to LOTOS translator used by cress_lotos
cress_parser.pm	CRESS diagram parser (syntax analyser) for parsing and checking of CRESS diagrams produced with "cress_lexer.pm")
cress_vocab.pm	CRESS vocabulary support for all domains, e.g. web services vocabulary defines reserved names and checks diagrams

Table 2.3: CRESS Tools Relevant to Web/Grid Services [104]

implementation aspects of the methodology are thoroughly discussed in Chapter 5 and 6 respectively.

2.8 Summary

Significant technologies, languages, and methodologies have been evaluated to establish concretely the various aspects of the thesis methodology objectives and how to achieve a practical, effective and rigorous development framework for creating composed web/grid services with confidence.

CRESS was found to be suitable, in view of its practical capabilities and framework, for use

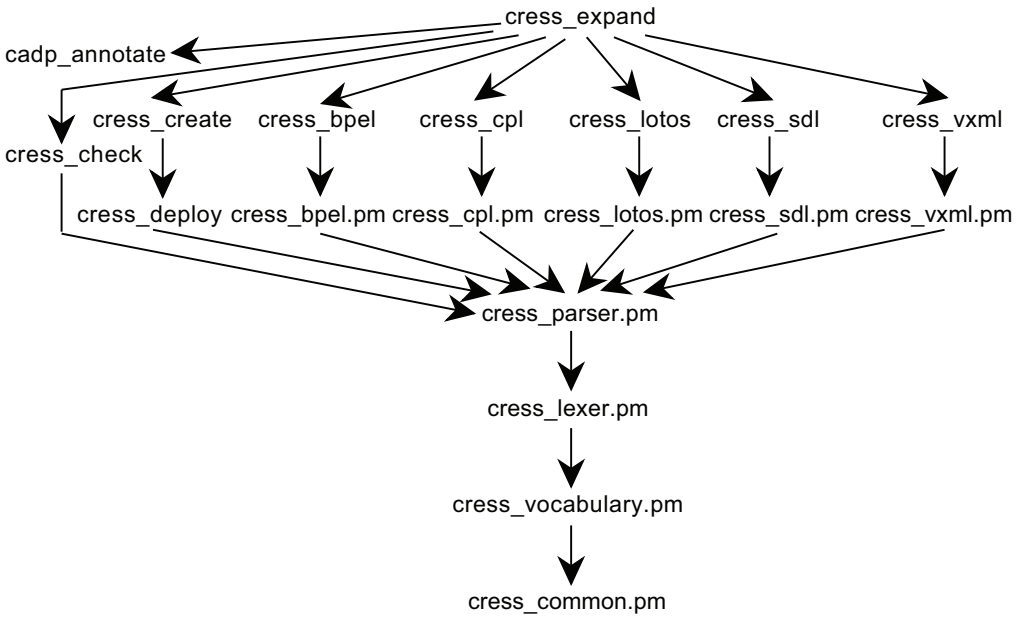

Figure 2.5: CRESS Modules Dependency [104]

as a basis for the thesis work. It has a high-level graphical notation for describing composed services; automated approach to formalisation; support for the widely adopted BPEL standard; and automated implementation of services, which is a pragmatic solution for rigorously developing composed web/grid services. It has an extensible framework which allowed the thesis work to develop aspects that were lacking in the CRESS existing approach to meet the required methodology. This has resulted in a more thorough rigorous development environment. CRESS is complementary to many of the evaluated technologies, which can be supported in CRESS as target implementations via its extensible framework. Developers can therefore exploit the CRESS capabilities within a single environment.

Chapter 3

An Integrated Methodology

This chapter describes the integrated methodology for creating composed web/grid services which underpins high-level design, automated formalisation, high-level formal validation and verification descriptions with automated analysis, automated implementation, and also automated implementation validation.

3.1 Goals

The thesis aim was to create an integrated environment for developers to create dependable composed web/grid services. The goals are:

- easy specification and development of realistic composed web/grid services using the CRESS high-level graphical notation and associated tools
- automated design-time (formal) analysis where specification, validation and verification are made simple for non-experts and accessible within the same development environment
- automated implementation with easy functional and performance evaluation, reusing the design-time validation tests

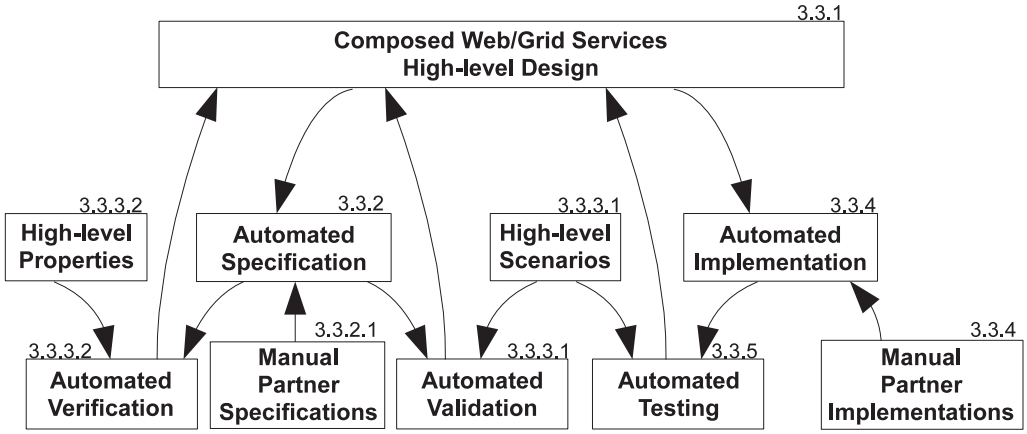

Figure 3.1: Integrated Methodology Development Lifecycle For Composed Web/Grid Services

3.2 Methodology Overview

Development of software in general goes through an iterative lifecycle of design and implementation, most likely with a variety of intermediate phases for analysis (e.g. verification, validation, and testing).

The thesis work has developed a methodology as a thorough development lifecycle for creating composed web/grid services within a single environment. The methodology is summarised in figure 3.1.

The methodology exhibits the following iterative phases: high-level design, abstract model specification, design-phase analysis, implementation, and post-implementation analysis. From an overall perspective, the methodology supports development iteration. Transitions from design to analysis, to implementation, to testing are supported in a convenient manner. This automates a lot of work on behalf of the developers and analysts, enabling them to focus on important tasks such as analysis, re-design, implementation, and testing.

The developer starts with the high-level design of the composed services and their configuration as CRESS service diagrams and configuration diagram. This is where the developer defines application data, partner involvement, service behaviour flow, and deployment configuration. These high-level descriptions are the foundation for the rest of the methodology – specifically seen in the automated formal specification and implementation, on which further activities are based.

Formal specification of the service behaviour is required in order to perform design anal-

ysis using validation (scenario tests) and verification (property assertions). A specification is automatically generated by CRESS tools that translate the high-level CRESS diagrams configuration descriptions into LOTOS specification. The generated specification fully specifies the behaviour of all CRESS services, and provides outline behaviour for non-CRESS partner services. The developer can provide handcrafted specifications for the non-CRESS partner services, which are usually created for a thorough analysis. The range of analyses that can be performed depends on the depends on the details of the specification – more thorough analysis for more detailed specification. Validation and verification of the services can be performed in either order as they are independent.

The developer defines high-level scenarios for the services to be validated using MUSTARD. The outcomes of validation, specifically diagnostic traces of validation failures, provide the developer with feedback on the CRESS service design and the overall specification. The developer can iterate the design, specification, and validation phases until satisfactory results are achieved: the approach is entirely up to the developer's preference for development. For example, a developer who prefers a progressive spiral approach may add more details to the specification of non-CRESS partner services and define more validation scenarios to be validated at the next iteration.

The developer specifies high-level properties to be verified using CLOVE. Usually CLOVE's predefined property templates are used as they are commonly verified properties. The CLOVE properties may be defined at any time and are only used when verification is performed. The outcomes of verification, in particular for compromised properties, provide counterexamples that the developer uses to diagnose and address problems. The developer can iterate design, specification, and verification phases as described above.

Once satisfied with the formal analysis, the developer should be more confident of the service design, particularly for the composed service. The developer then configures the services for implementation, and generates their implementation. CRESS composed services are generated as BPEL services, and non-CRESS partner services are generated for Java implementation. The developer then provides the detailed Java implementation of web or grid partner services, usually by adding code to generated code skeletons. Once implementation is complete, the services are then deployed into their respective service hosting containers (BPEL in ActiveBPEL,

partner web services in Axis, and partner grid services in Globus Toolkit 4). CRESS automates the service deployment if so requested by the developer.

The developer can validate functionality and evaluate performance on the deployed services. Validation of service implementations is similar to formal validation, re-using the same sets of MUSTARD scenarios but executing them against actual services using the MINT tool. This allows developers to be confident that what worked in the design (specification validation) is also exhibited by the service implementations (the ultimate products). The testing outcomes, specifically diagnostic traces of failure, provide feedback to the developer who uses it to diagnose and address the problems. The design, implementation and testing are iterated until satisfactory: the choice is up to the developer's preference as described earlier for the formal analysis, except that usually at this point all the MUSTARD scenarios are already specified. Once satisfied with the functional validation, the developer can proceed to performance evaluation, putting the target service under load, executing multiple tests either sequentially or simultaneously as specified by the developer. This evaluates the service behaviour consistency as well as obtaining insights into configuration issues such as resource allocation on hosting environments, in order to meet the non-functional requirements of the services.

Although the methodology does not mandate that all phases be followed or in a strict pattern, convenient and automated support for most of the phases will encourage and motivate their application. For example, developers have the choice not to validate designs. However, design validation is simple, high-level and highly automated; the analysis contributes to service quality. It is therefore advantageous that all phases be applied, as direct benefits such as improved service quality can be achieved with limited effort. The methodology is based on the CRESS toolset and therefore its framework, including directory structure, filename conventions, and framework-specific files [104].

3.3 Development Lifecycle of Composed Web/Grid Services

This section prescribes the details of a typical development flow to create a composed web/grid service using the methodology.

3.3.1 Design

3.3.1.1 Describing Service Diagrams

From the designer's perspective, the activities in the design phase are high-level descriptions of services and service configuration. The description notation is discussed in detail in Chapter 4. Briefly, a composite web or grid service behaviour is described by a CRESS diagram, which may use other CRESS diagrams (dependent diagrams) and feature templates (modular development, and reset features), varying according to the nature of the service that is being designed and the development preference. In any composition there is definitely the involvement of at least a root diagram which is the main description of the composed service. It is typical to start development from the root diagram(s) – a natural flow of development as it is the basis for service creation. There are no mandatory steps in development scenarios involving multiple CRESS diagrams. It may be preferred to start from the the lowest level root diagrams that do not use other root diagrams as service partners (i.e. a bottom-up approach). A top-down approach may also be used, or even both simultaneously in situations when resources (number of developers) permit. Ultimately these diagrams are combined by CRESS and parsed for purposes such as syntax checking and generation of code. When feature diagrams are involved (e.g. for reset features or modular development), it may be easier if the root diagram is first described as it provides a clear guide as to what the features should modify.

The web service examples in Chapter 7, figures 7.2, 7.3, and 7.4, show three CRESS root diagrams created using the CHIVE editor, depicting a possible flow of composed web services along with their interaction with other services. Chapter 4 presents the CRESS notation; for now, these diagrams contain rule boxes, nodes, arcs, and labels. At this point, the diagrams are designs with specified dependencies between them which will be used only when translating to specified target languages.

3.3.1.2 Describing Service Configuration

The service configuration diagram for the web and grid service domain is defined with deployment options and parameters, along with the specific namespace configuration for each service in the composed services. The configuration diagram does not play a direct role in this

phase but underpins the later phases, providing details such namespaces for implementation generation, style of formalisation (e.g. repeated top-level behaviour in a LOTOS specification), specification annotation for verification, settings to generate implementation validation, etc.

Developers should therefore describe the service configuration at this point as it will be convenient in later phases. Particularly the **Deploys** options and parameters, explained in detail in Chapter 4, depend on the immediate phase that developers want to perform. Logically, though not mandated, the next phase is formal analysis: the deployment (**Deploys**) options should thus concern those pertaining to formal specification and analysis. The configuration can be adjusted to specify implementation deployment, which has options such as the target language (version of BPEL standard), code comments, and service timeout threshold.

3.3.2 Specification

Following the design of services, the abstract model of the service behaviour is created as the foundation for the design-phase analysis. This is a highly automated phase. Developers do not have to build the model from scratch, which is potentially tedious and prone to errors.

The behaviour of composed web/grid services described as CRESS diagrams are automatically and fully specified in LOTOS as the diagrams fully describe the composite behaviour. At this point, the developer would have already specified the deployment options and services in the CRESS service configuration diagram; this information is used by the tools that generate the LOTOS specification. There are two ways to achieve automated specification, the explicit and implicit approach, and they can be used in combination. Developers can explicitly execute the tools to generate the specification, often used when partner and phantom (used for auxiliary behaviour such as resources for example a database) services have not yet been specified. Analysts may use the specification directly, such as specifying behaviour details for partner services, and performing their own analysis. The implicit approach is more advanced, going straight to executing the formal analysis (implies analysis descriptions have been specified), which will automatically generate the specification prior to performing the analysis. This approach is likely to be used in iterative design wherein most of the specifications, especially those of the partner/phantom services, have been completed and analysis with minor changes to

design/behaviour are repeated until satisfactory. Both approaches share the same specification generation process, but only the second approach achieves automated analysis.

3.3.2.1 Specification of Partner Services

This step is used to include detailed behaviour for partners services that are not described as CRESS diagrams. The automated generation of the LOTOS specification of the composed service includes the default interface behaviour of these partners, thereby completing the specification to an extent that immediately supports a limited range of analysis. Detailed analysis can be performed only if detailed behaviour is specified.

Partner service and phantom partner specifications are created according to the format and file structure as described in section 5.2.1.4. These specifications are automatically included into the generated LOTOS specification of the composed service.

While developers and analysts can specify the behaviour of partners from scratch, some effort can be saved by using the default behaviour of the services that was generated. Figures 3.2 and 3.3 illustrate the two approaches to specify a partner's behaviour. The first approach uses the explicit approach described in section 5.2.1.4 to complete the detailed specification. The partner's manual specification is initially absent. The automated generation of the LOTOS specification for the composed service is executed once. The partner's default LOTOS service behaviour interface will be generated as a LOTOS process within the overall specification. The analyst adds detailed behaviour to the default interface in this file. Subsequent automated generation of the composed service will include the partner's specification from the file instead of generating the default interface. This method applies for any number of partners. The second approach is to fully specify the partner behaviour without initial specification (to obtain the interface). CRESS will detect the partner specification and include its specification instead of generating its interface behaviour, thereby achieving the overall detailed specification in the first generation.

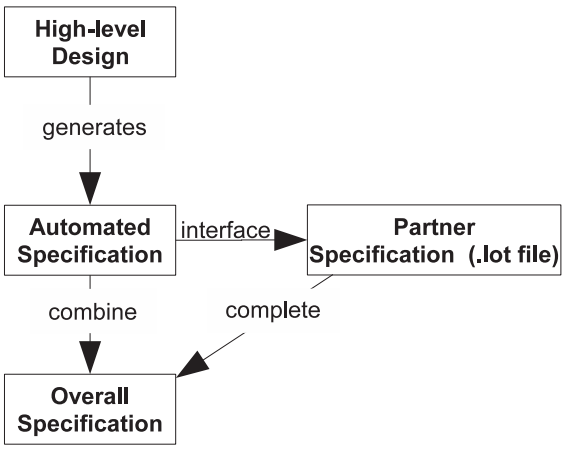

Figure 3.2: Partner Specification Using Generated Interface

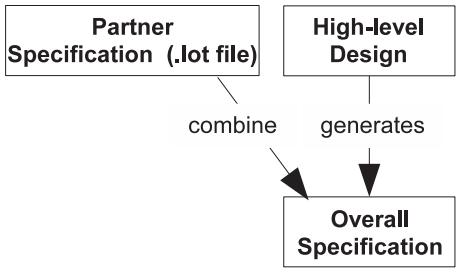

Figure 3.3: Include Partner Specification Directly

3.3.3 Formal Analysis

3.3.3.1 Formal Validation

In this phase, analysts specify high-level validation scenarios for composed and partner web/grid services using the MUSTARD notation. Each service may have its own set of validation scenarios saved as MUSTARD files according to their service name. The scenarios can be specified at any time, for example together with the CRESS diagram. Partner service MUSTARD files should reside in the same location as the composing service diagram. Validation of a partner service will yield productive results only if its behaviour has been specified, not using the automatically generated outline behaviour. If MUSTARD scenarios are not defined for a service or partner, then no validation will be attempted for this.

Analysts perform automated validation using the tools (e.g. cress_validate) within the integrated development environment. They can choose the services to be validated as it may not be always necessary to validate all services involved in the composition. The validation is carried out by the MUSTARD tool. Scenarios that do not pass will have diagnostic traces printed in

MUSTARD notation, which can be used to analyse and correct the composite service CRESS diagram and partner specifications that are manually specified.

3.3.3.2 Formal Verification

The methodology's approach to support automated verification is rather similar to automated validation. Formal verification of the composed service needs its CLOVE properties and annotated LOTOS specification. This implies that the service configuration has specified automatic generation of the annotated LOTOS specification.

In this phase, analysts specify high-level verification properties and data value enumerations for the composed service using the CLOVE notation, saved with the same service name with the CLOVE file suffix (.clove) in the same location as the CRESS service diagram. The properties can specified at any time; for example together with the CRESS diagram.

The following general guidelines may be followed for efficient verification. Verification of a composed web/grid service that has a simple LOTOS specification and small value range of data types (specified in its CLOVE description) may not require the compositional mode of verification as it may be less effective, particularly in generating the explicit LTS that is to be verified. Generally, in this verification context, a composed service that has only a few partner services, that does not have array data types, and does not have too large a range of value enumerations in its CLOVE description, is suitable for the non-compositional mode. Compositional mode is suitable otherwise, where the composite service (a CRESS diagram) comprises further composite services (e.g. other CRESS root diagrams), and may have large data value ranges for data types and/or array data structures.

Analysts can perform automated verification using the tools (e.g. cress_verify) of the integrated development environment. Automated verification will be carried out by the CLOVE tool. By default, deadlock and livelock freedom are checked automatically. A verification outcome of a property is either TRUE or FALSE, meaning that the property is respected or not. In the event of FALSE, CLOVE will initiate diagnostics by generating a pseudo-LOTOS behaviour that gives a counterexample path trace to the property that was evaluated, exhibiting a specific behaviour in the specification that is in contradiction. Such diagnostics are used to correct the composite service CRESS diagram and manual partner specifications.

3.3.4 Implementation

This is the phase where the actual composed and partner services are created and deployed, and is the ultimate objective in development. The service configuration and implementation code of owned partner services should be provided prior to invoking automated service implementation and deployment. It is recommended that the deployment option for WS-BPEL 2.0 be used as the adoption and implementation support for BPEL is moving towards this standard. The service configuration diagram must also be configured with the implementation details for the services, namely the namespaces, address, and resources if any.

Implementation for CRESS-described composed web/grid services is fully automated, with the CRESS framework supporting the automatic compilation and deployment packaging for partner services that are developed in-house. Developers provide the implementation code for a partner service in a directory of the same name in the same location as the composing CRESS diagram (see sections 6.1.1.5 and 6.1.2.5 for full details). For a web partner service, the implementation code is provided as a Java file under the same name in the partner service implementation directory. For a grid partner service, the Globus Toolkit implementation directory structure should be followed, providing the service code implementation (Java files), required libraries (e.g. JAR files), and actual deployment configuration (in the case of grid services, JNDI (Java Naming and Directory Interface) and WSDD (Web Service Deployment Descriptor) files are needed). If the implementation code is not yet provided, some effort can be saved by running the automated implementation once. This will produce a skeleton implementation that developers can build on and then run the automated implementation again to complete it.

For illustrative purposes, assume there is a composed web service A that uses an internally owned partner service B, and an externally owned partner service C. The composed service is to be implemented in the WS-BPEL 2.0 specification. Implementation of partner service B is to be provided by the developer, but has not been done yet. There is no need for a new implementation of partner service C as it is externally owned and is therefore already coded.

The implementation phase is automated by invoking the CRESS cress_expand command-line tool. The services that are specified in the **Deploys** clause of CRESS configuration diagram will be translated according to the specified target language, for example WS-BPEL 2.0. As

the above example has the situation wherein the partner service B implementation was not provided, the service cannot yet be deployed. However the skeleton code of the service will be generated by automatically translating to Java the service's WSDL, which was also generated. A developer can then build on the skeleton Java code thereby completing the implementation. Running the above command again for the second time will achieve a full implementation that can be deployed.

CRESS will do the necessary tasks involving further code generation (e.g. translating code from WSDL), compilation, and packaging into deployable archives. Implementations of composite services (CRESS diagrams) are packaged in deployable archives for ActiveBPEL as .bpr files, web partner services as .wsr archives, and grid partner services as .gar archives. Developers may choose immediate deployment to the appropriate container (.wsr and .bpr into ActiveBPEL, .gar to Globus Toolkit) as part of the process.

3.3.5 Implementation Validation

Following the implementation and deployment of (composed) web/grid services, they can then be validated. In this phase, the analyst may manually specify validation scenarios using the MINT notation, as well as a MINT configuration for each service that is to have its implementation validated. The notation and properties configuration parameters are described in section 6.3. Usually both validation scenarios and property files are automatically generated for each service target if formal validation has been applied (implying MUSTARD scenarios have been specified) and if service configuration is defined with an implementation validation option. The methodology's automated validation phase will translate the respective service MUSTARD scenarios into MINT. The MINT service configuration is automatically generated for each service from the CRESS diagram descriptions and domain service configuration during the automated implementation phase.

It is logical that services be functionally evaluated prior to measuring performance, as the latter is invalid should the former not be satisfied. Similar to formal validation, analysts perform automated implementation validation using the tools (e.g. cress_validate) within the integrated development environment. They can choose the services to be validated as it may not be always

necessary to validate all services involved in the composition. The validation will ultimately be carried out by the MINT tool.

The following is an example for illustration. Suppose a composed grid service A described in CRESS uses partner grid services B and C, and that all services have been successfully deployed with the methodology's automated implementation. MINT property files have been automatically generated for all services. Formal validation has been previously performed for all three services, implying their MUSTARD files have already been specified, and therefore implementation validation is fully automatic.

Functional validation may undergo iterations, and may involve bug fixes by making changes to CRESS diagrams, partner implementation code, and re-deployment. This is done until developers and analysts are satisfied that the services are 'doing the right thing', as the functionality have been tested. Once satisfied with the functionality of the service implementations, the developer can run the same set of tests for performance evaluation, which can help evaluate the consistency and response times of the target service. This also assists adjustment of the resource configuration. The analyst specifies for validation the mode of performance evaluation (sequential or concurrent) and the number of runs per test. Sequential mode executes test runs successively upon completion of the previous run. Sequential mode aids in evaluating behaviour consistency and the average response time under a series of successive invocations. It may provide insights into detecting resource management issues such as a resource locked but not released upon operation completion. Concurrent mode executes test runs of the same test simultaneously to aid in evaluating behaviour consistency and the average response time under simultaneous load. This provides insights into resource management and configuration to meet the scalability requirements of the application.

3.4 Evaluation

The integrated methodology developed by the thesis was realised through the integration of the tools developed, producing a thorough development lifecycle for creating composed web/grid services. Developers can graphically compose services, validate formal test scenarios, verify service properties, obtain an implementation, and perform functional and performance eval-

uation of the implementation, all within a single environment. The approach of high-level description (for design, validate, verify, and test) and the highly automated phases results in a high-level and rigorous development lifecycle for creating composed web/grid services of high quality. The CRESS framework offers extensibility to exploit features and advantages of new and other existing technologies (e.g. theorem proving, symbolic model checking, unit testing). This can strengthen and expand the methodology's capabilities to support service development and migration.

The high-level CRESS notation for describing web/grid service compositions enables developers to design language-independent services that can be automatically translated into actual representations such as LOTOS specifications and BPEL implementations. The use of the CHIVE graphical editor provides a visual environment for the design and configuration of composed and partner web/grid services, and access to other CRESS tools.

The automated formal specification significantly enables developers to focus on design analysis. This is achieved by the automatic generation of formal specifications from the CRESS diagrams. This can save time and effort that is required if done manually, thereby motivating the application of formal methods to development. The formal validation is high-level and automated, and can be applied to composed services and partner services. Complex specifications of validation scenarios are supported by the language-independent MUSTARD notation. Formal validation against a LOTOS specification is fully automated, with diagnostics in MUSTARD notation provided as information for the analyst to address issues. The formal verification is also high-level and automated. High-level properties are specified and automatically verified using the CLOVE notation and tool, which will provide counter-example traces as diagnostics for compromised properties. These can be used by the developer to improve service behaviour or design. Common specification properties are made available as templates to make the approach even simpler. CLOVE also uses CADP's compositional verification for LOTOS, which potentially speeds up the verification process by reducing unnecessary state space.

Implementation is mostly automated apart from the manually written code to be provided for partner services. The framework automatically generates the majority of code, compiles the implementations including those of partner services, resulting in composed services and partner services packaged in deployable forms for their respective hosting containers.

The methodology certainly could be improved. The following suggestions apply to individual phases but collectively contribute to the entire potential of the methodology from the perspective of development. The formal specification phase could be improved in usability, specifically in the area where the behaviour of partner services (non-CRESS) has to be manually specified in the target formal language, which in this case is LOTOS. A high-level abstract specification approach, perhaps graphically-oriented (maybe even with CRESS notation), could enable developers to easily describe the partner service behaviour. This abstract behaviour could be translated to the desired target language.

The relationship amongst the phases could be made dynamic in the sense that information/feedback produced could be proactive and propagate into related phases which pick up, analyse, and use the information to support their inner activities. For example, syntactical errors detected in a formal specification could proactively feed back to the design phase, such as highlighting specific parts of a CRESS diagram that need attention, making it easier and quicker for developers/analysts. Formal validation and verification could proactively relate diagnostic information back to the design (CRESS) description and make suggestions as to where potential issues may be found.

The manual implementation procedure for partner service implementation could be improved, especially for developers who have to rename filename implementation stubs to the service name and then build the code. This could be improved in two ways. Firstly this manual but simple step could be automated so that developers can focus on the actual code development. Secondly, considering the earlier suggestion of high-level description of partner service behaviour, this may translated to actual code. This could be inserted into the code skeletons or even directly, thereby covering more development effort consistent with the design. Implementation validation could be improved to provide more details for test runs such as a validation audit log in the form of interaction (e.g. SOAP) messages, with attached timings and traces of each step in the validation behaviour. Such information may provide more insights into the investigation of issues found in validation.

The methodology is currently symmetrical, apart from the verification aspect which is lacking in the implementation. Application of verification to the implementation could contribute to the quality of the actual products which are the ultimate target. Existing or new techniques

could be investigated (e.g. runtime verification and assertions of BPEL workflows [7]) as part of the integrated methodology. A plausible approach is exploring the potential of using CADP's interfaces to add in hooks for evaluating properties on implementations corresponding to the explicit state space. A symmetrical methodology would underpin a more rounded development lifecycle for creating composed web/grid services, especially in the consistency of rigorous analysis between design and implementation.

Chapter 4

Describing Composed Web/Grid Services

4.1 Introduction

There are many approaches to composing services, ranging from programming, through high-level orchestration languages, and to visual programming. For example, programming offers a high degree of control and capabilities, but it is tedious to code and maintain the coordination of behaviour. High-level composition languages on the other hand give specific focus to describing the functional interaction of composition units, and support the logic that constitutes the behaviour of the composed services. Visual programming enables graphical description of system behaviour using graphical notations such as UML (Unified Modeling Language [57]), SDL (Specification and Description Language [53]), Petri Nets [83], MSC (Message Sequence Chart [51]), and BPMN (Business Process Modeling Notation [14]). There is growth in the use of orchestration languages with support for graphical design, as seen from the ongoing and increasing development and adoption of service orchestration specification standards such as BPEL and implementations such as ActiveBPEL [3] and Oracle BPM [77]. Graphical interfaces and notations have been used to implement the composition of web/grid services. Most of them are bound to the underlying target specification language and deployment specification. For example, ActiveDesigner [2] is a graphical interface using the BPMN notation to specify BPEL business processes for deployment in ActiveBPEL.

The thesis work has aimed for an abstract design approach that supports automated formalisation and implementation, all integrated into one development methodology for creating composed web/grid services. A graphical approach that is independent from actual languages and is able to be formalised can support a range of target orchestration languages and associated deployment methodologies, while maintaining the same high-level service description. Similarly for formalisation, different formal representations can be obtained using the same graphical service specification and development environment.

CRESS [100] is a notation and toolset that was developed by Prof. Kenneth J. Turner for graphical specification and analysis of features and services. It was designed to be a flexible way of describing and combining services and features. The CRESS approach was assessed to be suitable as a basis for meeting the thesis goals, especially with its existing capability to compose web services in a rigorous manner, comprising support for automated specification, and formal validation of composed services. The thesis work is an extension of the original CRESS approach, and has extended it to support formal validation of partner services, verification, implementation validation and performance evaluation, which the original CRESS approach did not have.

This chapter describes the CRESS notation in general and for web services, and also its use by the CRESS framework – specifically how it underpins the abstract approach. Extensions made to the notation for the research are then discussed in detail. This is followed by a high-level discussion of the extensions made to the framework to meet the thesis objectives – their technical detail is covered in the corresponding later chapters.

4.2 The Original CRESS Notation

The CRESS notation was loosely based on the Chisel notation, which was developed by Bell-Core for describing telephony features to meet industrial needs [99]. The notation is not bound to any language or application service domain. Given the association with the context of an application domain (target framework), the notation will have the semantics of services in that domain.

The CRESS notation consists of rounded rectangles, ellipses, arcs, and comment boxes.

Rounded rectangles (rule boxes) are used for configuration, and in individual service diagrams for usage of variables, parameters, constants, macros, and diagram uses. Ellipses or nodes are for describing service activities, and are linked by arcs to describe the flow of behaviour. These CRESS graphical components generally depict the service definitions, behaviour and flow. CRESS has a general set of rules for describing and parsing a diagram syntactically. Domain-specific constructs (such as keywords and data structure syntax) are parsed according to their target context linked into the CRESS framework.

The graphical description of a service using the CRESS notation is generically known as a service diagram. CRESS supports two types of service diagrams: root and feature. A root diagram is the actual description of the service behaviour, which include activities, parameters, service flow, assignments, etc. depending on the context of the domain. A feature diagram, as illustrated in 4.1 and discussed in 4.2.2.1, is similar but describes separately the functionality which can be included into a service (a root or even a feature) diagram prior to service generation, thereby resulting in a service having additional functionality. Features can be added or withdrawn easily as they are not part of the root diagram. An example use is for POTS (Plain Old Telephony Service), where the call-forwarding feature can be added or withdrawn from the service as required without changing the service (root) diagram.

Generally, a CRESS root or feature diagram contains a rule box (see 4.2.1) and a graphical behaviour description (see 4.2.2). The description of a composed web service is typically a CRESS root diagram where the entire process behaviour is fully described, comprising definitions (data types and structures, variables, constants, inclusion of other CRESS diagrams, etc.) in a rule box, and the service logical behaviour (assignments, invocations, fault handling, replies, etc.) using nodes and arcs. Concrete examples of root diagrams are given in chapter 7. Feature diagrams may be used in a modular development of composed web services. For example, several teams could develop their designated set of ports and operations for a composed web service separately as features, for merging by CRESS tools into the root diagram. This may benefit development where different behaviour can be engaged and disengaged readily with little effort, for example to engage and evaluate the use of different behaviour descriptions of one functionality.

For each domain there is a specific CRESS graphical configuration diagram which is used

to define the deployment details for services. Service configuration for web services specifies the exact services to be generated and their deployment options, service namespaces, prefixes, and deployment addresses. The service configuration is also where features can be specified for the services that are to be generated.

CRESS was originally and successfully used to compose web services as a domain that is plugged into the CRESS extensible framework [102]. Formalisation (specification and rigorous validation) and implementation are automated for composite web services. The CRESS notation for web services supports description of data and variable definitions, usage of other composed web services described in CRESS, constants, service ports and operations, requests, service invocations, data value assignments, conditions, fault and event handling, (scoped) compensation, parallelism, loops, and termination (as described in detail in section 4.2). The CRESS notation was evaluated to be appropriate for composing grid services as well, due to their similarities to the service-oriented paradigm and their standards, potentially this also means automated formalisation and implementation. For these reasons, CRESS was adopted as the foundation of an integrated rigorous environment for creating composed web/grid services. The notation was extended by the thesis work with the aim of supporting realistic service composition issues such dynamic partners and use of existing services.

4.2.1 Rule Box

The structure of the web service rule box begins with **Uses**, followed by the definition of named (variable) structures and constants, followed by partner services usage. CRESS supports data types for web services which map straightforwardly to XSD types as listed in table 4.1. Complex data structures are specified within '{' and '}' and can be nested. Arrays are supported using '[' and ']'. For example the following syntax defines a complex type Parcel that has fields: name as **String**, address as **String**, itemlist as an array of items which are of type **String**:

{**String** name **String** address [**String** item] itemlist} parcel

Named constants can be defined as well simply by using the '*name* <- *value*' notation in a rule box, for the convenience of specification. For example *basicRate* <- *3.5* means the constant *basicRate* has 3.5 as its value. CRESS supports the inclusion of other CRESS diagrams, which

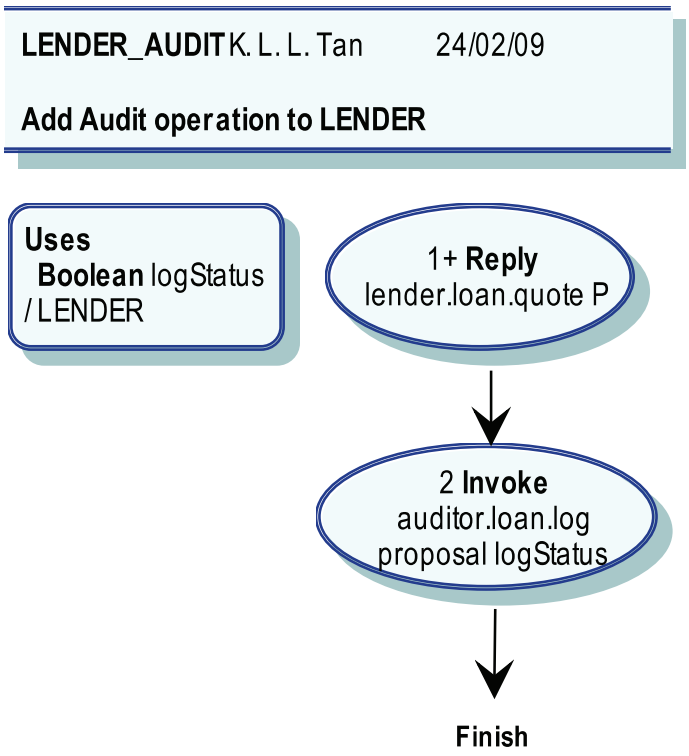

Figure 4.1: CRESS Feature Diagram Example

allows a nested form of composition, specified using '/' followed by the names of other CRESS diagrams to be included with the current web service composition. A brief but typical example is a composed web service diagram A that uses an existing composed web service B, already described in CRESS, by stating **Uses** /B in its rule box. Diagram B is then considered a partner service to A. These kind of partner services have all information automatically available and included into the composed service, such as the behaviour specification and service interfaces (WSDL). Partner services that are not described as CRESS diagrams are not specified in the **Uses** clause of a Rule Box as such information may not be readily available and may have to be inferred or manually provided by developer. The WSDL of this type of partner service can be constructed from their operation signatures invoked by the composed service. For formal specification, skeleton interface behaviour can be generated for partners based on their use in the composing CRESS service diagram as it is not known how they behave. Developers will have to manually provide the formal behaviour, replacing the skeleton one, to be included into the behaviour of the composing service to better exploit formal analysis.

Boolean	Byte	Date	DateTime
Decimal	Double	Int	Integer
Long	Natural	NegativeInteger	NonNegativeInteger
NonPositiveInteger	PositiveInteger	Short	String
Time	UnsignedByte	UnsignedInt	UnsignedLong
UnsignedShort			

Table 4.1: CRESS Supported Data Types For Web Services

4.2.2 Service Behaviour Description

The composed web service or 'business process' behaviour is described using ellipses, which are activity nodes, and (labelled) arrowed arcs depicting the process workflow and interaction with other units of the service. Nodes are uniquely numbered, with labels and parameters defining the process activities such as receiving requests, invoking services, parallelism, assignments, loops, compensation, etc. These process activities and their syntax are described in table 4.2. The usual syntax convention applies here: '?' means optional, '*' means zero or more, 'l' means alternative, '(' and ')' means grouping. Many node labels seem to follow closely the BPEL activity constructs, but they are not necessarily constrained to BPEL as the CRESS notation is abstract. The names were adopted as they are appropriate for defining process flow semantics. With BPEL as a widely adopted standard, this implicitly serves the purpose of convenience with regard to understanding the semantics.

Nodes are joined by arrowed arcs depicting the process flow from one activity to another. Arcs may be labelled with expression guards, event guards, and assignments. Expression guards are Boolean expressions specified for path choices in the composed service. An example is *loan.amount >= 10000*. Boolean operators such as && and || along with parentheses can be specified for complex boolean expressions. In general, operators do not have precedences and should be parenthesised as required. Multiple expression guards on different arcs may be specified for alternative choices of paths. If more than one expression guard is specified from the source node to the same target node, then their expressions are combined together with &&. An **Else** guard from the same source node represents the negation of all the specified guards.

Event guards (**Catch**, **CatchAll**, **Compensation**) specify the fault handling behaviour for

Activity Syntax	Description
Invoke operation output (input faults*)?	An asynchronous (one-way) invocation sends only an output to the target service operation. A synchronous (two-way) invocation exchanges an output and an input with a partner web service where the input is the result from the operation invocation. CRESS requires potential faults to be declared statically, though their occurrence is dynamic. The faults that may arise in a business process are implied by **Invoke**, **Reply** and **Throw**.
Receive operation input	Typically this is used at the start of a business process to receive a request for service. An initial **Receive** creates a new instance of the process. Each such **Receive** is matched by a **Reply** for the same operation. **Receive** also accepts an asynchronous response to an earlier one-way **Invoke**.
Reply operation output | fault	Typically this is used at the end of a business process to provide a response, representing the output to the client. Alternatively, a fault may be signalled.
Fork strictness?	This is used to introduce parallel paths; further forks may be nested to any depth. Normally, failure to complete parallel paths is expected to lead to a fault. This is strict parallelism, and may be indicated explicitly as 'strict' (the default). If this is too stringent, 'loose' may be used instead.
Join condition?	Each **Fork** is matched by **Join**. By default, only one of the parallel paths leading to **Join** must terminate successfully. However, an explicit join condition may be defined over the termination status of parallel activities. In CRESS, the expression uses the node numbers of immediately prior activities. For example, 1 && (2 || 3) means that activity 1 and either activity 2 or 3 must terminate successfully. In turn, this means that activities prior to 1, 2 and 3 must also succeed.
Throw fault	This reports a fault as an event to be caught elsewhere by a fault handler.
Compensate scope?	This is called after a fault to undo previous work. An explicit scope (CRESS node number) indicates which compensation to perform. In the absence of this, compensation handlers are called in reverse order of completion.
While condition	The while is associated with the specified condition. The **While** node should have two outgoing arcs labelled True and False. The True arc is traversed to the activities performed if the condition is satisfied, and repeats until the condition is not satifisfied - thereby following the False arc.
Terminate	Ends the process behaviour
Empty	A void node used to link others

Table 4.2: CRESS Node Activity Syntax

the associated scope that is conditional upon occurrence of the specified event. **Catch** has the syntax '**Catch** *fault*', to specify handling for faults thrown. Variations of **Catch** faults can be specified. If a fault only has a name and no value, it is handled by a **Catch** with the matching fault name only. A named fault with value will be handled by the **Catch** with matching fault name and value type that corresponds to the value, or by a **Catch** without a fault name but with the matching type. **CatchAll** handles any fault. **Catch** and **CatchAll** apply where they are defined, for example after an **Invoke** (local scope) or as a global fault handler. If a fault occurs, the current scope (e.g. at the **Invoke**) is first considered. If no matching fault handler is found, it escalates to the higher-level scope until a matching fault handler is found. The process terminates in the absence of a matching fault handler. **Compensation** defines a compensation handler to undo work (e.g. when a fault occurs), and applies where it is defined. Compensation handlers and their scopes, which comprise subsequent activities for undoing work, are only enabled when the associated activity completes successfully. Only enabled compensation handlers can be invoked, and it is by means of an explicit **Compensate** action which is specified in a node. **Compensate** can be specified with a scope which will invoke only the compensation handlers for the scope. If no scope is specified, then compensation is carried out in the reverse order of the activities that have occurred.

Assignments are for data parameter manipulation, and can be specified partially for complex data types. An assignment expression has the syntax '/variable <- expression'. Structured variable access is supported using the '.' operator. For example */msg* <- *forecast.description* specifies the assignment of the *description* field of the *forecast* variable to the *msg* variable. Assignments specified within guards mean that the assignments will be executed if the guards, expressions or events, are satisfied or have occurred. Arcs with only assignments are considered to have empty guards.

4.2.2.1 Features

A CRESS feature diagram can also use the service description constructs described above, and overall look similar graphically. A feature may define a template. The initial node of a template has a numeric label that ends with '+' (append to matching node), '-' (prefix to matching node), '=' (replace matching node). The initial node defines a single event. Any binding associated

with the start is appended to the corresponding original node. The template must end with a single **Finish** (or empty) node, though other non-empty leaf nodes are allowed. A node whose numeric label ends with '!' is not template-expanded. Template event nodes are subject to further template expansion. Figure 4.1 is an example of a simple feature created for a web service Lender that adds in the behaviour of logging to an auditing service whenever loan proposal replies are made. The inclusion of features is specified in the domain configuration diagram, which combines these with the root diagrams for service generation.

4.2.3 Service Configuration

Figure 4.2 is an example of a CRESS web service configuration diagram, where all configuration information is specified within the rounded rectangle. The first line in the configuration specifies the deployment options, which are specified as command-line switches, followed by a '/' and the names of services and features to be deployed. In this figure, the service LENDER is to be deployed. If the feature that was described in figure 4.1 is to be used in LENDER, then its name is specified. Table 4.3 describes the original switches for web service deployment. As CRESS is an abstraction for formalisation and implementation, there are options available for both in the web service domain. The services designated for translation (e.g. LOTOS and BPEL) are specified using the syntax ***/ services***.

CRESS supports the notion of 'phantom' partners. This is to provide an approach in the formalisation to describe additional aspects of components which are shared by partners, for example a database behind the scenes that is shared between two or more services. This is specified by a -m <phantom> option, which generates a LOTOS process for the phantom partner whose gate is synchronised with all the services thereby establishing a means of communication for interaction by the services if required.

The lower rectangle specifies implementation-related information for all services, including partners. Each configuration is space delimited in the order of: service name, namespace prefix, namespace, and deployment URL. The service name is a (case-insensitive) lookup identifier for the services during translation. The namespace prefix and namespace are for the generation of BPEL and WSDL files, which will contain relevant imports and references to files, data types,

Option	Usage
-a	annotation of LOTOS specification
-b mode	BPEL specification version (1 - BPEL4WS, 2 WS-BPEL 2.0)
-c	generate comments in code
-e	error report level (3 – internal errors, 2 – also with user errors, 1 (default) – also with informative notes, 0 Ű- also with diagnostics)
-m partners	merge partners as a comma-separated list; this is needed only if one partner is shared by several business processes, the merged processes being extracted to the top level of the specification
-n	number of top-level call instances (*formalisation*, default 3)
-o seconds	generate of configuration for implementation validation, with service timeout specified in seconds
-r	repeat behaviour (*formalisation*, default is stop at a leaf node)
-l	levels of code shown by indenting (default is no level indenting)

Table 4.3: CRESS Deployment Options

partner links, bindings, etc. The deployment URL defines the base address of the deployed service, which will be made public via the WSDL.

```
Deploys -b 2 -c -n 1 -o 5 -r /  LENDER

APPROVER    app     um:FirstRate      localhost:8080/active-bpel
ASSESSOR    ass     um:RiskTaker      localhost:8080/active-bpel
BROKER      brok    um:CarMen         localhost:8080/active-bpel
DEALER1     deal1   um:BigDeal        localhost:8080/active-bpel
DEALER2     deal2   um:WheelerDealer  localhost:8080/active-bpel
LENDER      lend    um:LoanStar       localhost:8080/active-bpel
SUPPLIER    supp    um:DoubleQuote    localhost:8080/active-bpel
```

Figure 4.2: CRESS Web Service Domain Configuration Diagram

4.2.4 CRESS Diagram Editors

The CRESS framework defines the graphical notation but does not mandate any particular visual editing tool for describing CRESS diagrams, giving freedom of choice for editors. This is achieved with an extensible approach having the requirement that the editors support formats

that CRESS can parse into an intermediate internal representation for use by the CRESS tools, illustrated in figure 4.3. Currently CRESS supports service diagrams prepared by the following editors: yEd, Diagram!, and CHIVE (CRESS Home-grown Interactive Visual Editor).

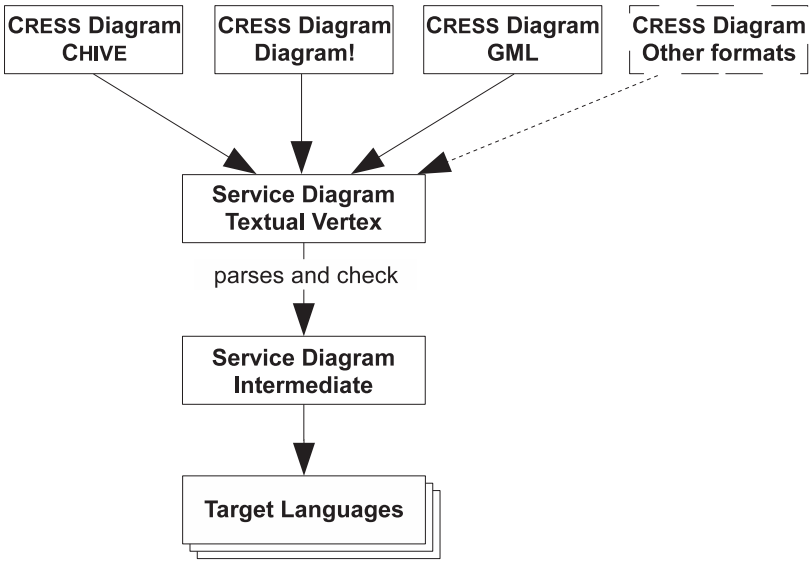

Figure 4.3: CRESS Diagram Translation Approach

yEd is a graph editor developed by yWorks as a Java application that supports the popular text-based and portable graph format GML (Graph Modelling Language) that CRESS supports and therefore parses. Diagram! is a drawing tool from Lighthouse Design that runs under NextStep/OpenStep systems. CHIVE is a Java-based visual editor specifically developed for CRESS, It is the best alternative amongst the editors as it is well integrated with the CRESS tools. It provides an integrated visual environment for development of services in the various CRESS-supported application domains.

4.2.4.1 CHIVE Graphical Editor

CHIVE is a Java application that is the recommended visual editor and environment for creating composed web/grid services. Figure 4.4 is a sample CHIVE window, opened with a service diagram describing a composed web service. CHIVE supports the CRESS notation and has been applied across all the CRESS supported domains including web services and, in this thesis, grid services as well. Conventional graphical editing capabilities are available to conveniently design CRESS diagrams, such as: cut; copy; paste; undo; fonts; snapping arcs to nodes; arc labelling; grid lines; and arc curve editing. CHIVE provides a visual interface for the CRESS

tools, constituting an integrated visual development environment for the CRESS-supported application domains. Domains, target languages, and tool options are configured in CHIVE via its File -> Preference menu which sets the context for the service diagram that is being edited, shown in figure 4.5. These settings have explicit influence on the CRESS tools; depending on the configured application domain, target language, and tool options, invoking these supported actions through CHIVE will achieve the appropriate actions in the corresponding CRESS tools for the specified context.

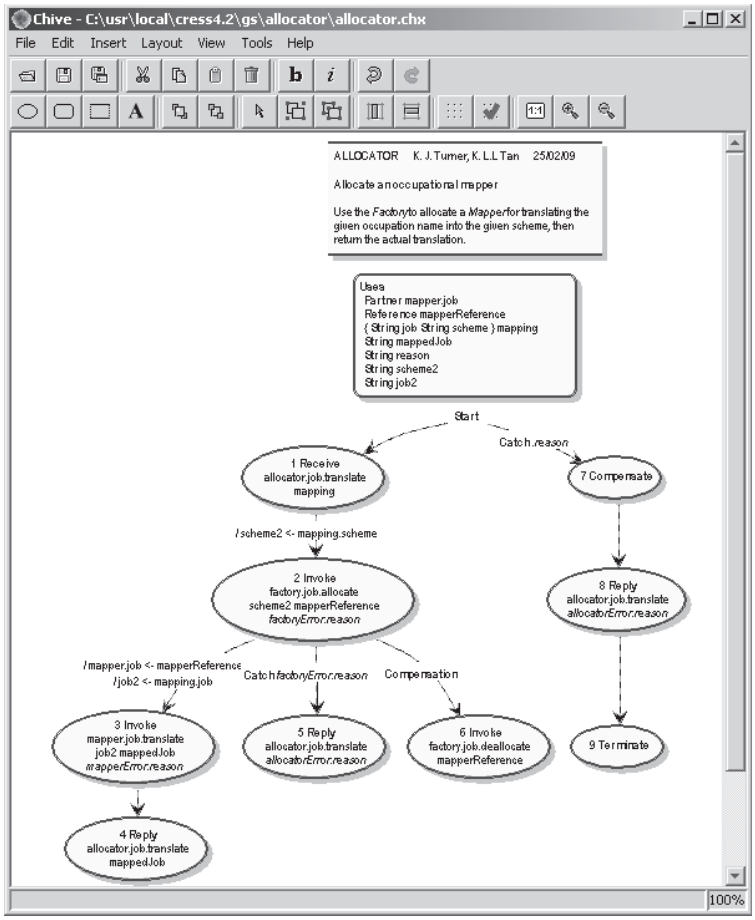

Figure 4.4: CHIVE Window

Originally, CHIVE's interfaces for CRESS tools were only Check and Validate in its Tools menu, which correspond to invocation of the CRESS tools cress_check and cress_validate that were described in table 2.3. Results from these command-line tools are displayed in the visual environment of CHIVE. These meet part of the thesis goal of an integrated methodology for composing web/grid services. The thesis work extended CHIVE's existing interface to support the CRESS notation extended by the thesis, and new work in automated verification and

implementation validation in fulfilment of the methodology's new design-phase analysis and post-deployment testing. The CHIVE manual and installation guide are found in [103].

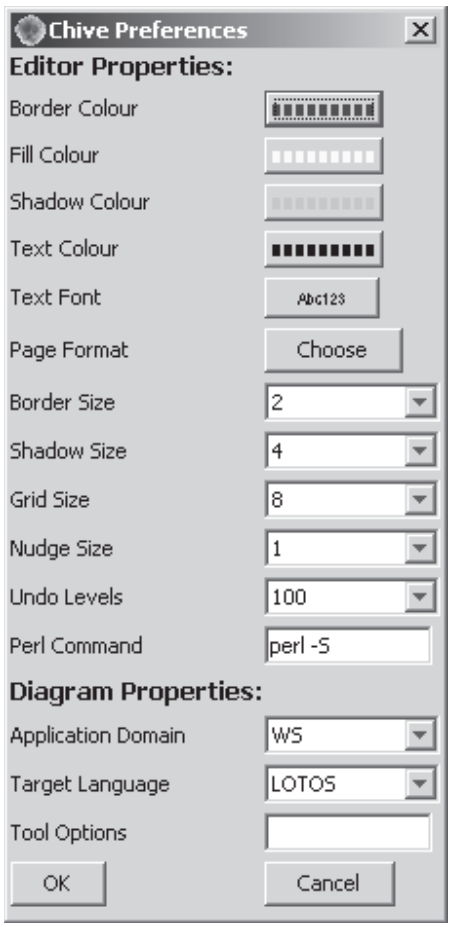

Figure 4.5: CHIVE Preferences Domain Configuration (web service)

4.2.5 Checking Service Description

CRESS diagrams can be syntactically checked within the context of their respective domains, using cress_check or via the CHIVE editor. For example, a web service (WS domain) CRESS diagram will be checked with the rules, keywords and context of web service composition such as supported base data types (e.g. String), complex data type syntax, service activities (e.g. **Invoke**), and service flow (e.g. compensation, and error handling). For web services the checking of the diagram can detect syntactic errors such as duplicated numbered nodes, invalid activity keywords, used of reserved names, invalid operation signatures, invalid partner declaration, and incorrect assignment syntax. The errors are displayed as text which the developer will use to correct the diagrams.

4.3 Extensions To The CRESS Notation

While the CRESS notation and methodology were suitable for the thesis research, it required several extensions and developments to realise the goals set out, namely: grid service support, dynamic partner services, bottom-up approach of composition using existing services, and supporting the new WS-BPEL 2.0 standard. The thesis work developed these extensions which collectively support realistic web/grid compositions as well as a choice of implementation standard. These extensions are also supported by the formalisation and implementation aspects of the thesis methodology.

CRESS originally supported web services and not grid services. Preliminary investigations demonstrated that the CRESS notation for web services could be potentially and conveniently applied to grid services, with support for automated formalisation (LOTOS) and implementation (BPEL4WS) [91, 106]. The investigations showed that, whilst the BPEL4WS orchestration of grid services developed in Globus Toolkit 4 was possible, the implementation of composed services was awkward as workarounds were needed due to the standards used by underlying technologies differing in version, especially the WS-Addressing standard. It was not possible to invoke dynamic WS-Resources using endpoint references, which *ideally* could be done by setting the endpoints in BPEL dynamically bound partner links. The workaround without introducing any form of software extension was to have the endpoints as explicit operation parameters. The target services could then manually infer and marshall the actual WS-Resources for invocation, which was awkward. This was not a letdown but rather showed the potential of BPEL for grid services, despite BPEL4WS being developed only for web services. The thesis work anticipated that the WS-BPEL 2.0 release and the harmonisation effort with grid service standards would address the standards problem and enable more seamless orchestration of grid services. Indeed it was so when the WS-BPEL 2.0 specification was officially released and the Globus Toolkit 4 was further developed using standards compatible with WS-BPEL 2.0.

Dynamic partner services may be involved in the composition of services, i.e. partner web services may be dynamically bound at runtime. This activity is often seen of grid resources (WS-Resources) also. The original CRESS notation had not previously supported this. This was a worthy and important composition capability, so the thesis work extended the CRESS notation

to define, bind and invoke dynamic partners. As an example, a grid resource may be created when required for a specific purpose and then destroyed when no longer required. Therefore the endpoints to the WS-Resources are dynamic, and the composed service can dynamically set their endpoints for use. As a web example, a composite service can choose from a set of available partners that implements the same service interface depending on certain criteria (e.g. supplier service that offers the same functionality at a cheaper cost or faster performance).

CRESS originally adopted a top-down approach to create composed services, whereby partner services were also newly developed. Realistically it is very often the case that new composite services are created using already deployed existing services as partners. Using existing services implies their definitions, especially data types, messages, and namespaces. The thesis work extended the CRESS notation to allow specification of type ownership, where data structures may have explicit definitions of service owners and will consequently use the owners' namespaces.

4.3.1 Service Diagrams

4.3.1.1 CRESS Notation For Dynamic Partners

There are two main aspects of this capability: their definition and their binding. Firstly, there has to be a means of expressing a partner service port as dynamically bound. Secondly, there has to be a way to express the binding, which is required prior to invocation of dynamic partners. The extension introduces two new keyword types that can be defined in a rule box: **Partner** and **Reference**. **Partner** is the keyword for defining 'partner.port' as dynamic. For example **Partner** *weather.forecast* designates that the *forecast* port of partner service *weather* is dynamic. **Reference** is the keyword for defining endpoint-typed variables which then can be used as the source in assignments to express binding of dynamic partners where *partner.port* is the target.

Naturally a designated dynamic partner is initially unbound. Therefore prior to the first invocation by the composed service, the dynamic partner is bound to a target endpoint. This implies the source of the assignment naturally has to represent the reference service endpoint. Pragmatically this is how services are bound at runtime in implementations such as BPEL,

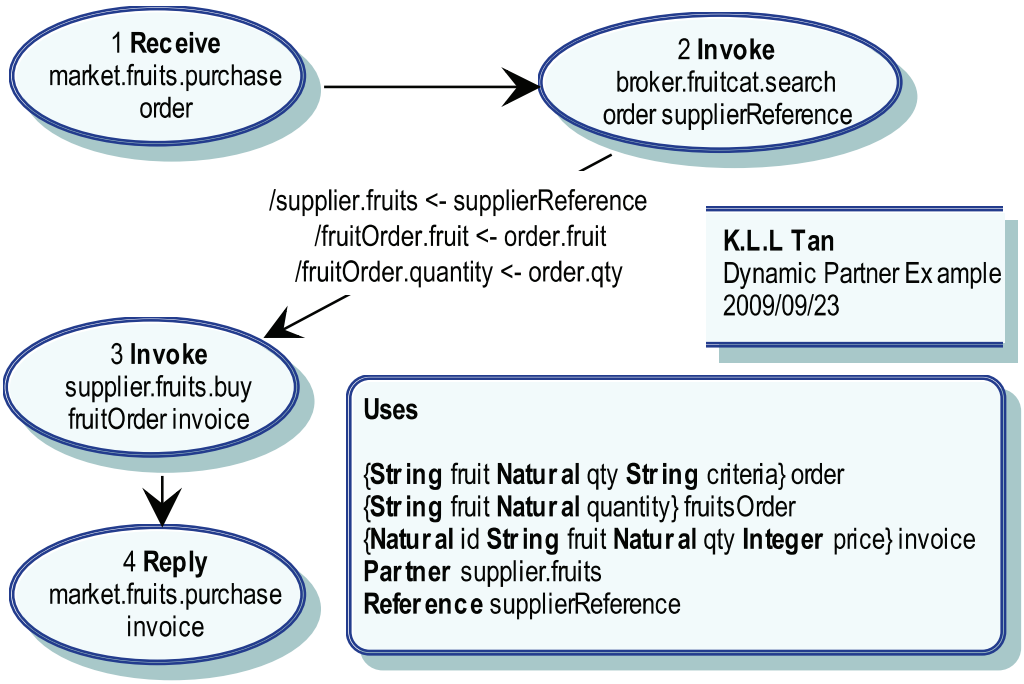

Figure 4.6: Dynamic Partner Example

where the partner link is assigned the value of type EndpointReference (which may be a value created and returned by a service). For example, a brokerage service might return the endpoint to the service that best fits the criteria for the composed service. The composed service will bind its designated dynamic partner to an endpoint prior to invocation and hence establish the interaction dynamically. The original CRESS notation had no type definition for service endpoint, and therefore had no means to specify such bindings. The CRESS notation was extended to support such a notion via assignments to **Partner** types as the target. Invocation of dynamic partners is the same as invoking a statically defined service partner.

Figure 4.6 shows a simple example of a CRESS service diagram specified to define, bind and invoke dynamic partners. It describes a simple composed service (market) that uses a broker service for finding a supplier that best matches the customer's criteria. The composite market service then uses the supplier recommended by the broker, and proceeds with the order on behalf of the customer. The partner service *supplier* with its *fruits* port is the dynamically bound port. The expression '**Partner** *supplier.fruits*' within the rule box describes this dynamic nature. The *broker* service returns the *supplier* endpoint in node 2 as *supplierReference*, which is then assigned to *supplier.fruits* in its outgoing arc, thereby specifying the binding of the

dynamic partner. The invocation of the specified supplier follows as normal.

4.3.1.2 Type Ownership

Existing services are often part of a newly composed service, which is a very pragmatic and common development flow in service-oriented computing. This implies the data structures and messages of the existing web/grid services as they are the core elements of the contractual agreement to consume services. The CRESS notation was extended with type ownership to support such descriptions where data structures defined in a rule box can be specified explicitly with partner service names indicating ownership. The notation for data definition is extended to *name[:owner]* where *owner* is the name of the service partner. If no owner is specified then the data structure ownership defaults to the current service diagram.

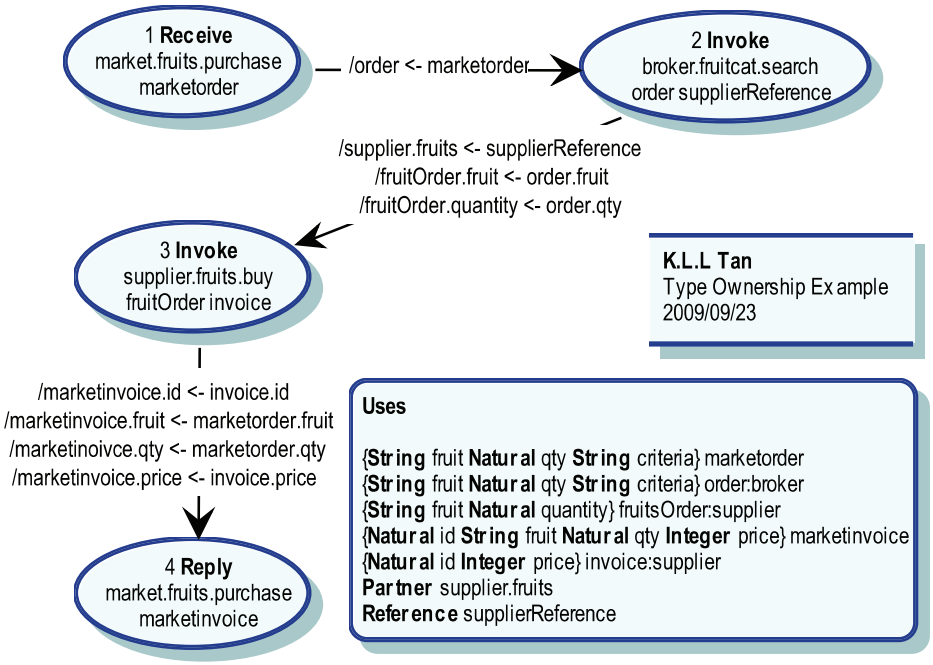

Figure 4.7: Type Ownership Example

The CRESS diagram in figure 4.7, modified from figure 4.6, illustrates that the broker and supplier are already existing services which are then involved in the newly composed market service. This implies that the broker and supplier services have their service interfaces already defined prior to the development of market service. This means that the market service has to comply with both partner service definitions in order to consume their services successfully. The rule box data definitions for *order* and *fruitOrder* are specified to be owned by *broker*

and *supplier* respectively. However, the market service would like not to expose its business dealings with the companies that developed broker and supplier service for corporate reasons. In this diagram, the market service defines an identical structure to *order* named *marketorder* but has ownership by market to represent its service inputs rather than using those of broker. The market service therefore 'hides' away the view of broker, particularly its namespace and data types, from its service consumers as a trade secret. The identical structure allows direct assignment of the entire complex structure easily, as seen in the assignment arc between node 1 and 2. It is for a similar reason that *invoice* of supplier is hidden, with *marketinvoice* also containing additional information constructed using values from other data as the return data structure of *market.fruits.purchase* operation. This type ownership notion is simple in terms of notation description, but its effects will be seen in Chapter 6 in the automated implementation with regard to WSDL/BPEL generation, where the namespaces of the data schema definitions reflect ownership explicitly.

4.3.2 Service Configuration

Additional options were added to the service configuration diagrams. These new CRESS options are general switches of CRESS tools which were developed to support the goals of the thesis, specifically automated verification and implementation support for WS-BPEL 2.0, applicable to both web and grid services. Table 4.4 describes the new options and their use. The automated verification aspect of the thesis uses CADP as the verification technology. This requires pragma-annotated LOTOS specifications. The thesis uses a Perl script cadp_annotate which will be described in detail in Chapter 5 to automated the annotation.

CRESS was extended to support the grid service domain, which is largely similar to the web service domain. Grid services or WS-Resources may have state information held in data structures that can be optionally queried through the WSRF-defined WSDL interfaces. The service configuration for grid services now allows definition of resource data structures for state and query capabilities. Data structures defined in such a manner are regarded as the same as normal data types. In addition they will be configured as resource properties of the services. Resource data types are defined the last column after the location URL of the associated service.

Option	Purpose
-a	Automated annotation of LOTOS specification. Potentially applicable to support formalisation for other CRESS domains that use LOTOS as target specification language
-b <version>	BPEL translation target. 1 for BPEL4WS, 2 for WS-BPEL 2.0
-o <timeout>	Timeout setting in number of seconds. This is used for the implementation testing (see section 6.3.2.1), where a timeout is imposed upon the invocation of a service operation if the response is expected to be within the threshold. Default no timeout.

Table 4.4: CRESS Service Configuration

4.4 Extensions To CRESS Framework

The thesis work has made new developments to extend the CRESS framework which corresponds to the requirements described earlier in figure 2.3. These new developments involve the extension of existing CRESS tools, and development of new tools which were added to the CRESS framework. Their technical details are presented according to their respective impact area: formalisation and analysis (validation and verification) in Chapter 5, implementation and analysis (validation) in Chapter 6. A new application domain was added in CRESS for creating composed grid services. The grid service domain reused and adapted the web service application domain due to similarities in paradigm and standards. As the CRESS notation was extended to support realistic service compositions, the existing CRESS tools were also extended (especially cress_vocab.pm, cress_parser.pm, cress_check, cress_lotos.pm, and cress_bpel.pm) to reflect the new developments. cress_vocab.pm and cress_parser.pm were extended to support the new keywords (**Partner** and **Reference**) and label constructs (type ownership and dynamic partner assignment). cress_check was extended to support the checking of these new notations, allowing **Partner** and **Reference** types, etc. cress_lotos and cress_lotos.pm were extended to support the new extensions to the notation in the automated generation of LOTOS specification code. cress_bpel and cress_bpel.pm were extended for to support the new extensions for the automated generation of implementation code, and in addition were significantly developed to support the WS-BPEL 2.0 standard as a target for automated implementation.

The support for formal verification was added into the framework via the new CRESS tool cress_verify that was developed during the thesis work. cress_verify integrates the verifica-

tion language and tool CLOVE (CRESS Language Oriented Verification Environment), which was developed by the thesis work as a high-level approach to automate formal verification of composed services, independent from the underlying verification techniques. cress_validate was extended to support the capability of validating partner services, and integrated with the thesis newly developed tool MINT (MUSTARD Interpreter) which automates validation of an implementation.

4.5 Evaluation

Composed web and grid services can be created using CRESS using a notation that provides a high-level abstract graphical description. The notation underpinned by the tools framework has resulted in a rigorous and highly automated development methodology comprising high-level design, formalisation, design-time analysis, implementation, and testing.

The original CRESS notation for composing web services supported the main constructs commonly used in service development. During the thesis work, the notation for web services was adapted to the grid service domain, and therefore to the development of composed grid services. The CRESS notation was extended to support dynamic partners and type ownership for describing composed web/grid services, extending the notation for realistic compositions. Service configuration was extended with new options to support automated formalisation and implementation of grid services. Grid partner services can employ resource data structures through service configuration. The extended notation therefore supports typical description of composed web/grid services and partner services, along with the capability to define resource data structures.

The CRESS notation could certainly be improved. The following are explicit and pragmatic suggestions as future work on various aspects of the notation. The first suggestion is in security which is an aspect that is potentially crucial and also critical in service-oriented computing. This enables owners to protect their service resources by security mechanisms. Therefore it is possible that a composed service might involved a partner service that implements security for authorised access, and that the composed service itself requires security. The CRESS notation for service description and/or configuration may be improved with the capability to describe

security mechanisms. The second suggestion for improvement is directed at supporting the collection of WSRF specifications as first class citizen for the grid service domain. A plausible approach would be the support of ports and operations as keywords in the notation. For example 'wsrp' as a port keyword could explicitly indicate that the associated grid service implements the ResourceProperties (WSDL specification) port and operations. The third suggestion is directed at the composed service itself: to have the notation support correlation which may be required in the composed service. Lastly, the web and grid service domain could be merged as one. This would be a realistic move due to their converging definitions and specifications, and that a combination of web and grid services may be involved in a composed service. Certainly these improvements in the approach should also be reflected in the later phases of the methodology.

Chapter 5

Formalising Composed Web/Grid Services

5.1 Introduction

The next stage following the description of composed web/grid services is design-time analysis. Applying design-time analysis in the development lifecycle of composed web/grid services has the advantage of early error detection, thereby reducing the cost to remedy problems at later stages especially after implementation. Formalising services can achieve this effect, whereby their behaviour can be rigorously analysed prior to actual implementation.

Conventionally the stages in formalisation involve specifying behaviour, defining the required analysis, and performing and evaluating the analysis. Specification creates an abstract model to describe the behaviour of the system at the desired level of detail. A high-level model can be analysed quickly. Analyses can be more detailed as finer behaviour is captured in the model. Validation and verification are key techniques frequently applied in the analysis of a formal model. Validation uses scenario simulation which can analyse behaviour on specific test cases. Verification can check general properties exhibited by the behaviour such as deadlock freedom, liveness, and safety from undesired actions. Verification can also check specific properties. Validation can be completed very quickly as scenarios are usually small and finite parts of the behaviour. Verification is usually used to check the entire behaviour for properties and usually takes much longer to complete. Their purposes and analysis orientation are com-

plementary. For example an exhaustive set of validation scenarios may be equivalent to full finite state verification. However it is tedious to specify the exhaustive set of scenarios, and each scenario is simulated separately, so there is a need to consolidate their results back into the same evaluation context. Some analyses such as deadlock freedom are simple to evaluate through verification but are more difficult with validation. Conversely, validation may be applied to an infinite state space if the available verification techniques cannot be used. Therefore exploiting validation and verification together is thorough and beneficial to the development. Validation and verification can be performed in either order as they are independent from each other. Both forms of analysis can produce feedback that identifies problematic behaviour that analysts can use to correct the model, therefore reducing the possibility of errors and the costs of correction at later stages. These stages are iterated till the user is satisfied and confident of the service quality. The task of formalisation may require a very significant amount of time and effort especially in cases where engineers are not well-educated in the techniques [1, 75].

This thesis has developed an automated, integrated approach to formalisation of web/grid services, which have practical benefits for both formal methods trained and untrained developers. Formal specification is automatically obtained, which enables both types of developers to arrive at the actual intent of analysis immediately. This approach supports abstraction and automation of the underlying techniques, thereby enabling service developers (who are usually not formalists [1]) to perform analysis. The approach has combined design-time analysis support with service implementation, resulting in a single integrated environment with a systematic development lifecycle.

This chapter discusses how the automated formalisation of composite web/grid services was achieved, in particular: the strategy extensions, the extended capabilities for formal validation, and entirely new work on formal verification. These were developed by the author to underpin the automated formalisation aspect of the integrated methodology.

5.2 Automatic Formalisation

Obtaining the formal model is the basis of formal analysis. This also applies to composite web/grid service behaviour with regard to formalisation. As the model can be readily created,

the effort on design-time analysis is reduced. LOTOS (Language of Temporal Ordering Specification) is an ISO standard specification language designed for specifying formal behaviour of distributed processes, and so is appropriate for web/grid services [101]. The Full LOTOS language [50] can specify behaviour as well as data types and values, which can model and analyse web/grid service compositions and data. Tools that support formal validation and verification techniques for LOTOS are available and are still being maintained by their developers, thus assuring tool stability. For these reasons LOTOS was evaluated to be well suited for formalising web/grid service compositions and was therefore used as the foundation for the formal aspects of this thesis.

The work of CRESS prior to this thesis demonstrated the automatic formalisation of composed web services [101] whereby their descriptions as CRESS diagrams are translated into LOTOS specifications. The following LOTOS code illustrates the translated high-level LOTOS specification structure for a web/grid service.

```
Specification NAME [SERVICE] : Exit(States)

Library                                          (* library *)
...
Behaviour
  SERVICE [SERVICE]                              (* call main process *)

Where
                                                 (* local definitions *)
definition of types such as events, operations, ports, user data types

Process                                          (* partner processes *)
...
Process SERVICE [service] : Exit(States) :=      (* main service *)
  Hide ... In                                    (* hide internal gates *)
    (
      interleaved partners
    )
  |[partner gates]|                              (* synchronised with partners *)
    SERVICE_1 [gates]                            (* call main service process *)

Where                                            (* local definitions *)

local definitions of main service process behaviour
...
```

CRESS predefines a set of LOTOS library data types which are automatically included during translation for the specification of composed web services. These formal data types are used to abstract the data primitives, and also to support the formal specification of complex

data types defined in CRESS diagram descriptions in a rule box. CRESS also generates the LOTOS high-level behaviour describing a composed web/grid service. This is viewed as the only service by the external world, hiding away (**Hide** ... **In**) all partner services and their gates from the clients of the composed service. The behaviour of all partner services are interleaved, and synchronise at their respective gates with the composed service behaviour which is the main behaviour, resulting in the entire LOTOS specification of the composed service.

Formal analysis can then be applied to the LOTOS specification using high-level analysis descriptions that can be easily defined, with analysis being automated. Automated validation of a composed web service was the only form of formal analysis supported by the original CRESS approach. There was no support for validation of partner services and no support for verification. Grid service analysis was also rudimentary for the same reasons. The automated LOTOS translation also needed to incorporate the requirement for realistic composition added during the thesis work. The thesis work used the original CRESS automated formalisation approach as the basis, and added new developments to realise the following objectives:

- formalisation of composed grid services
- extending automated validation to composed web/grid services and their partner services
- supporting automated verification of composed service behaviour
- integrating these design-time analysis features into the development environment for creating composed web/grid services.

Composite web and grid services can involve dynamic partners, especially in grid computing. Composed services usually interact with existing partners, implying the use of existing data type definitions. The thesis work has extended the CRESS notation to support dynamic partners and type ownership when composing web/grid services, requiring the corresponding extensions for automated formal specification. This section discuss how automatic formalisation is achieved for composed web/grid services, also dealing with the new extensions. The automated formalisation approach of CRESS prior to the thesis work is presented first, followed by the extended work developed by the thesis.

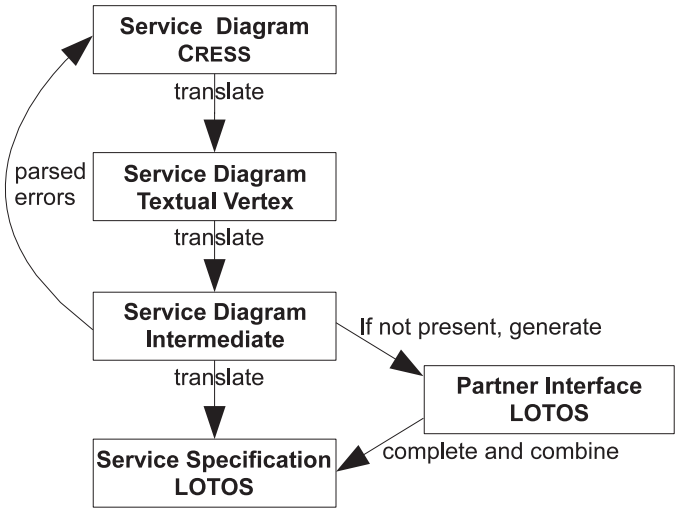

Figure 5.1: CRESS Translation Strategy

5.2.1 Original Translation Strategy

CRESS had an existing strategy developed by Prof. Kenneth J. Turner (figure 5.1) to generate LOTOS specifications that model composed web services [101]. The LOTOS specification includes abstract data type definitions and process behaviour, underpinned by predefined library data types. The CRESS-to-LOTOS translation tool parses the CRESS diagram into an internal intermediate textual format. The information obtained from the parsing comprises data type definitions, partner usage, ports, and operations. These determine the resultant LOTOS specification framework, overall behaviour, and data types. Parsing of nodes and their associated (labelled) arcs produces the detailed behavioural part of the LOTOS specification such as assignments, invocation, event handling, loops, and parallelism. Static semantic errors are also detected as part of the process, and fed back to the developer for correction. If a service partner is a root CRESS diagram, the behaviour will be fully specified by CRESS. If a service partner is not a CRESS diagram (i.e. is external), then its formal behaviour will be included if it has been specified in LOTOS. Otherwise a basic interface will be generated by CRESS for the service partner specification.

5.2.1.1 Data Type Translation

There is a strategy for translating into abstract data types the variables declared in the rule box of a CRESS diagram. CRESS has a library of abstract data types defined for the data primitives it

CRESS Data Primitive	LOTOS Data Type Name
Boolean	Boolean
Char	Char
Double	Number
Float	Number
Int	Number
Integer	Number
Long	Number
Natural	Natural
NegativeInteger	Number
NonNegativeInteger	Number
NonPositiveInteger	Number
Partner	Natural
PositiveInteger	Number
Short	Number
String	Text
UnsignedByte	Number
UnsignedInt	Number
UnsignedLong	Number
UnsignedShort	Number
arrays	Array (instantiated definition) and LimitedArray (with limited capacity)

Table 5.1: CRESS Data Types

supports. The strategy also supports generalised type definitions for complex structures (arrays and records). Table 5.1 shows the abstract data type names that correspond to the supported primitive types intended for the composed web/grid service domain. Many of the abstract data types are used in other domains such as IN (Intelligent Networks), but are not covered in the thesis here. This explains the differences in the names of Integer and String with their associated abstract data type name, as more harmonised names were chosen. This does not affect the formal model in any way. All these formal types are defined in the 'stir' (Stirling) LOTOS library specification in a form easily included by LOTOS analysis tools such as Lola.

Records (which may be nested) are structures defined through the combination of primitive types. CRESS has a simple strategy for automatically describing record types. A LOTOS type is created with the name of the record. This also applies recursively for inner record definitions. Each record element has "accessor" (get/set) operations defined. Each record has a constructor

operation with an identical name to the type for the purpose of creating its data value. A comparator operation is also defined for the record, which recursively uses the comparator operations of its elements. These operations naturally model a complex type implementation as they are also generated in a similar ('Java bean-like') fashion from their XSD complex types. Array types have 'getIndex' and 'setIndex' operations to access array elements via a numerical index.

Below is an example of a record definition Proposal type comprising **String** name, **String** address, and **Integer** amount. This proposal type is defined in CRESS as '{**String** name **String** address **Integer** amount} proposal'. The automated translation is as follows.

```
Type Proposal Is BaseTypes                          (* proposal record *)
     Sorts Proposal                                 (* proposal sort *)
     Opns                                           (* proposal operations *)
       AnyProposal: -> Proposal
       _eq_,_ne_: Proposal,Proposal -> Bool
       proposal: Text,Text,Number -> Proposal
       getName: Proposal -> Text
       setName: Proposal,Text -> Proposal
       getAddress: Proposal -> Text
       setAddress: Proposal,Text -> Proposal
       getAmount: Proposal -> Number
       setAmount: Proposal,Number -> Proposal
     Eqns                                           (* equations *)
     ForAll                                         (* equation variables *)
      proposalA,proposalB: Proposal,
      nameA,nameB: Text,
      addressA,addressB: Text,
      amountA,amountB: Number

      OfSort Bool
       proposal(nameA,addressA,amountA) eq
       proposal(nameB,addressB,amountB) =
        (nameA eq nameB) and
        (addressA eq addressB) and
        (amountA eq amountB);
       proposalA ne proposalB = not(proposalA eq proposalB);

      OfSort Text
       getName(Proposal(nameA,addressA,amountA)) = nameA;

      OfSort Text
       getAddress(Proposal(nameA,addressA,amountA)) = addressA;

      OfSort Number
       getAmount(Proposal(nameA,addressA,amountA)) = amountA;

      OfSort Proposal
       AnyProposal = Proposal(AnyText,AnyText,AnyNumber);
```

```
        setName(Proposal(nameB,addressB,amountB),nameA) =
         Proposal(nameA,addressB,amountB);
        setAddress(Proposal(nameB,addressB,amountB),addressA) =
         Proposal(nameB,addressA,amountB);
        setAmount(Proposal(nameB,addressB,amountB),amountA) =
         Proposal(nameB,addressB,amountA);
EndType                                          (* end Proposal *)
```

5.2.1.2 Partner, Port and Operation Translation

Actual interactions, for example invocations with web/grid service partners, are in the form of message exchanges at specific communication endpoints. These communications are specified using gate synchronisation with event offers and value passing. For every service, a gate with the identical name is used in the LOTOS specification as an abstraction of the service endpoint where communications will take place.

There is no explicit definition of the ports and operations required when composing with CRESS. Their use directly implies the ports and operations. In the parsing of a CRESS diagram, all the ports and operations are recorded and translated respectively as LOTOS data values of the Port and Operation types, which are types specifically generated for the web/grid services domain. These data values are used in the behavioural part of the LOTOS specification, as described in section 5.2.1.3.

As an example, suppose there are **Invoke** nodes of two partner services providing language translations: goodenglish and fluentchinese. For simplicity, both define the 'language' port and 'translate' operation. LOTOS gates named 'goodenglish' and 'fluentchinese' are LOTOS gates. CRESS creates a LOTOS type named Port that defines the values of the port(s) used, and a type Operations representing the operation values, listed in the following LOTOS data specification. These values will be used in the LOTOS behaviour expression to model the interaction with the specific service, port and operation.

```
Type Port Is                    (* port name *)
 Sorts Port                     (* port name sort *)
 Opns                           (* port name operations *)
  ...
  language: -> Port             (* port name constants *)
  ....
EndType                         (* end Port *)

Type Operation Is               (* operation name *)
```

```
Sorts Operation              (* operation name sort *)
Opns                         (* operation name operations *)
 translate: -> Operation     (* operation name constants *)
EndType                      (* end Operation *)
```

5.2.1.3 Behavioural Translation

The CRESS description of the process behaviour is denoted by the semantics of the flow across (labelled) arcs and nodes. Generally a node in CRESS approximates to one LOTOS process with the name *SERVICE_NODE* where 'SERVICE' is the composed service name, and 'NODE' is the node number.

A port and an operation value are used as value offers gate synchronisation with a service gate. This normally specifies a request or response for the partner's specified port and operation, where the actual operation parameter data is appended as a negotiation offer or match. Note that LOTOS event synchronisation is essentially directionless.

The LOTOS translation of the major constructs are illustrated in Appendix A. A **Receive** is translated as value passing in LOTOS *partner !port !operation ?var:type*, where it represents the reception of the value into a variable 'var' of 'type' for the partner's port and operation. Multiple **Receive** nodes originating from a **Start** node are translated as choices. An **Invoke** is translated as value offer of the data to the target partner using *partner !port !operation !value*. If the **Invoke** is synchronous, meaning a blocking return, it is immediately followed by a value offer like a **Receive** where the returned result of the invocation is captured by *var*. This also applies to fault responses. A **Reply** by the composed service is translated as an event offer.

The order of LOTOS events follow the flow of (labelled) arcs and nodes depicted in the CRESS diagram. An activity sequence in a CRESS diagram becomes a sequential composition in LOTOS. Parts of a CRESS diagram may be translated as separate LOTOS processes when part of a diagram is reached by different paths or is invoked as an event handler [101]. The LOTOS processes can exit with state true or false to specify the outcome of a BPEL activity as these processes have a state (*States* exit parameter) for successful completion or failure. These states are also used in concurrency and event handling, when they carry a state.

Guarded arcs are translated as LOTOS guarded choices. Assignments are specified using the LOTOS **Let** keyword, binding variables of the specified names. Guarded assignments are

specified within the scope of the associated guarded choice.

A **Fork** is represented using the LOTOS interleave operator '|||' with the behaviour expressions of the **Fork** branches as operands. A **Join** condition is specified using Boolean operators, combining the status of each branch that is returned during its exit, as found in the States parameter of a LOTOS process exit. This join expression is translated as a pair of LOTOS guarded choices, with the negated form indicating a join failure.

CRESS generates an event dispatcher LOTOS process to deal with event handlers and their scopes. The event dispatcher maintains a list of parameters describing the state of the service, variables, event, faults and messages, and current scope (associated node activity). For example, if the event is Compensate and the scope is 2 then the appropriate compensation behaviour is invoked. If a fault handler does not exist for the current scope, then the global handler will be tried. Faults are handled in the following order as in BPEL: **Catch** with a matching fault name, **Catch** with a matching fault name and type, **Catch** with a matching fault type, **CatchAll**. As faults mean unsuccessful termination, event handlers always exit with a False status.

The event dispatcher LOTOS process may be invoked by **Compensate** or faults, where information about the scope, fault name and fault value type are provided. In BPEL the fault name and fault type are used to distinguish the appropriate fault handler in a variety of **Catch** statements. For example there might be a two **Catch** statements, both with the same name but dealing with different types. This is captured in the LOTOS specification by matching the correct fault handler using the fault name and fault value type, where the latter (a pre-defined CRESS *Value* type) corresponds to the fault value type name.

Compensation handling is more complex. In BPEL, an activity is considered "compensatable" if it terminates successfully, for example an **Invoke** returning instead of faulting. Compensation handlers are stacked into the States parameter according to the order of activity completion, carrying the compensation scope and parameter information in the flow between LOTOS processes. When compensation is called, the stack is popped and the scope is obtained. A **Compensate** action for a given scope invokes the event dispatcher. This searches the stored states for a matching compensation state; if found, the corresponding handler for the state is invoked. If not found, or the **Compensate** had not specified a scope, then by default all the compensation handlers are invoked in reverse order of activity completion, thus ensuring

roll-back.

All CRESS diagrams are automatically and fully specified with this translation strategy. Following the service-oriented paradigm, both composite and non-composite services can be service partners. Service partners described in CRESS are of course compositions and their formal behaviour can be included into any higher-level compositions specified in CRESS.

5.2.1.4 Partner Service And Phantom Partner Specification

For non-CRESS specified services, their behaviour is not known except for their interface, i.e. operation signatures. CRESS provides a framework with automated support to specify partner service behaviour. There are two manners in which their formal model are specified for the overall composite specification: default interface behaviour generation, and inclusion of manually specified behaviour. If a manual specification of the service partner behaviour exists, then this specification will be included as part of the composite behaviour. Its behaviour is defined as a LOTOS process with the process name and gate using the name of the service partner.

If the manual specification does not exist, then an outline interface behaviour is automatically generated within the overall specification of the composite service. This readily supports a basic form of formal analysis. Behavioural details can be handcrafted upon this interface behaviour, contributing to a more solid model and thorough analysis as more information is captured. This also saves some effort from specifying the behaviour from scratch. The outline behaviour is inferred from the composing CRESS diagram where the service is used, consolidating its ports, operations, inputs, outputs, and faults to formulate the default behaviour. Suppose a partner service C was used in part of the composing CRESS diagram in node(s) describing: '**Invoke** c.example.cal calPar result calfault.error', where calPar is a user-defined type CalPar, result is an **Integer**, error is a **String**. This means the invocation of the 'cal' operation with input type CalPar either returns an Integer result or a fault with String value. The default behaviour is generated as the following LOTOS code. Generally the possible paths of the behaviour inferred from this analysis are specified.

```
Process C [c] : Exit(States) :=
   c !example !cal ?calPar:CalPar;
```

```
  (
    c !example !cal !AnyNumber;
    C [c]
  []
    c !example !cal !CalFault !AnyText;
    C [c]
  )
EndProc
```

5.2.2 Extended Translation Strategy

The thesis work extended the translation strategy to support the specification of the thesis extensions to the CRESS notation. The specific extensions were the **Reference** type (endpoint reference), **Partner** type, dynamic partner assignment, dynamic partner invocation, and type ownership.

5.2.2.1 Dynamic Partners and Endpoints

A CRESS **Reference** type declares an endpoint variable. In a web service implementation, an EPR (endpint reference) contains a variety information representing a unique binding to a service (resource) with which there is communication. The information usually has a service location (e.g. URL) and possibly a resource identifier in the case of WS-Resource. An EPR is abstracted using the **Natural** type in LOTOS. The rationale for using **Natural** is its existing simple definition and it can uniquely represent a binding of the actual EPR, even though in practice it may use a combination of types and values. **Reference** variables are carried throughout the specification behaviour as a **Natural** typed value in LOTOS process parameters.

A CRESS **Partner** type declares a service partner and its port to be dynamic (defined at run time). It is dynamically assigned a binding through use of a **Reference** in a dynamic partner assignment. This corresponds to the BPEL use of partner links where it is necessary to define the partner role and a port pair as a partner link type in the WSDL. As a **Partner** is analogous to a **Reference** it can also be represented using the **Natural** type. An internal variable with the naming convention<*partnerPort*>EPR of type **Natural** is created as a placeholder for storing the currently assigned value of the dynamic partner and port. The CRESS assignment expression '*/partner.port* <- *referenceVar*' assigns the value of referenceVar to partnerPortEPR using the LOTOS **Let** partnerPortEPR:Nat = referenceVar.

Invoking a dynamic service partner requires a LOTOS behaviour expression that synchronises with the behaviour of the partner process uniquely identified by the EPR. As an example, there may be several potential stock price forecast services that implement identical service interfaces, and can be invoked at runtime by a composite brokerage service. As different entities, they have unique endpoints. In the LOTOS translation, a synchronisation with the endpoint is made prior to invocation. In this example, after the dynamic partner EPR is set, its invocation uses the following two LOTOS events which first synchronises only with the stock service check port that is identified by the stockCheckEPR, then the invocation.

```
stock !check !stockCheckEPR;
stock !check !forecast !stockName;
```

The strategy for generating interface behaviour for partners is also extended to capture this notion. If a **Partner** is declared, then the interface behaviour generated will have an extra event specifying the synchronisation of the dynamic endpoint prior to its operation invocation events.

5.2.2.2 Type Ownership

Type ownership does not have a separate LOTOS translation as the original strategy for data structures can be used. Identically typed data structures but with distinguished owners are flattened into a single type specification by the CRESS translator. Assignments are then straightforward in LOTOS, even for the complex types. This is reasonable as the implementation (BPEL specification) implements this same capability whereby assignments between data types of identical structures with different namespaces are possible, resulting in a more compact specification as there is no need to express an assignment for each individual element for the target.

5.2.3 Automatic Specification

There are two ways to automatically create specifications: explicit and implicit. The explicit approach gives control – developers and analysts may want to obtain just the specification, study it, and then apply their analysis, or to follow activities that formal methods experts can engage in with their expertise. The explicit approach is used to realise a complete formal behaviour for rigorous analysis, through generating a specification in particular for partner

services where outline behaviour is automatically generated and adding handcrafted behaviour. This generation is performed by executing cress_expand tool.

The implicit approach gives immediate focus to analysis where obtaining the specification is an automated and intermediate procedure, allowing analysts to focus on validation scenarios and verification properties to be automatically evaluated. Both approaches can be used in combination within the thesis methodology. The implicit approach is initiated by performing the methodology's automated formal analysis directly. This implies that the generation of the specification is an automated and intermediate procedure in the automated validation (section 5.3.3.3) and verification (section 5.4.4). This approach is used when a CRESS description is determined by developers/analysts as mature enough to be analysed, sometimes in a progressive and iterative way.

5.3 Rigorous Validation using MUSTARD

5.3.1 Original MUSTARD Overview

MUSTARD was originally developed by Prof. Kenneth J. Turner. MUSTARD is a domain and language-independent notation and tool for specifying and validating test scenarios. Its approach allows high-level use on actual validating techniques. Interpreting test results in the high-level MUSTARD notation avoids having to be familiar with the target languages and technologies. The notation is success/failure assertion-oriented with a concise and rich syntax for complex expressions such as parallelism, deterministic and non-deterministic choices. MUSTARD has three validation outcome definitions: Pass, Inconclusive, and Fail. Pass means that all aspects of the validation have passed. Inconclusive means there is a mixture of successful and unsuccessful paths. Fail means that the validation is completely unsuccessful. MUSTARD has been used to validate CRESS descriptions in several domains including web services [101]. MUSTARD has achieved this through an abstract notation, and through a tool interface to the underlying validation tools. The abstract notation gives a direct focus to test scenario specification. This is translated to the designated target language where the technical requirements are automatically fulfilled. The automation interacts with the target underlying validation technology on behalf of analysts. Execution directives are coordinated, and results are interpreted

MUSTARD	Description
% text	Explanatory comment
decide(*behaviour*)	Non-deterministic (scenario-decided) choice of alternative behaviours
interleave(*behaviour*)	Concurrent execution of behaviours
offer(*behaviour*)	Deterministic (system-decided) choice of alternative behaviours
read(*partner.port.op* [,faultname],*parameters*)	Read from the partner's port and operation the the expected parameter value with optional specification of faultname.
refuse(*behaviour*)	Sequential behaviour with abrupt termination if the final behaviour occurs, or successful termination if not
send(*partner.port.op*, *parameters*)	Invoke the partner's port and operation with the parameter values
sequence(*behaviour*)	Sequential behaviour with abrupt termination
succeed(*behaviour*)	Sequential behaviour with successful termination
test(*name*,*behaviour*)	Defines a test for the given name and behaviour

Table 5.2: Original MUSTARD Notation

into a high-level presentation. Diagnostics are automatically collected through this process and presented back in MUSTARD notation. MUSTARD uses Lola as the underlying tool for validating LOTOS specifications, abstracting much technical knowledge of the validation technique and so appealing to non-specialists. These characteristics and approach suits well to the thesis objective of achieving automated formal validation for composed web/grid services and partner services.

5.3.1.1 Original MUSTARD Notation

The subset of the core MUSTARD notation relevant to web/grid services is covered in table 5.2. A scenario is specified using the **test** construct where the behaviour is identified by the test name. The root of the behaviour is usually a **succeed** or **refuse** construct that asserts scenario acceptance or rejection respectively. MUSTARD has **send** and **read** primitives specifying the actual interaction taking place in the form of signal output and input respectively from the point of view of the environment. For web and grid services, these corresponds to service request and response. Compact and expressive combinators (**sequence**, **decide/offer**, **interleave**) support the specification of complex scenarios involving sequence, (non-deterministic) choice, and parallelism. These combinators can be nested within the *behaviour* parameters.

5.3.1.2 Lola Validation Technique

Lola's validation technique is to synchronise test behaviour specified as a LOTOS process with the entire behaviour, thereby executing a simulation of the constrained LOTOS specification guided by the test behaviour. Lola uses a special gate (normally 'OK') to assert a successful outcome of a test. A simple example is used to illustrate this technique. We assume a LOTOS specification whose behaviour is either a then b, or c then d:

a;b [] c;d

To specify a validation scenario that the sequence of event a then b is successful, a LOTOS test process needs to be defined in the specification, for example:

```
Process Validation_A_B [a,b,OK] : NoExit :=
 a;
 b;
 OK;
 Stop
EndProc
```

To perform the validation in Lola, commands are manually executed in the Lola environment to specify the process designated as the test. Simulation is then carried by Lola, whereby the test process is synchronised with the main behaviour. In this simple example, if the behaviour of Validation_A_B reaches OK, then this indicates a successful path – OK is the key event Lola uses to denote success. This scenario is possible as the behaviour is able to synchronise with the event sequence specified by the scenario. Suppose the scenario specified 'a;c;OK;stop', then the test would fail as the behaviour can only synchronise on event 'a' and cannot proceed. Complicated validation may involve choices, non-determinism, and concurrency. All outcomes of the validation will be displayed by Lola, reporting the number of paths tried, the number of successes, the number of stops. The Lola interactive interface has options for printing path details, which are useful especially when contradictory scenarios are found. However achieving these requires knowledge of LOTOS as well as all the tools underpinning the analysis, and not just Lola. This is rather impractical considering that service developers are usually implementation-inclined; even formally-trained personnel may think twice about the manual effort involved.

Abstracting and automating these procedures is a more appealing and pragmatic approach.

Validation should be more accessible and convenient for developers, allowing them to focus directly on scenario functionality, hiding the formal techniques and automating the validation. Abstracting away the technical knowledge is a practical move as development teams may not have access to formal methods personnel and knowledge, but design-time analysis is still conveniently possible through automation and easy-to-specify definition of tests. Automation has a direct and motivating contribution to formal analysis but also supports abstraction in a way that hides the underlying techniques from the user.

5.3.2 Extended MUSTARD

The original MUSTARD was not adequate for the purposes of the thesis as application to composed grid services and also to web/grid partner services was required. There was also no support for dynamic partners. The thesis work extended MUSTARD to meet these requirements. The LOTOS translation for composed grid services is very similar to web services, so the same MUSTARD validation approach is applicable. The strategy allows inclusion of a partner service as a LOTOS process, either user-defined or automatically generated. The test techniques of Lola also support validation of specified processes, and not just the entire behaviour, therefore the validation of partner services can be readily achieved using the same approach as composed services. The original MUSTARD notation had no support for dynamic partner validation, where the endpoints are only known during execution. The notation was extended with the capability to set endpoint variables and use them in invocations. This specific extension was generalised to support test variables at no additional effort, and yet offering the capability to specify general and dynamic validation scenarios. It is possible to specify general validation criteria such as: a return result should be of type String regardless of the value, which is suitable to describe non-deterministic and more generalised scenarios in a compact manner. In addition, the MUSTARD tool is extended to call for implementation validation using MINT, which is discussed in section 6.3. Extensions to the MUSTARD notation are listed in table 5.3.

MUSTARD	Description
read(partner.port.op/*epr* [, *fault*], *parameter*)	Same as original MUSTARD **read**, with extra option to to specify the read from an endpoint EPR [121] (variable of type Reference)
send(partner.port.op/*epr*, *parameter*)	Same as original MUSTARD **send**, with extra option to specify the invoke to an endpoint EPR [121] (variable of type Reference)
?type	Specify a type for parameter, with arbitrary values
?varname:type	Specify a variable of type for parameter, with arbitrary values
!varname	Specify the value use of a variable for parameter
[StructImpl/]Struct(...)	Specifies complex data values (Struct). The structure actual implementation name can be optionally prefixed with / to specify the actual data type for implementation validation.

Table 5.3: Extensions to MUSTARD Notation

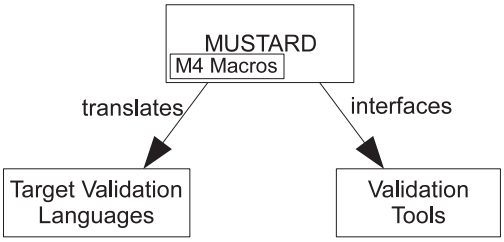

Figure 5.2: MUSTARD Architecture

5.3.3 Tool Design Overview

MUSTARD has a simple but extensible architecture illustrated in figure 5.2. The tool is developed in Perl, and the notation is based on the M4 macro language [58], which was found very suitable to support translations between languages [98]. The tool is the user interface and coordinates the validation process automatically. Perl is used as it is portable, easy to develop and execute. M4 macro "modules" are developed for the purpose of translating the high-level MUSTARD notation into the target validation syntax (e.g. LOTOS test scenarios for Lola) which will be used in the formal validation.

5.3.3.1 MUSTARD Translation Strategy to LOTOS

Table 5.4 shows the high-level MUSTARD constructs and their corresponding translated LOTOS structure and semantics.

MUSTARD	Translated LOTOS
decide(*behaviour*, ...)	(**I**; *behaviour* [] **I**; ...)
interleave(*behaviour*, ...)	(*behaviour* |[OK]| ...)
offer(*behaviour*, ...)	(*behaviour*; **Exit** [] ...)
read(partner.port.op, *value*) (see type translations below) **read**(partner.port.op, *fault*, *value*) **read**(partner.port.op/*epr*, *value*) (*epr* is Reference variable)	partner !port !op !*value*; partner !port !op !fault !*value*; partner !port !*epr*; partner !port !op !*value*;
refuse(*behaviour*)	(*translated behaviour*; **Stop** [] **I**; OK; **Stop**)
send(partner.port.op, *value*) **send**(partner.port.op/*epr*, *value*)	partner !port !op !*value* partner !port !*epr*; partner !port !op !*value*;
succeed(*behaviour*)	*behaviour*; OK; **Stop** *OR if behaviour contains Exit then* (*behaviour*) $\gg$ OK; **Stop**
test(test_name, *behaviour*)	**Process** SERVICE_test_name [service,OK] : **NoExit** := *behaviour* **EndProc** (* SERVICE_test_name *)
?type	?type:type
?varname:type	?varname:type
!varname	!varname
'string value	t(s)∼t∼r∼ii∼n∼g∼^∼v∼a∼l∼u∼e *ii is used as i is* LOTOS *internal event, ^ represents space*
7	7 **of** Nat
-7.	Number(-,t(7),<>) *Number comprises of sign character,* *textual (t(n)) whole and fraction, <> represents empty*
1234.5	Number(+,t(1)∼2∼3∼4,t(5))
StructImpl/Struct(...)	Struct(...)

Table 5.4: MUSTARD Translation Strategy for LOTOS

5.3.3.2 Diagnostic Support

MUSTARD provides path traces that leads to failure points. As an example, Simple_Result is a straightforward MUSTARD scenario, describing success if the invocation to maths.calc.add with the complex Pars object returns the value 7. The LOTOS behaviour is as simple, which specifies that if these two events synchronise, then OK (Lola success event) will be offered to indicate to Lola that this is a successful path.

```
test(Simple_Result,
 succeed(
  send(maths.calc.add, Pars(4, 3)),
  read(maths.calc.add, 7)))
```

translates into

```
Process MATHS_ Simple_ Result [lender,maths,OK] : NoExit :=
   maths !calc !add !pars(4 Of Nat,3 Of Nat);
   maths !calc !add !7 Of Nat;
   OK;
   Stop
EndProc (* MATHS_ Simple_ Result *)
```

Suppose the above scenario cannot go beyond the first action (meaning that the read cannot be performed), then this validation fails as it cannot reach the OK event which Lola uses to assert a pass. The diagnostic, which may vary in the completion time, will be printed as follows:

```
Test MATHS Simple Result ...                Fail    0 succ   1 fail   1.3 secs
send(maths.calc.add, Pars(4, 3))
<failure point>
```

5.3.3.3 Tool Integration

The MUSTARD tool was integrated with CRESS as part of the thesis work, so that validation can be performed using the cress_validate tool. This implies that graphical editor CHIVE can be used also to validate diagrams directly. There are therefore three ways to initiate the automated validation. Figure 5.3 shows the integration of the tools with respect to their programmatic flow of control, which also illustrates the level of user interface and automation. A box with dashes denotes a task unit of the tool. A box with a bold outline represents a tool that the user interfaces with. The arcs represent tool interactions. The straight lines represent the outputs from the originating box.

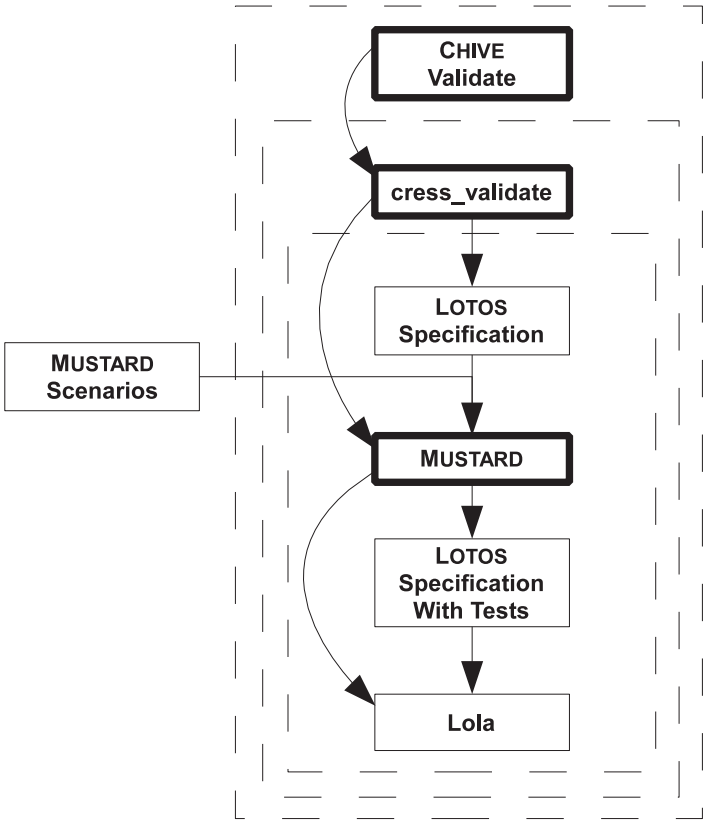

Figure 5.3: Levels of Automated Validation Procedure

Analysts can start the automated formal validation with: validation via CHIVE, cress_validate command-line tool, or using the MUSTARD tool directly. The first two methods are identical functionally but activated by different means. Validation can be executed via the CHIVE visual development environment, which invokes the cress_validate tool with parameters and tool options configured in the editor's preferences. Invoking validation by cress_validate will generate the LOTOS specification for the services prior to performing validation (which is actually carried out by MUSTARD). Finally, MUSTARD can be invoked directly at the command-line. This has the same effect, but omitting the generation of a LOTOS specification, implying that the specification is already present. Validation results are returned along the chain of command to where the validation was initiated.

5.3.4 Examples

Double_Booking_Refusal is a MUSTARD refusal scenario that a double booking of the same badminton courts should not happen. The first invocation receives a booking confirmation with

the price 9.5. If the second booking of the court regardless of its value (?Booking) happens, then this indicates a refusal. The LOTOS code has the last event as a choice with the event negotiation 'sports !badminton !book ?booking:Booking' not followed by the OK event, and the alternative as an internal event 'I' leading to OK event. If 'sports !badminton !book ?booking:Booking' is able to synchronise, the outcome will be Inconclusive as there is a failure path and a success path (the internal event will always proceed). Otherwise, the outcome is Pass if all the paths (only one here) are successful.

```
test(Double_Booking_Refusal,
 refuse(
  send(sports.badminton.book, 'CourtA),
  read(sports.badminton.book, Booking('CourtA,9.5)),
  send(sports.badminton.book, 'CourtA),
  read(sports.badminton.book, ?Booking)))
```

translates into

```
Process SPORTS_ Double_ Booking_ Refusal [lender,sports,OK] : NoExit :=
   sports !badminton !book !t(C)∼o∼u∼r∼t∼A;
   sports !badminton !book !booking(t(C)∼o∼u∼r∼t∼A,Number(+,t(9),t(5)));
   sports !badminton !book !t(C)∼o∼u∼r∼t∼A;
   (
      sports !badminton !book ?booking:Booking;
      Stop
   []
      I;
      OK;
      Stop
   )
EndProc (* SPORTS_ Double_ Booking_ Refusal *)
```

Either_Greeting describes a scenario that has a deterministic choice of responses, where any of the defined responses is valid. In its translated LOTOS, if either of these two events can synchronise, then the behaviour exits and thereby enables (≫ operator) the OK event.

```
test(Either_Greeting,
 succeed(
  send(chatter.chat.login, 'John),
  offer(
   read(chatter.chat.login, Greeting('Hi John)),
   read(chatter.chat.login, Greeting('Good Day)))))
```

translates into

```
Process CHATTER_ Either_ Greeting [lender,chatter,OK] : NoExit :=
   chatter !chat !login !t(J)∼o∼h∼n;
   (
      chatter !chat !login !greeting(t(H)∼ii∼ ^ ∼J∼o∼h∼n);
      Exit
```

```
  []
    chatter !chat !login !greeting(t(G)~o~o~d~ ^ ~D~a~y);
    Exit
  )
≫
  OK;
  Stop
EndProc (* CHATTER_Either_Greeting *)
```

Class_Parallel describes a success scenario with concurrency that has two simple invoke-response sequences. Each sequence is simply to validate the return code for a job. These two sequences are interleaved in LOTOS (**Exit** is implicitly synchronised in |[OK]|). When these two sequences complete and can exit, then the OK event can be enabled and thereby be reached in this case.

```
test(Class_Parallel,
 succeed(
  interleave(
   sequence(
    send(classify.job.code, Job('Cab Driver)),
    read(classify.job.code, '8214)),
   sequence(
    send(classify.job.code, Job('Private Detective)),
    read(classify.job.code, '9241)))))
```

translates into

```
Process CLASSIFY_Class_Parallel [lender,classify,OK] : NoExit :=
  (
    classify !job !code !job(t(C)~a~b~ ^ ~D~r~ii~v~e~r);
    classify !job !code !t(8)~2~1~4;
    Exit
  |[OK]|
    classify !job !code !job(t(P)~r~ii~v~a~t~e~ ^ ~D~e~t~e~c~t~ii~v~e);
    classify !job !code !t(9)~2~4~1;
    Exit
  )
≫
  OK;
  Stop
EndProc (* CLASSIFY_Class_Parallel *)
```

Dynamic_Resource scenario is a demonstration of using dynamic endpoints as well as endpoint variables. This scenario describes grid factory service resource creation that returns an endpoint to the resource, which is stored as a variable named ‘resourceEPR’. The variable is used to interact with the instance service which will work on the resource associated. In the LOTOS specification, the event has a prior synchronisation with the endpoint which models

in abstraction the establishment of communication to the service or resource, then the actual interaction (invocation or reply) happens.

```
test(Dynamic_Resource,
 succeed(
  send(factory.create.createResource, CreateResource('SOC2000)),
  read(factory.create.createResource, ?resourceEPR:Reference),
  send(instance.job.translate/resourceEPR, Job('Nurse)),
  read(instance.job.translate/resourceEPR, '3211)))
```

translates into

```
Process FACTORY_ Dynamic_ Resource [lender,factory,instance,OK] : NoExit :=
   factory !create !createResource !createResource(t(S)∼O∼C∼2∼0∼0∼0);
   factory !create !createResource ?resourceEPR:Nat;
   instance !job !resourceEPR;
   instance !job !translate !job(t(N)∼u∼r∼s∼e);
   instance !job !resourceEPR;
   instance !job !translate !t(3)∼2∼1∼1;
   OK;
   Stop
EndProc (* FACTORY_ Dynamic_ Resource *)
```

Variable_Use demonstrates the use of variables in a more flexible way which is suitable in some non-deterministic cases, such as in this example. This scenario validates a car rental service, where the rent operation returns a RentConfirm complex data structure (suppose it has been defined in CRESS diagram) that contains reference number (refNo) which is an arbitrary value. The cancel operation requires the reference number which can be obtained from the RentConfirm value returned. This is specified by using a named variable (rentConf) of type RentConfirm, then using the access to its refNo in the cancel operation. The LOTOS specification of this test reflects this identically.

```
test(Variable_Use,
 succeed(
  send(car.loan.rent, Rental('BMW 320,'3Days)),
  read(car.loan.rent, ?rentConf:RentConfirm),
  send(car.loan.cancel, !rentConf.refNo),
  read(car.loan.cancel, 'Rental Cancelled)))
```

translates into

```
Process CAR_ Variable_ Use [lender,car,OK] : NoExit :=
   car !loan !rent !rental(t(B)∼M∼W∼ ^ ∼3∼2∼0,t(3)∼D∼a∼y∼s);
   car !loan !rent ?rentConf:RentConfirm;
   car !loan !cancel !getRefNo(rentConf);
   car !loan !cancel !t(R)∼e∼n∼t∼a∼l∼ ^ ∼C∼a∼n∼c∼e∼l∼l∼e∼d;
   OK;
   Stop
EndProc (* CAR_ Variable_ Use *)
```

5.4 Formal Verification using CLOVE

CADP is well-known tool for the verification of LOTOS specifications. CADP implements enumerative verification (also known as explicit state verification) to enable automatic detection of errors in complex systems. CADP has been widely used in the design of hardware architecture and protocols [34]. CADP (Construction and Analysis of Distributed Processes) comprises many well-developed tools supporting a range of analyses, especially CAESAR, CAESAR.ADT, BCG (Binary Coded Graph) tools, PROJECTOR, EVALUATOR, and SVL (Script Verification Language), which were used in the thesis work. CAESAR and CAESAR.ADT are used to generate the explicit state space of the behaviour (using BCG format). The BCG tools provide command-like interfaces to manipulate the graphs, such as obtaining label information and state space reduction. PROJECTOR computes abstractions of behaviour which are applied in compositional verification. EVALUATOR [66] checks the state space to evaluate properties specified in μ-calculus and to generate diagnostics. Verification scripts in SVL notation [35] define a batch of verification procedures; this is very handy as it can access all the CADP tools in one file with a convenient syntax. The CADP tool suite is still being maintained by the development team and therefore has assurance in its stability and sustainability. For these reasons, CADP was the chosen as the tool for the formal verification work reported in this thesis.

Even using SVL scripts, operational and conceptual knowledge is required in order to use the tools of CADP. Preparation usually has to be made prior to the actual verification of properties, and can be the bulk of effort in the entire verification work. CADP uses C to implement and execute specifications. Typical groundwork involves annotating the LOTOS specification, possibly implementing data types (in the C language according to CAESAR.ADT specification), and also constraining their value ranges as finite if required. These steps are not required for LOTOS specifications without data types, but this is rarely the case as data is often important and is hence often specified – especially in web/grid services. These tools may be difficult, particularly if developers do not have formal expertise [111].

Technical abstraction and automation of verification is the thesis approach to motivate, automate, and simplify the verification process from the developer's perspective for creating composed web/grid services. Technical interfaces to CADP tools and languages are abstracted and

automated as much as possible. Annotation of the CRESS-specified LOTOS is automated. The thesis work has developed the CLOVE notation and tool to support verification of property specifications using automated verification. Through CLOVE, high-level verification properties can be specified independent of actual languages and tools. CLOVE properties are translated to the desired target languages to exploit the advantages of the supporting tools. Common verification properties are available as parameterised templates for quick specification. At present, CLOVE properties are translated into μ-calculus for verification in CADP, but the generic framework can be potentially applied to other temporal logic languages. CLOVE supports the high-level specification of data ranges and automatically translates them into a C implementation specified by CAESAR.ADT, therefore minimising the technical and possibly intensive effort. Finally CLOVE automates the verification process where the explicit state space is automatically generated, evaluation of properties is carried out, followed by diagnostics if any properties do not hold. Various reduction techniques supported by CADP for generated models are automatically available as CLOVE options. CLOVE also supports automated compositional verification for CRESS-translated LOTOS specifications, which is a useful technique to minimise state space explosion.

5.4.1 CLOVE Notation

5.4.1.1 CLOVE Property Verification Syntax

The CLOVE property verification syntax draws inspiration largely from the temporal verification language semantics of RAFMC (Regular Alternation-Free μ-Calculus), but defining as an abstract syntax intended for non-expert use. CLOVE can also be translated into other (temporal) verification languages. RAFMC has a range of operators to construct action, regular and state formulae as part of the verification property. The CLOVE notation adapts and abstracts from RAFMC its boolean operators, modal logic operators, and part of the action/regular/state formulae. Table 5.5 list the CLOVE constructs.

A CLOVE property is specified using the **property** keyword labelled with a user-selected identifier for the verification property. As will be seen in the syntax, a CLOVE verification expression is oriented towards temporal logic, having modal logic operators, sequences, and

CLOVE	Description
property(*name*, *behaviour*)	Define a property with given name and behaviour
forall(*behaviour*)	The necessity modal operator that specifies all transition sequences starting at the state satisfy the defined behaviour that is the property
signal(partner.port.op [,*fault*], *value*)	Definition of a single action behaviour
or(*behaviour*)	Binary alternative in the behaviour
and(*behaviour*)	Binary combination of the behaviour
choice_any(*behaviour*)	Shorthand for multiple alternatives
any_signal	Match any one signal behaviour
sequence(*behaviour*)	Defines a sequential behaviour
exists(*behaviour*)	The possibility modal operator – specifies there is (at least) one transition sequence starting at the state satisfying the defined behaviour
not(*behaviour*)	Not matching the behaviour
true	Asserting the truth of the property if the specification satisfies the behaviour specified earlier in the property
false	Asserting that the property is false if the specification satisfies the behaviour specified earlier in the property

Table 5.5: CLOVE Verification Constructs

combinations of signals using boolean binary operators. It also supports multiplicity, one or more '+', zero or more '*', by appending the appropriate multiplicity operator.

A signal denotes an action in the behaviour of the service with a specific port and operation and data value. The signal may represent a fault behaviour. The data value may be specified as a regular expression for matching a range of actions. Any signal satisfies "true", and never "false". The boolean binary operators for signals are used to express filters over behaviour. Sequences of signals express consecutive behaviour. Signals, sequences, and boolean operators may be repeated using multiplicity operators.

5.4.1.2 CLOVE Verification Patterns

Service developers are more likely to be interested in pragmatic implementation than formal aspects [75]. A high-level approach to specifying verification properties for composed web/grid services may not provide adequate abstraction to motivate use. Though the 'how to' is simplified, the 'what' to specify remains a conceptual challenge. Having templates for common

properties and ones useful for for web/grid services can help verification without the need to understand in detail complex technical concepts (e.g. temporal logic). Verification templates, in addition to the high-level CLOVE approach, are provided for commonly specified properties.

Several properties are common and applicable even across different domains. This was the finding of the Patterns project [67], which to a large extent is applicable also to web and grid services. The Patterns project published categorised sets of the most frequently specified verification properties as pattern references, which formal analysts can readily adopt and adapt to verify their specifications. The RAFMC patterns [65] are μ-calculus template formulae, based on these sets of pattern references, which can be used to perform verification via the CADP toolkit. The RAFMC formulae templates are based on single predicates (individual transition labels). CLOVE adapts these formulae as abstract notation and support multiple predicates. The pattern references fall into many categories. Amongst these CLOVE adapts the response, universality, absence, precedence, and existence categories as they were found by the Patterns project to be useful 92% of the time. They are also applicable to the web/grid service domain.

Response is usually for describing a cause-effect relationship between a pair of actions. It sometimes may conceived as a safety property as it is similar to the converse of Precedence though not equivalent. Universality is used for describing a property that always (or henceforth) holds for a part of the system's execution. Absence employs safety reasoning where "things should not happen". Precedence is also a form of safety specification where it enforces the precedence of specified actions prior to the given predicates. Existence is the reachability of actions, which is a form of liveness property. There are five scopes for all patterns: *global*, *before* a specific behaviour, *after* a specific behaviour, *between* two specific behaviour, and *after* a specific behaviour *until* another specific behaviour. The adaptation primarily uses single predicates as placeholders, made available via macro calls as parameters. Table 5.6 show CLOVE's adaptation of the five selected patterns of RAFMC and their scopes. Of these five scopes the global scope plays a major role, with the other scopes being less commonly used.

Each category has its required number of predicates. Response has two (S responds to P); Absence has one (P is false); Existence has one (P occurs), Universality has one (P is true); and Precedence has two (S precedes P). Each scope has its required number of predicates. The

CLOVE Pattern	Description
initials(**signal**(...), ...)	A specific safety property for verifying the permitted initials of the specification, meaning that the behaviour only starts with the signals specified in initials.
response(scope, behaviour, ...)	The Response Pattern: behaviour S responds to behaviour P. The first two parameters after the scope are for S and P; thereafter the parameters are as required by the scope.
absence(scope, behaviour, ...)	The Absence (safety) Pattern: P is false where P is a behaviour. The first parameter after *scope* is for P, thereafter the parameters as required by the scope.
existence(scope, behaviour, ...)	The Existence Pattern: behaviour P occurs. The first parameter after *scope* is is for P, thereafter the parameters are as required by the scope.
universality(scope, behaviour, ...)	The Universality Pattern: behaviour P is true. The first parameter after *scope* is for P, thereafter the parameters are as required by the scope.
precedence(scope, behaviour, ...)	The Precedence Pattern: behaviour S precedes P. The first two parameters after *scope* are for S and P, thereafter the parameters are as required by the scope.

Table 5.6: CLOVE Supported Patterns

global scope requires no predicates; before has one, after has one, between has two, after until has two. Therefore the number of parameters (predicates) required in a pattern is the total of the parameters needed by the category and specified scope. For example, the CLOVE Response pattern (S respond to P) using the before scope (before R) requires three parameters.

The translation strategy for these CLOVE pattern templates is described in section 5.4.2.6, specifically in tables 5.10, 5.11, 5.12, 5.13, and 5.14 which are adaptations for CLOVE based on the RAFMC patterns.

5.4.1.3 CLOVE Data Type Range Syntax

CLOVE allows specification of data values. Considering CADP, which uses a finite state space model checking technique, this feature plays a major part in the automated verification of composed web/grid services where their formal behaviour is generated in LOTOS. As every (composed) web/grid service will define data structures and values to describe their information

Syntax	Description
enum_complex(...)	Complex type enumeration
enum_naturals(...)	Natural enumeration
enum_numbers(...)	Floating point number enumeration
enum_strings(...)	String enumeration

Table 5.7: CLOVE Data Enumeration Syntax

exchange, the support data in the formal model enables the analysis which is possible with LOTOS and CADP.

CADP (specifically its CAESAR tools) will generate the state space of the behaviour, incorporating data values. The CAESAR tools have the capability to use user-defined data values that are implemented in C code in adherence to its framework, thereby generating the corresponding state space as directed by user inputs. Manual coding of data values is possible, but the effort required may be significant and prone to errors. CLOVE abstracts the technical details of the framework and C code, allowing users to focus on specifying data values which are automatically translated for verification. There are CLOVE constructs for specifying data values for base and complex types. There are also built-in functions to automatically capture and use data values in specified properties. Table 5.7 describes the CLOVE syntax to define data values.

Literal values and regular expressions can be specified for the entire range for a particular type, for example the CLOVE syntax below specifies three literal floating point values (1, 3 and 7) and the range of values defined in the regular expression such as 10, 12, 14, etc.:

enum_numbers(1,3,7, [1-9][02468])

With regard to CAESAR, CLOVE-specified data values are translated into a C implementation (value construction and iteration functions) which CAESAR includes in its verification framework. It is not always necessary to specify values for complex types; by default their values are based on the range of base types provided by CLOVE.

CLOVE provides a way to pick up literal data values of base types within specified properties. This is done via a simple "record" functionality whereby data values are stored if the functionality is on turned on while parsing the CLOVE file in a top-down manner. This functionality can start and stop multiple times between each property, giving the user more control

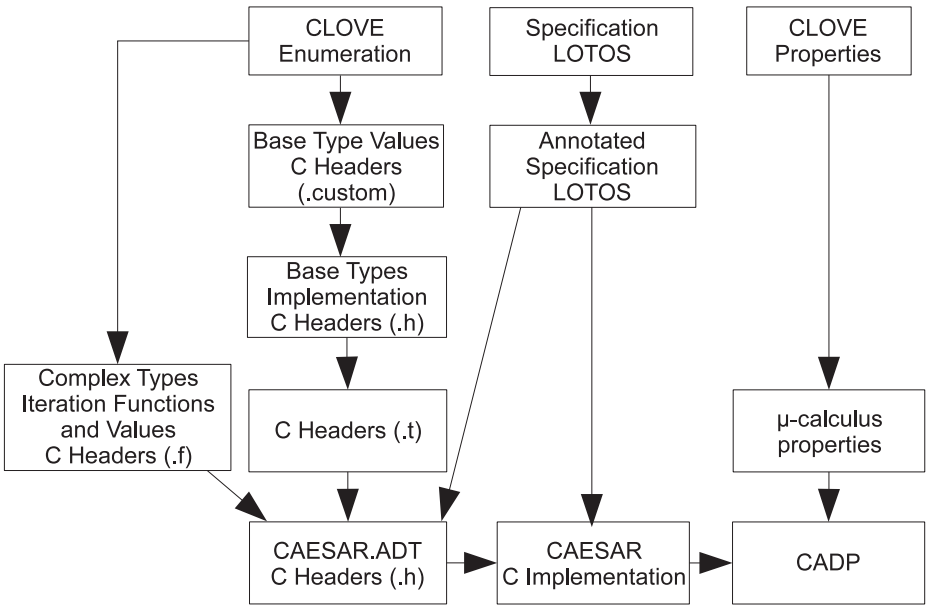

Figure 5.4: CLOVE's Generic Framework

over the inclusion of values. These recorded values are consolidated with the literal values and those generated from patterns as a duplicate-free set for the type.

5.4.2 Tool Support

5.4.2.1 Tool Design Overview

Figure 5.4 shows the components in the CLOVE architecture and how they collectively automate verification and abstract CADP as the underlying toolset. CADP's enumerative verification is achieved by generating C code from an annotated LOTOS specification. The execution of the code simulates the behaviour, generating the state space for model checking as the means of analysis. This is achieved by the CAESAR/CAESAR.ADT tools. CAESAR.ADT generates an implementation of the data types as a C header file, and provides hooks to include external implementations (such as values and iteration functions) by including '.f' and '.t' files which are respectively for functions and types. Verification can then be performed on the generated state space; for example the evaluation of μ-calculus properties.

CLOVE makes use of the the CRESS cadp_annotate tool to automate the annotation of a LOTOS specification to achieve a compatible specification for CADP. A suite of C implementations of the CRESS base types and enumeration functions were developed as C headers files to underpin the complex types implementation. They are implemented in such a way that

user-defined enumeration values for base types can be loaded in, achieved by inserting base type values into a '.custom' file which will be read in by the base type C implementations to use the specified values for iteration. These are consolidated under the '.t' file which will be generated by CLOVE. This is then automatically included into the C implementation generated by CAESAR.ADT. The framework supports the enumeration of data values even for complex data types. CLOVE enumeration constructs for complex types are generated with C iteration functions and enumeration values in the '.f' file. These enable the simulation of the behaviour by CADP, specifically CAESAR, which will generate the state space representing the behaviour based on the implementation of the data types and enumeration values. The thesis work developed a translation strategy for CLOVE properties into μ-calculus properties, which is the target property verification specification language for web and grid service LOTOS specifications. CLOVE invokes the various tools in CADP to execute verification: invoking CAESAR.ADT to generate a C implementation; invoking CAESAR to simulate the specification; invoking EVALUATOR to verify the translated μ-calculus properties; and extracting counter-examples for diagnostics. In addition, a strategy was developed and implemented to support compositional verification automatically for the generated LOTOS specifications.

Many of these components are potentially useful even for formal methods personnel, which can significantly reduce the preparation work for verification such as annotating a specification, handcrafting C enumeration code for each data type, and even performing compositional verification.

Although CLOVE currently supports web/grid services, LOTOS (target specification language), CADP (target underlying tool), and μ-calculus (property language). Its tool design is extensible, allowing addition of languages, techniques and technologies, to achieve abstraction and automation of verification.

5.4.2.2 Annotation Strategy for LOTOS Specification

There are two types of annotation: sort, and constructor operation. Sort annotation comprises C macro names for implementation, comparison, iteration (first and next value), and printing, annotated on the LOTOS Sort. Constructor operation annotation deals with the type's constructor operations such as 'proposal: Text,Text,Number -> Proposal' for the Proposal sort described

in section 5.2.1.1. The CADP pragmas annotation syntax is described in table 5.8. In dealing with data types, a mandatory requirement of CAESAR.ADT is the identification of constructor operations. Constructors in LOTOS are data type operations that are basic to constructing values. The non-basic expressions or equations in LOTOS are reduced/rewritten into canonical form when reasoning. CAESAR.ADT requires these constructors to be annotated in order to generate a C program that correctly handles data values. Sorts are annotated if there is a need to implement data value handling functions. Another mandatory requirement is that data type definitions must not be in generalised or parameterised form, which are template definitions for actual type instantiation in LOTOS. This further requires that type definitions be already instantiated from generalised types. The solution is to instantiate formal types as actual types.

Manually adapting LOTOS specifications can take time and effort, especially when a specification is large, contains generalised data types, and has a lot of constructor operations to be annotated. The cadp_annotate tool performs automated annotation of a LOTOS specification. cadp_annotate performs the task of instantiating type definitions as normal types using a flattening tool, and then applies the annotations. The flattening tool takes the generated LOTOS specification, which may contain generalised LOTOS types, and collapses these into actualised definitions. The cadp_annotate tool contains pre-defined signatures with associated annotation information. This pre-defined information contains the annotation details for CRESS library types. The cadp_annotate tool accepts user-defined types in the same signature format to annotate user-defined data structures. The annotation is then automatically applied to the flattened LOTOS specification using these two sets of information.

The thesis work has developed support for user-defined annotation information to be automatically generated in a separate file during the translation of the CRESS diagram. This is feasible as the CRESS translation strategy for a data structure in LOTOS is known.

5.4.2.3 Abstract Base Type Implementation for CADP

The CAESAR tools offer the flexibility of user external implementation, which gives total control over the types, such as their value range, iteration algorithm, value comparison, and label printing as described in table 5.8. These implementations are specified in C header files

Pragmas	Description
implementedby	The value of implementedby specifies the name of the sort or operation that will be found in the C code. If also annotated external, then CADP expects an implementation to be provided. Otherwise CAESAR.ADT will generate the C code and associate the name of the sort or operation using the value specified.
comparedby	For sort annotation only. The value specifies the name of the C function/macro used to perform comparisons between two values of the sort.
iteratedby ... **and** ...	For sort annotation only. The values specify the names of the C functions/macros used to calculate respectively the first and the next data value of the sort.
printedby	For sort annotation only. The value specifies the name of the C function/macro used to print the value of the sort.
external	Applicable to both sort and operation annotations. If used, CAESAR will expect to be provided with an external implementation instead of generating one automatically.
constructor	For constructor operation annotation only. If specified, CAESAR will expect the operation not to have equations (rewriting rules) defined for it.

Table 5.8: CADP Pragmas Description

where the functions correspond to annotations in the LOTOS specification. CAESAR.ADT can infer implementations of complex types that are based on existing types. By default, CADP uses natural numbers as abstract values for all types. Though this form of abstraction is very simple, it can be difficult to specify readable properties – especially if complex data values equations are involved. Having external implementations can help in the effort of readable property specification and control of analysis, for example simplify data value printing in BCG.

An overview of the CAESAR.ADT framework for data implementation is first described to establish how it underpins CADP's enumerative verification. The transition labels in a generated labelled transition system (BCG) denote the gate events with the values offered. All possible values of an event will be enumerated, for example 'gate ?var:Nat' will exhaust all possible values that the Nat type may be represented in its C implementation. CAESAR.ADT defines an API for the implementation of sorts: functions for first (returns first value), next (returns next value), printed (printing of the data value in a transition), and compare (comparing two values). The data type and function names follow what has been defined in the CADP

pragmas during annotation. For example, if a sort Proposal is annotated with **implementedby** PROPOSAL_SORT **iteratedby** PROPOSAL_FIRST and PROPOSAL_NEXT, then the C data type is PROPOSAL_SORT with *first* function as PROPOSAL_FIRST and the *next* function PROPOSAL_NEXT. The print function provides the analyst with a way to print the values as desired. By default, values are printed as defined by the equations. For example, using the LOTOS library definition of natural number, the value 3 is printed as succ(succ(succ(0))). The print function could be used to improve readability by printing this as "3". This potentially can improve readability when defining a property involving data values.

The thesis work developed C implementations for CRESS-defined LOTOS types: Boolean, Natural, Char, String, and Integer. These LOTOS base types are annotated 'external' by cadp_annotate so that CAESAR.ADT will not generate their C code and will expect the implementation to be linked via C #include directives. Complex types are built using these base types, implying that the external implementations of the base types will be invoked by the generated implementation that represents the LOTOS equations of the operations. These C implementations are used by CAESAR to generate a simulation of the model, creating the BCG.

Figure 5.4 shows how the developed C implementations for CRESS base types are included into the verification procedure. CAESAR.ADT produces the C implementation of the LOTOS data types as a C header file that has the same name as the LOTOS specification file but with a ".h" extension. This file contains a #include directive for a file with the same name but a ".t" extension. This is a feature that allows the inclusion of user-defined type implementations, especially those annotated "external" (which is the case for the CRESS base types). Conventionally a formal analyst would have to manually specify the #include directives for external implementation. The thesis work has automated this process, specifying the #include directives of the five base types in the ".t" file.

5.4.2.4 Approach to Constraining Infinite Types

The CAESAR and CAESAR.ADT tools in CADP support finite state model checking. This implies finite data values in the LOTOS abstract data types. It is very common for types to have an infinite range. Basic types such as (natural) numbers and strings are obvious examples. By default, CAESAR.ADT generates the the value iteration functions of a complex type based

on the enumeration of its attributes. The CRESS-defined and generated C implementation of data types must be constrained for model checking with CADP. This is achieved by intervening in the C implementation code, implying a good understanding of CAESAR and LOTOS. The range of the data types should be easily specified and achieved by an analyst at a high level.

The approach (figure 5.4) allows specification of enumeration values for base types via a specialised ".custom" file that is included framework. The approach automates modification of the complex type iteration functions by overwriting the functions defined in the ".h" file, via the ".f" file. The reason for the ".custom" file for base type iteration is that all the operations have been fully predefined externally, with their iteration having the ability to incorporate user-defined enumeration values via designated macro names. Therefore there is no need to overwrite these functions, since they already are "external" by definition. The ".custom" file will contain macros with predefined names recognised by the base types C implementation.

These mechanisms form the basis for automation and abstraction of user-defined enumeration in a convenient manner. This is achieved through CLOVE's data type range notation syntax. Data value ranges are automatically translated as C code in the ".custom" and ".f" files.

5.4.2.5 Automated Implementation of Data Enumeration

Complex data types can have user-specified enumeration implemented. Their required data enumeration and C value implementation are obtained from the CLOVE file, specifically from the data values description (notation described in section 5.4.1.3).

For each data type, the implementation of first its and next enumeration function is automatically generated. Both functions act upon a specific enumeration macro which is an array of unique data values translated from the CLOVE sybtax. The first function simply returns the first value in the array. The next function simply looks up the array index of the current value and, if it is not the last element, then returns the value of the next element in the array.

Three styles of data value expressions (explicit, recorded, patterns using regular expressions) may be used in CLOVE. Explicit and recorded values are simpler as they are actually explicit data values. Data values expressed as patterns using regular expression usually represent a range of values. The range is fully generated using an external tool 'regldg' [20] which can generate the entire range of specified regular expression resulting in static individual val-

CLOVE Construct	Represented μ-calculus
forall(behaviour)	[behaviour]
exists(behaviour)	<behaviour>
sequence(behaviour, ...)	(behaviour **.** ...)
and(behaviour1, behaviour2)	((behaviour1) and (behaviour2))
or(behaviour1, behaviour2)	((behaviour1) or (behaviour2))
choice_any(behaviour ...)	((behaviour1) or ...)
not(behaviour)	not((behaviour))

Table 5.9: CLOVE Translation to μ-calculus

Scope	**CLOVE Macro**	**RAFMC Pattern** [65]
globally	**response**(global, P, S) *S responds to P*	[true*. P] mu X. <true> true and [not S] X
before R	**response**(before, P, S, R) *S responds to P before R*	[(not R)*. P. (not (S or R))*. R] false
after Q	**response**(after, P, S, Q) *S responds to P after Q*	[(not Q)*. Q. true*. P] mu X. <true> true and [not S] X
between Q and R	**response**(between, P, S, Q, R) *S responds to P between Q and R*	[true*. Q. (not R)*. P. (not (S or R))*. R] false
after Q until R	**response**(afteruntil, P, S, Q, R) *S responds to P after Q until R*	[true*. Q. (not R)*. P] mu X. <true> true and [R] false and [not S] X

Table 5.10: CLOVE's Adaptation of RAFMC Response Pattern '*S responds to P*'

ues. All the data values are then consolidated as a set with no duplicates, resulting in the enumeration array.

5.4.2.6 Translation Strategy for CLOVE Properties to μ-calculus

CLOVE high-level properties are translated into μ-calculus to verify the generated LOTOS specifications. Table 5.9 illustrates the corresponding μ-calculus translations of CLOVE constructs.

CLOVE property templates/patterns are translated based on the adaptation of the RAFMC patterns as illustrated in tables 5.10, 5.11, 5.12, 5.13, and 5.14. The parameters S, P, Q, and R are placeholders representing predicates.

Scope	**CLOVE Macro**	**RAFMC Pattern** [65]
globally	**universality**(global, P)	[true*. not P] false
before R	**universality**(before, P, R) *P is true before R*	[(not R)*. not (P or R). (not R)*. R] false
after Q	**universality**(after, P, Q) *P is true after Q*	[(not Q)*. Q. true*. not P] false
between Q and R	**universality**(between, P, Q, R) *P is true between Q and R*	[true*. Q. (not R)*. not (P or R). (not R)*. R] false
after Q until R	**universality**(afteruntil, P, Q, R) *P is true after Q until R*	[true*. Q. (not R)*. not (P or R)] false

Table 5.11: CLOVE's Adaption of RAFMC Universality Pattern '*P is true*'

Scope	**CLOVE Macro**	**RAFMC Pattern** [65]
globally	**absence**(global, P)	[true*. P] false
before R	**absence**(before, P, R) *P false before R*	[(not R)*. P. (not R)*. R] false
after Q	**absence**(after, P, Q) *P is false after Q*	[(not Q)*. Q. true*. P] false
between Q and R	**absence**(between, P, Q, R) *P is false between Q and R*	[true*. Q. (not R)*. P. (not R)*. R] false
after Q until R	**absence**(afteruntil, P, Q, R) *P is false after Q until R*	[true*. Q. (not R)*. P] false

Table 5.12: CLOVE's Adaption of RAFMC Absence Pattern '*P is false*'

Scope	**CLOVE Macro**	**RAFMC Pattern** [65]
globally	**precedence**(global, P, S) *where S precedes P globally*	[(not S)*. P] false
before R	**precedence**(before, P, S, R) *S precedes P before R*	[(not (S or R))*. P. (not R)*. R] false
after Q	**precedence**(after, P, S, Q) *S precedes P after Q*	[(not Q)*. Q. (not S)*. P] false
between Q and R	**precedence**(between, P, S, Q, R) *S precedes P between Q and R*	[true*. Q. (not (S or R))*. P. (not R)*. R] false
after Q until R	**precedence**(afteruntil, P, S, Q, R) *S precedes P after Q until R*	[true*. Q. (not (S or R))*. P] false

Table 5.13: CLOVE's Adaption of RAFMC Precedence Pattern '*S precedes P*'

Scope	CLOVE Macro	RAFMC Pattern [65]
globally	**existence**(global, P)	mu X. <true> true and [not P] X
before R	**existence**(before, P, R) *P occurs before R*	[(not P)*. R] false
after Q	**existence**(after, P, Q) *P occurs after Q*	[(not Q)*. Q] mu X. <true> true and [not P] X
between Q and R	**existence**(between, P, Q, R) *P occurs between Q and R*	[true*. Q. (not (P or R))*. R] false
after Q until R	**existence**(afteruntil, P, Q, R) *P occurs after Q until R*	[true*. Q] mu X. <true> true and [R] false and [not P] X

Table 5.14: CLOVE's Adaption of RAFMC Existence Pattern '*P occurs*'

5.4.2.7 Compositional Verification

The CADP tool supports compositional verification, a technique which uses abstract interfaces and process constraints in conjunction, resulting in possible speed-up of verification. The LOTOS specification is divided into smaller behavioural units (in BCG form) and then recomposed. By doing so, the entire state space is not generated from scratch, but is built from the synchronisation of the generated state space of each constrained behavioural unit.

An abstract interface is usually a process whose interactions with others constrain their behaviour, often through data values. For example process A interacts with process B, passing a Natural number. Process B would normally generate a state with all possible Natural numbers at the point of interaction with A, for example 0 to 255 in a finite context. This may not be the case, e.g. process A only sends even natural numbers between 1 and 24, which is far fewer than the possibilities offered by process B in this situation. It is possible to generate only the necessary states for process B's behaviour when used in conjunction with A, thereby increasing the efficiency of state space generation. This is achieved by having an abstract interface to Process A for Process B, as a state space with all transition labels used for synchronising with B's behaviour. Process B is then constrained, only generating the state space based on the synchronisation with the labels of Process A's abstraction.

This technique is very useful for specifications with several process interactions and a very large state space. The strategy for LOTOS specification of composed web/grid services is a

natural fit to CADP's compositional verification, and also a demonstration of how CLOVE's abstraction approach takes advantage of the underlying tool capability. Services themselves are distributed and specified by CRESS as processes having interactions through synchronisation.

CLOVE's strategy for applying compositional verification to composed web/grid services is an automated approach that systematically break the LOTOS specification behaviour for all CRESS-specified services recursively into units. Partner services that are not composite (not a CRESS diagram) will not be broken down further as there is no information to determine if they are compositions themselves. Process abstraction and constraints are therefore recursive until non-CRESS partners are found. The composition of these state spaces constitute the entire state space of the whole LOTOS specification. This state space (BCG) is then subjected to property verification. This work is automatically carried out by CLOVE, through a combination of automatically adapting the LOTOS specification and generating an SVL script describing the compositional state space.

Compositional verification works best in the case of multiply-nested compositions and large data value ranges, and potentially can have verification to execute sooner. As it may not always be the case that compositional verification works efficiently, it is an option in CLOVE. This allows the user to have a choice between compositional and conventional state space generation methods.

5.4.2.8 Diagnostic Support

CLOVE coordinates the verification process with the underlying verification tool on behalf of the user. CLOVE automatically obtain diagnostics for analysis, using CADP with regard to LOTOS as the target language. The thesis work specifically used CADP's EVALUATOR to verify properties. EVALUATOR returns a TRUE or FALSE outcome from the verification process, with diagnostics available in BCG format. CLOVE enters into a diagnostic procedure should it read FALSE for a verification outcome.

CLOVE invokes CADP to extract the textual form of the diagnostic BCG as pseudo-LOTOS. The pseudo-LOTOS output usually contain LOTOS event labels, processes, internal events (as the LOTOS *i* keyword) and invocations of processes. CLOVE formats the pseudo-LOTOS into a textual label-oriented form prior to display. The LOTOS process definitions are formatted as

labels. LOTOS *i* events are displayed more meaningfully as "internal event". As an example, the pseudo-LOTOS may look like the following. This is a counterexample describing a cyclic behaviour of event A, followed by two *i* events, then event B.

```
A;
 P_1
where
process P_1 :=
 i;
  i;
   B;
    A;
     P_1
endproc
```

CLOVE formats it as:

```
A;
 jump to label P_1
label: P_1
 internal event;
  internal event;
   B;
    A;
     jump to label P_1
end of label: P_1
```

5.4.3 Examples

5.4.3.1 CLOVE Data Value Enumeration

The following is a CLOVE example that specifies data values for base types (String, Natural, etc.) and complex types using static values and regular expressions.

```
enum_ numbers(3.5, 3.7, 4.1, 4.4)

enum_ strings('loan unacceptable,'low,'medium,'high)
enum_complex(
proposal("(KEN TURNER|LARRY TAN)","UK",[5|7]0000.0),
proposal("(KEN TURNER|LARRY TAN)","UK",[5|7]000.0) )
```

The enum_numbers, and enum_strings constructs are used to define the values for integers and strings. These values are used translated into C code in the '.custom' file for use by the String, and Number C implementations. The '.custom' file may look like the following.

#define CRESS_TEXT_ENUM "HIGH","LOAN UNACCEPTABLE","LOW","MEDIUM"

#define CRESS_ENUM_NUMBER_PERMUTATIONS 3.5,3.7,4.1 ...

The Proposal complex type's enumeration values are defined using patterns, which will be expanded by the 'regldg' (regular expression grammar language dictionary generator [20]). The first pattern generates Proposal ("KEN TURNER", "UK",50000.0), Proposal ("KEN TURNER","UK",70000.0) etc. The generated values are consolidated, and translated as the C macro PROPOSAL_ENUM. This will be used by the iteration C code for Proposal that is generated by CLOVE in the '.f' file.

5.4.3.2 CLOVE Properties

The following response pattern specifies the property that "throughout the entire system behaviour any proposal is replied to with a numerical value or a refusal with a loan unacceptable fault". The first parameter is 'global' which indicates the scope of the property applies throughout the entire behaviour. A response pattern using global scope requires two parameters. The first is the predicate that is responded to, the second specifies the response. This verification property can produce counter-examples for responses that are not numerical or faults that are not loan unacceptable refusal.

property(General_Response,
  response(global,
    signal(lender.loan.quote,?proposal),
    choice_any(**signal** (lender.loan.quote,?number),
      signal(lender.loan.quote,refusal,'loan unacceptable))))

This CLOVE property is automatically translated into μ-calculus syntax:

(* Property name : GENERAL RESPONSE *)

[true* . ("LENDER !LOAN !QUOTE" # " !" # 'PROPOSAL (.*)')]
    mu X. (<true> true and [not(("LENDER !
    LOAN !QUOTE" # " !" # '[0−9][0−9]*[.][0−9]*') or
    ("LENDER !LOAN !QUOTE" # " !" # "REFUSAL" # " !" # "\"LOAN
    UNACCEPTABLE\""))] X)

The same pattern template could be adjusted with specific values to refine the property objective, such as the following property for which a response of 1.0 as loan rate would be a counterexample.

```
property(Specific_Response,
  response(global,
    signal(lender.loan.quote,?proposal),
    choice_any(signal(lender.loan.quote,3.5),
      signal(lender.loan.quote,3.7),
      signal(lender.loan.quote,4.1),
      signal(lender.loan.quote,4.4),
      signal(lender.loan.quote,refusal,'loan unacceptable'))))
```

The following is the translated μ-calculus:

```
(* Property name : SPECIFIC RESPONSE *)

[true* . "LENDER !LOAN !QUOTE" # " !" # 'PROPOSAL (.*)']
      mu X. (<true> true and [not(("LENDER !
      LOAN !QUOTE" # " !" # "3.5") or
      ("LENDER !LOAN !QUOTE" # " !" # "3.7") or
      ("LENDER !LOAN !QUOTE" # " !" # "4.1") or
      ("LENDER !LOAN !QUOTE" # " !" # "4.4") or
      ("LENDER !LOAN !QUOTE" # " !" # "REFUSAL" #
      " !" # "\"LOAN UNACCEPTABLE\""))] X)
```

CLOVE has made the initials safety property into a convenient shorthand, using the **initials** macro where all permitted initial signals are specified. The above could be specified as the following and executed automatically using CLOVE's -i option.

```
initials(signal(lender.loan.quote,?proposal))
```

The **initials** definition is translated to the following μ-calculus syntax:

```
(* Property name : INITIALS SAFETY *)

[not("LENDER !LOAN !QUOTE" # " !" # 'PROPOSAL (.*)')] false
```

Two other well-known and used properties are the deadlock and livelock freedom. Though they are not patterns, they are so general that they are specified in most analyses, and are provided by verification tools including CADP for ready use. A system is deadlocked when it cannot proceed further. A system enters into a livelock when it can be indefinitely doing something without responding (e.g. an internal event cycle). Deadlock and livelock freedom are properties very applicable to web/grid services. It is undesirable for services to make no progress in execution and therefore unable to render further service (deadlock). It is also undesirable that the service is indefinitely in execution, for example unable to terminate from loops, and therefore be unable to respond (livelock). Given that these are likely to be specified for analysis, CLOVE will check for deadlock and livelock freedom without even requiring to specify a property or use a pattern macro.

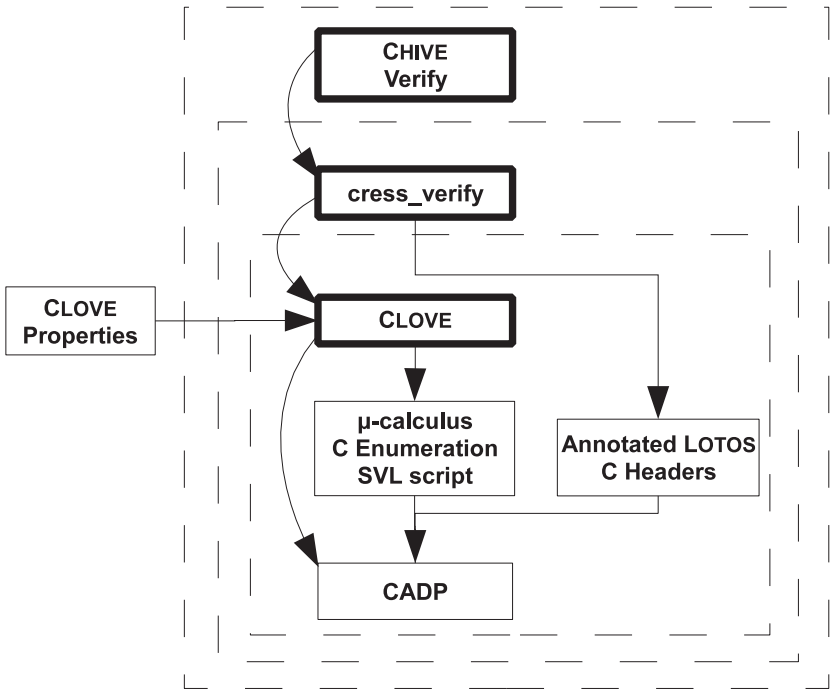

Figure 5.5: Levels of Automated Verification Procedure

5.4.4 Tool Integration

Verification is an entirely new CRESS feature for composed web/grid services, realised by CLOVE with tool support for the LOTOS specification. CLOVE was integrated into CRESS to realise the objective of an integrated rigorous development environment as defined by the thesis methodology. The thesis work extended the CHIVE editor graphical interface to call for verification. A new command-line tool cress_verify, which is analogous to cress_validate for validation, was developed as the verification interface in the CRESS environment; this can be invoked from the CHIVE editor.

Figure 5.5 illustrates the different levels of integration of CLOVE, which is identical to the integration for validation concerning the level of tool interaction and control. It also demonstrates the various approaches to automated verification, including the activities carried out and the chain of command. A box with dashes denotes a task unit of the tool. A box with a bold outline represents a tool that the user interfaces with. The arcs represent tool interactions. The straight lines represent the outputs from the originating box.

A “Verify” menu item was added to the CHIVE editor Tool menu as a graphical callout to cress_verify. cress_verify options are specified through the CHIVE Preferences ‘tool options’

dialogue box. The outputs from cress_verify will be captured by CHIVE as a results window. This provides push-button verification via a graphical development environment.

cress_verify interfaces CLOVE, invoking its verification procedure where the actual verification then will then be carried out by the CADP toolset. Within the scope of cress_verify there is the annotation of the LOTOS specification (performed by cadp_annotate), and generation of C header directives (by cress_lotos.pm) for the inclusion of the base data types implementation and provision for hooking in base data type enumeration values. This prepares the LOTOS and data implementation as required by CADP. The CLOVE tool is invoked to coordinate verification. The results gathered by CLOVE are returned to cress_verify.

The CLOVE tool translates CLOVE data enumeration and property definitions respectively into C enumeration code, values, and μ-calculus properties. An SVL script will be generated if compositional verification is required, or when manual mode is specified. CLOVE automatically invokes the appropriate CADP tools using the annotated specification, C implementation, μ-calculus properties, and SVL script (if necessary) as inputs to verification. Results and diagnostics returned by the CADP tools are interpreted at a high-level. If CLOVE is used directly, then the annotated specification and C headers are presumed to be present.

All three approaches support a manual mode whereby there is automatic generation of the required verification inputs (μ-calculus properties, C data enumeration and values, SVL script), but automated verification is not carried out. This is useful especially for the formal methods personnel who can make refinements and perform verification analysis by hand. The tool options for cress_verify (applies to CHIVE) and CLOVE can be found in manuals of CRESS [104] and CLOVE [105]. Although CLOVE was developed for the verification of web/grid service specifications using LOTOS and associated tools, this approach can be generalised and applied to other domains and support other formal techniques.

5.5 Evaluation

The formal analysis of composed web/grid service specification has been extended to cover automated validation and automated verification, respectively achieved by MUSTARD and CLOVE. Both aspects are complementary to one another. Although validation is formally based, testing

is necessarily limited. Its main advantage is that validation is practical: automated validation of even a complex service is performed in seconds or minutes. The practicality applies even more so now with the extended MUSTARD validation of service partners, dynamic resources, and dynamic specification supported by variables. Through CLOVE's automation of formal verification using CADP tools, it is easy and quick to prove properties in general for classes of tests for "system" level confidence. Technical challenges of the underlying technologies have been abstracted and automated, lowering the barrier to entry for maximum exploitation. Verification property specification is aided with parameterised pattern/template macros for frequently specified properties. Although in this case the formal verification deals with a finite state space, validation can be used to check cases beyond this. Their combined use supports practical formal analysis.

Automated verification with CADP currently has a limitation that specifications containing compensation activities cannot be readily handled, though for validation it is entirely accomplished with the same generated LOTOS specification. This is not a limitation of the LOTOS language, but rather a restriction mandated by the CADP tool: there cannot be unguarded recursive process instantiation. CADP imposes this constraint as it cannot guarantee finite state generation with such a specification. This is reasonable as CADP uses explicit enumerative verification. Subsequent to completion of the thesis work, this restriction has now been removed.

There are future plans for improvement and new developments to the formalisation and analysis. Specification for partner services could be simplified further with support for graphical design. This could be supported with automated bisimulation, for example with the generated interface behaviour, to ensure that these partner specifications implement the expected behaviour. The specification generation strategy can be extended with the capability to adapt the specification readily for verification with CADP tool. Information produced (e.g. outcomes, statistics, and diagnostics) from analysis could be elaborated and depicted graphically for easier comprehension.

Chapter 6

Implementing Composed Web/Grid Services

The implementation of composed services is the ultimate outcome of development, where service functionality and performance are subjected to actual evaluation and experience by real users who are probably not aware of or have concerns about any prior design activities involved – although they are important and have major influence on the implementation. The design phase in development has moved on from being a traditionally separate activity to being more integrated with the lifecycle. For example, high-level graphical notations are now common to depict the functionality and relationship of components, even generating entire code or code skeletons. Such an approach can be used for implementing composed web/grid services in practice. For example some BPEL standard vendors provide visual design tools in addition to their enactment engines to develop and deploy orchestrated services.

CRESS supports the development of composed web services where implementation code is automatically generated from the CRESS graphical service description. Using the same service description for formalisation and implementation has the advantages of being simple, consistent and integrated. Changes in service description are automatically reflected in the formal behaviour and the actual implementation code.

Originally CRESS supported BPEL4WS (version 1.1) as the target implementation standard for composed web services, obtaining the BPEL code from translating CRESS diagrams. The

thesis work has adapted the original translation strategy for web services to composed grid services. The thesis work developed the translation strategy for the successor WS-BPEL 2.0 standard which is applicable to both web and grid service domains. The CRESS notation was extended as described in Chapter 4, so these automatic implementation translation strategies have also been extended to support the CRESS semantics correspondingly.

The original CRESS did not have an integrated and automated approach for testing implemented services as a counterpart to MUSTARD for specification validation. Hence developers had to deal with the task of testing composed services and partner services. The thesis work developed tool support for implementation validation in the methodology to increase the productivity and quality of service development, especially in terms of time and effort. There are advantages in having an approach to easily specify tests cases, and to automatically test and evaluate the performance of implemented composed and partner web/grid services. It gives focus to test specification unlike test tools such as JUnit where much programming unrelated to an actual test is required. Improved productivity is needed so that low-level technical details are automated such as translating WSDL to actual program code and compiling this, which by itself may be considered as a specific form of validation. Diagnostics for validation and performance evaluation are supported.

This chapter discusses the implementation aspect of composite web/grid services. In particular, the chapter presents extensions and new work on the CRESS translation strategy for implementation, harmonising compatibility issues, and achieving implementation validation.

6.1 Automatic Implementation

Automatic implementation was achieved for web services by CRESS with its original strategy [101]. This included full code generation for services, with a framework that is capable of incorporating manually-provided implementations for partner services, and of compiling service implementations into readily deployable archives. The thesis work has adapted the approach and applied it to the grid service domain. The extensions also had the goal to support the latest BPEL standard (WS-BPEL 2.0). In addition to the development for web services to support WS-BPEL 2.0, there was also a parallel development to support both standards for grid

services, including support of implementation for the extended CRESS notation. Collectively these realised the thesis objective of automatic implementation for web/grid services.

6.1.1 Original Translation Strategy

The original strategy followed a service deployment plan that coordinates code generation, compilation and deployment of (partner/composed) services, using information given in and inferred from the CRESS service and configuration diagrams. The generated code consists of data types (XSD), service interfaces (WSDL), process behaviour (BPEL), and deployment configuration (e.g. PDD for ActiveBPEL). This subsection describes in detail the original strategy that achieved automated implementation of composed web services [102].

6.1.1.1 Web Service Deployment Plan

Figure 6.1 illustrates the plan for how a composed web service described as a CRESS diagram is automatically translated into a significant amount of XML code that developers do not have to write; this code comprises WSDL, BPEL and configuration files. A WSDL file is created to describe a business process interface as it is also a web service. Partner web services also have their respective service interfaces created as WSDL files. Another WSDL file is also created for message and data type definitions that are shared by the composite service and its partner services. This WSDL is used by the services interface WSDL files. A BPEL file describing the process behaviour is created. Finally these files are packaged into a deployable archive for the composed service. Composed services are then automatically deployed in ActiveBPEL.

A partner web service will require a one-off implementation by the developer if the service is new. CRESS will create the partner's WSDD (Web Service Deployment Descriptor) automatically. The WSDL service interface is translated into a Java service skeleton using the AXIS *wsdl2Java* tool. The developer can then manually implement the service using the skeleton. If a Java implementation was already provided, that code is used instead of the skeleton in automatic deployment. Partner web services are deployed using ActiveBPEL.

If a partner service already exists, meaning that it is already a deployed and running service, then there is no need for the one-off implementation. The generated WSDL can be used directly by the composed process as there is an existing implementation for service interface. It is likely

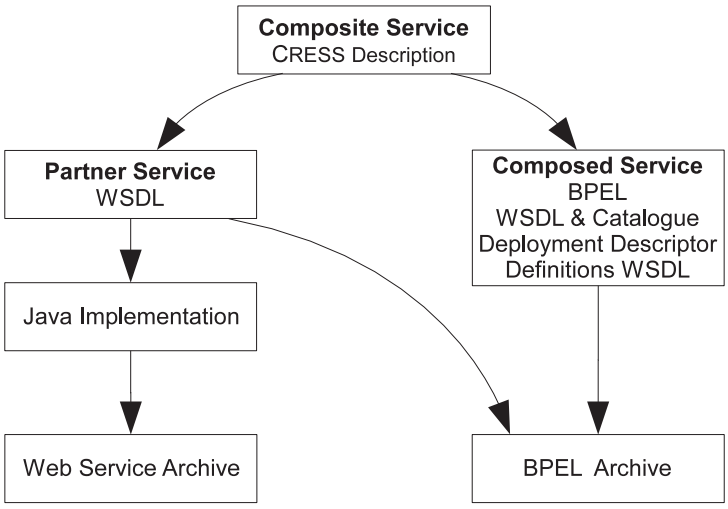

Figure 6.1: Web Service Deployment Plan

that only certain ports and operations of the partner will appear in CRESS diagrams. The WSDL generated for the partner web service may a subset of its actual WSDL, containing only the necessary information required for the composite service deployment. This subset WSDL does not affect the actual partner service in any manner, but rather reflects the CRESS perspective on the partner service interfaces visible to the composite service.

6.1.1.2 Service Interfaces (WSDL)

For each service, CRESS generates the service interface WSDL which syntactically defines the ports, operations and their corresponding message parameters and service bindings. These services share the common definitions, therefore their WSDL will have import directive for these common definition which are also WSDL files.

The service port is defined in the WSDL <portType> specification. Within the <operation> specification its elements (input, output, faults) are given. The definitions are derived from the consolidation of their use in the CRESS diagram which captures all ports, operations, and corresponding parameters and faults for the particular service, including partner services. The port type name follows the convention <*port*>Port. For example, the consolidation of a service use in a CRESS diagram may have the signatures for a composed service *supplier*: 'dealer1.car.quote need offer' and 'dealer1.car.cancel offer'. The <portType> is generated as the following WSDL <portType> code. The 'message' attribute values refer to the <message> definitions that are based on the data types definitions, as described in section 6.1.1.3.

```
<portType name="carPort">
  <operation name="cancel">
    <input message="defs:offerMessage"/>
  </operation>
  <operation name="quote" >
    <input message="defs:needMessage"/>
    <output message="defs:offerMessage"/>
  </operation>
</portType>
```

The service binding defines the use of transport protocol, SOAP parameters, and namespace for the port's operations and parameters; these establish the protocol contract for service communication. The XML prefix and the value of the namespace attribute are obtained from the service's definition in the CRESS configuration diagram where the prefix and namespace are defined, in this case 'deal1' and 'urn:BigDeal'. The following is an example of <bindings> for the above portType definitions:

```
<binding name="carBinding" type="deal1:carPort">
  <soap:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="cancel">
    <soap:operation
    soapAction="http://www.cs.stir.ac.uk/schemas/Dealer1Service/cancel"/>
    <input>
      <soap:body use="encoded" namespace="urn:BigDeal"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
  </operation>
  <operation name="quote" >
    <soap:operation
    soapAction="http://www.cs.stir.ac.uk/schemas/Dealer1Service/quote"/>
    <input>
      <soap:body use="encoded" namespace="urn:BigDeal"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <soap:body use="encoded" namespace="urn:BigDeal"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>
```

For a composed service described in CRESS, it is natural that the description specifies all its ports and operations. This is not necessarily so for a partner service whose behaviour is not described in CRESS, where the generation of the WSDL takes the view of the composed

service which may be a subset of what the partner service supports. The generated WSDL is sufficient for use by the composed service which is implemented in BPEL.

Also for a composed service, its WSDL service interface will contain definitions of partner links, including itself, which are referenced by the BPEL process when interacting with the partners. A BPEL service considers itself as a partner to its client and therefore has a partner link describing itself. A partner link is a named association to a partner's role which is defined with its port types. The name for a partner link follows the convention <*partner*><*Port*>Link. As an example, the code below gives the partner link for the above 'dealer1' service with 'car' port.

```
<plnk:partnerLinkType name="dealer1CarLink">
  <plnk:role name="dealer1">
    <plnk:portType name="deal1:carPort"/>
  </plnk:role>
</plnk:partnerLinkType>
```

All services will have a <service> definition which declares its port name and the **<binding>** it uses, along with the default service address location. The naming convention for the service name is <*Partner*>Service, and the port name <*Partner*><*Port*>. The port name is used in the generation of code stubs. There is, for example, a getDealer1Car accessor method for obtaining an instance of the port for invocation in programming languages such as Java. As an example, the <service> definition for the 'dealer1' example illustrated above is:

```
<service name="Dealer1Service">
  <port name="Dealer1Car" binding="deal1:carBinding">
    <soap:address
     location="http://localhost:8080/active-bpel/services/Dealer1Car"/>
  </port>
</service>
```

6.1.1.3 Common Definitions

Composed services and their partner services may share some data type definitions. As it happens, a composed service typically uses partner service operations, thereby implying the latter's binding via the service interface, especially the messages and data structures for information exchange. This information is specified as a WSDL document and then shared by the service

interfaces (WSDL files) via imports. Data structures are defined in XSD schema elements under the WSDL <types> element.

CRESS-supported web service basic types are translated directly into XSD primitive types, mostly using identical names, such as Float translated to xsd:float, Natural to xsd:nonNegative-Integer. Data structures in CRESS are translated into XSD complex structures using XSD's complexType tag, with each element representing the fields which can be nested. Fields that have structured definitions will have their own complexType definitions. Multiple variables of identical type will share the same type definition, where the name of the first variable found of that type is used to define the complex type name. For example the following data definition in CRESS:

{**String** name **String** address [{**String** description **Natural** quantity} item] contents} parcel, parcel2

will be translated into various complexType(s) definitions as shown below.

```
<complexType name="item" >
    <sequence>
        <element name="description" type="xsd:string"/>
        <element name="quantity" type="xsd:nonNegativeInteger"/>
    </sequence>
</complexType>

<complexType name="contents" >
    <sequence>
        <element name="item" type="defs:item"
            minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
</complexType>

<complexType name="parcel" >
    <sequence>
        <element name="name" type="xsd:string"/>
        <element name="address" type="xsd:string"/>
        <element name="contents" type="defs:contents"/>
    </sequence>
</complexType>
```

The *parcel* is defined as a data structure that has fields *name*, *address* and *contents*, of which contents is an array of *item* which by itself is a data structure consisting of fields *description* and *quantity*. The complexType definitions for *contents* and *item* are under the same namespace and

are instantiated by the elements of *parcel*. Here 'defs' is used as the prefix for this namespace in the XML document, pointing to the namespace configured in the CRESS configuration diagram. The variable *parcel2* shares the same type as *parcel*, therefore the type definition *parcel* is used.

These data structures are used in the specification of interactions with the service operations. A WSDL <message> is used for this, where the message elements define the parameter order, names and types. These message elements are then used in the service portType and operation definition thereby establishing the operation signature. These message elements are also used by the BPEL process specification to instantiate variables. The definitions of message elements are inferred from the service operation signatures described in the CRESS diagram which contain the input/output/fault variables, and their types are then used. For example, consider the invocation 'postal.post.quote parcel price' where the output to the service is *parcel* which is given above and the input from the service invocation is *price* which is of **Float** type. The message element definitions related to the service interface are as follows.

```
<message name="floatMessage" >
    <part name="float" type="xsd:float"/>
</message>

<message name="parcelMessage" >
    <part name="parcel" type="defs:parcel"/>
</message>
```

As seen above, messages that represent primitive typed variables use the type names with 'Message' appended to the name, and simply use the type name again for the part names. Messages with parts that refer to complex types (such as *parcel*) are defined in a similar fashion, with the exception that the type refers to the complex type. The *quote* operation definition of *post* <portType> will have <input> and <output> message attributes referring to *parcelMessage* and *floatMessage* respectively.

These common definitions are specified using the naming convention *service*_defs.wsdl where *service* is the name of the CRESS diagram where the types are initially defined. These common definition WSDL files are imported by the service interface WSDL files.

6.1.1.4 Process Behaviour

The process behaviour is generated according to the BPEL4WS specification. The nature of the CRESS notation for describing the composite web services closely matches that in BPEL. The following XML code illustrates the high-level structure of an executable BPEL4WS process. Typically there is the declaration of partner links, variables, then the flow of behaviour including definitions of links. Partner links associate the roles and ports of partners which are used by the BPEL process for interaction. Variables contain the instantiation of named bindings to data structures used in the behaviour. There is usually a <flow> which describe the activities within the process (e.g. invoke, reply, assign etc.). The flow between activities is described by <links> which contains definitions of named labels attached to source and target activities, indicating the direction of the process flow.

```
<process name="name" ...>
  <partnerLinks>
  ...
  <variables>
  ...
  <flow>
    <links>
    ...
    ... process behaviour contructs ...
</process>
```

The partner links are translated according to the following template, where partner, port, and composedservice_prefix respectively correspond to the partner service name, its port, and the configured prefix for the composed services.

```
<partnerLink name="<partner>Port"
 partnerLinkType="<composedservice_prefix>:<partner><Port>Link"
 partnerRole="<partner>"/>
```

Variables are translated from the CRESS service diagram rule box where data structures and variables are defined. The variable names are taken from the rule box, and normally instantiated as *type*Message which is the name of the <message> defined in the common definitions, for example, '<variable name="need" messageType="defs:needMessage"/>'.

The links are created with names corresponding to the nodes associated by the arcs in CRESS service diagram. For an arc from node 1 to node 2 in a 'supplier' CRESS service

diagram, the link will be named SUPPLIER.1-SUPPLIER.2. This link is then designated as the source in the BPEL activity described with node 1, and as the target for node 2.

The CRESS notation for web service diagrams is translated quite straightforwardly into BPEL activities along with their required attributes, as the syntax and parameters map directly to the corresponding BPEL constructs. Details of the translation for the major constructs are listed in Appendix B. The **Compensate**, **Empty**, **Invoke**, **Reply**, **Terminate**, and **While** CRESS node activities are directly translated into the BPEL construct of the same name, configured with the source and target links to establish the flow. If the CRESS service specifies only one **Receive** activity, then it is translated to the <receive> BPEL construct. If there is more than one **Receive** from the **Start** node, each **Receive** is translated to the <onMessage> BPEL construct. Event (**Catch**, **CatchAll**) and compensation handlers (**Compensation**) are specified within the corresponding BPEL construct (e.g. **Invoke**). Event handlers can be in the global scope for the process and not associated with any activity. Conditional guards are translated as an expression in the 'transitionCondition' attribute of the <source> link of the activity. The **Else** guard comprises the negation of all the conditional guards combined from the same originating node. The **Fork** activity that describes concurrent activities is simply translated as an <empty> activity linked to the respective activities to be executed in parallel, which means they have more than one source link defined. The **Join** *condition* activity is translated as an <empty> activity with the *joinCondition* attribute that is translated from *condition*, describing the (composite) condition of the completion of the links that converge into the **Join**.

As a concrete example of translation, assume there is a node 2 of a CRESS lender service diagram with '**Invoke** approver.loan.quote proposal rate refusal.error'. Node 2 is linked from node 1 and 6. It has a **Catch** refusal.error event handler associated with it leading to node 3, which will **Reply** 'refusal.error' followed by **Terminate** at node 4. Node 2 leads to node 5 if the **Catch** event has not occurred. This is translated to the BPEL <invoke> activity as the following BPEL code snippet.

```
<invoke name="LENDER.2"
 partnerLink="approverLoan" portType="app:loanPort"
 operation="approve" inputVariable="proposal" outputVariable="rate" >

  <target linkName="LENDER.1-LENDER.2"/>
```

```
<target linkName="LENDER.6-LENDER.2"/>
<source linkName="LENDER.2-LENDER.5"/>

<catch faultName="app:refusal" faultVariable="error" >
  <flow>
    <reply name="LENDER.3"
     partnerLink="lenderLoan" portType="lend:loanPort"
     operation="quote" variable="error" faultName="lend:refusal" >
      <source linkName="LENDER.3-LENDER.4"/>
    </reply>
    <terminate name="LENDER.4" >
      <target linkName="LENDER.3-LENDER.4"/>
    </terminate>
  </flow>
</catch>
</invoke>
```

6.1.1.5 Inclusion of Web Service Partner Implementation

Figure 6.2 shows the flow of how the the one-off implementation for a web service partner is included into the process of automated implementation. The 'cress_expand' command executes the automated implementation process for the composed and partner web services. The automated translation of the CRESS diagram produce the service interfaces and definitions as WSDL files, translates them into Java service stubs, and compiles them for use. As a concrete example, assume 'dealer1' service with a 'car' port is a supplier partner and is not a CRESS diagram. The developer will provide Java code for dealer1. The source file for the partner is named dealer1.java in the supplier directory under the composed diagram's path. Its Java class definition overwrites the implementation stub class that was produced in the translation, as seen in the Java code below. This dealer1.java file replaces the generated implementation skeleton, and is compiled into a Java binary class file together with all other Java sources in the dealer1 directory. Deployment meta files are also created using AXIS (WSDD files) under the service package directory; a META-INF directory is also created. Together the compiled class files and WSDD files are packaged in a WSR archive for deployment. If the deployment option is set in the service configuration diagram, then the WSR file is deployed into the ActiveBPEL container whose path is determined by the environment variable CATALINA_HOME, which is the location of the Apache Tomcat servlet container where ActiveBPEL is installed.

```
package namespace;
```

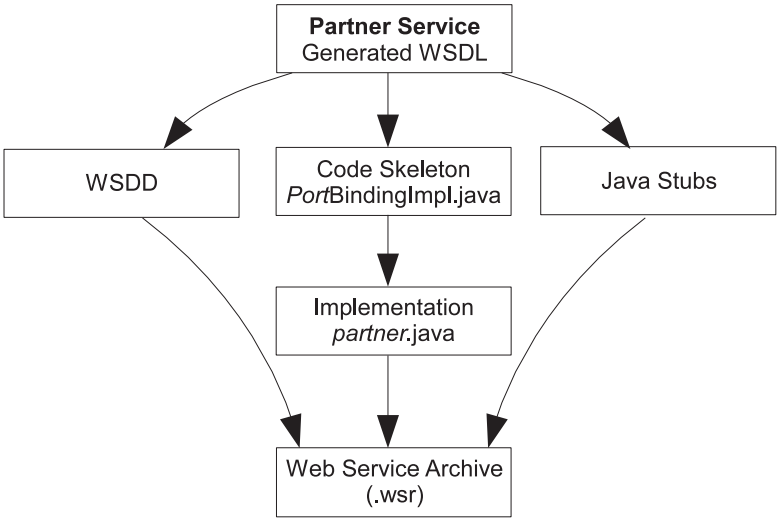

Figure 6.2: Include Partner Service Implementation

... other imports ...

class *Port*BindingImpl **implements** *Port*Port {
// italics represent the actual name of the port, e.g. Car
... operation signatures and implementation ...

6.1.2 Extended Translation Strategy

The thesis work has made several extensions to the CRESS notation to support realistic web/grid service composition, namely: dynamic partner definition, dynamic partner binding and invocation, and ownership of types. The extended strategy supports these extensions in the automated implementation strategy, and supports the WS-BPEL 2.0 specification.

6.1.2.1 Dynamic Partner Definition

In a CRESS Rule Box the new **Partner** keyword with the specified partner.port introduces a dynamic partner. The BPEL standard supports specification of dynamic partners, implying runtime binding and invocation. The configuration of dynamic partners is in a separate specification from the BPEL behaviour in ActiveBPEL, which is where the translation comes into effect. A dynamic partner is configured in the ActiveBPEL PDD (Process Deployment Descriptor) of the composite service in the following manner. In its partnerRole element the value endpointReference attribute is set to “dynamic”, and the invokeHandler value is set to “default:Address” as shown below. This is less detailed in comparison to a static partner configuration. In this example, **Partner** mapper.job is specified in the CRESS Rule Box, resulting

in the code generation for PDD file being:

<**partnerLink** name=″mapperJob″ >
<**partnerRole** endpointReference=″dynamic″
invokeHandler=″default:Address″/>
</**partnerLink**>

In addition, an intermediate EndpointReference variable is also created for facilitating bindings to the partner using the naming convention dynamic<*Partner*><*Port*>EPR. This is a mechanism for implicit validation and compatibility harmonisation. The reason and workaround are contextually described in the following subsection where the binding of dynamic partner is covered.

6.1.2.2 Dynamic Partner Binding

The endpoint of a dynamic partner is initially not bound. An assignment to the **Partner** in the CRESS diagram defines its binding in the composition at the point of specification. This implies that the source of assignment is of CRESS-type **Reference**, which will be an EndpointReference when implemented. The thesis work has developed a straightforward translation into BPEL code that specifies an assignment from the source EndpointReference variable to the target partner link. The CRESS assignment expression 'mapper.job <- epr' is translated into BPEL as illustrated in the BPEL code below. As will be seen, an assignment from a variable of EndpointReference to the intermediate dynamicMapperJobEPR variable is made prior to binding the endpoint of a partner. It was found that an assignment made directly to a partner link resulted in an error: the ActiveBPEL container reported unsuccessful schema validation when assigning directly from the source. It was observed that assignments of returned EndpointReference values directly into partner links is accepted, but fail at invocation. It was found that the addressing headers were sent as part of the SOAP body rather than as headers and therefore could not be resolved at the destination. Furthermore, the assigned value retained the structural information of the reference part (e.g. name of <part> in WSDL <message>) that was returned; this is incompatible with partner link types but was not validated. An empirical solution was found. The workaround is an assignment to the intermediate variable of EndpointReference which is then used to set the partner link. This results in addressing information now within SOAP headers, and therefore the endpoints can be invoked.

```
<copy>
  <from variable="mapperReference" part="reference"/>
  <to variable="dynamicMapperJobEPR"/>
</copy>
<copy>
  <from variable="dynamicMapperJobEPR"/>
  <to partnerLink="mapperJob"/>
</copy>
```

6.1.2.3 Type Ownership

The original strategy's translation of data variables applies by default (i.e. a variable is owned by its defining diagram). If variables are explicitly declared in association with the owning partner service via the CRESS diagram Rule Box, then the type ownership translation rule applies. Variables expressed with ownership have their type definitions translated within the XSD schema under the namespace of the specified owner. This namespace is defined by the CRESS domain (WS or GS) configuration diagram. The corresponding WSDL message definitions will be generated accordingly, pointing to the right schema and namespace. These definitions are written to the WSDL file that contains the common data type and definitions. In the generated BPEL specification, these variables are declared with the corresponding message elements.

6.1.2.4 Translation Strategy for WS-BPEL 2.0

CRESS had an existing translation strategy for BPEL4WS, the major constructs being listed in Appendix B. The thesis work has extended this strategy for translation into WS-BPEL 2.0. The WS-BPEL 2.0 standard is an update of its predecessor BPEL4WS in several respects, with significant improvements and refactoring in the language constructs and use of standards compatible with that of WSRF, ensuring compatibility with grid services. The extended translation strategy does not support all the new BPEL updates, but maintains compatibility between BPEL4WS and WS-BPEL 2.0.

One of the main improvements in WS-BPEL 2.0 is variable access. In BPEL4WS the access to a (part of a) variable is via the function call 'bpws:getVariableData' with the variable and XPath query as parameters. WS-BPEL 2.0 simplifies XPath expressions by introducing the '$' and '.' notation, where the former denotes the use of a variable of a given name and the latter is used for access, using the format $variable[.part]/location. The WS-BPEL 2.0

variable access notation, in comparison to BPEL4WS, is an improvement in conciseness and simpler syntax. For example the BPEL4WS syntax for an expression 'offer.price != 100000' is expressed as 'bpws:getVariableData('offer','offer','/offer/price') != 1000000', whilst the WS-BPEL 2.0 is simpler with '$offer.offer/price != 1000000'. Despite the differences in the actual implementation syntax, variable access expressions in CRESS diagrams are not affected at all, as they are semantically the same at a high-level. This the CRESS syntax is more alike to WS-BPEL 2.0 in this respect.

Some constructs in WS-BPEL 2.0 are refactored (mostly renamed) from BPEL4WS, such as **switch/case** to **if/else**, and **terminate** to **exit**. This means that the extended translation to BPEL4WS (Appendix B) could be used as a guideline for translating CRESS diagrams to WS-BPEL 2.0 syntax. As the majority of other constructs are similar, the existing translation strategy to BPEL code needed only small changes to support WS-BPEL 2.0. The source and target links in WS-BPEL 2.0 are defined within <sources> and <targets> elements.

Several BPEL-related namespaces are also different in WS-BPEL 2.0, and these are reflected in the translation strategy. WS-BPEL 2.0 added <import> functionality to support WSDL and XSD formally in the BPEL specification. This gives a direct description of how WSDL and XSD are explicitly used in BPEL. These <import> statements are also automatically and directly generated by the translation strategy as it has information about the WSDL definitions and namespaces.

The WS-BPEL 2.0 standard uses a later version of the WS-Addressing specification. The use of its namespace and its import specification details have been incorporated into the new translation, with the gain of being more compatible with the WSRF specification. This helps to realise the interoperability with grid services more seamlessly, as anticipated by the preliminary investigations with BPEL4WS [91].

6.1.2.5 Inclusion of Grid Partner Service Implementation

Grid partner services are implemented and deployed rather similarly to that of web partner services, illustrated in figure 6.3. The WSDL service interface and common definitions are automatically generated, along with their service stubs. There are however no implementation skeleton and deployment descriptors generated, as the grid service implementation follows the

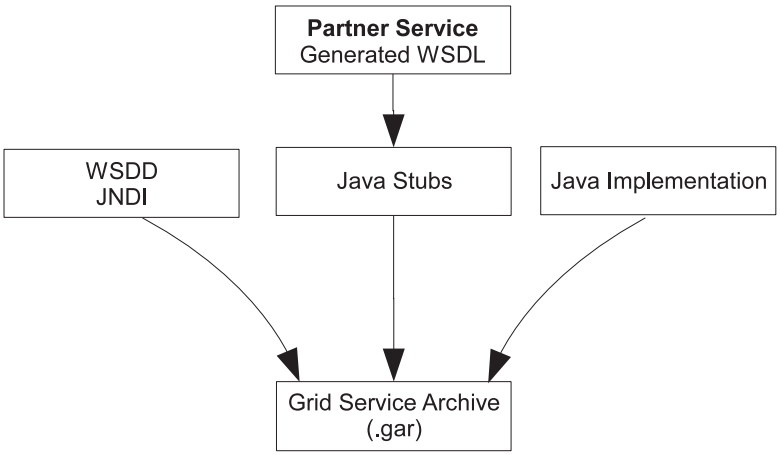

Figure 6.3: Grid Service Partner Deployment Plan

Globus Toolkit development framework which is different from that of web services. The developer fully provides the one-off implementation for each grid partner service, along with the JNDI (Java Naming and Directory Interface) and WSDD descriptors which are required by GT4. Once completed, the CRESS implementation framework was able to automatically compile and package a provided implementation as a grid service archive (.gar) using the Globus Toolkit packages.

The CRESS tools, specifically cress_expand and cress_create were extended by my supervisor Prof. Kenneth J. Turner with the capability to execute this service deployment plan for composed services and grid service partners in the grid service domain. The composed services in the grid service domain are implemented as BPEL archives, which follows the service deployment plan for composed web services. The grid service partners are compiled and deployed as grid service archives as described here.

6.2 Compatibility

The standards used by ActiveBPEL and GT4 have evolved to a mostly harmonised state; however, there is an compatibility that results in an interaction failure between ActiveBPEL and GT4. This has been addressed as described in the following subsection for GT4.

6.2.1 Interworking of ActiveBPEL and GT4

The first interworking issue revolves around the need for ActiveBPEL version 3, which just supported WS-BPEL 2.0, to interact with dynamic resources deployed in GT4 [107]. To illustrate this, a BPEL service and some dynamically created resources are assumed in ActiveBPEL and GT4 respectively. The endpoint reference of the target service is set during the execution of the BPEL process and prior to the target service invocation. This implies dynamic service binding in the BPEL description of the composite service. As part of the invocation, ActiveBPEL will send a SOAP message to the target service (dynamic resource) that is hosted in GT4. ActiveBPEL sends information (the service actor is empty) that results in GT4 not being able to identify the service resource and throws "No Action Header" AXIS faults, even though the required information is present. Even though the information complies with the standards, GT4 is unable to extract the correct WS-Addressing information. GT4 sets the 'addressing.required' property to true in its message context, which originally disallowed this format of SOAP messages sent by ActiveBPEL. A solution was to relax this requirement by setting it to not mandatory, which does not affect other services and the container behaviour. This problem was resolved in ActiveBPEL version 5.0.2 which sends the headers correctly, and is therefore recommended for use.

The second issue is another SOAP-level incompatibility, but from GT4 to ActiveBPEL. A BPEL service and a grid service are assumed in ActiveBPEL and GT4 respectively. Under circumstances where a complex type value is returned by the grid service as a response to the invocation by the BPEL service, an empty namespace in the form of xmlns="" is found in the structure operation response when its contents are empty. This is incorrect according to the validation of SOAP messages which the ActiveBPEL container implements. As a result, although invocation is successful from the perspective of getting a response, the BPEL process will fault due to invalidation of the SOAP response.

6.2.2 SOAP-Level Message Harmonisation

The thesis work has implemented solutions to address these two incompatibilities, driven by factors of minimal development and interference, and maximum impact. This was possible

because of the configurable deployment flexibility offered by AXIS and GT4.

The first issue was addressed by plugging handlers into the AXIS request handler chain of GT4, which is a WSDD file. A handler named CressAddressingHandler was implemented and configured at the top of the existing request handler chain of the GT4 container. This handler simply sets 'addressing.required' to false for the context of requests. As the first handler, the effect of CressAddressingHandler is naturally propagated to the successive handlers configured in the container, and deals with the first addressing incompatibility. A permanent fix in GT4 (version 4.0.1) is viable; however, considering the amount of effort to understand the existing code, to rebuild the container, and the impact of the change on existing deployments, the solution implemented in this work fares better. The solution implemented in the thesis can also be tried out immediately in a more deterministic manner as there is only a single point of change and this configuration can be easily withdrawn when desired.

The solution to the second incompatibility involves creation of a Java class (named CressGlobusAxisRPCProvider) that is to be configured into grid service WSDD files. The purpose of this class is to eliminate the empty namespace problem by filling in the correct namespace obtainable from the namespace of response objects. CressGlobusAxisRPCProvider extends the existing default provider, overriding the existing code that is responsible for creating the SOAP body part, filling in the namespace if it is found to be empty, it then passes message validation. This is also a configurable solution which can be easily withdrawn without other code changes and redeployment.

6.3 Validation using MINT

6.3.1 MINT Notation

An important objective with regard to automated validation was to be able to share the same set of test scenarios in both the formalisation and implementation of service compositions, which is achieved by translating MUSTARD scenarios into MINT. The syntax description below describes the list of MINT keywords which are largely based on MUSTARD notation. The **test** keyword creates a test specified behaviour. Primitive constructs are **send** and **read**. **INT** and **OK** are actual actions executed in a test. Combinator constructs are **sequence**, **interleave**,

and **offer** that provide powerful composite expressions for sequence, parallelism and choice respectively to describe a test. Combinators may also be nested for describing complex tests. Additional syntax expressing variable usage and endpoints is used to support specification of versatile tests.

send(partner.port.op[/epr], value) Invokes the target service partner at an optional endpoint (see /epr_name), the port, operation and the specified parameter. The type of parameter depends on the operation requirements. The response from an invocation will be stored. The value parameter can be a static expression, e.g. **send**(smartmaths.calc.pow, Pow(7, 5)) invokes the *power* operation of partner *smartmaths* via the *calc* port using a complex type Pow which is instantiated with 7 and 5 as parameters (i.e. 7^5, The value parameter can use a variable (see !name).

read(partner.port.op[/epr], [fault,] value) Read from the stored response returned by a specified partner at the optional endpoint (see /epr_name), for the given port and operation. Fault responses are read with the fault name. A read implies there should have been an invocation to the specified partner, port and operation. The stored response should then match what is specified in the value parameter, indicating a successful read. The value parameter can be a static expression, e.g. **read**(smartmaths.calc.pow, 16807) means the stored response should match 16807 returned by partner *smathmaths* via the *calc* port for the operation *pow*. The value parameter can be an arbitrary value of some type (see ?type). The value parameter can be instantiated as a typed variable (see ?name:type). The value parameter can use a variable (see !name).

INT Represents an event internal to the MINT tool which is always a successful action. Used in conjunction with **offer** to describe MUSTARD **decide** semantics.

OK Indicates a path succeeds.

sequence Child constructs occur in sequence.

interleave Child constructs occur with interleaving parallelism, all possible sequences being traversed exhaustively.

offer A choice of child constructs where one branch occurring suffices for a successful path.

test(name, specification) Creates a test with a given name and test specification described as a composition of primitive and (nested) combinator constructs.

?name:type This syntax is only used in a **read** primitive to instantiate a named variable bounded to a type, matching any value of the given type in the read. This implies that the type of the read value is the same as the type of the variable.

read(smartmaths.calc.pow, ?answer:natural) will be successful iff the read value is from smartmaths.calc.pow and the type is Natural. The variable *answer* will then be instantiated with the value read.

?type This syntax is usually used in a **read** primitive, occasionally in **send**. This is like ?name:type with the exception that it does not instantiate a variable. This is often used for a one-off **read** accepting arbitrary values of a specified type.

!name[.field]... This syntax expresses the use of (part of) a variable. !name means a variable that was instantiated. The '.' operator accesses the fields within the associated type structure in succession. Suppose variable cust has type structure {{Natural hse String street}contact Integer balance} customer, then: !cust returns the entire cust record; !cust.balance returns the value of field balance in cust; !cust.contact.street returns the value of street in contact that is in cust; !cust.contact returns the entire contact of cust.

/epr_name This syntax is useful for dynamically binding endpoints to target service partners specified just after partner.port.op. epr_name is a variable name that must be of type EndpointReference, implying a prior instantiation via ?epr_name:EndpointReference When used as part of a send primitive, the endpoint of the service partner is set with the value of the specified EPR variable prior to invocation. When used as part of a **read** primitive, the endpoint serves as a part of the match for a response.

There are three MUSTARD constructs that are not part of the MINT constructs described above: **succeed**, **refuse**, and **decide**. This is because their semantics can be described in a more fundamental form using MINT constructs and keywords, described in table 6.1. The user does not have to manually adapt existing MUSTARD validation specifications in order use MINT to execute the implementation validation. A macro script was developed, using M4, in conjunction with the thesis supervisor to translate MUSTARD tests into MINT notation automatically. This facilitates the flow of automated validation, with an intermediate step of translating MUSTARD test scenarios into MINT notation with the macro script prior to MINT interpreting and executing the tests.

6.3.2 Tool Support

MINT is also a tool that interprets and executes test scenarios automatically. This section describes the design of the tool and its capabilities for implementation validation of the composite web/grid services and service partners that are developed and deployed. A guide to MINT is available on its web page [90].

6.3.2.1 Tool Design

The approach towards automating implementation validation with the MINT tool is shown in figure 6.4. The MINT tool requires the following inputs: service configuration files, and MINT notation test files. A service configuration is a properties file that contains information that relates to the actual implementation: service location, timeouts, and abstract mappings

MUSTARD Construct	MINT Description
succeed	**OK** appended as leaf node, i.e. the sequence of behaviour must succeed, e.g. **succeed**(**send**(params), **read**(params)) is specified in MINT as **sequence**(**send**(params),**read**(params),OK)
refuse	A sequence where last action is replaced with an offer containing the last action and a branch of **OK**, e.g. **refuse**(**send**(param1), **read**(param2)) is specified in MINT as **sequence**(**send**(param1), **offer**(**read**(param2), OK))
decide	An offer with all the branches beginning with **INT** (internal event) e.g. **decide**(**sequence**(**send**(param1), **read**(param1)), **sequence**(**send**(param2), **read**(param2))) is specified in MINT as **offer**(**sequence**(**INT**, **send**(param1), **read**(param1)), **sequence**(**INT**, **send**(param2), **read**(param2)))

Table 6.1: Representation of MUSTARD Combinators in MINT

to actual implementation classes. The MINT tool uses this information to set up and execute the tests automatically. Service interfaces are downloaded from actual implementations, and service stubs are generated and compiled. The scenarios are interpreted by MINT, where the execution is underpinned by the actual implementation of the stubs. The parameters of a service configuration file are described in table 6.2, written in Java properties format *name=value*.

A deployed web/grid service is validated as follows. MINT downloads and generates services stubs for all the service interfaces from the locations specified in the service configuration property files. MINT delegates this activity to the AXIS WSDL2Java class which exists for the purpose of generating Java service stubs for communicating with target services. These stubs are immediately compiled by MINT (using the Java compiler) for actual use. These two steps set up the necessary technical framework for executing the tests.

MINT then interprets the test files and parses the results. MINT executes each test on-the-

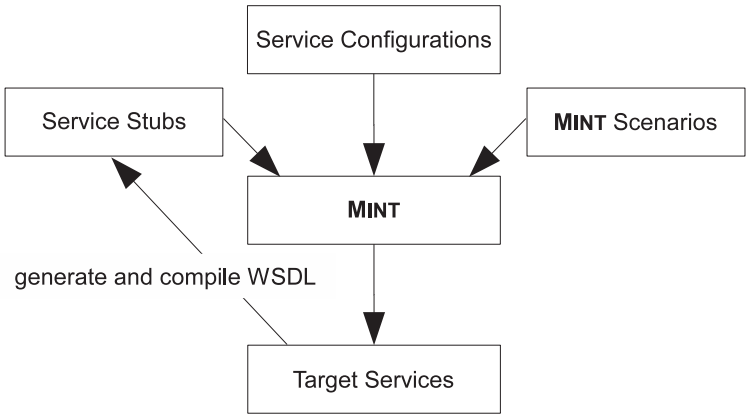

Figure 6.4: MINT Approach to Automated Implementation Validation

fly using the recursive algorithm described in table 6.3. The MINT algorithm uses initials, potentials and (success, complete, failure) path trace results for execution, achieving the same interpretation as validation of a specification with MUSTARD.

An initial is a MINT primitive action (**INT**, **send**, or **read**) representing the first possible step at the current point in test execution. Progressing from an initial results in post-initial behaviour. This represents contextually the state of the test description considering actions already performed, determining the actual visibility of the following initials. The following description illustrates the initials and post-effect of progressing an initial (post-initial) of a MINT combinator construct.

sequence: Union of all initials of the first child construct. If the first child is a primitive then the child is the only initial.

initials:
sequence(A, B) -> A
sequence(**offer**(A, B), C) -> A, B
sequence(**interleave**(A, B), C) -> A, B

Post-initials:
sequence(A, B) -> **sequence**(B)
sequence(**offer**(A ,B), C) -> **sequence**(C) having done A or B
sequence(**interleave**(A, B), C) -> **sequence**(**interleave**(B), C) having done A *or*
sequence(**interleave**(A), C) having done B.

offer: Union of all immediate child primitive constructs and the initials of each child combinator construct.

initials:
offer(A,B) -> A, B
offer(**sequence**(A, B), **sequence**(C, D)) -> A, C

Configuration Property	Purpose
target.url	The (remote) location of the service which will be subjected to implementation validation. This is also the root location where MINT will download the WSDL service interface from, thereafter translating it into the client stubs with AXIS or GT4 *wsdl2java* for actual use.
service.timeout	This is the timeout in milliseconds which will be set into the client stub prior to invocation. The AXIS-generated client stubs contain operations to set the timeout parameter. The default value is 0 meaning no timeout.
service.package.*partner*	Specifies the Java package name in which the service stubs reside. E.g. service.package.lender=LoanStar if Lender WSDL service interface will generate stubs into the LoanStar package.
class.*partner.port.operation* .fault.*faultname*	Specifies the fault class that is associated with a fault used in MINT. E.g. class.lender.loan.quote.fault.refusal = LoanStarDefs.StringMessage refers to a 'refusal' fault used in a MINT scenario with the StringMessage class.

Table 6.2: MINT Service Configuration File Properties

offer(**interleave**(A,B), **sequence**(X, Y)) -> A, B, X

Post-initials:
offer(A,B) -> empty
offer(**sequence**(A, B), **sequence**(C, D)) -> **sequence**(B) having done A OR **sequence**(D) having done C

offer(**interleave**(A,B), **sequence**(X, Y)) -> **interleave**(B) having done A OR **interleave**(A) having done B OR **sequence**(Y) having done X

interleave: Union of all immediate child primitive constructs and the initials of all child combinator constructs.

Initials:
interleave(A, B, C, D) -> A, B, C, D
interleave(**sequence**(A,B), **sequence**(C,D)) -> A, C
interleave(**offer**(A,B), **sequence**(C,D)) -> A, B, C

Post-initials:
interleave(A, B, C, D) -> **interleave**(B, C, D) having done A *or*
interleave(A, C, D) having done B *or*

interleave(A,B, D) having done C *or*
interleave(A,B,C) having done D

interleave(**sequence**(A,B), **sequence**(C,D)) ->
interleave(**sequence**(B), **sequence**(C,D)) having done A *or*
interleave(**sequence**(A,B), **sequence**(D)) having done C

interleave(**offer**(A,B), **sequence**(C,D)) ->
interleave(B, **sequence**(C,D)) having done A *or*
interleave(A, **sequence**(C,D)) having done B *or*
interleave(**offer**(A,B), **sequence**(D)) having done C

Potentials are initials that can indeed be actualised at the current point in behaviour, indicating possible paths that can be followed during execution. For example, a **read** is not possible if there are no prior replies to read. When a potential action is performed successfully, it is considered actualised. If the action does not actualise, then it is appended to the path trace which is considered a failure path. For example a **send** has always potential to execute but the actual invocation might result in exceptions such as service not found, therefore it cannot actualise.

If a path trace leads to the special **OK** event it is added to the set of successful paths. Otherwise, a path trace that has no following initials but also does not end with **OK** is considered complete, which is distinct from being a successful path. Complete paths offer diagnostics that inform about possible test paths but with no assertion of success or failure. The algorithm recurses for an actualised pathway with the current path trace and the post-initial behaviour, until it is concluded with one of the three categories of path traces (success, failure, or complete).

Actualised actions can be reversed via an operation name prefixed with 'reset', if such an operation exist. This is a function specifically for potentially backtracking the state of an implementation system during test execution. The reset function is discussed in section 6.3.2.2.

MINT maintains a buffer storing responses from performing **send**s (service operation invocations). The buffer is read by **read** actions that match the next immediate response value in the buffer. MINT also maintains a space where named variables are instantiated with their associated values. A variable is instantiated on a read with the given name, as defined by the MINT notation.

The criteria for considering if an action can be actualised is the following. **INT** (internal event) will always actualise successfully as it emulates an internal event typically used for

```
execute (path_ trace, test)
   initialise results
   get <initial_ action, post_ initial_ test> tuples from test
   potential_ tuples = filter(tuples)
   foreach potential_ tuple
      if actualise(potential_ tuple.initial_ action)
         if initial_ action is OK
            results.success_ paths.add(path_ trace + initial_ action)
         else
            if post_ initial_ test of potential_ tuple is null
               results.complete_ paths.add(path_ trace + initial_ action)
            else
               child_ results = execute(path_ trace + initial_ action,
                                        potential_ tuple.post_ initial_ test)
               if child_ results is empty
                  results.failure_ paths.add(path_ trace + initial_ action)
               else
                  results.success_ paths.add(child_ results.success_ paths)
                  results.complete_ paths.add(child_ results.complete_ paths)
                  results.failure_ paths.add(child_ results.failure_ paths)
               end-if
            end-if
         reset(initial_ action)
         end-if
      else
         results.failure_ paths.add(path_ trace + initial_ action)
      end-if
   end-foreach
   return results
end execute
```

Table 6.3: MINT Pseudo-Algorithm

directing a path definitely tried, for example in the case of specifying a mandatory attempt on all branches of a given path (e.g. **decide** which is simplified into **offer** with **INT**). A potential **send** representing a service operation invocation is actualised iff the service invocation succeeds. A potential **read** will always actualise, as the criteria for a determining if a read is potential is that there is a match in the next element of the response buffer.

6.3.2.2 Reset Feature

The reset feature is a specialised capability of MINT which aims to backtrack the state of a service. Though some services are stateless, some services maintain state either explicitly

or implicitly via other components. Transactions on databases are an example, even though the using service might indeed be considered stateless. This may result in unexpected false interference and inconsistency between tests and their order of occurrence if unperceived by the test specifiers. Suppose a car rental service has an early bird discount for the first three customers. The first three invocations (**send**s) to the rental service get the early bird response, and subsequently the service stop offering the discount. The following test specification, if executed several times (e.g. for a load test) results in inconsistency:

test(Test1,
 succeed(**send**(car.rental.book, Car('John Smith, 'BMW, '3 days)),
  read(car.rental.book, Booking('Discount, 210.75))))

If the service is working correctly, the test will not pass from the fourth time onwards. If the service is implemented wrongly in that it contradicts the requirement of only three early bird offers, the test is consistent but is a false positive. Similar situations can also happen when several tests are executed in sequence.

MINT provides a simple reset feature demonstrating the benefit of state rollback if possible, which can ensure test independence as previous state changes are removed to avoid future interference. The reset feature only needs to be applied to **send** constructs as they are invocations that influence services directly. To reverse a **send**, MINT searches the same service partner port for the operation name used in the **send** but prefixed with a 'reset'; this takes the same parameter as the **send**. An invocation is then made to the reset operation using the value specified in the **send** as a parameter. By doing this, the execution of a test is then made independent of other tests.

Is it possible to specify reset operations during composition in a pluggable manner? The idea of the reset feature also inspired another part of the thesis technique to readily pluggable reset operations when specifying composite services in CRESS. Usually a service would have specified explicit compensating operations, for example a 'cancel' operation to void a reservation already made. However the reset technique provides flexibility in terms of development and validation in the absence of these operations.

CRESS supports templates whereby features can be defined and added to specifications when necessary. The development of templates was initially intended for modelling telephony

features. The thesis work has used CRESS templates as a novel way for flexible addition of functionality in composing web/grid services, whereby features can be added as required. Corresponding reset operations for operations specified in CRESS diagrams can be defined using a CRESS feature diagram which is a separate diagram from the main behaviour. The feature is specified in the CRESS configuration diagram, indicating that the feature behaviour to be included in the automatic translation. This way the reset operation features are defined separately from the main specification, and can be easily pulled out by reconfiguration.

6.3.2.3 Diagnostics

Diagnostic test execution is similar to that of formal validation with MUSTARD. All possible paths in a scenario are considered and tried. The outcomes of the path results are consolidated as a final outcome of the scenario: fail, inconclusive, or pass. A consolidated outcome is accompanied by test execution time. A fail outcome indicates that all paths have failed for the specified scenario. An inconclusive outcome indicates a mixture of path failures and path successes. A pass indicates all the paths are successful. In the event of a scenario not achieving a pass outcome, path traces are produced for analysis.

Path traces that compromise test intentions are printed textually, showing the paths until the point of failure. These are similar to the formal validation path traces. The diagnostics help to narrow down the points of failure, giving feedback to developers who can readily address them. MINT provides additional diagnostic output in view of engaging with implementations where operation timeouts are potential failure points. This gives useful feedback in terms of service status and system configuration. For example, the following are diagnostics of a validation –reported ConnectException, which indicates that the invocation failed because it was not able to establish a connection to the service.

```
Test APPROVER High Rate ...
Execution error of
send(approver.loan.approve,Proposal2("Nancy Turner","Manchester England",14999.))
    caused by java.net.ConnectException: Connection refused: connect
Cause is java.net.ConnectException: Connection refused: connect
Fail                              0 succ1 fail2.0 secs

Warning:Cause is java.net.ConnectException: Connection refused: connect
```

```
send(approver.loan.approve,Proposal2("Nancy Turner","Manchester England",14999.))
    <failure point (Cannot connect to service)>
```

6.3.2.4 Performance Evaluation

Load tests provide another level of evaluation of an actual service deployment. MINT was also developed with the intent to run many test scenarios on a target service, using the same MUSTARD test specifications to evaluate performance. MINT supports two modes of load or performance testing: sequential and concurrent. Both modes are defined by a positive number that indicates the number of times for test scenario execution. A test can performed successively for the specified number of times in sequential mode, or simultaneously in the case of concurrent mode. Each test run is instantiated as a Java thread. The final outcome of each test run belongs to the same test specification (fail/inconclusive/pass). These results are aggregated into the performance diagnostics, comprising average completion time, fastest and slowest completion time, number of different outcomes, and finally a consistency flag indicating identical outcomes throughout. The performance diagnostics provide insight into the service deployment such as settings for supporting the desired simultaneous requests. The following is an example of a successful performance evaluation with a concurrent load of 150 runs, of which all runs of the same scenario are successful (no inconclusives or failures), with an average completion time of 4.5 seconds, the fastest completion time at 3.6 second, and the slowest at 4.8 seconds.

```
Test APPROVER Low Rate ...          Pass                150 succ 0 fail      4.5 secs
                              Concurrent   0 inco true cons 3.6 secs .. 4.8 secs
```

The following diagnostic reports inconsistency in the performance mode in that some runs of the same scenario passed and some failed. The captured errors are given (ConnectException and SocketTimeoutException) reported by some of the failures.

```
Test LENDER Little Low Risk ...
Execution error of
send(lender.loan.quote,Proposal("Nancy Turner","Manchester England",9999.))
caused by java.net.SocketTimeoutException: Read timed out
...ConnectException: Connection refused: connect ...
Inconclusive        129 succ            21 fail 8.1 secs
                              Concurrent            0 inco false cons 3.5 secs .. 8.9 secs
```

6.3.3 Examples

6.3.3.1 Acceptance Test

As a deterministic example, a maths service providing a natural number power operation should always return the right calculation. The *pow* operation takes in a complex type PowParam that contains the natural and the power. The first test description is a MUSTARD test specification that asserts success if the natural number 16807 is received as a response to the *pow* invocation using a PowParam containing fields 7 and 5 (i.e. 7^5). This is followed by the translated MINT description which is executed in sequence.

```
test(SEVEN_FIVE,
  succeed(send(math.calc.pow, PowParam(7,5)),
    read(math.calc.pow, 16807)))
```

translates to

```
test(MATH_SEVEN_FIVE,
  sequence(send(math.calc.pow,PowParam(7,5)),
    read(math.calc.pow,16807),OK))
```

As a deterministic example, a weather service provides an operation that returns a simple textual forecast (sunny, cloudy, showers) for the given hour of the current day at the specific location. The test scenario will be successful iff the forecast on 20091225 returns either sunny, cloudy, showers or a metFault with "sensor faulty" message as an offer response. The MUSTARD description is first listed, followed by the MINT description.

```
test(FORECAST_OUTPUTS,
  succeed(send(met.weather.forecast,'20091225),
    offer(read(met.weather.forecast,'sunny),
    read(met.weather.forecast,'cloudy),
    read(met.weather.forecast,'showers),
    read(met.weather.forecast,metError,'sensor faulty))))
```

translates to

```
test(MET_FORECAST_OUTPUTS,
  sequence(send(met.weather.forecast,"20091225"),
    offer(read(met.weather.forecast,"sunny"),
    read(met.weather.forecast,"cloudy"),
    read(met.weather.forecast,"showers"),
```

read(met.weather.forecast,metError,"sensor faulty")),
OK))

6.3.3.2 Refusal Test

The example refusal test specifies that a client from Stirling should not get an outright rejection for a loan application regardless of the rejection message; rather, the loan application must be successful. Should a response be a reject fault with arbitrary message (the use of ?String matches any String value), then the test will not pass. The translation follows the semantics of the MUSTARD **refuse** construct, where the last action becomes a child of an **offer** with **OK** as an alternative. If the **read** actualises, it becomes a complete path but is not successful. If it does not actualise, the only initial that can be offered is **OK** which then asserts that the **read** did not actualise.

test(STIRLING_CLIENT_NO_REJECT,
  refuse(**send**(bank.personal.credit, Loan('John,'Stirling Scotland, 1000.)),
    read(bank.personal.credit, reject, ?String)))

translates to

test(BANK_STIRLING_CLIENT_NO_REJECT,
  sequence(**send**(bank.personal.credit,Loan("John","Stirling Scotland",1000.)),
    offer(**read**(bank.personal.credit,reject,?String),
    OK)))

6.3.4 Tool Integration

The integration of MINT is similar to MUSTARD and CLOVE, which are according to the CRESS methodology. The existing CRESS and CHIVE integration of formal validation, that is MUSTARD, can be reused but for implementation validation instead. Figure 6.5 illustrates the integration as well as the control flow of the four approaches to execute implementation validation and performance evaluation.

By integrating MINT into MUSTARD, the validation feature was broadened to both formal and implementation with reduced effort. The interface to MINT through MUSTARD is analogous to Lola's interface through MUSTARD. The MUSTARD tool was modified to switch between formal validation and testing with MINT. The MUSTARD command-line criterion to

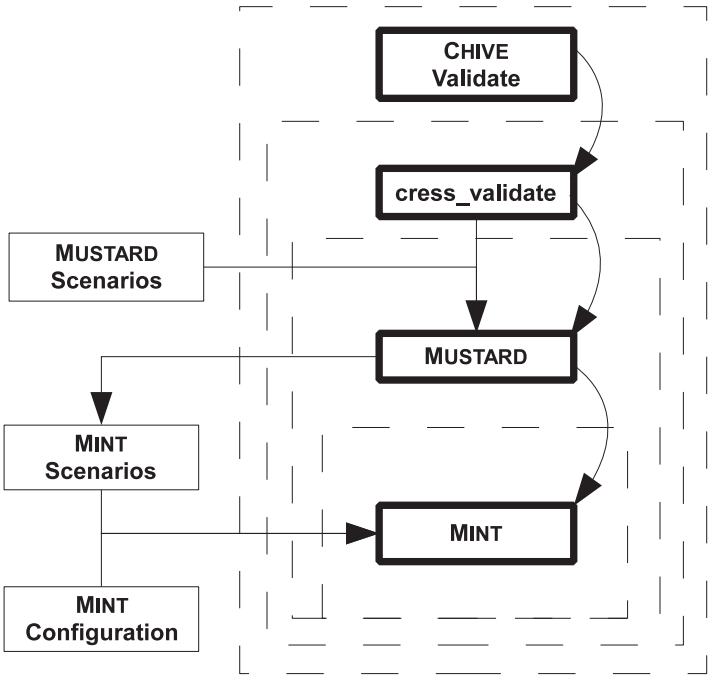

Figure 6.5: Integration of Implementation Validation Tools and Execution Approaches

perform implementation validation is the presence of the diagram file name with the '.bpel' suffix. MUSTARD will translate the .mstd test specification files into MINT-compatible form prior to invoking MINT to execute the actual implementation validation. The cress_validate tool was given a minor extension with the capability to direct MUSTARD to formal or implementation validation based on the respective specified target languages LOTOS or BPEL. If the target language in CHIVE's Preference is LOTOS then formal validation will be carried out, cress_validate will invoke MUSTARD. If BPEL, then MUSTARD will invoke MINT. No extensions were required in CHIVE to use the implementation validation feature as the existing integration invokes cress_validate with the target language that was set in CHIVE's Preferences. By setting the target language in CHIVE to BPEL, the same Validate menu item will invoke cress_validate. This will perform implementation validation by directing MUSTARD to invoke MINT; the results of validation will be displayed in a dialogue box.

6.4 Evaluation

The thesis developed an approach to meet the objective of automated implementation. The seamless orchestration of composite web and grid services is possible, including the capability

for implementation validation, within an integrated development methodology. Implementation is fully automated for composed web/grid services, and now supports the BPEL4WS and WS-BPEL 2.0 standards. The development methodology provides a framework with a service deployment plan whereby implementation of partner web/grid services is automated as much as possible.

The grid service domain added to the CRESS framework to support the creation of composed and partner grid services. The web and grid service domain support automated implementation for composed and partner services, underpinned by translation strategies and service deployment plans, resulting in readily deployable services for their respective hosting environments (ActiveBPEL and GT4). The thesis work has developed and implemented a translation strategy for latest WS-BPEL standard for composed web/grid services whereby code is fully and automatically generated from CRESS diagrams, supporting also the CRESS notation that was extended by the thesis for realistic service compositions such as dynamic partners and type ownership.

The compatibility issues between ActiveBPEL and GT4 have been addressed effectively, leading to seamless service interoperability in their respective hosting environments. The thesis work developed solutions for both issues which were implemented in GT4. These solutions that do not interfere directly at the implementation code of the hosting environment but at the configuration level, which is neater than providing an adapted container; and the solutions give developers configuration control. The solutions are general and simple, involving only two Java classes with only a few lines of code which are packaged as Java archives (JAR files) for immediate use apart of the integrated methodology developed by the thesis.

The thesis work has also led to MINT which automates functional and performance evaluation for (composite) web/grid services, providing support for post-development evaluation. The aspect of automated implementation testing and performance evaluation has advantages as part of an integrated development methodology for creating composed web/grid services. Firstly, it shares the definitions for validation with the formalisation aspect, meaning that the same set of analysis performed for the design phase is performed at the post-deployment phase. This enables developers to ensure the same confidence level in the functionality. Secondly, validation specification is abstract using MUSTARD and MINT notation. Analysts can focus on

specifying scenarios and support choices, non-determinism, interleaving parallelism, dynamic partners, and variables. these require low-level programming in testing tools such as JUnit. Diagnostics are automatically generated, narrowing down the potential areas of errors. Performance evaluation provides insight into better resource configuration when services undergo stress tests. This can reveal feedback on service functional consistency and response times (fastest, slowest, average).

The integrated methodology, specifically in the implementation aspect, provides an environment where an implementation is automatically obtained with support for post-deployment analysis. This offers a thorough development lifecycle for creating composed web/grid services. Certainly there is room for improvement and several practical and interrelated suggestions can be made, namely in: security, reset capability, implementation validation technique, and support for more service orchestration engines.

Security is an important aspect in distributed computing, protecting resource owners and communication by facilitating access at various levels (such as authentication, authorisation, trust federation, etc.) and encrypting modes. Security is used in realistic development of (composed) web and grid services. Supporting security within this integrated methodology extending to automated implementation and testing will support more realistic applications.

The 'reset' feature used for backtracking service states is quite rigid in that the corresponding reset operations are only bound to names starting with 'reset' followed by the name of the operation used in the **send**, and also using the same parameters. This aspect can be improved with flexibility by having support to allow analysts to configure reset operation names and the construction of parameter types and values. This could be totally different from the operation signature that initially influenced service state.

Automated implementation validation using MINT can be improved in several aspects. Currently evaluations of tests are provided after the execution. This can be improved with real-time response and detailed feedback, for example by showing the progress during execution: the part of the behaviour currently being executed; time taken for each single part of a test; graphical visualisation of test execution. The performance evaluation is currently carried out from a single point which may not be suitable to carry out stress tests on a large scale, especially with concurrent testing which incurs more resources with more executions. This can

be improved with a distributed approach where the test executions are remotely coordinated across different machines, and the test statistics are then consolidated for analysts.

There are many orchestration engines or workflow enactors available, varying in their capabilities and advantages. Automated support for more target implementations enables developers to test and choose the deployment environments that will suit the service requirements. A constructive support strategy should be considered, favouring implementation environments that use open standards and are widely used by developers, for example OMII-BPEL and Oracle BPEL Process Manager. These two are potential candidates for the reason that they already support the widely adopted BPEL standard and are highly used, respectively in research and commercial activities. OMII-BPEL provides a BPEL workflow enacting environment to support scientific research that may involve large magnitudes of processes, enabling large scale research collaboration enabled with WS-Security mechanisms. Oracle BPEL Process Manager provides commercial infrastructure support to users, ensuring stability and performance which is one key business-critical requirements highly sought after in the commercial arena. Extending support for automated implementation to a variety of service orchestration engines in the integrated methodology will enable developers to readily exploit their respective benefits, and the benefits of the methodology such as automated implementation validation. Other environments, standards and approaches such REST services will be considered in future developments.

Chapter 7

Case Studies

7.1 Introduction

Web and grid service compositions are used in combination in two case studies to demonstrate each individual aspect of the integrated methodology from the perspective of the development lifecycle. The case studies here are a subset of those developed during the thesis work. The case studies have been realised using the Apache Tomcat servlet container for the deployment of web partners (AXIS) and BPEL services (ActiveBPEL), and using the GT4 container for grid partner services. A more detailed explanation of these case studies are available in [89], which includes their automatically generated LOTOS specifications, implementations (BPEL, WSDL, and deployment descriptors), translated validation scenarios (LOTOS) and verification properties (μ-calculus), and SVL verification scripts.

7.2 Development Of Composed Web Services

LoanStar is a *lender* service that is similar to the classic loan approval process used as an example in the BPEL4WS standard [4]. LoanStar is a business process that combines two individual partner services FirstRate and RiskTaker as part of its behaviour.

FirstRate is the *approver* that evaluates loans and returns a loan rate when a loan is approved. FirstRate operates as follows (purely as a hypothetical example):

- if the applicant's name starts with Ken, the loan rate is 3.7%

- otherwise, the loan rate of 4.1% is given if the applicant's address includes Scotland
- otherwise, the loan rate of 4.4% is given if the proposal amount is less than 15000
- otherwise the loan is refused with "loan unacceptable" as the reason.

RiskTaker is an *assessor* that assesses the risk of a proposal request, and returns the risk as a string. RiskTaker operates as follows (purely as a hypothetical example):

- if the applicant's name ends with Turner, the risk is low
- otherwise, if the applicant's address is a UK address, the risk is medium
- otherwise, the risk is high.

LoanStar combines FirstRate and RiskTaker in the following business logic. Loan approval evaluation is assumed to be costly, therefore an approval request is only made to FirstRate for proposals having an amount greater than or equal to 10000. For proposals seeking loans of 10000 or less, a risk assessment is made through the RiskTaker. If RiskTaker evaluates a low risk, then LoanStar immediately approves the loan proposal with a rate of 3.5%. All other risk levels will direct the proposal request to FirstRate for approval.

DoubleQuote is a *supplier* service that involves two car dealer partners: BigDeal (*dealer1*) and WheelerDealer (*dealer2*). All three services are to be developed under the assumption that there is no specific service ownership. Both dealers have a variety of offers for cars, giving the price and the delivery as quotes for car needs. The DoubleQuote business process starts with a car need request which comprises the customer's name, address, and the car model. Upon receiving the need for a car, DoubleQuote will seek quotations from both dealers. A quote offer contains a reference number, dealer identifier, price and delivery (in days). The returned quotes are compared by price, favouring the cheaper. If the prices are identical, then the quote offering faster delivery is favoured. The dealer for the chosen quote is then invoked with an order by DoubleQuote. The offer is returned to DoubleQuote's customer as a reference. The DoubleQuote service allows its customers to cancel the car order, which leads to arranging the car order cancellation with the associated dealer.

BigDeal (*dealer1*) has the following offers for cars (purely as a hypothetical example):

- Mondeo: price 20000, 15-day delivery
- A5: price 33000, 30-day delivery

- Megane: price 11000, 5-day delivery
- Others: price 1000000, 0 day delivery (indicates no offer)

WheelerDealer (*dealer2*) operates as follows (purely as a hypothetical example):

- Mondeo: price 20000, 10-day delivery
- A5: price 35000, 20-day delivery
- Astra: price 18000, 30-day delivery
- Others: price 1000000, 0-day delivery (indicates no offer).

CarMen is a *broker* service that combines DoubleQuote and LoanStar services to offer a car purchase complete with a financing solution. CarMen organises the car order with DoubleQuote followed by requesting a loan at LoanStar. A loan request to the nearest rounded figure to LoanStar will only proceed should the desired car be ordered. If the loan is refused by LoanStar, then the car order made with DoubleQuote is cancelled. A successful coordinated car-finance purchase then returns schedule information to the customer. The schedule contains the order reference, car dealer, price of car, delivery period, and the loan rate. These are all obtained from the information returned by DoubleQuote and LoanStar.

Figure 7.1 shows the process of the service development for the web services case studies. The development activities are labelled with corresponding section numbers (in round brackets) for the methodology given in section 3.3. The developer first describes the composed behaviour in CRESS, provides the partner specification and implementation for Approver (FirstRate) and Assessor (RiskTaker), and specifies the MUSTARD scenarios and CLOVE properties. The specification is then automatically generated for analysis. The developer executes the automated validation and verification, analyses the results, and addresses any issues identified by the analyses. Once satisfied with the analyses, the developer starts the automated implementation. The developer provides the actual implementation for the code skeletons generated for partner services Approver and Assessor. The developer now performs automated implementation validation and performance evaluation for the deployed services using the same MUSTARD scenarios as for the specification, addressing implementation and resource configuration issues that are discovered. This development methodology applies similarly to Supplier (DoubleQuote), where Dealer1 (BigDeal) and Dealer2 (WheelerDealer) are the partner services. As

partner services of Broker (CarMen) are CRESS-described services, there is no need for their manual specification and implementation.

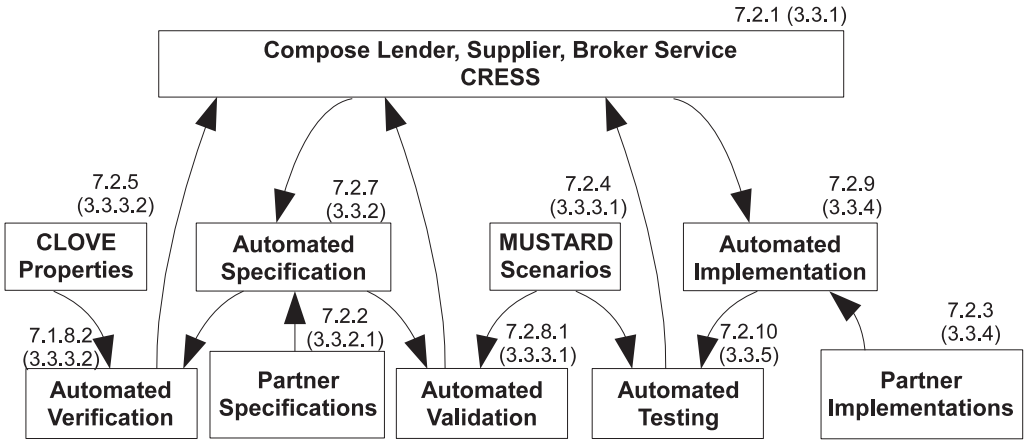

Figure 7.1: Composite Web Service Development

7.2.1 Service Diagrams

The LoanStar, DoubleQuote and CarMen composed service behaviours use the diagram names lender, supplier, and broker in their CRESS root service diagrams illustrated in figures 7.2, 7.3 and 7.4 respectively. FirstRate, RiskTaker, BigDeal and WheelerDealer use the diagram names approver, assessor, dealer1 and dealer2. These service names are also used in the web service configuration diagram.

7.2.1.1 Lender

The Lender service behaviour (figure 7.2) is described as follows. In its rule box definition, the data type and variable proposal2 are declared with Approver as owner, implying Approver's namespace, using the syntax 'proposal2:approver'. This is the input data structure for the approver.loan.approve operation. Lender also defines its own Proposal type with identical structure but using the Lender namespace, the syntax omitting '*:owner*', which defaults to Lender as the current diagram. By doing so, Lender has 're-packaged' the entire service representation for its client, hiding away information about partners for the reason of trade secrets. Lender makes the 'proposal2' variable assignment from its 'proposal' data type in order to invoke the Approver quote operation. The Assessor service has the same (LoanStar) owner and is developed with the Lender's namespace. Following these definitions are three variables named *risk*,

rate and *error*, which are of the CRESS primitive types String and Float. A constant/macro *basicRate* is defined with the float value 3.5.

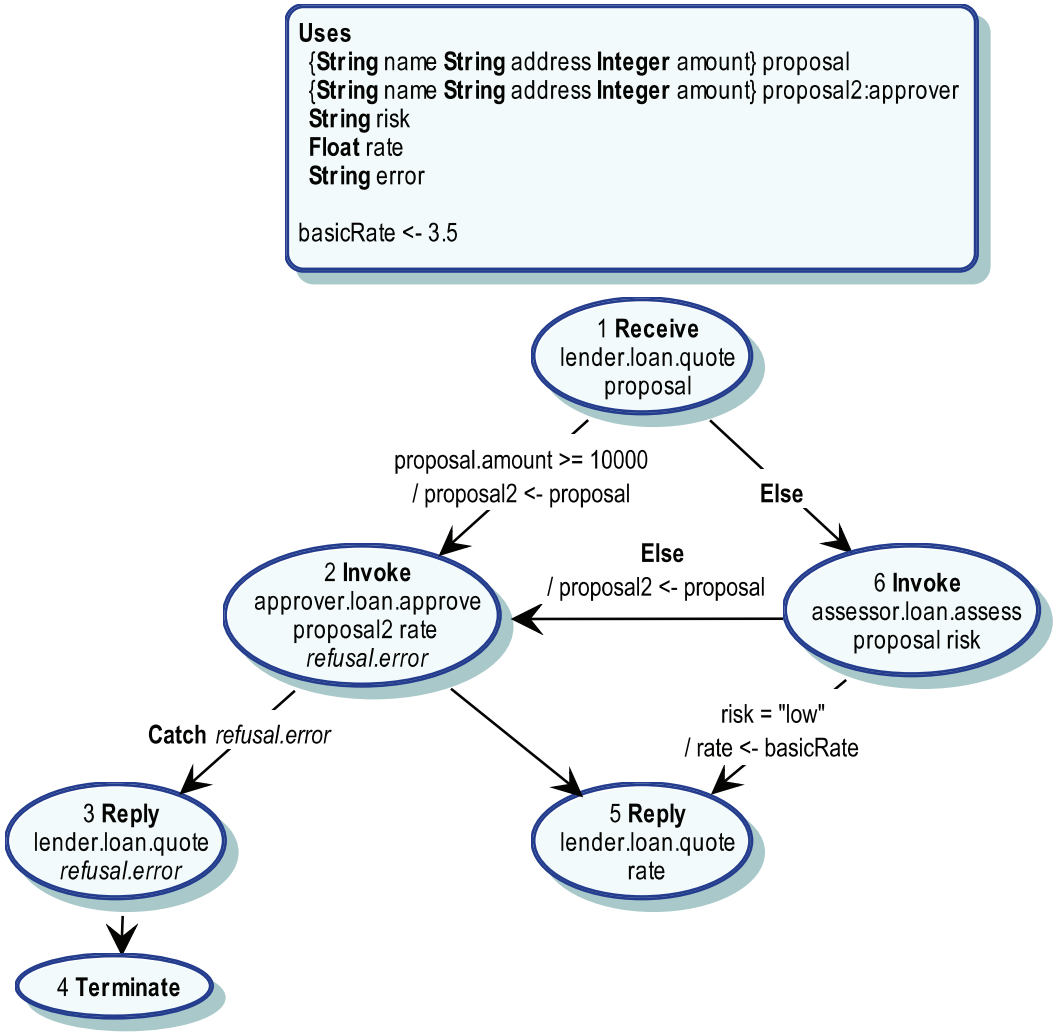

Figure 7.2: Lender CRESS Diagram

The process behaviour starts from node 1, where Lender receives a loan quote request through the lender.loan.quote operation with a *proposal* data value. The value guard from node 1 to node 2 is followed if the proposal amount is 10000 and above, and an assignment is made to set the value of the 'proposal' variable to 'proposal2', which is used in the invocation of 'approver' at node 2. If the invocation at node 2 throws a fault with name 'refusal' and a String type value, the arc labelled with '**Catch** *refusal.error*' is followed, leading to node 3. In this arc, the fault value of the String is set into variable *error*. The lender.loan.quote **Reply** returns the fault (node 3) and the process terminates (node 4). The arc to node 5 is followed if the

invocation at node 2 is successful – the lender.loan.quote **Reply** returns the loan rate. At node 6, the Assessor is invoked for risk evaluation. A "low" risk satisfies the risk = "low" guarded arc, with an associated assignment of 3.5 to the rate: node 4 then returns. Otherwise the '**Else**' is followed, with an assignment made to 'proposal2' for the invocation to Approver at node 2.

7.2.1.2 Supplier

The Supplier service behaviour (figure 7.3) is described as follows. The rule box specifies two user-defined data types *need* and *offer*, which are also named variables used in the service. The *need* is a complex data structure comprising name, address, and model, all of **String** type. The *offer* is also a complex data structure comprising: **Natural** *reference* as order reference number; **String** *dealer* which is the name of the dealer who offers the car deal; **Float** *price* as the cost of the car; and **Natural** *delivery* which is the number of days to deliver the car. The variable *offer2* also uses the same data type as *offer*.

Supplier has a *car* port with two operations named *order* and *cancel*, which are for executing the process of car ordering and order cancellation respectively. Both dealers have a *car* port which has three operations named *quote*, *order*, and *cancel*, respectively for obtaining car quotations, ordering cars, and cancelling car orders. The Supplier's order operation and both dealer's quote operations are synchronous. All other operations are asynchronous, having only input and no output. The use of the data definitions in these service operations is described in the nodes of the service behaviour description.

Nodes 1 and 10 are the entry points of the Supplier's service behaviour, which are for its clients to order cars and cancel orders respectively. As there is more a one starting point, these nodes are therefore specified as branches from the **Start** node. Upon receiving a *need* for the car at node 1, simultaneous requests are made to both dealers for quotes described from nodes 2 to 5. Node 2 explicitly defines a **Fork** with parallel branches outgoing to nodes 3 and 4, indicating the simultaneous invocation of dealer1.car.quote and dealer2.car.quote operations. These return offer-typed data values as variables *offer* and *offer2* respectively. The parallel invocations are synchronised back to the Supplier's business process as successful execution at this point, with the **Join** condition '3&&4' at node 5 thereby completing join (i.e. nodes 3 and 4 must complete successfully). The guarded arc from node 5 to node 6 compares the two

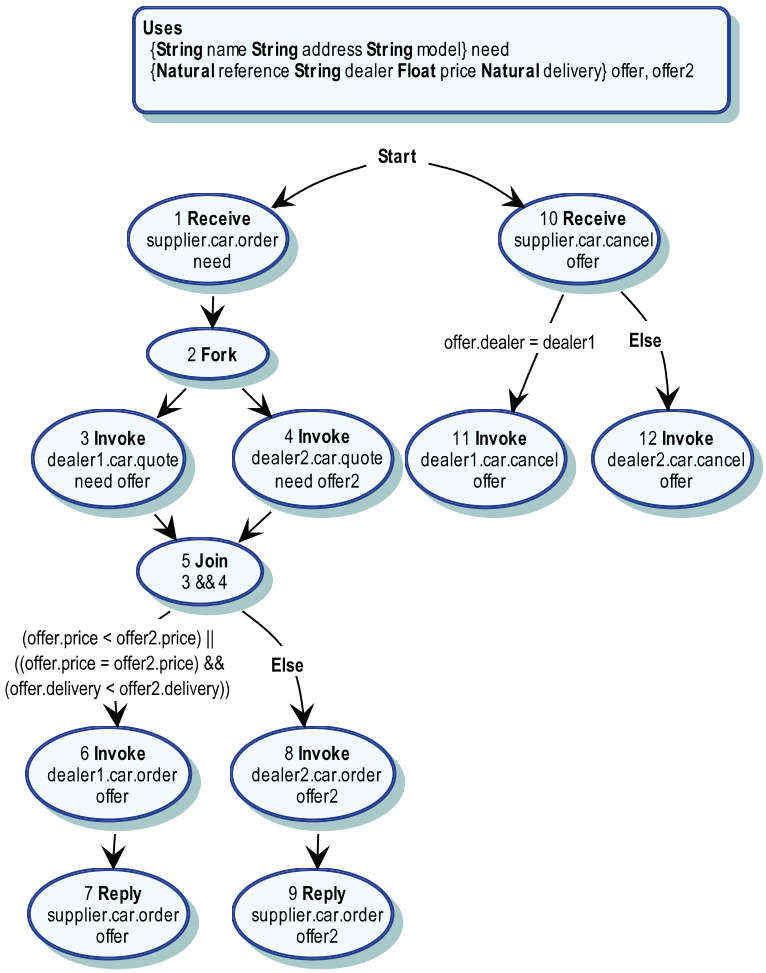

Figure 7.3: Supplier CRESS Diagram

offers from the dealers, stating that offers are selected in the order of lower price followed by faster delivery. If the prices and delivery are the same then an order is placed with dealer2. This expression is specified from the perspective of the offer by dealer1. If satisfied, the Supplier process will place an order with dealer1 using the *offer* information (node 6), followed by replying to its customer at node 7. Otherwise the offer of dealer2 is chosen (**Else** branch), with Supplier placing an order at node 8, and replying to the customer at node 9.

The *cancel* operation receives cancel requests at node 10, where the guarded expression is satisfied if the dealer of the offer parameter is dealer1 (an implicit macro which refers to **String** value BigDeal) therefore leading to invoking its 'cancel' operation at node 11 using the same offer parameter. Otherwise the 'cancel' operation of dealer2 is invoked at node 12.

7.2.1.3 Broker

The Broker service behaviour (figure 7.4) is described as follows. The rule box contains one data structure and variable definition, partner use declaration, and a definition of a constant. The complex data structure *schedule* is made up of: **Natural** reference which is the car order number; **String** dealer which identifies the dealer who organises the car; **Float** price that is the cost of the car; **Natural** delivery as the number of days to deliver the car; and **Float** rate as the loan rate of the car finance. The Broker diagram uses the syntax '*/ LENDER SUPPLIER*' which means it uses the Lender and Supplier CRESS diagram descriptions, implying the definitions in their rule boxes. Therefore the Broker diagram will have access to data types such as *proposal* and *need*. The macro/constant *unpriced* is defined with the value 1000000 to represent no offer of a car. The Broker service has only one operation broker.car.arrange which is synchronous, having *need* as the input and *schedule* as the output.

Broker receives the car *need* through broker.car.arrange in node 1. The car order is made with Supplier at node 2 invocation with the *need*, which returns the car offer as variable *offer*. A successful car order will have a price that is not of value *unpriced* (node 2 to 3) and the value for *proposal* variable is constructed. The fields in *proposal* are assigned the corresponding fields from *need*, with *amount* being the rounded value of *price*. Lender is invoked at node 3 with the *proposal* variable. If the invocation is successful, the *schedule* fields are set with the corresponding values from *offer* and *rate*, and broker.car.arrange replies with *schedule* at node 4. If the car order is unsuccessful (meaning *offer.price* is *unpriced*), *error* is assigned 'car unavailable' (**Else** branch). Broker then returns from broker.car.arrange operation with a fault named *refusal* and the value of *error*. Node 2 also specifies a **Compensation** handler (**Compensation** arc to node 6) which invokes supplier.car.cancel with the *offer*. The compensation behaviour at node 6 is enabled only if the **Invoke** at node 2 is successful. Only enabled compensation handlers can be invoked, via the **Compensate** activity (e.g. node 7).

The entire process has a global fault handler for fault name *refusal* of String type, specified by the **Catch** *refusal.error* from the **Start** node, which sets the String value of the fault to the variable *error* (which is defined in Lender). This fault handler is specified for any potential fault that could be thrown by the Lender process in the case of rejected loans. If the fault

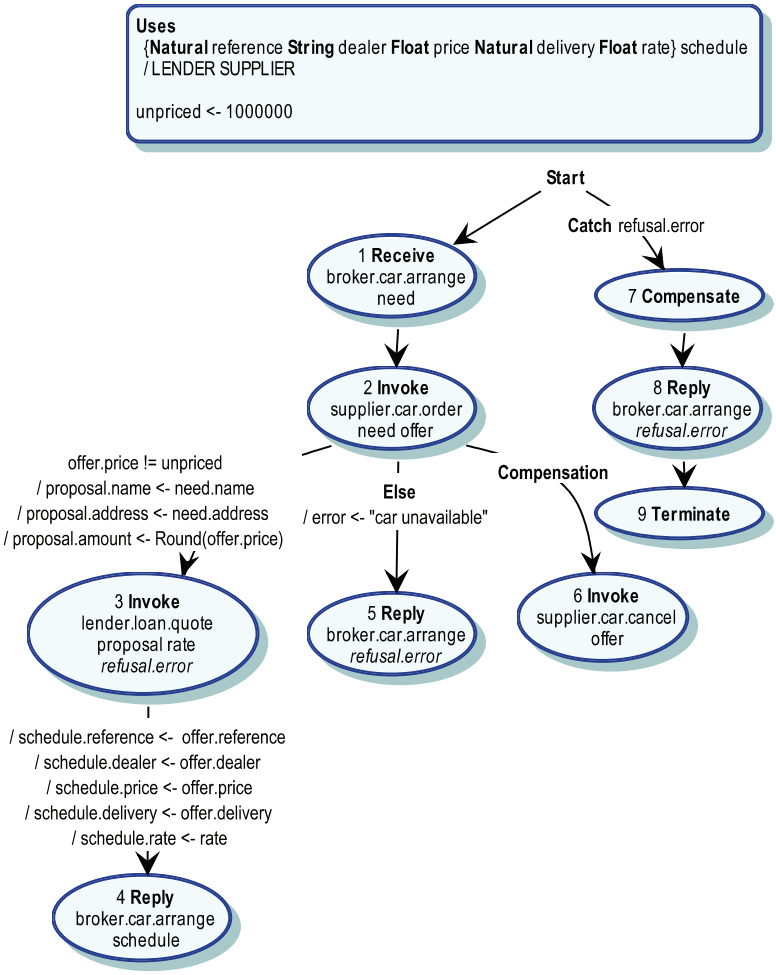

Figure 7.4: Broker CRESS Diagram

arises, the global fault handler is enabled and the Broker process will be directed to node 7. This explicitly calls the **Compensate** action which means all **Compensation** handlers that are enabled are executed in reverse order of completion. In this case, the only compensation handler (at node 2) will be called as it is already enabled after successful invocation at node 2. After compensation completes, Broker returns the fault for its operation and the process terminates (nodes 8 and 9).

7.2.1.4 Web Service Configuration

The developer configures the composite web services and partners services for formal specification and analysis. This is illustrated in the configuration diagram of figure 7.5. One instance of the services is sufficient for formal validation analysis, specified with '-n 1' (number

of instances: 1). As web services, these services should be always ongoing; the -r repeated option specifies that behaviour will repeat after the leaf node actions in the CRESS diagram. Comments will be generated in the specification, indicated by the '-c' comment option. The services to 'deploy' are the Lender, Supplier and Broker, which implies their partner services (Approver, Assessor, Dealer1, Dealer2). Following the **Deploys** clause are each service's parameters comprising the service name, namespace prefix, namespace, and deployment location.

Deploys -b 2 -c -a -n 1 -o 5 -r / LENDER SUPPLIER BROKER

APPROVER	app	urn:FirstRate	localhost:8080/active-bpel
ASSESSOR	ass	urn:RiskTaker	localhost:8080/active-bpel
BROKER	brok	urn:CarMen	localhost:8080/active-bpel
DEALER1	deal1	urn:BigDeal	localhost:8080/active-bpel
DEALER2	deal2	urn:WheelerDealer	localhost:8080/active-bpel
LENDER	lend	urn:LoanStar	localhost:8080/active-bpel
SUPPLIER	supp	urn:DoubleQuote	localhost:8080/active-bpel

Figure 7.5: Web Service Configuration Diagram

7.2.2 Partner Specification

The specifications of Approver, Assessor, Dealer1 and Dealer2 are provided by the developer. Their specifications can be extended from interface behaviour in the automated specification of Lender and Supplier. For illustrative purposes, the Approver interface behaviour and complete LOTOS specification are presented here.

The following will be the generated interface behaviour for Approver if there is no existing behaviour manually specified.

```
Process APPROVER [approver] : Exit(States) :=      (* APPROVER partner *)          1
  approver !loan !approve ?proposal2:Proposal;     (* 'approve' input *)           2
  (                                                                                 3
    approver !loan !approve !AnyNumber;            (* 'approve' output *)          4
    APPROVER [approver]                            (* repeat behaviour *)          5
  []                                               (* or *)                        6
    approver !loan !approve !Refusal !AnyText;     (* 'Refusal' fault *)           7
    APPROVER [approver]                            (* repeat behaviour *)          8
  )                                                                                 9
EndProc                                            (* end APPROVER *)              10
```

The following is the complete LOTOS specification for Approver. The specification uses CRESS-defined LOTOS abstract data types and operations which are defined in the 'stir' (Stirling) library, and also the Proposal type which was automatically generated. These library

and user-defined types are automatically declared and defined in the overall specification, i.e. Lender. The FirstRate service approves loans based on specified conditions described in the requirements. For example, lines 4-5 specifies that a client whose name starts with “Ken” will be offered a rate of 3.7%.

```
Process APPROVER [approver] : Exit(States) :=                    (* start APPROVER *) 1
  approver !loan !approve ?proposal:Proposal;                    (* proposal request *) 2
  (                                                                                  3
    [starts(getName(proposal),t(K)∼e∼n)] ≫                       (* Ken? *)           4
      approver !loan !approve !number(+,t(3),t(7));              (* rate 3.7% *)      5
      APPROVER [approver]                                        (* repeat *)         6
  []                                                                                 7
    [not(starts(getName(proposal),t(K)∼e∼n))] ≫                  (* else *)           8
      (                                                                              9
    [contains(getAddress(proposal),t(S)∼c∼o∼t∼l∼a∼n∼d)] ≫        (* Scotland? *)      10
      approver !loan !approve !number(+,t(4),t(1));(* rate 4.1% *)                   11
      APPROVER [approver]                                        (* repeat *)         12
      []                                                                             13
    [not(contains(getAddress(proposal),t(S)∼c∼o∼t∼l∼a∼n∼d))] ≫ (* else *)           14
      (                                                                              15
        [getAmount(proposal) lt number(+,t(1)∼5∼0∼0∼0,<>)] ≫ (* < 15000? *)         16
          approver !loan !approve !number(+,t(4),t(4));          (* rate 4.4% *)      17
          APPROVER [approver]                                    (* repeat *)         18
      []                                                                             19
        [getAmount(proposal) ge number(+,t(1)∼5∼0∼0∼0,<>)] ≫ (* else *)            20
          approver !loan !approve !refusal                       (* refuse *)         21
        !t(L)∼o∼a∼n∼ ^ ∼u∼n∼a∼c∼c∼e∼p∼t∼a∼b∼l∼e;                                   22
          APPROVER [approver]                                    (* repeat *)         23
      )                                                                              24
      )                                                                              25
  )                                                                                  26
EndProc                                                          (* end APPROVER *)27
```

7.2.3 Partner Implementation

The implementation of Approver, Assessor, Dealer1 and Dealer2 are provided by the developer. These implementation can be extended from the code skeletons generated in the implementation of the services. For illustration the Approver partner skeleton and implementation code are provided here. The following is the code skeleton generated for Approver, which can be extended to complete the implementation described above.

```
package FirstRate;
...
public class LoanBindingImpl implements LoanPort{
  public float approve(Proposal2 proposal2)
    throws RemoteException, StringMessage {
```

```
    return -3;
  }
}
```

The following is the Approver completed Java code extended from the above code skeleton.

```
package FirstRate;                                                          1
...                                                                         2
class LoanBindingImpl implements LoanPort {          // loan port binding   3
  ...                                                                       4
                                                                            5
  public float approve(Proposal2 proposal) throws RemoteException {         6
    String name = proposal.getName();                // get proposal name   7
    String address = proposal.getAddress();          // get proposal address 8
    int amount = proposal.getAmount().intValue();    // get proposal amount 9
                                                                            10
    float rate = 0.0f;                               // declare loan rate   11
    if (name.startsWith("Ken"))                      // name starts "Ken"?  12
      rate = 3.7f;                                                          13
    else if (address.indexOf("Scotland") != -1)      // address has "Scotland"? 14
      rate = 4.1f;                                                          15
    else if (amount <= 15000)                        // amount less than 15000? 16
      rate = 4.4f;                                                          17
    else {                                           // otherwise           18
      ... throw fault loan unacceptable                                     19
    return (rate);                                   // return loan rate    20
  }                                                                         21
}                                                                           22
```

7.2.4 MUSTARD Scenarios

7.2.4.1 Approver Scenarios

- A client named 'Ken Smith' from 'Liverpool UK' seeking a loan amount of 6000 should receive a rate 3.7%
- A client named 'Angus Og' from 'Airth Scotland' seeking a loan amount of 20000 should receive a rate 4.1%
- A client named 'Nancy Turner' from 'Manchester England' seeking a loan amount of 14999 should receive a rate 4.4%
- A client named 'Ian Carey' from 'Croydon England' seeking a loan amount of 15000 should be rejected with a refusal fault containing a string message 'loan acceptable'.

The following are its MUSTARD scenarios.

```
test(Low_Rate,
  succeed(
    send(approver.loan.approve,Proposal2/Proposal('Ken Smith,'Liverpool UK,6000.)),
    read(approver.loan.approve,3.7)))
```

```
test(Medium_Rate,
  succeed(
    send(approver.loan.approve,Proposal2/Proposal('Angus Og,'Airth Scotland,20000.)),
    read(approver.loan.approve,4.1)))

test(High_Rate,
  succeed(
    send(approver.loan.approve,
         Proposal2/Proposal('Nancy Turner,'Manchester England,14999.)),
    read(approver.loan.approve,4.4)))

test(Loan_Unacceptable,
  succeed(
    send(approver.loan.approve,
         Proposal2/Proposal('Ian Carey,'Croydon England,15000.)),
    read(approver.loan.approve,refusal,'loan unacceptable)))
```

The above MUSTARD scenarios each comprise a **send** and **read** primitive. The service name (approver), port (loan), operation (approve), data types (proposal, string, float), and fault name (refusal) correspond to those defined in the Lender CRESS diagram. The Proposal2/Proposal syntax was specified as the data type definition for Proposal (i.e. type 'Proposal2' constructs 'Proposal'). As Proposal and Proposal2 have the same structure, the abstract data type was collapsed during translation to one data type using Proposal, even for the Approver service, which simplifies the specification (e.g. assignment) but does not affect the behaviour from the perspective of validation. 'Proposal2/' is still specified as the MUSTARD scenarios are reused in the validation of Approver's implementation, where the actual data type has to be used in the web service interaction.

7.2.4.2 Assessor Scenarios

The following are the MUSTARD scenarios specified for Assessor. They are similar apart from values which validate the three possible responses of Assessor.

```
test(Low_Risk,
  succeed(
    send(assessor.loan.assess,Proposal('Mike Turner,'Carlisle UK,20000.)),
    read(assessor.loan.assess,'low)))

test(Medium_Risk,
  succeed(
    send(assessor.loan.assess,Proposal('Fred Hoyle,'UK,5000.)),
    read(assessor.loan.assess,'medium)))
```

```
test(High_ Risk,
  succeed(
    send(assessor.loan.assess,Proposal('Patrice Touvet,'Paris France,1000.)),
    read(assessor.loan.assess,'high)))
```

7.2.4.3 Lender Scenarios

The following MUSTARD scenarios are defined for Lender.

```
test(Little_ Low_ Risk,
  succeed(
    send(lender.loan.quote,Proposal('Nancy Turner,'Manchester England,9999.)),
    read(lender.loan.quote,3.5)))

test(No_ Risk_ Assess_ Low,
  refuse(
    send(lender.loan.quote,Proposal('Nancy Turner,'Manchester England,10000.)),
    read(lender.loan.quote,3.5)))

test(Lots_ Ken,
  succeed(
    send(lender.loan.quote,Proposal('Ken Boyle,'Dublin Ireland,10000.)),
    read(lender.loan.quote,3.7)))

test(Lots_ Scotland,
  succeed(
    send(lender.loan.quote,Proposal('Mary Duncan,'Wick Scotland,20000.)),
    read(lender.loan.quote,4.1)))

test(Lots_ Under_ 15000,
  succeed(
    send(lender.loan.quote,Proposal('Sally Dean,'Cardiff Wales,14999.)),
    read(lender.loan.quote,4.4)))

test(Lots_ Exceeds_ 15000,
  succeed(
    send(lender.loan.quote,Proposal('Ian Carey,'Croydon England,15000.)),
    read(lender.loan.quote,refusal,'loan unacceptable)))
```

The definition of the No_Risk_Assess_Low scenario is particularly different, and therefore is discussed in detail using the LOTOS specification automatically translated from the MUSTARD scenario. The scenario uses the **refuse** construct. This indicates that the last event, which is **read**(lender.loan.quote,3.5), should not happen as the Lender behaviour logic should not lead to any risk assessment for amounts greater or equal to 10000. The **refuse** is translated in the Lola test process as a choice behaviour. If the event 'lender !loan !quote !Number(+,t(3),t(5))' should synchronise, then the behaviour will stop without an OK event, implying the validation

did not pass. Otherwise the alternative behaviour path is followed which asserts the OK event, indicating the scenario passes.

```
Process LENDER_No_Risk_Assess_Low [lender,OK] : NoExit :=                    1
  lender !loan !quote !proposal(t(N)~a~n~c~y~ ^ ~T~u~r~n~e~r,                 2
  t(M)~a~n~c~h~e~s~t~e~r~ ^ ~E~n~g~l~a~n~d,                                   3
  Number(+,t(1)~0~0~0~0,<>));                                                4
  (                                                                          5
    lender !loan !quote !Number(+,t(3),t(5));                                6
    Stop                                                                     7
  []                                                                         8
    I;                                                                       9
    OK;                                                                      10
    Stop                                                                     11
  )                                                                          12
EndProc (* LENDER_No_Risk_Assess_Low *)                                      13
```

7.2.4.4 Dealer1 Scenarios

Dealer1's MUSTARD scenarios are defined below and are straightforward. Scenarios for Dealer2 are similar and hence not shown. The use of ?Natural in the **read** indicates that it accepts any value of Natural as the reference number in the Offer data type. In this situation, the reference number of the Offer returned by Dealer1 is dynamic, which is reasonable in a business quotation. Therefore the use of the variable notation is convenient to describe such scenarios in a compact way. It also supports a general form of validation in addition to specific scenarios. For example, the Mondeo test specifies: an invocation of the Need should receive an Offer of 20000 in value and 15 days from BigDeal, regardless of the reference number as long as it is a Natural.

```
test(Mondeo,
  succeed(
    send(dealer1.car.quote,Need('Mark Flowers,'Uist Scotland,'Mondeo)),
    read(dealer1.car.quote,Offer(?Natural,'BigDeal,20000.,15))))

test(A5,
  succeed(
    send(dealer1.car.quote,Need('Peter Gough,'Congleton UK,'A5)),
    read(dealer1.car.quote,Offer(?Natural,'BigDeal,33000.,30))))

test(Megane,
  succeed(
    send(dealer1.car.quote,Need('Jan Hiddink,'Hengelo Netherlands,'Megane)),
    read(dealer1.car.quote,Offer(?Natural,'BigDeal,11000.,5))))
```

```
test(XJ6,
   succeed(
      send(dealer1.car.quote,Need('Iain MacKay,'Throsk Scotland,'XJ6)),
      read(dealer1.car.quote,Offer(?Natural,'BigDeal,1000000.,0))))
```

7.2.4.5 Supplier Scenarios

```
test(Mondeo,
   succeed(
      send(supplier.car.order,Need('Mark Flowers,'Uist Scotland,'Mondeo)),
      read(supplier.car.order,Offer(?Natural,'WheelerDealer,20000.,10))))

test(A5,
   succeed(
      send(supplier.car.order,Need('Peter Gough,'Congleton UK,'A5)),
      read(supplier.car.order,Offer(?Natural,'BigDeal,33000.,30))))

test(Megane,
   succeed(
      send(supplier.car.order,Need('Jan Hiddink,'Hengelo Netherlands,'Megane)),
      read(supplier.car.order,Offer(?Natural,'BigDeal,11000.,5))))

test(Astra,
   succeed(
      send(supplier.car.order,Need('Hywel Thomas,'Swansea Wales,'Astra)),
      read(supplier.car.order,Offer(?Natural,'WheelerDealer,18000.,30))))

test(XJ6,
   succeed(
      send(supplier.car.order,Need('Iain MacKay,'Throsk Scotland,'XJ6)),
      read(supplier.car.order,Offer(?Natural,'WheelerDealer,1000000.,0))))
```

7.2.4.6 Broker Scenarios

```
test(Mondeo_ Ken,
   succeed(
      send(broker.car.arrange,Need('Ken Boyle,'Dublin Ireland,'Mondeo)),
      read(broker.car.arrange,Schedule(?Natural,'WheelerDealer,20000.,10,3.7))))

test(A5_ Scotland,
   succeed(
      send(broker.car.arrange,Need('Mary Duncan,'Wick Scotland,'A5)),
      read(broker.car.arrange,Schedule(?Natural,'BigDeal,33000.,30,4.1))))

test(Megane,
   succeed(
      send(broker.car.arrange,Need('Sally Dean,'Cardiff Wales,'Megane)),
      read(broker.car.arrange,Schedule(?Natural,'BigDeal,11000.,5,4.4))))

test(Astra,
   succeed(
      send(broker.car.arrange,Need('Ian Carey,'Croydon England,'Astra)),
```

```
        read(broker.car.arrange,refusal,'loan unacceptable)))

    test(XJ6,
      succeed(
        send(broker.car.arrange,Need('Iain MacKay,'Throsk Scotland,'XJ6)),
        read(broker.car.arrange,refusal,'car unavailable)))
```

7.2.5 CLOVE Properties

7.2.5.1 Lender Properties

- The service should be free from deadlock.
- The service should be free from livelock.
- The service should start only with the signal lender.loan.quote accepting only Proposal values.
- All Proposal requests should receive either a reply of a rate (any number) or a refusal fault with message "loan unacceptable".
- A more specific response property is that valid responses for any Proposal values are either the rates 3.5, 3.7, 4.1, 4.4, or the refusal fault of message "loan unacceptable".

The following are the CLOVE value enumerations and properties defined by the developer. The **initials** define the permitted signals during the start of the service, which will be used by the initial safety property that is built into CLOVE. The other explicitly specified properties were constructed using the global response verification template. Deadlock and livelock freedom are checked by default in CLOVE.

```
initials(signal(lender.loan.quote,?proposal))

enum_complex(
  proposal("(KEN TURNER|LARRY TAN)","UK",[5|7]0000.0),
  proposal("(KEN TURNER|LARRY TAN)","UK",[5|7]000.0)
...
)

property(General_Response,
  response(global,
    signal(lender.loan.quote,?proposal),
    choice_any(signal(lender.loan.quote,?number),
      signal(lender.loan.quote,refusal,'loan unacceptable))))

property(Specific_Response,
  response(global,
    signal(lender.loan.quote,?proposal),
```

```
choice_any(signal(lender.loan.quote,3.5),
  signal(lender.loan.quote,3.7),
  signal(lender.loan.quote,4.1),
  signal(lender.loan.quote,4.4),
  signal(lender.loan.quote,refusal,'loan unacceptable'))))
```

7.2.5.2 Supplier Properties

The following is the informal description of properties specified for Supplier.

- Freedom from deadlock.
- Freedom from livelock.
- All car orders must be responded to with offers.

The CLOVE file is specified with the data enumeration, and properties which explicitly specify the latter two properties described above.

```
enum_complex(
  need("KEN TURNER","SCOTLAND","MONDEO"),
  need("LARRY TAN","UK","MONDEO"),
  offer([0−9], "BIGDEAL", 20000.0, 15),
  offer([0−9], "WHEELERDEALER", 20000.0, 10),
...
)

property(General_Response,
  response(global,
    signal(supplier.car.order,?need),
    signal(supplier.car.order,?offer)))
```

7.2.5.3 Broker Properties

The following is the informal description of properties specified for Broker.

- Freedom from deadlock
- Freedom from livelock
- The Broker service should only have the 'arrange' operation.
- All 'need' requests should be responded to with, either success schedules or faults (loan unacceptable or car unavailable)

Broker is a composite service comprising composed services, i.e. Lender and Supplier, that are all specified in CRESS. Compositional verification should therefore be used to achieve

effective analysis. Below is the CLOVE file specified by the developer, comprising the enumerations and properties only for the last two properties, as deadlock and livelock freedom will be checked by default. These properties are general and difficult to check with validation, but verification is well-suited for this purpose.

```
initials(signal(broker.car.arrange,?need))

enum_ numbers(3.7,3.7,4.1,4.4)
enum_ strings('loan unacceptable,'car unavailable,'low,'medium,'high)

enum_complex(
   need("KEN TURNER","SCOTLAND","MONDEO"),
...
   proposal("LARRY TAN", "UK", 20000.0), ...
   offer([0−9], "BIGDEAL", 20000.0, 15),
...
   schedule([0−9], "BIGDEAL", 20000.0, 15),
...
)

property(General_ Response,
   response(global,
      signal(broker.car.arrange,?need),
      choice_ any(signal(broker.car.arrange,?schedule),
         signal(broker.car.arrange,refusal,'loan unacceptable),
         signal(broker.car.arrange,refusal,'car unavailable))))
```

7.2.6 Formal Specification

The following show the high-level LOTOS behaviour specification generated for the Lender composite service. The specification of the behaviour is centred on the Lender's composed behaviour as described in its CRESS diagram and service configuration that **Deploys** the Lender service. Therefore the specification has one gate (endpoint) which is 'lender' that is externally visible, with Approver and Assessor hidden from the view of Lender's clients. The specification includes CRESS-defined LOTOS library data types (e.g. String), ports (e.g. loan), operations (e.g. assess), fault events (e.g. refusal), and user-defined complex data types (e.g. proposal).

```
Behaviour
   LENDER [lender]                          (* call LENDER process *)
...

Type Proposal Is BaseTypes                  (* proposal record *)
...
```

```
Process APPROVER [approver] : Exit(States) := (* APPROVER partner *)
...

Process ASSESSOR [assessor] : Exit(States) :=  (* ASSESSOR partner *)
...

Process LENDER [lender] : Exit(States) :=        (* LENDER service *)
  Hide approver,assessor In                      (* hide internal gates *)
    (
      APPROVER [approver]                        (* call APPROVER partner *)
    |||                                          (* interleaved with *)
      ASSESSOR [assessor]                        (* call ASSESSOR partner *)
    )
  ||[approver,assessor]||                        (* synchronised with partners *)
    LENDER_ 1 [approver,assessor,lender]         (* call main process *)
      ...

Where                                            (* local definitions *)
...

Process LENDER_ 1 [approver,assessor,lender] ...
  lender !loan !quote ?proposal:Proposal;        (* LENDER receive 1 *)
...
```

For the overall behaviour expression, the 'LENDER [lender]' process defines the behaviour of Approver and Assessor synchronised with the behaviour of Lender. These are respectively specified as APPROVER, ASSESSOR and LENDER_1 (i.e. Lender from node 1) processes. The services and their behaviour are autonomous and therefore independent from one another. Through the behaviour synchronisation via gates '|[approver, assessor]|', service communication is established whereby LENDER_1 (the main entry to Lender behaviour) can communicate with APPROVER and ASSESSOR. The partners Approver and Assessor do not communicate with each other and therefore they are interleaved using the '|||' operator.

Apart from manual specification of partner services, the formal specification is fully automated such that all data types and behaviour are specified. Even the partner specification itself can be extended from the interface behaviour that is automatically generated. Using this specification, formal validation and verification can be performed. Table 7.1 lists the code summary in terms of lines of code generated and manually provided.

Automated (Lender)	**Manual** (Approver & Assessor)	**Automated** (Supplier)	**Manual** (Dealer1 & Dealer2)	**Automated** Broker
412	47	775	366	1349

Table 7.1: Specification of Web Services (no. of lines)

7.2.7 Formal Analysis

7.2.7.1 Formal Validation

Validation is performed on the composed and partner services, which corresponds to the methodology described in section 3.3.3.1. There is no strict order of validation; however, given the relationship of the services, it is logical to validate the partner services first before the composed service. For the Lender composed service, the order of validation was Approver, Assessor and then Lender. The order is similar for Supplier – validate Dealer1, Dealer2, then Supplier. Broker us the last to be validated, its partners Lender and Supplier having been validated. The following are the validation results, which may differ in terms of time completion.

```
Test APPROVER Low Rate ...                Pass    1 succ    0 fail    1.7 secs
...
Test ASSESSOR Low Risk ...                Pass    1 succ    0 fail    0.6 secs
...
Test LENDER Little Low Risk ...           Pass    1 succ    0 fail    0.5 secs
...
Test DEALER1 Mondeo ...                   Pass    1 succ    0 fail    0.6 secs
...
Test DEALER2 Mondeo ...                   Pass    1 succ    0 fail    0.5 secs
...
Test SUPPLIER Mondeo ...                  Pass    1 succ    0 fail    0.7 secs
...
Test BROKER Mondeo Ken ...                Pass    1 succ    0 fail    0.6 secs
...
```

The automated formal validation translates the MUSTARD scenarios into 220 lines of LOTOS tests (distribution listed in table 7.2), and executes with results in a few seconds, which is very effective for analysing the specification and detecting errors contradicting the scenarios specified. Achieving passes in the validation means that the service behaviours have demonstrated that they are doing the right thing with regard to these specified scenarios. If there are any failures or inconclusive outcomes in the validation, then the diagnostic traces are used to trace and address the problem, i.e. partner specification or CRESS diagram.

Lender	Approver	Assessor	Supplier	Dealer1	Dealer2	Broker
47	28	20	35	28	28	35

Table 7.2: Generated LOTOS Tests (no. of lines)

Suppose that the the Low_Rate scenario did not pass the validation, and there is a diagnostic trace of the following:

```
Test APPROVER Low Rate ...          Fail              0 succ   1 fail    1.3 secs

send(approver.loan.approve,Proposal2/Proposal('Ken Smith, 'Liverpool UK, 6000.))
<failure point>
```

This demonstrates that the scenario is unable proceed after the first action that is **send** (approver.loan.approve, Proposal2/Proposal('Ken Smith, 'Liverpool UK, 6000.)). The next action is **read**(approver.loan.approve, 3.7). This is unsuccessful, indicating that the value returned is not 3.7. As the validation is targeted at Approver, its specification is to be inspected and corrected (or perhaps the test is wrong). The validation should be executed again and all the Approver MUSTARD scenarios passed.

Although the validation of a composite service may imply validation of its partner services, it may not directly validate their behaviour, which is why they should have their own validation scenarios. The following is an example to emphasis the need for partner service validation. Suppose Assessor returns another risk assessment outcome instead of "medium" (e.g. mid), the behaviour of Lender is not affected and passes its validation scenarios. This is because the Lender's behaviour does not deal directly with any other risk assessments apart from "low". If Assessor does not have its own validation scenarios (e.g. Medium_Rate), then this error may not detected in its specification. Suppose that the condition arc from node 1 to node 2 is specified incorrectly (e.g. 'proposal.amount > 10000'), the validation of No_Risk_Assess_Low scenario does not pass, and the diagnostic trace is as follows:

```
Test LENDER No Risk Assess Low ...  Inconclusive      1 succ 1 fail     0.4 secs

send(lender.loan.quote,Proposal(Nancy Turner,Manchester England,10000.))
read(lender.loan.quote,3.5)
<failure point>
```

The validation outcome for No_Risk_Assess_Low is Inconclusive, with one success path and one failure path. The diagnostic trace shows the failure path which indicated that the rate

3.5% returned by the Lender behaviour was unexpected. As a result the path behaves as 'stop' (line 7 of its translated LOTOS scenario described in section 7.2.4.3). The reason for the one success path is due to the non-determinism specified in the internal event 'i' as one of the first events in the choice operator. This event will be tried since all possible paths in the validation behaviour are tried. As 'i' events are always successful, this path leads to OK indicating a success. Should the read for rate 3.5% fail to synchronise, then this is the only path that is successful, therefore the validation scenario will pass.

The failure of No_Risk_Assess_Low indicates that the behaviour of Lender is not doing the right thing. Given the context of the problem, rate 3.5% is returned by Lender only if a "low" risk assessment is made by Assessor, as there is no such rate returned by the Approver specification. The Lender CRESS diagram is inspected as Assessor was invoked as part of its logical behaviour. There are only two guards from node 1 in the Lender diagram where there is a condition leading to node 2; otherwise it follows the **Else** guard leading to node 6 that invokes Assessor. Therefore the condition arc from node 1 to node 2 is specified incorrectly (e.g. 'proposal.amount > 10000') and therefore fails this specified validation, and should corrected.

7.2.7.2 Formal Verification

The same LOTOS specifications are used for formal verification of the composed services in the methodology, as described in section 3.3.3.2; however, the specifications has to be automatically annotated for CADP. This is achieved in the web service configuration with the '-a' annotation option for specifications to be generated with annotations.

The composed services are verified separately for the following reason. Lender has the simplest behaviour and does not require the compositional mode of verification. Supplier uses array data types to maintain the list of cars, quotations and orders, therefore these data types require constraints on their sizes (e.g. specifying bounded size of 3 for arrays), with data operations adapted to these constraints. Compositional mode is used in the verification of Supplier and Broker. The web service configuration for Lender is configured as:

Deploys -a -c -n 1 -r / LENDER

The above configuration is similar for Supplier and Broker, only changing to the deployed service (e.g. SUPPLIER).

The following shows the results of verifying the Lender properties.

```
Verifying LENDER GENERAL RESPONSE ...              TRUE
Verifying LENDER SPECIFIC RESPONSE ...             TRUE
Verifying LENDER ALL KEN NO RISK APPROVE ...       TRUE
Verifying          DEADLOCK FREEDOM ...            TRUE
Verifying          LIVELOCK FREEDOM ...            TRUE
Verifying          INITIALS SAFETY ...             TRUE
```

The following shows the results of verifying the Supplier properties.

```
Verifying SUPPLIER GENERAL RESPONSE ...            TRUE
Verifying SUPPLIER REQUEST LIVENESS ...            TRUE
Verifying          DEADLOCK FREEDOM ...            TRUE
Verifying          LIVELOCK FREEDOM ...            TRUE
```

Prior to verifying Broker, its behavioural specification has to be adapted to be free from the semantics of recursive process instantiation, which is not compatible with CADP requirements – this is not a limitation of the LOTOS language but the CADP tool. The manual mode in cress_verify is first used, where there is a high degree of automated support in the LOTOS specification annotation, and CLOVE is invoked for: generation of the C implementation of data enumeration; translation of properties into μ-calculus; and generation of the SVL script which describes all the tasks in the compositional verification.

In the Broker LOTOS behaviour specification, there are uses of the enabling operator '$\gg$' which specify compensation behaviour in a way that the processes are being called but through recursive process instantiation that CADP does not allow. The specification was manually adapted to be compatible with CADP, as given in [89]. An automated solution has been implemented in later work. The analyst executes the generated SVL script. The summary of the verification outcome is:

```
... compositional generation of state space ...

"broker_general_response.bcg" = verify "broker_general_response.mcl" in "ws.bcg"
TRUE
"_deadlock_freedom.bcg" = verify "_deadlock_freedom.mcl" in "ws.bcg"
TRUE
"_livelock_freedom.bcg" = verify "_livelock_freedom.mcl" in "ws.bcg"
TRUE
"_initials_safety.bcg" = verify "_initials_safety.mcl" in "ws.bcg"
TRUE
```

where verification outcomes assert that the four specified properties are exhibited by the service behaviour. These are the CLOVE properties individually translated into .mcl (μ-calculus) files.

In particular the ‘general response’ property demonstrates that the service will always reply with only the specified responses.

Table 7.3 describes the effect of automated verification in the methodology with regards to the lines of code that are automatically generated.

Service	**Annotated LOTOS**	**μ-calculus**	**C Implementation**	**SVL**
Lender	1721	5 files (15 lines)	64	–
Supplier	2055	4 files (9 lines)	98	32
Broker	2514	4 files (12 lines)	184	48

Table 7.3: Generated Verification Code (no. of lines) for Lender, Supplier and Broker

The following is an illustration of verification complementing validation. Suppose the behaviour of Approver stops after returning the first refusal (“loan unacceptable”) fault. The model checking of the Lender behaviour will produce a counter-example for freedom from deadlock:

```
Verifying              DEADLOCK FREEDOM ... FALSE
"LENDER !LOAN !QUOTE !PROPOSAL ("LARRY TAN", "UK", 50000.0)"
internal event
internal event
"LENDER !LOAN !QUOTE !REFUSAL !"LOAN UNACCEPTABLE""
"LENDER !LOAN !QUOTE !PROPOSAL ("LARRY TAN", "UK", 70000.0)"
DEADLOCK
```

The path trace above shows that the first loan request results in a response, but a subsequent request which is denoted by ‘LENDER !LOAN !QUOTE !PROPOSAL ("LARRY TAN", "UK", 70000.0)’ can proceed no further, nor even any internal events. From the CRESS diagram, all loan requests value greater than or equals to 10000 will not be directed to the Assessor at all, which narrows down the behaviour to check. A MUSTARD scenario can be quickly defined to represent the same value as the subsequent request. The following scenario is expected to pass the validation.

```
test(Lots_Larry,
  succeed(
    send(lender.loan.quote,Proposal('Larry Tan,'UK,70000.)),
    read(lender.loan.quote,refusal,'loan unacceptable)))

Test LENDER Lots Larry ...          Pass          1 succ   0 fail   0.5 secs
```

This result narrows down to the behaviour handling of a second request, where there is no way to proceed further, in this case with Approver. The scenario above could have appended a second pair of **send** and **read** with the same values, which will detect the same deadlock path found through verification, confirming the same problem. The Approver specification will have to be inspected and corrected, in this case by repeating its behaviour instead of stop. It may be argued that validation sufficed for this deadlock example; however, such problems are usually generalised. For example in this case deadlock does not happen at the second request onwards but hypothetically after any arbitrary number, therefore it is impractical to use validation. Validation may be used as in this example to quickly confirm a counter-example if required, although the verification results are adequate. The generalised properties such as General_ Response(s) and livelock freedom are practical and concise in verification, where they would be impossible using validation.

7.2.8 Implementation

This step corresponds to the implementation aspect of the methodology in section 3.3.4 where there is automated generation of code and deployment. As the implementation for the partner services has been provided, they can be immediately compiled for deployment. The implementation of the composed services is fully automated.

To engage the implementation phase, the developer configures the service configuration diagram with the following parameters:

Deploys -c -b 2 -o 5 / LENDER SUPPLIER BROKER

This configuration **Deploys** the three CRESS composed services which implies their partner services as they are used in the composed behaviour. The BPEL implementations of all composed services are in the WS-BPEL 2.0 standard (-b 2). As there is intent to perform implementation validation, the timeout threshold is set to five seconds (-o 5), which generates the appropriate MINT validation configuration with service timeout of 5 seconds for all the services to be deployed. The code will be generated with comments (-c). Partner services (e.g. approver, dealer1, etc.) and composed services (e.g. lender, supplier and broker) are compiled for AXIS and ActiveBPEL deployment respectively.

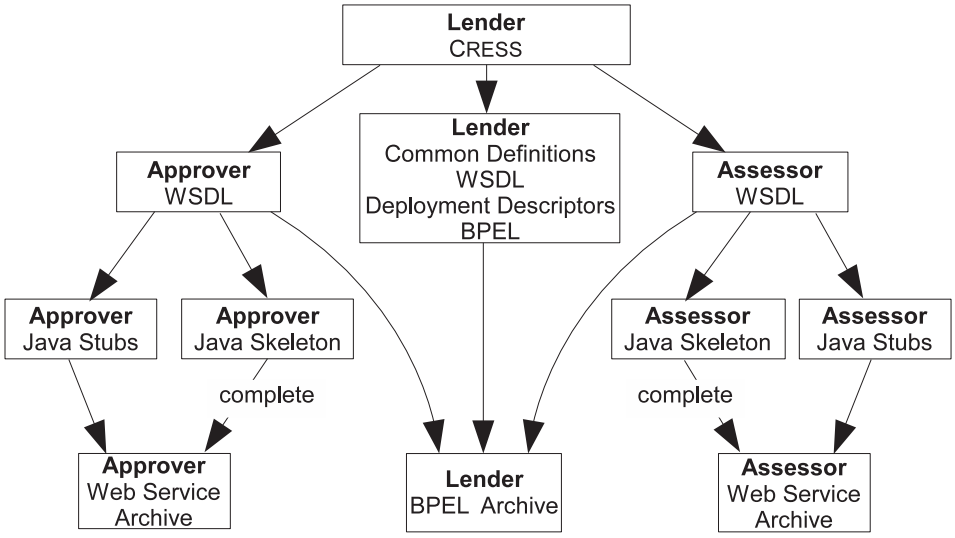

Figure 7.6: Automated Implementation Process for Lender, Approver and Assessor

Table 7.4 shows the number of lines of code and files that are automatically produced in the implementation phase. The implementation for the services is highly automated apart from the manual implementations of partners, which are added into the generated code skeletons. The automation of the implementation took less than a minute to produce the deployable service archives.

Service	**WSDL** (incl. partners)	**Java**	**Deployment**	**BPEL**
Approver	2 files (136 lines)	8 files (738 lines)	34 lines	–
Assessor	2 files (129 lines)	8 files (657 lines)	34 lines	–
Lender	4 files (284 lines)	–	2 files (66 lines)	192 lines
Dealer1	2 files (140 lines)	8 files (997 lines)	42 lines	–
Dealer2	2 files (140 lines)	8 files (997 lines)	42 lines	–
Supplier	4 files (314 lines)	–	2 files (67 lines)	222 lines
Broker	6 files (420 lines)	–	2 files (73 lines)	292 lines

Table 7.4: Code Generation Summary of Implemented Web Services

7.2.9 Implementation Validation

7.2.9.1 Functionality Validation

The implementation of the web services can be validated/tested after deployment in the AXIS and ActiveBPEL container; this activity corresponds to the description discussed section 3.3.5.

Implementation validation requires the MINT configuration properties and scenarios. The MINT validation configuration properties were generated as part of the implementation process (-o option in the configuration diagram). The MUSTARD scenarios for these web services were already defined in the formal validation; these are automatically translated into MINT scenarios for implementation.

The following is an example of what the validation outcomes may be:

```
Test APPROVER Low Rate ...              Pass    1 succ    0 fail    1.1 secs
Test APPROVER Medium Rate ...           Pass    1 succ    0 fail    1.1 secs
Test APPROVER High Rate ...             Pass    1 succ    0 fail    1.1 secs
Test APPROVER Loan Unacceptable ...Pass         1 succ    0 fail    1.0 secs
Test ASSESSOR Low Risk ...
...
Test LENDER Lots Exceeds 15000 ...      Pass    1 succ    0 fail    1.2 secs
```

Lender & partners Properties	**MINT** Scenario	**Supplier & partners** Properties	**MINT** Scenario	**Broker** Properties	**MINT** Scenario
3 files (59 lines)	68	3 files (57 lines)	81	20	33

Table 7.5: Code Generation Summary for Implementation Validation (no. of lines)

The results from implementation validation should agree with formal validation – the scenarios should pass. This will confirm that the service implementation is behaving as expected, considering these scenarios. Otherwise there will be diagnostics provided to trace the problem. For example, suppose the Assessor is expected return a “low” risk but does not, then its implementation is to be investigated. Although formal validation may correct the composed service diagram, this does not imply that the implementation validation will achieve the same validation results. This is because the partner service implementations may not behave as expected, such as the previous example. This has a direct impact on Lender behaviour and therefore some of its scenarios may not pass. Implementation validation also tests (black-box) the availability of the services which can gain insight into the deployment of the targeted service, e.g. Lender is not deployed successfully and therefore WSDL download by MINT fails. In addition, there is testing for the service response within the timeout threshold.

7.2.9.2 Performance Evaluation

After the functional validation is satisfied, the performance evaluation can be carried out to inspect the deployment configuration of the service environment and consistency of service behaviour under load. The sequential performance evaluation may be carried out first to establish consistency in a series of invocations, followed by concurrent execution.

The following is one result obtained from actual concurrent performance evaluation. This scenario passes the concurrent performance evaluation of 150 runs: all are successful, none failed, and none were inconclusive (inco), with an average scenario completion approximately in 4.5 seconds. The results are consistent (cons), the fastest scenario completion time is approximately 3.6 seconds, and the slowest is 4.8 seconds.

```
Test APPROVER Low Rate ...        Pass                150 succ 0 fail      4.5 secs
                          Concurrent  0 inco true cons 3.6 secs .. 4.8 secs
```

Resource configuration issues may also arise during performance evaluation, particularly concurrent mode, which may be the cause for the service's inconsistent behaviour. The following is an example that was actually found. The concurrent performance evaluation with a load of 150 of a particular scenario for Lender has reported results of:

```
Test LENDER Little Low Risk ...
Execution error of
send(lender.loan.quote,Proposal("Nancy Turner","Manchester England",9999.))
caused by java.net.SocketTimeoutException: Read timed out
...ConnectException: Connection refused: connect ...
                          Inconclusive        129 succ          21 fail 8.1 secs
                          Concurrent          0 inco false cons 3.5 secs .. 8.9 secs
```

which is Inconclusive, meaning inconsistency of behaviour, with a mix of successes and fails, and reported Java exceptions of "java.net.SocketTimeoutException: Read timed out" and "java.net.ConnectException: Connection refused: connect". The former means invocation timeout and the latter means unable to connect at all. Overall, the results indicate that under the specified concurrent load of 150 the Lender service is not able to function consistently. The hosting server (Apache Tomcat) was observed to report "All threads are busy, waiting. Please increase maxThreads or check the servlet status" and "java.lang.OutOfMemoryError: Java heap space" errors when concurrent validation was the only activity in progress at the server. This reveals that the server is not configured with adequate resources to host the web services with the expected performance requirements. The java.net.SocketTimeoutException was most likely due

to the server not being able to process the request in time within the timeout threshold. Over time, the Java heap space ran out for the engine, which led to the java.net.ConnectException as the container could not acquire memory resources for processing the load of tests. The server was then configured with adequate threads and Java heap size by respectively increasing the maxThreads attribute in Tomcat's server.xml and Java heap size for the server (e.g. via environment variable CATALINA_OPTS). The concurrent performance evaluation was executed again, and the configuration monitored and altered until the service exhibited consistency under this performance load.

7.2.10 Evaluation Compared To Other Approaches

A comparison is made with respect to the thesis goal of an integrated rigorous methodology for development of composite web/grid services, with automated support for specification, validation, verification, implementation and evaluation, and with abstraction of the underlying implementation and analysis techniques. Related work in (WSAT [31], the LOTOS approach by Ferrara [24], PEPA [71], and LTSA-WS [28]) are most relevant with regard to this comparison. This related work also applies or supports different degrees of rigorous development using the same Loan Approval Process as discussed above [4, 6]. Other web service composition methodologies, especially the implementation ones such as JOpera, support many if not most of the compositional constructs, but they generally do not have a rigorous aspect; there is a degree of systematic testing in ActiveBPEL (BUnit). The specific characteristics in the comparison are the coverage of BPEL orchestration constructs, support for interacting processes, automated formalism, automated support for validation and verification, automated support for implementation and deployment, automated support for testing, and the abstraction framework.

7.2.10.1 Supported Composition Activities

BPEL constructs are categorised into basic, structured, data handling, scopes, and correlation [4, 6]. The coverage of constructs is based on that stated by authors of the related work. A tick indicates an aspect is fully supported; a minus sign indicates partial support. Basic activities include receive, invoke and reply. Structured activities includes alternative (switch, if-else), iteration, sequence, pick and flow. Data handling refers to the notion of data types, variables,

assignment, and expressions. Error, event and compensation handling are scope activities, which in some cases are generally referred to as event handling [78]. In addition there is data structure definition in WSDL which BPEL processes will use. Table 7.9 gives a comparison showing the coverage of BPEL constructs of the related approaches. The Loan Approval example [4, 6] does not involve the use of all the BPEL constructs such as compensation and correlation.

	WSAT	**Ferrara et al.**	**PEPA**	**LTSA-WS**	**Thesis**
Basic	✓	✓	✓	✓	✓
Structured	✓	✓	✓	✓	✓
Scope		–	–	–	–
Correlation					
Data Structure	✓	✓			✓
Dynamic Partners					✓
BPEL4WS	✓	✓	✓	✓	✓
WS-BPEL 2.0					✓
Interacting BPEL	✓			✓	✓

Table 7.6: Support of BPEL Constructs

7.2.10.2 Analysis and Tool Support

Table 7.7 shows the comparison with regard to the automated and abstract support for specification, implementation, analysis of web service compositions. The thesis shares similarities in the type analysis that can be performed with the related work as they are also process algebra based, except for PEPA (performance-focused). The automated support and strength of analysis differs.

	WSAT	**Ferrara et. al.**	**PEPA**	**LTSA-WS**	**Thesis**
Specification	✓	✓	✓	✓	✓
Abstraction				✓	✓
Verification			✓	✓	✓
Validation				✓	✓
Implementation	✓	✓			✓
Testing					✓

Table 7.7: Automated Tool Support of Web Service Composition Approaches

- WSAT only has automated support for the translation of BPEL implementation to specification into Promela. This was applied in the formalisation of the Loan Approval composition. Two temporal logic LTL properties are handcrafted and manually verify the model using SPIN: a loan request will eventually get an output or fault message; if the request amount is greater than 2, it will eventually get rejected (a false property). The authors limit the integer domain to 4 for the (seven) integer variables generated in the specification. The manual analysis includes the notion of data values, as seen from the second property. The tradeoff for this analysis is the state space explosion. Verification of the first property completes in about 2 minutes, while the second property takes 0.2 seconds.

- Ferrara's approach automates translation between a LOTOS specification and a BPEL implementation. The approach demonstrates translation of a Loan Approval specification into BPEL4WS. The approach does not provide any (automated or abstracted) tool support or guidelines for analysis, implying a requirement for expertise (e.g. CADP) that developers may not have [75]. It provides only general suggestions for formal analysis such as bisimulation for service redundancy, request-response property verification, and black box testing, which could be manually applied to the Loan Approval composition.

- LTSA-WS abstracts underlying techniques and tools in an effort to simplify and make analysis more accessible. This was achieved by using high-level notations, automated specification and analysis. LTSA-WS uses UML in design and analysis (validation and verification) diagnostics. Validation is performed interactively in LTSA-WS by animating the model. Deadlock freedom property can also be checked on the LTS translated from BPEL4WS code. Trace equivalence is automatically checked between the LTS models generated from MSCs and BPEL4WS; the latter implies code is already implemented. Other properties are specified in FSP for verification, such as request-response and safety properties. Although not demonstrated for Loan Approval, the automated analysis of trace equivalence verification can be performed in LTSA-WS if the MSCs of the composition (designer role) are specified, which can detect errors such as interface incompatibility in the LTS translated from BPEL4WS. The LTS models can be animated

for validation through the LTSA-WS tool. The supported analysis, however, does not include data semantics like WSAT which can detect errors in the composition specific to data values. There is no support for implementation validation.

- The PEPA approach automates annotated BPEL/WSDL translation to PEPA. Analysis is directed at performance evaluation in consideration of latency and operating timing. This was applied to the Loan Approval composition to demonstrate the automated translation and analysis. However, the analysis does not demonstrate how this automated analysis benefited the development of this service composition.

- The thesis uses CRESS, CLOVE with verification patterns, MUSTARD, and MINT for design and automated support of specification, implementation, deployment and analysis. The methodology was applied on the Loan Approval composition (lender service), including its partner services. The thesis approach enables verification of properties similar to WSAT, but the thesis approach can be more specific such as with data values. In addition, there is support for abstract property specification and tool automation which WSAT does not provide. Verification templates are provided for well-known property patterns [67] (e.g. Lender request-response properties), abstracting the temporal logic syntax and supporting data values. By default a specification is checked for freedom from deadlock and livelock. The verification shares the same tradeoff as WSAT in model checking; depending on the domain data values, the time and state space of verification varies. The verification of similar response properties to WSAT, for Loan Approval composition is completed in several minutes. This is achieved with data range of 2000 distinct numbers and five distinct enumerations of strings which are used in the complex Proposal type, generating a state space comprising $\sim$104 thousand states, $\sim$204 thousand transitions and $\sim$50 thousand labels. Validation is automated for the specified scenarios; these can include data values. The approach has demonstrated that it can detect errors in in specification and implementation. The latter uses black box testing with a degree of load testing which is not supported or integrated in the other compared approaches. Formal performance analysis is not supported, although it is possible as performance analysis can be undertaken on a LOTOS specification using CADP, this could be investigated in

future work.

7.3 Development of Allocator Composed Grid Service

The Allocator service brings together two resource-related grid services, Factory and Mapper, that perform occupational matching of survey datasets. The Factory grid service coordinates the allocation of resources to map occupations. The Mapper grid service performs the task of mapping occupational information using the selected resource. The Factory and Mapper services are developed according to the WS-Resource factory pattern whereby the former is the factory that creates resources and the latter is the 'instance' service that performs operations. Factory and Mapper therefore share the same resource context.

The Allocator service combines the two services within its process behaviour to provide a complete service for mapping an occupation, which includes the coordination of resource allocation, data mapping, resource deallocation, and event handling. The Allocator has a 'translate' operation that maps an occupational title to a specific classification code. The process flow is to have the Factory service allocate a mapping resource that is accessible via an endpoint. This addresses a particular Mapper service resource to perform the mapping. Upon successful mapping by the Mapper, the Allocator then returns the result to its client. Otherwise, if the invocation of Mapper results in a fault, then the resource is deallocated through the Factory, and Allocator returns a fault to its client.

The developer describes the composed behaviour in CRESS, partner specifications and implementations, MUSTARD scenarios, and CLOVE properties. The composite service specification is then automatically generated for analysis. The developer executes the automated validation and verification, analyses the respective results, and addresses the issues identified by the analyses. Once satisfied with the analyses, the developer starts the automated implementation. This is followed by the automated implementation validation and performance evaluation on the deployed services using the same MUSTARD scenarios used for specification, addressing implementation and resource configuration issues that are discovered.

7.3.1 Service Diagrams

7.3.1.1 Allocator

The developer describes the Allocator service behaviour in CRESS, as illustrated in figure 7.7. This is a representation of the requirements given in the informal description. The rule box declares the user-defined data types and variables used by Allocator and its partner services, similar to those in the previous case studies. In particular, there is a declaration of dynamic partner Mapper at its port *job*. There is a **Reference** type variable named mapperReference which is used to hold the endpoint reference to the resource created by Factory. This is associated with Mapper's *job* port after Factory invocation.

The Allocator has one operation, *translate*, that is described in node 1 – **Receive** with the complex type *mapping*. The variable *scheme* is assigned the field *scheme* in *mapping*, followed by invoking factory.job.allocate (node 2) which allocates a resource according to the value in *scheme*. The reference to the created resource is returned and held in variable *mapperReference*. This reference is assigned to *mapper.job* which binds to the WS-Resource. The variable *job2* is assigned the value of the field *mapping.job* which is used in the **Invoke** activity in node 3. This asks the Mapper to perform the occupational translation of *job2* according to the classification indicated by *scheme2*. The translated result *mappedJob* is returned by Allocator in node 4. The allocation by Factory in node 2 may throw a fault, which is handled by the **Catch** *factoryError.reason* arc leading to node 5. The translation in node 3 may throw a fault, which is handled by a **Catch** .reason matching any fault name and leading to node 6. If this event occurs, then an invocation of Factory is made to deallocate the resource, followed by returning the fault value with *allocatorError* fault name at node 7 before terminating at node 8.

7.3.1.2 Service Configuration

The developer defines the service configuration illustrated in figure 7.8. Most of the **Deploys** options should now be familiar, with the exception of the '-m RESOURCE' parameterised option for 'merge partners', which will generate a behaviour specification for a phantom partner RESOURCE. This is used to describe the interaction of the shared resource context between Factory and Mapper.

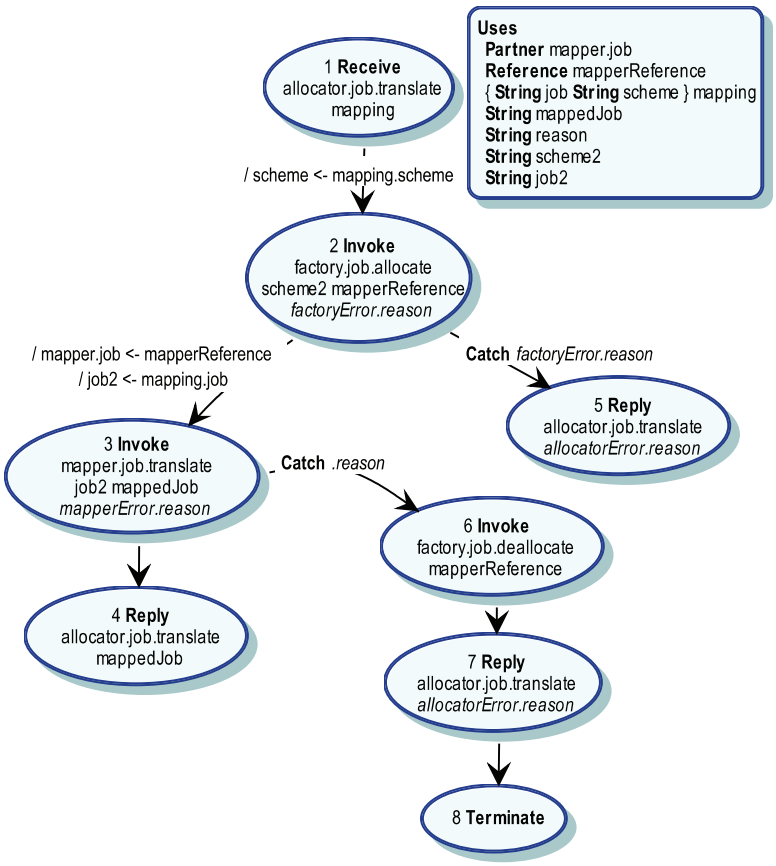

Figure 7.7: Allocator CRESS Diagram

The deployment location of Factory and Mapper is the Globus Toolkit container, distinguished by the base URL at port 8880 as the operating port used in this example.

```
Deploys -a -b 2 -c -n 1 -o 5 -r -m RESOURCE / ALLOCATOR

ALLOCATOR   allo     urn:Allocator   localhost:8080/active-bpel
FACTORY     factory  urn:Factory     localhost:8880/wsrf        -
MAPPER      mapper   urn:Mapper      localhost:8880/wsrf        -
```

Figure 7.8: Grid Service Configuration

7.3.2 Partner Specification

The Resource, Factory and Mapper partner specifications are manually provided by the developer. The complete Factory specification is illustrated below:

```
Process FACTORY [factory,resource] : Exit(States) :=    (* FACTORY partner *)          1
  factory !job !allocate ?scheme:Text;                  (* get scheme name *)          2
  resource !scheme;                                     (* send scheme name *)         3
  (                                                                                     4
```

```
    resource !True ?epr:Nat;                          (* get EPR *)                  5
    factory !job !allocate !epr;                      (* return mapping EPR *)       6
    FACTORY [factory,resource]                        (* repeat behaviour *)         7
  []                                                  (* or *)                       8
    resource !False ?reason:Text;                     (* get fault *)                9
    factory !job !allocate !factoryError !reason;     (* return factory fault *)     10
    FACTORY [factory,resource]                        (* repeat behaviour *)         11
  )                                                                                   12
[]                                                                                    13
  factory !job !deallocate ?epr:Nat;                                                  14
  FACTORY [factory,resource]                          (* repeat behaviour *)         15
EndProc                                               (* end FACTORY *)              16
```

Assuming the above specification is not provided, executing automated formal specification will generate the following interface behaviour.

```
Process FACTORY [factory,resource] : Exit(States) :=  (* FACTORY partner *)          1
  factory !job !allocate ?scheme2:Text;               (* 'allocate' input *)         2
  (                                                                                   3
    factory !job !allocate !AnyNat;                   (* 'allocate' output *)        4
    FACTORY [factory,resource]                        (* repeat behaviour *)         5
  []                                                  (* or *)                       6
    factory !job !allocate !FactoryError !AnyText;    (* 'FactoryError' fault *)     7
    FACTORY [factory,resource]                        (* repeat behaviour *)         8
  )                                                                                   9
[]                                                    (* or *)                      10
  factory !job !deallocate ?mapperReference:Nat;      (* 'deallocate' input *)      11
  FACTORY [factory,resource]                          (* repeat behaviour *)        12
EndProc                                               (* end FACTORY *)             13
```

7.3.3 Partner Implementation

This section presents the implementation provided for Factory and Mapper. Resource is actually implemented as the AllocatorResource WS-Resource in the Factory implementation and configured to be shared with Mapper.

7.3.3.1 Factory Service

The Factory service implements the operations 'allocate' and 'deallocate'. There is also the need to implement the 'resource home' and 'resource' that the Factory uses to manage resources. This follows the GT4 resource context framework, where the former creates the resource in the container and the latter is the resource itself which contains the allocated scheme information to return mapped values. These are respectively implemented as AllocatorResourceHome and AllocatorResource Java classes. The implementation uses the code stubs

generated from its WSDL service interface, along with common definitions which were also automatically generated. These are packaged into the grid service archive for deployment, together with the developer's specified WSDD (Web Service Deployment Descriptor) and JNDI (Java Naming and Directory Interface) resource configuration. The WSDD is a basic one for GT4, with values configured only for the Factory service name and service WSDL location. The JNDI configures the namespace and resource class implementation that the Factory uses.

The 'allocate' operation takes a String which contains the occupational scheme to be allocated, and invokes AllocatorResourceHome to create an instance of AllocatorResource hosting the scheme for mapping use. Upon successful resource creation, the operation returns an endpoint to the WS-Resource which the Mapper service can use. The 'deallocate' operation takes the endpoint to the WS-Resource, and uses AllocatorResourceHome to remove the identified resource. The Factory code snippet is shown in the Java code below. The Factory's grid service archive is deployed into the GT4 container.

```
package uk.ac.stir.cs.factory;

... imports ...

public class FactoryService {

   public EndpointReferenceType allocate(String scheme)
            throws RemoteException, StringMessage {

      ...
      home = (AllocatorResourceHome) ctx.getResourceHome();
      key = home.create(scheme);
      ...

   EndpointReferenceType epr = null;

   ... create the value of epr ...

   return(epr);
}

   public void deallocate(EndpointReferenceType reference)
            throws java.rmi.RemoteException {
      ResourceContext context = ResourceContext.getResourceContext();
      AllocatorResourceHome home =
                  (AllocatorResourceHome) context.getResourceHome();
      ResourceKey resourceKey = this.getAsResourceKey(reference);
      if (resourceKey != null) {
         AllocatorResource resource = (AllocatorResource) home.find(resourceKey);
         home.remove(resourceKey);
```

```
            ...
        }

    private ResourceKey getAsResourceKey(EndpointReferenceType epr)
            throws Exception {
        ...
    }
}
```

7.3.3.2 Mapper Service

The Mapper service implements the 'translate' operation which associates resources dynamically. This is done by Allocator setting mapper.job dynamically with the endpoint value returned by Factory. The 'translate' operation uses the associated resource to obtain the mapped value as a String and returns it. This is just the few lines of code shown below. Mapper uses the resource implementation already developed within the Factory service. Its generated WSDL service interface and common definitions are packaged into a grid service archive for deployment, together with the developer's specified WSDD and JNDI resource configuration. The WSDD is mostly similar to that of Factory apart from values pertaining to the Mapper service and its WSDL location. Its JNDI file is simply configured as a reference to the resource configured in Factory, establishing sharing of the resource context. The Mapper's grid service archive is deployed into the GT4 container.

```
AllocatorResourceHome home = (AllocatorResourceHome) context.getResourceHome();
AllocatorResource allocatorResource = (AllocatorResource) context.getResource();
return allocatorResource.mapValue(string.value);
```

7.3.4 CLOVE Properties

The informal description of the properties desired of the service is as follows:

- Allocator's translation request should result in either a successful translation, or faults of unknown job or scheme.
- Allocator should begin its service with no other operations apart from the 'translate' operation that takes a Mapping request.
- Freedom from deadlock.
- Freedom from livelock.

The developer specifies the value enumerations and properties in the following CLOVE syntax. There is a specification of the enumeration of strings. Note that there is no explicit

definition of the Mapping values; however, these string values will be used in the construction of the value of Mapping, enumerating the two string fields in the structure to a total combination of 36 different Mapping values (6×6) where there are both valid and invalid values. These are sufficient to verify all of Allocator's key behaviour. In executing the verification, there is automated annotation of the LOTOS specification, property translation to μ-calculus, and generation of a C implementation for data enumeration. There is use of a regular expression in the General_Response property to match the string values for successful job mapping translations returned by Allocator, instead of using '?string' which only caters for spaces and alphanumeric characters. This expression is specified verbatim in μ-calculus syntax instead of using the **signal** construct. The C data enumeration is only one of line code; it is effectively a macro that lists the values of the six strings specified.

```
enum_strings('SOC2000,'SIC92,'bookbinder,
                'cab driver,'nurse,'private detective)

initials(signal(allocator.job.translate,?mapping))

property(General_Response,
  response(global,
    signal(allocator.job.translate,?mapping),
      choice_any(signal(allocator.job.translate, '"[./a-zA-Z0-9]*"'),
        signal(allocator.job.translate,allocatorError,'unknown scheme),
        signal(allocator.job.translate,allocatorError,'unknown job))))
```

7.3.5 MUSTARD Scenarios

Validation of grid partner services is not performed as the composed service is rather straightforward and will definitely test both the partner services indirectly with a variety of Mapping values to Allocator. The Unknown_Nurse test is a specific safety validation, where a possible mapping of a nurse must be unsuccessful.

```
test(SOC2_Nurse,
  succeed(
    send(allocator.job.translate, Mapping('Nurse,'SOC2000)),
    read(allocator.job.translate, '3211)))

test(SIC_Nurse,
  succeed(
    send(allocator.job.translate, Mapping('Nurse,'SIC92)),
    read(allocator.job.translate, '95.14)))
```

```
test(SOC2_ Unknown,
   succeed(
      send(allocator.job.translate, Mapping('Sailor,'SOC2000)),
      read(allocator.job.translate, allocatorError, 'Unknown job)))

test(Unknown_ Nurse,
   refuse(
      send(allocator.job.translate, Mapping('Nurse,'Unknown)),
      read(allocator.job.translate, '3211)))

test(SOC2_ Unknown_ Scheme,
   succeed(
      send(allocator.job.translate, Mapping('Sailor,'SOC20000)),
      read(allocator.job.translate, allocatorError, 'Unknown scheme)))
```

7.3.6 Formal Specification

The developer executes the automated formal specification. The behaviour for Allocator is fully specified, and includes the manually specified behaviour of all the partners. For the overall behaviour listed in the LOTOS code below, the ALLOCATOR process is synchronised with the RESOURCE process at its gate. The actual behaviour of the Allocator begins from the ALLOCATOR_1 process call, and does not interact with RESOURCE. FACTORY and MAPPER are exposed to the 'resource' gate in the behaviour expression of the ALLOCATOR process, which allows them to synchronise.

```
Specification GSSystem [allocator] : Exit(States)        (* GS system *)

Library                                                  (* library *)
...
Behaviour
Hide resource In                                         (* hide internal gates *)
   RESOURCE [resource]                                   (* call RESOURCE process *)
|[resource]|                                             (* synchronised with services *)
   ALLOCATOR [allocator,resource]                        (* call ALLOCATOR process *)

Where                                                    (* local definitions *)
Type Mapping Is BaseTypes                                (* mapping record *)
   Sorts Mapping                                         (* mapping sort *)
   Opns                                                  (* mapping operations *)
      AnyMapping: -> Mapping
      _eq_,_ne_: Mapping,Mapping -> Bool
      mapping: Text,Text -> Mapping
      getJob: Mapping -> Text
      setJob: Mapping,Text -> Mapping
      getScheme: Mapping -> Text
      setScheme: Mapping,Text -> Mapping
...
```

```
Process ALLOCATOR [allocator,resource] : Exit(States) := (* ALLOCATOR service *)
  Hide factory,mapper In                                 (* hide internal gates *)
    (
      FACTORY [factory,resource]                         (* call FACTORY partner *)
    |||                                                  (* interleaved with *)
      MAPPER [mapper,resource]                           (* call MAPPER partner *)
    )
  ||[factory,mapper]||                                   (* synchronised with partners *)
    ALLOCATOR_1 [allocator,factory,mapper]               (* call main process *)
     (AnyText,AnyText,AnyNat,AnyNat,AnyMapping,AnyText,AnyText)

Where                                                    (* local definitions *)
Process FACTORY [factory,resource] : Exit(States) :=     (* FACTORY partner *)
...
Process MAPPER [mapper,resource] : Exit(States) :=       (* MAPPER partner *)
...
Process RESOURCE [resource] : Exit(States) :=            (* RESOURCE phantom *)
...
```

Apart from manual specification of partners Factory, Mapper, and Resource, the formal specification is fully automated with all data types and behaviour specified. Even the manual specification itself is added to the interface behaviour that is automatically generated. Using this specification, formal validation and verification can be performed. Table 7.8 lists the specification summary in terms of lines of code generated and manually provided.

Automated (Allocator)	**Manual** Factory	**Manual** Mapper	**Manual** Resource
479	24	22	91

Table 7.8: Specification Summary of Allocator, Factory, Mapper, and Resource

7.3.7 Formal Analysis

For illustration the Allocator composed service is subject to verification followed by validation, as analysis can be performed in any order.

7.3.7.1 Formal Verification

Verification is performed in non-compositional mode with the following results:

```
Verifying ALLOCATOR GENERAL RESPONSE ...    TRUE
Verifying ALLOCATOR SOC2000 SAFETY ...      TRUE
Verifying          DEADLOCK FREEDOM ...     TRUE
Verifying          LIVELOCK FREEDOM ...     TRUE
```

```
Verifying        INITIALS SAFETY ...        TRUE
```

7.3.7.2 Formal Validation

Validation is executed for the specified scenarios with the results listed below. These validation results, along with the verification outcomes, establish a level of confidence in the behaviour of the services, where specified properties are satisfied and the tests demonstrate correct functionality. As a complement to verification, the validation checks for scenarios beyond the finite state space of the Allocator, using values such as “Sailor”and “SOC20000” not found in the enumeration for verification – this gives confidence that the behaviour is still correct.

Test ALLOCATOR SOC2 Nurse ...	Pass	1 succ	0 fail	0.6 secs
Test ALLOCATOR SIC Nurse ...	Pass	1 succ	0 fail	0.5 secs
Test ALLOCATOR SOC2 Unknown ...	Pass	1 succ	0 fail	0.4 secs
Test ALLOCATOR Unknown Nurse ...	Pass	1 succ	0 fail	0.4 secs
Test ALLOCATOR SOC2 Unknown Scheme ...	Pass	1 succ	0 fail	0.4 secs

7.3.8 Implementation

Implementation of the Allocator service is fully automated. It compiles the developer-provided implementation for the Factory and Mapper grid service partners which are deployed to the GT4 framework. This is illustrated in figure 7.9.

The grid service implementations provided by the developer for Factory and Mapper are automatically compiled and built using GT4, and packaged as grid service archives (.gar) for deployment in GT4 container. For the Allocator service, its WSDL service interface, deployment descriptors (PDD and WSDL catalogue), and BPEL code are automatically generated, compiled into a BPEL archive, and deployed in ActiveBPEL. The PDD (Process Deployment Description) code specifies the Mapper partner as dynamic, where its endpoint (WS-Addressing) for the WS-Resource is bound during Allocator’s execution in order to invoke it.

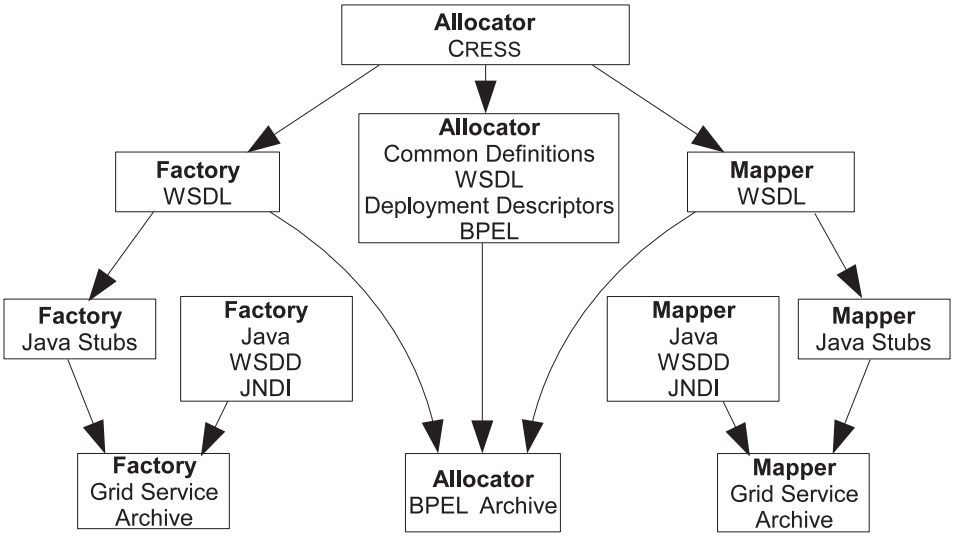

Figure 7.9: Implementation of Factory and Mapper

7.3.9 Implementation Validation

7.3.9.1 Allocator Implementation Validation

The deployed service can be validated much as for the formal validation. This requires the MINT properties that were also generated previously in the automated implementation, and translation of the MUSTARD scenarios into MINT scenarios for test interpretation by MINT. The Allocator's service is expected to respond within 5 seconds.

The results below show that validation passes, which agrees with specification validation, confirming that the implementation behaviour is functioning the same as the specification with regard to these scenarios. Otherwise, the Allocator service diagram and the partner service implementations should be inspected and corrected using the diagnostic traces provided as feedback.

Test ALLOCATOR SOC2 Nurse ...	Pass	1 succ	0 fail	1.8 secs
Test ALLOCATOR SIC Nurse ...	Pass	1 succ	0 fail	1.3 secs
Test ALLOCATOR SOC2 Unknown ...	Pass	1 succ	0 fail	1.3 secs
Test ALLOCATOR Unknown Nurse ...	Pass	1 succ	0 fail	1.2 secs
Test ALLOCATOR SOC2 Unknown Scheme ...	Pass	1 succ	0 fail	1.2 secs

7.3.9.2 Performance Evaluation

Suppose the service is expected to handle up to 200 mappings simultaneously. The developer first evaluates the consistency of behaviour by sequential performance testing, where the results

reported below show that behaviour is consistent.

Test ALLOCATOR SOC2 Nurse ...	Pass	200 succ	0 fail 0.1 secs
Sequential	0 inco true cons 0.1 secs .. 1.1 secs		
Test ALLOCATOR SIC Nurse ...	Pass	200 succ	0 fail 0.1 secs
Sequential	0 inco true cons 0.1 secs .. 1.1 secs		
Test ALLOCATOR SOC2 Unknown ...	Pass	200 succ	0 fail 0.1 secs
Sequential	0 inco true cons 0.1 secs .. 1.2 secs		
Test ALLOCATOR Unknown Nurse ...	Pass	200 succ	0 fail 0.3 secs
Sequential	0 inco true cons 0.2 secs .. 1.1 secs		
Test ALLOCATOR SOC2 Unknown Scheme ...	Pass	200 succ	0 fail 0.4 secs
Sequential	0 inco true cons 0.2 secs .. 1.1 secs		

The concurrent performance evaluation with a load of 200 requests also passes, but with a much slower completion time than the sequential evaluation (which is reasonable). The grid partner services were also indirectly tested through invocations from the Allocator BPEL service, which implies that the grid partner services and their environment (GT4 container) can handle this load.

Test ALLOCATOR SOC2 Nurse ...	Pass	200 succ	0 fail 8.6 secs
Concurrent	0 inco true cons 7.5 secs .. 9.2 secs		
Test ALLOCATOR SIC Nurse ...	Pass	200 succ	0 fail 6.2 secs
Concurrent	0 inco true cons 5.0 secs .. 7.7 secs		
Test ALLOCATOR SOC2 Unknown ...	Pass	200 succ	0 fail 5.3 secs
Concurrent	0 inco true cons 4.7 secs .. 6.4 secs		
Test ALLOCATOR Unknown Nurse ...	Pass	200 succ	0 fail 7.8 secs
Concurrent	0 inco true cons 4.6 secs .. 9.5 secs		
Test ALLOCATOR SOC2 Unknown Scheme ...	Pass	200 succ	0 fail 6.8 secs
Concurrent	0 inco true cons 4.2 secs .. 8.4 secs		

7.3.10 Evaluation Compared To Other Approaches

There has been more work on realising grid service composition in contrast to the limited related work that supports their rigorous development [42, 127]. Considering automated support for specification, validation, verification, and implementation (particularly BPEL), evaluation, and abstraction of the underlying implementation and analysis techniques, the PGSCV [42] and CPi-calculus [127] approaches are relevant to the formal aspects. However these approaches focus on formalising grid service compositions in general, and do not involve implementation (WSDL/BPEL) of a specification. Their support of composition constructs are not clearly stated and demonstrated. For example, event, compensation and fault handling are not explicitly presented; only the constructs similar to BPEL basic and structured activities are explicit.

JOpera, OMII-BPEL, and ActiveBPEL are methodologies supporting the implementation of grid service compositions in addition to web services, the latter two implementing the BPEL standard.

7.3.10.1 Supported Composition Activities

The comparison of support compositions is made with regard to the BPEL constructs. The PGSVC and CPi-calculus do not explicitly support BPEL, but their semantics are related in the compared categories. A tick indicates an aspect is fully supported; a minus sign indicates partial support.

	PGSCV	**CPi-calculus**	**JOpera**	**OMII-BPEL**	**ActiveBPEL**	**Thesis**
Basic	✓	✓	✓	✓	✓	✓
Structured	✓	✓	✓	✓	✓	✓
Scope			✓	✓	✓	–
Correlation			✓	✓	✓	
Data Structure			✓	✓	✓	✓
Dynamic Partners		✓	✓	✓	✓	✓
Interacting Processes	✓	✓	✓	✓	✓	✓
BPEL				✓	✓	✓

Table 7.9: Support of Composition Constructs

7.3.10.2 Analysis and Tool Support

Table 7.10 shows the comparison with regard to the automated and abstract support for specification, implementation, and analysis of grid service composition.

- PGSCV has automated support for specification of the interaction patterns; individual behaviour has to be manually specified. Verification is automatically performed to see if the grid service composition is correct, meaning the interaction or coordination can be smoothly performed among grid services. This could (in principle) be applied to the Allocator composition, where an analyst specifies the interaction patterns and executes

	PGSCV	**CPi-** calculus	**JOpera**	**OMII-** BPEL	**ActiveBPEL**	**Thesis**
Specification	–					✓
Abstraction (F/I)			–	–	–	✓
Verification	✓					✓
Validation						✓
Implementation			✓	✓	✓	✓
Testing					✓	✓

Table 7.10: Automated Tool Support of Grid Service Composition Approaches

the PGSCV tool to analyse these for coordination correctness. PGSCV does not cover implementation, deployment, and implementation analysis (testing), as the focus is on formal aspects; these aspects are thus not applicable to the Allocator composition and its partners.

- CPi-calculus work not implement automated tool support for compositions as it only proposes the foundation framework and semantics for modelling. It could be applied to formalisation of Allocator only on a manual basis. The analyst would specify the Allocator composition based on the proposed semantics.

- JOpera, OMII-BPEL and ActiveBPEL Designer are focused on implementation. They have graphical abstraction to the underlying composition languages SCUFL (for JOpera) and BPEL, and automated support for their implementation and deployment. In addition, they provide the composition execution/enactment environment and over runtime support such as monitoring. These approaches could fully describe and implement the Allocator composite service, only requiring implementation of the partner services (Mapper and Factory) separately. ActiveBPEL Designer supports a degree of testing within its development environment through its BUnit module. This enables invocation of services with configured ranges of data, which would be applicable to the Allocator composition and partner services. There is overlap in the approach to abstraction and automated implementation of composed services, but these three approaches are by no means compete with the thesis approach. They complement it as potential candidates for target service deployment environments of the thesis approach, and could benefit from its rigorous development features.

- BPEL4WS has been demonstrated to be applicable for grid service composition [126] and the WS-BPEL 2.0 has a more harmonised compatibility with WSRF specification. Approaches such as WSAT and Ferrara's approach as well as others [78, 82] that support these BPEL standards could potentially be applied to a rigorous development of the Allocator case study to some extent, notably in the formal aspects.

7.4 Evaluation

Two case studies have been used to illustrate the methodology in action from the developer's perspective. There are other service composition case studies produced earlier in [91, 92, 106, 107, 108] to demonstrate the methodology in practice. Service compositions are described in the CRESS high-level graphical notation. Formal specifications and implementations are automatically generated, enabling service prototyping and development. Partner service specifications and implementations are semi-automated, generating code when possible to reduce the developer's effort. Formal validation and verification are supported with high-level analysis descriptions (scenarios and properties). Formal analyses are automatically executed and interpreted on behalf of users, which simplifies the technical interface to underlying tools yet supports rigorous service development. Post-implementation analysis is available as implementation validation where functionality testing and performance can be evaluated, supported by a high-level notation and automated tool support. The integration into a methodology enables rigorous development of composed services.

Chapter 8

Conclusions

8.1 Thesis Summary

The thesis has presented an integrated methodology for the rigorous development of composed web and grid services. The overall approach is based on the high-level and domain-independent CRESS methodology, which originally supported automated specification, validation, deployment and implementation of composite web services. The thesis work has extended the methodology to support extended descriptions, extended standard support, grid services, formal verification, implementation validation and performance evaluation, which together offer a more rounded development methodology. Chapter 3 illustrated how the integrated methodology is used, covering the development lifecycle of web and grid service compositions.

Chapter 4 showed how CRESS is used for describing composite web services, the approach that underpins translations to actual languages, the thesis extensions to the notation and framework to support grid services, and the integration of the new tools developed by the thesis.

Chapter 5 discussed the approach to the automated specification of composed services, the exploitation of formal validation which was extended to partner services, and the approach and development of CLOVE that acts as an automated verification tool. Rigorous analysis is automated and abstracts the underlying techniques. It can help the effort of analysis as specification and analysis are automated. Integration of the tools into the methodology was also outlined.

The strategy of automated implementation, and extensions to support WS-BPEL 2.0, were covered in Chapter 6. This also discussed in detail the MINT tool which was developed to automate implementation validation and also performance evaluation that were integrated into the methodology. This was supplemented by case studies demonstrating the methodology in Chapter 7.

8.2 Evaluation

8.2.1 Strengths and Weaknesses

The methodology automatically generates the formal specification and executable implementation for composite and partner services by translating the graphical, language-independent CRESS service descriptions. This approach is extensible to other target languages and their technologies, e.g. the support of WS-BPEL 2.0. Formal specifications in LOTOS are automatically generated which can be readily analysed by tools (MUSTARD and CLOVE) integrated into the methodology. This automation helps in the effort to support developers in formal analysis, and also to help formal experts who can apply their own analyses apart from those supported by the methodology. Likewise the high-level and automated approach to formal validation and verification enable effective and productive analysis of the generated specifications, contributing to service quality. In addition, MUSTARD and CLOVE can be used independently from the methodology to translate high-level and easy-to-define scenarios and properties into actual analysis syntax, which can be hard to read and write without the automated help.

Validation is carried out on the basis of specific scenarios which may not catch some errors; however, it provides opportunities to do so with an automated approach to quickly analyse and understand the model. Verification complements validation where general analysis such as deadlock freedom and property checking across the scope of the system can be easily verified. There is also support for frequently used verification properties, made available as templates. Verification using CADP has a limitation of not being able to analyse behaviour in large state spaces as it is a enumerative technique. This technique is pragmatic, cost effective, and has the potential to discover errors using finite state systems [18]. Verification is currently available for composed service behaviour but not partner services.

Service development is possible since implementation of composed services is fully automated. The methodology automates code generation to a high degree (WSDL, code stubs, skeletons), and supports the deployment of partner services. The CRESS notation for web and grid service composition does not support the entire BPEL specification, however the major constructs are supported and are sufficient to create realistic services. Implementation of services currently supports deployment in AXIS (partner web services), ActiveBPEL (composed services), and GT4 (partner grid services). The methodology is extensible and can be further developed to support other implementation languages and environments. There is post-deployment support in the methodology, where the integrated MINT tool can be used to execute implementation validation on the deployed services including partners, reusing the MUSTARD scenarios from the specification stage. Performance evaluation is supported by MINT, which automates load testing using scenarios from the implementation validation. This helps in gaining insight into the resource configuration that supports the target services. The black-box validation approach is limited to external analysis as it cannot inspect the target's internal behaviour, such as monitoring components within hosting environments [8] . The tradeoff is that the approach is lightweight, portable, and not constrained to any particular implementation or environment.

8.2.2 Future Work

The CRESS notation could be further extended to support the description of more realistic service compositions, such as timing (onAlarm) constructs and correlation in BPEL. This implies that the translation into formal specifications and implementations would also need extensions. Formal semantics will have to be defined for new constructs, and approaches that support timing semantics such as time extended LOTOS [60] might be used. The automated analysis could be improved such as interactive system simulation and graphical feedback (e.g. CRESS notation) on the analyses and their results. It is possible to address the constraints of manually adapting generated specifications in the case of recursive process instantiation (as has now been done). Symbolic verification [15, 68] could be applied to increase efficiency with regard to time and resources (memory) as they are techniques to address state space explosion. Automated

test generation [96] for LOTOS specifications can be used to assist the effort in validation, for example automatically specifying a set scenarios for the developer. Performance evaluation of LOTOS specification is supported in CADP [34]. If it could be automated, it would expand the range of verification such as deadlock freedom in context of probabilistic load. The CRESS notation could be extended to specify stochastic parameters which could be translated into the specification and analysed through CADP. Implementation performance evaluation could be improved by a means to distribute and coordinate performance tests across a network of machines instead of being limited to one machine as at present with MINT. Another aspect is looking at the analysis of service compositions in consideration of the environment, such as resource allocation [27]. There has been development of automatic synthesis for composable web service where automata modeling behaviour protocols can be derived from its WSDL [9]. This technique can be investigated for the automated formalism of existing auxiliary and externally owned partners services but with no formal specification, deriving information about their behaviour which is then used for service composition.

8.2.3 Concluding Remarks

There is increasing acknowledgement that formal methods are applicable to service-oriented computing such as web service composition [25], as is evident in the various research efforts that are trying to encourage the use of formal methods. It is hoped that the methodology and tools developed in this thesis will contribute to the development of web and grid service composition in a rigorous manner.

Appendix A

CRESS to LOTOS Translation

CRESS Syntax	**LOTOS Translation**
Text, e.g. “high”	t(h) ∼ii ∼g ∼h operation ‘t’ creates a Text value from a character and ‘∼’ concatenates characters with text resulting new text the letter ‘i’ is represented as ‘ii’ in LOTOS as ‘i’ is a reserved keyword representing an internal event
Number, e.g. 3.5	number(+,t(3),t(5)) Operation ‘number’ creates a number value (CRESS base type) with parameters comprising a sign character (+ or -), text for whole number, and text for fraction
Assignment simple /rate <- 3.5	**Let** rate:Number = number(+,t(3),t(5)) **In**
Assignment complex /contact <- contact2	**Let** contact:Contact = contact2 **In**
Assignment partial complex /contact.address <- ‘UK’	**Let** contact:Contact = setAddress(contact,t(U)∼K) **In**
loan.amount >= 10000	[getAmount(loan) ge number(+,t(1)∼0∼0∼0∼0, <>)] ->
Else	[not(*conjunction of all guards*)] ->

Table A.1: CRESS Assignment and Guards in LOTOS

CRESS Syntax	LOTOS Translation
Compensate *scope*	*SERVICE*_EVENT [...] (...,scope,Compensation,...) calls event dispatcher process with specified scope and Compensation as event parameter
Fork [strict | loose] (e.g. nodes 3 and 4)	for strict (PROCESS_3[...] ≫ **Accept** xstates:States **Exit**(xstates, Any States)) ||| (PROCESS_4[...] ≫ Accept xstates:States **Exit**(Any States, xstates)) for loose, join conditions are ignored (see join)
Invoke partner.port.op input output fault	partner !port !op !input; (partner !port !op ?output:Type; ... [] partner !port !op !fault; ...)
Join *condition* (e.g. 3&&4)	if fork was strict then ≫ **Accept** xstates0,xstates1:States ... get state information from returned xstates ... xstatus represents the success value of the node [not(xstatus0 and xstatus1)]->... ... join failure flow [xstatus0 and xstatus1]->... ... join success flow if loose, then the conditions are ignored
Receive partner.port.op input	partner !port !op ?input:Type;
Reply partner.port.op output **Reply** partner.port.op faultname.fault	partner !port !op !output; partner !port !op !faultname !fault;
Terminate	with -r repeat option, then process repeats otherwise behaviour Exit
While *condition*	[*condition*]-> ... *PROCESS_Node_Number* (* repeat while *) [] [not(*condition*)]-> ...

Table A.2: CRESS Activities in LOTOS

CRESS Syntax	LOTOS Translation
Catch fault.message	For Catch in a diagram: service !port !op !fault ?message:Type; where the service, port and op are specified in the associated node For Catch in an event handler, [match(xevent,xkind,*fault*,*messageKind*)]-> ... behaviour where xevent should match the fault name, and xkind (type) should match the *message* type
CatchAll	For an event handler: [match(xevent,CatchAll)]-> ...behaviour
Compensation	For an event handler: **Let** xstates:States = state(*service*,True,*node*,*variables*...) $\sim$ xstates **In** saves the compensation scope (*node*), along with the current state of variables inserted into an existing list of states (xstates) for the service behaviour

Table A.3: CRESS Event and Compensation in LOTOS

Appendix B

CRESS to WS-BPEL 2.0 Translation

CRESS Syntax	**WS-BPEL 2.0 Translation**
Assignment simple /rate <- 3.5	<**assign**> <**copy**> <**from** expression=″3.5″/> <**to** variable=″rate″ part=″float″/> ...
Assignment complex /contact <- contact2	<**assign**> <**copy**> <**from** variable=″contact2″/> <**to** variable=″contact″/> ...
Assignment parts /contact.address <- ‘UK’	<**assign**> <**copy**> <**from** expression=″ ′UK′ ″/> <**to** variable=″contact″ part=″contact″ query=″/contact/address″/> ...
Assignments	<**assign**> each assignment has a <**copy**> construct </**assign**>
proposal.amount >= 10000	<**if**> <**condition**> $proposal.proposal/amount >= 10000 </**condition**> ...
Else	<**else**> ... </**else**>

Table B.1: CRESS Assignment and Guards in WS-BPEL 2.0

CRESS Syntax	WS-BPEL 2.0 Translation
Compensate *scope*	<**compensate**>...</**compensate**> *OR* <**compensateScope** target=″scope″/>
Fork [strict|loose]	<**empty** name=″*SERVICE.NODE*″> <**target** linkName=″...″/> <**source** linkName=″...″/> <**source** ... other parallel activities </**empty**> strict (default) and loose apply in the Join
Invoke partner.port.op input output fault	<**invoke** name=″*SERVICE.NODE*″ partnerLink=″*partner*Port″ portType=″prefix:portPort″ operation=″op″ inputVariable=″input″ outputVariable=″output″> ... </**invoke**>
Join *condition*	<**empty** name=″*SERVICE.NODE*″ joinCondition=″boolean condition of the link status ″> ... </**empty**> if join is loose, then <empty> has suppressJoinFailure="yes"
Receive partner.port.op input	<**receive** name=″*SERVICE.NODE*″ partnerLink=″servicePort″ portType=″prefix:portPort″ operation=″op″ variable=″input″ createInstance=″yes″> ... </**receive**> if there are multiple Receives, then as **<onMessage>** nested within **<pick>** <**onMessage** partnerLink=″servicePort″ portType=″prefix:portPort″ operation=″op″ variable=″input″> ... </**onMessage**>
Reply partner.port.op output	<**reply** name=″*SERVICE.NODE*″ partnerLink=″servicePort″ portType=″prefix:portPort″ operation=″op″ variable=″output″> ... </**reply**>
Reply partner.port.op faultname.fault	<**reply** name=″*SERVICE.NODE*″ partnerLink=″servicePort″ portType=″prefix:portPort″ operation=″op″ variable=″fault″ faultName=″prefix:faultname″ > </**reply**>
Terminate	<**terminate** name=″*SERVICE.NODE*″ >...</**terminate**>
While condition	<**while** ...> <**condition**>condition</**condition**> ... <**flow**> ...

Table B.2: CRESS Activities in WS-BPEL 2.0

CRESS Syntax	WS-BPEL 2.0 Translation
Catch fault.variable	<**catch** faultName=″prefix:fault″ faultVariable=″variable″> ... </**catch**>
CatchAll	<**catchAll**> ... </**catchAll**>
Compensation	<**compensationHandler**> ... </**compensationHandler**>

Table B.3: CRESS Event and Compensation in WS-BPEL 2.0

Bibliography

[1] J.-R. Abrial. Formal methods in industry: achievements, problems, future. *Software Engineering, International Conference on*, 0:761–768, 2006.

[2] Active Endpoints. ActiveVOS BPMS from Active Endpoints. *http://www.activevos.com/*, 2009.

[3] Active Endpoints. BPEL open source engine. *http://www.activevos.com/community-open-source.php*, 2009.

[4] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, editors. *Business Process Execution Language for Web Services*. Version 1.1. BEA, IBM, Microsoft, SAP, Siebel, May 2003.

[5] Apache. Axis web page. *http://ws.apache.org/axis/*, 2009.

[6] A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Lie, S. Thatte, P. Yendluri, and A. Yiu, editors. *Web Services Business Process Execution Language*. Version 2.0. Organization for The Advancement of Structured Information Standards, Billerica, Massachusetts, USA, Apr. 2007.

[7] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Software*, 2007.

[8] L. Baresi, D. Bianculli, S. Guinea, and P. Spoletini. Keep it small, keep it real: Efficient run-time verification of web service compositions. In *FMOODS '09/FORTE '09: Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems*, pages 26–40, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *ESEC/SIGSOFT FSE*, pages 141–150, 2009.

[10] J. Billington and et. al. The petri net markup language: Concepts, technology, and tools. In W. van der Aalst and E. Best, editors, *Applications and Theory of Petri Nets 2003*, number 2679 in Lecture Notes in Computer Science, pages 483–505. Springer, 2003.

[11] B. W. Boehm. Verifying and validating software requirements and design specification. *IEEE Transactions on Software Engineering*, 1(1):75–88, Jan. 1984.

[12] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods: Dispelling industrial prejudices. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 105–117, Berlin, Germany, 1994. Springer.

[13] bpelpeople.com. BPEL Comparison - comparison between bpel-g and other BPEL engines. *http://code.google.com/p/bpel-g/wiki/BPELComparison*, Oct. 2009.

[14] BPMI. *Business Process Modeling Notation*. Version 1.0. Business Process Management Initiative, May 2004.

[15] M. Calder and C. E. Shankland. A symbolic semantics and bisimulation for full LOTOS. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XIV)*, pages 184–200. Kluwer Academic Publishers, London, UK, Sept. 2001.

[16] S.-C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. In *ACM Transactions on Software Engineering and Methodology*. 1999.

[17] A. Chirichiello and G. Salaün. Encoding abstract descriptions into executable web services: Towards A formal development. In *Proc. Web Intelligence 2005*. Institution of Electrical and Electronic Engineers Press, New York, USA, Dec. 2005.

[18] E. M. Clark, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[19] J. Coutaz, L. Balme, X. Alvaro, G. Valvary, A. Demeure, and J.-S. Sottet. An MDE-SOA approach to support plastic user interfaces in ambient spaces. In C. Stephanidis, editor, *Proc. 12th Universal Access in Human-Computer Interaction*, number 4555 in Lecture Notes in Computer Science, pages 63–72. Springer, Berlin, Germany, 2007.

[20] P. Cronin. A regular expression grammar language dictionary generator. *http://regldg.com/*, Feb. 2009.

[21] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, Germany, 1985.

[22] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. Grid service orchestration using the business process execution language (BPEL). *Grid Computing*, 3(3-4):283–304, Sept. 2005.

[23] Eviware. The web service, SOA and SOAP testing tool. *http://www.soapui.org/*, Dec. 2009.

[24] A. Ferrara. Web services: A process algebra approach (technical report). available from author's web page.

[25] A. Ferrara. Web services: A process algebra approach. In *Proc. 2nd. International Conference on Service-Oriented Computing*, pages 242–251. ACM Press, New York, USA, Nov. 2004.

[26] C. Fidge. A comparative introduction to CSP, CCS and LOTOS. Technical Report 93-24, Department of Computer Science, University of Queensland, Brisbane, Australia, Apr. 1994.

[27] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. Rosenblum, and S. Uchitel. Model checking service compositions under resource constraints. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 225–234, New York, NY, USA, 2007. ACM.

[28] H. Foster, S. Uchitel, J. Kramer, and J. Magee. Compatibility verification for web service choreography. In M. Aiello, editor, *Proc. 2nd. International Conference on Service-Oriented Computing*, New York, USA, Nov. 2004. ACM Press.

[29] I. Foster and C. Kesselman. *The Grid. Blueprint for a New Computing Infrastructure.: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[30] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Technical report, Global Grid Forum, Lemont, Illinois, USA, June 2002. http://www.globus.org/research/papers/ogsa.pdf.

[31] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proc. 13th. International World Wide Web Conference*, pages 621–630. ACM Press, New York, USA, May 2004.

[32] H. Garavel. Compilation of LOTOS abstract data types. In S. T. Vuong, editor, *Proc. Formal Description Techniques II*. North-Holland, Amsterdam, Netherlands, Dec. 1989.

[33] H. Garavel. An overview of the Eucalyptus toolbox. In Z. Brezocnik and T. Kapus, editors, *Applied Formal Methods in System Design*, pages 76–88, Maribor, Slovenia, June 1996. Action COST 247.

[34] H. Garavel and H. Hermanns. On combining functional verification and performance evaluation using CADP. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 410–429, London, UK, 2002. Springer-Verlag.

[35] H. Garavel and F. Lang. SVL: a scripting language for compositional verification. In *Proc. of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems, FORTE'2001*. August 2001.

[36] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. In *European Association for Software Science and Technology (EASST) Newsletter*, volume 4, pages 13–24. August 2002.

[37] H. Garavel and W. Serwe. State space reduction for process algebra specifications. In C. Rattray, S. Maharaj, and C. E. Shankland, editors, *Proc. 10th Int. Conf. on Algebraic Methodology and Software Technology*, number 3116 in Lecture Notes in Computer Science, pages 164–180. Springer, Berlin, Germany, June 2004.

[38] H. Garavel and J. Sifakis. Compilation and verification of LOTOS specifications. In L. M. S. Logrippo, R. L. Probert, and H. Ural, editors, *Proc. Protocol Specification, Testing and Verification X*. North-Holland, Amsterdam, Netherlands, June 1990.

[39] S. Graham, A. Marmakar, J. Mischinsky, I. Robinson, and I. Sedukhin, editors. *Web Services Resource*. Version 1.2. Organization for The Advancement of Structured Information Standards, Billerica, Massachusetts, USA, Apr. 2006.

[40] C. Grimm, R. Groeper, S. Makedanz, H. Pfeiffenberger, P. Gietz, M. Haase, M. Schiffers, and W. Ziegler. Trust issues in Shibboleth-enabled federated grid authentication and authorization infrastructures supporting multiple grid middleware. volume 0, pages 569–576, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[41] O. M. Group. *OMG Unified Modeling Language (OMG UML), Superstructure*. Open Management Group, Feb 2009.

[42] W. Guo and C. Lin. A formal approach to verify grid service composition based on interaction pattern. In *APSCC '08: Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference*, pages 69–74, Washington, DC, USA, 2008. IEEE Computer Society.

[43] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

[44] R. Hamadi and B. Benatallah. A petri net-based model for web service composition. In *Proc. 13th. International World Wide Web Conference*, pages 621–630. ACM Press, New York, USA, May 2004.

[45] S. Hinz, K. Schmidt, and C. Stahl. Transforming bpel to petri nets. In F. C. W. M. P. van der Aalst, B. Benatallah and F. Curbera, editors, *Proceedings of the 3rd Int'l Conference on Business Process Management (BPM 2005)*, pages 220–235. Springer-Verlag, 2005.

[46] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[47] G. J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[48] P. Inverardi and S. Scriboni. Connectors synthesis for deadlock-free component-based architectures. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, Coronado Island, San Diego, CA, USA, Nov. 2001. IEEE Computer Society.

[49] P. Inverardi and M. Tivoli. Deadlock-free software architectures for com/dcom applications. *Journal of Systems and Software*, 65(3):173–183, 2003.

[50] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807. International Organization for Standardization, Geneva, Switzerland, 1989.

[51] ITU. *Message Sequence Chart (MSC)*. ITU-T Z.120. International Telecommunications Union, Geneva, Switzerland, 2000.

[52] ITU. *SDL Combined with UML*. ITU-T Z.109. International Telecommunications Union, Geneva, Switzerland, 2000.

[53] ITU. *Specification and Description Language*. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, Aug. 2002.

[54] JUnit.org. JUnit home page. *http://www.junit.org/*, 2009.

[55] N. Kaveh and W. Emmerich. Deadlock detection in distributed object systems. In V. Gruhn, editor, *Joint Proc. of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 44–51. ACM Press, Vienna, Austria, 2001.

[56] N. Kaveh and W. Emmerich. Validating distributed object and component designs. In M. Bernardo and P. Inverardi, editors, *Formal Methods for Software Architecture*, number 2804 in Lecture Notes in Computer Science, pages 63–91. Springer, Berlin, Germany, 2003.

[57] S. Kent. The Unified Modeling Language. In H. Bowman and J. Derrick, editors, *Formal Methods for Distributed Processing*, pages 126–152. Cambridge University Press, UK, Dec. 2001.

[58] B. W. Kernighan and D. M. Ritchie. The *m4* macro processor. Technical report, Bell Laboratories, Murray Hill, New Jersey, USA, 1977.

[59] P. S. Lambert, K. L. L. Tan, K. J. Turner, V. Gayle, K. Prandy, and R. O. Sinnott. Development of a grid enabled occupational data environment. In R. Procter, editor, *Proc. 2nd. International Conference on e-Social Science*, pages 1–12, Manchester, UK, June 2006. National Centre for e-Social Science.

[60] L. Léonard and G. Leduc. A formal definition of time in LOTOS. *Formal Aspects of Computing*, 10:248–266, 1998.

[61] F. Leymann. *Web Services Flow Language (WSFL)*. 1.0. IBM Software Group, May 2001.

[62] P. A. Lindsay, R. C. Moore, and B. Ritchie. Review of existing theorem provers. Technical Report UMCS-87-8-2, Department of Computer Science, University of Manchester, Manchester, UK, Dec. 1986.

[63] R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. In *Journal of Logic and Algebraic Programming*, page 2006. Elsevier press, 2005.

[64] B. Margolis and J. L. Sharpe. *SOA for The Business Developer*. MC Press, Woodland, Texas, USA, 2007.

[65] R. Mateescu. Property pattern mappings for RAFMC. *http://www.inrialpes.fr/vasy/cadp/resources/evaluator/rafmc.html*, May 2009.

[66] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. In *Proc. of the 5th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2000*, Berlin, Germany, Apr. 2000.

[67] G. S. A. Matthew B. Dwyer and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of the 21st International Conference on Software Engineering*. May 1999.

[68] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, London, UK, 1993.

[69] A. J. R. G. Milner. A calculus of communicating systems. Number 92 in Lecture Notes in Computer Science. Springer, Berlin, Germany, 1980.

[70] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes – Parts I and II. *Information and Computation*, 100:1–77, 1992.

[71] B. Mitchell and J. Hillston. Analysing web service composition with pepa. In J. Bradley, editor, *Proceedings of the Third Workshop on Process Algebras and Stochastically Timed Activities*, pages 33–44, Edinburgh, Scotland, June 2004.

[72] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker, editors. *Web Services Security*. Version 1.1. Organization for The Advancement of Structured Information Standards, Billerica, Massachusetts, USA, Feb. 2006.

[73] S. Nakajima. Model-checking behavioral specification of BPEL applications. In *Proc. WLFM'05*, 2005.

[74] OASIS. *Universal Description, Discovery and Integration*. Version 2.0. Organization for The Advancement of Structured Information Standards, Billerica, Massachusetts, USA, July 2002.

[75] OASIS. WS BPEL issues list. *http://www.oasis-open.org*, 2004.

[76] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[77] Oracle. Oracle BPEL process manager. *http://www.oracle.com/technology/products/ias/bpel/index.html*, 2008.

[78] C. Ouyang, W. van der Aalst, S. Breutel, M. Dumas, A. ter Hofstede, and H. Verbeek. Formal semantics and analysis of control flow in WS-BPEL. In *Report BPM-05-15*, BPM Center, 2005.

[79] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. ter Hofstede. WofBPEL: A tool for automated analysis of BPEL processes. In P. T. Boualem Benatallah, Fabio Casati, editor, *Service-Oriented Computing - ICSOC 2005: Third International Conference*, volume 3826 of *Lecture Notes in Computer Science*, pages 484–489. Springer-Verlag, Nov. 2005.

[80] C. Pautasso. JOpera: An agile environment for web service composition with visual unit testing and refactoring. In *Proc. IEEE Symposium on Visual Languages and Human Centric Computing*. Institution of Electrical and Electronic Engineers Press, New York, USA, Nov. 2005.

[81] S. Pavón Gomez, D. Larrabeiti, and G. Rabay Filho. LOLA user manual (version 3R6). Technical report, Department of Telematic Systems Engineering, Polytechnic University of Madrid, Spain, Feb. 1995.

[82] Y. F. PengCheng Xiong and M. Zhou. A petri net approach to analysis and composition of web services.

[83] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, FRG, 1962.

[84] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *Proc. International Conference on Web Services*, pages 43–51. Institution of Electrical and Electronic Engineers Press, New York, USA, June 2004.

[85] G. Salaün, A. Ferrara, and A. Chirichiello. Negotiation among web services using LOTOS/CADP. In *Proc. European Conference on Web Services 2004*, volume 3250 of *Lecture Notes in Computer Science*, pages 198–212. Springer, Berlin, Germany, 2004.

[86] A. Slomiski. On using BPEL extensibility to implement OGSI and WSRF grid workflows. In *Proc. Global Grid Forum 10*, Berlin, Germany, Mar. 2005. Humboldt University.

[87] B. Sotomayor and L. Childers. *Globus Toolkit 4: Programming Java Services*. Morgan Kaufmann, San Francisco, USA, Mar. 2006.

[88] C. Stahl. A petri net semantics for bpel. Technical report, Humboldt-Universitat zu Berlin.

[89] K. L. L. Tan. Case studies using CRESS to develop web and grid services. Technical Report CSM-183, Department of Computing Science and Mathematics, University of Stirling, UK, Dec. 2009.

[90] K. L. L. Tan and K. J. Turner. MINT interpreter web page. *http://www.cs.stir.ac.uk/~kjt/software/lotos/mint.html*.

[91] K. L. L. Tan and K. J. Turner. Orchestrating grid services using BPEL and Globus Toolkit 4. In M. Merabti, R. Pereira, C. Oliver, and O. Abuelma'atti, editors, *Proc. 7th PGNet Symposium*, pages 31–36. School of Computing, Liverpool John Moores University, Liverpool, UK, June 2006.

[92] K. L. L. Tan and K. J. Turner. Automated analysis and implementation of composed grid services. In D. Dranidis and I. Sakellariou, editors, *Proc. 3rd South-East European Workshop on Formal Methods*, pages 51–64. South-East European Reseach Centre, Thessaloniki, Greece, Nov. 2007.

[93] S. Thatte. *XLANG - Web Services For Business Process Design*. Microsoft Corporation, 2001.

[94] The Apache Software Foundation. Apache ODE (Orchestration Director Engine). *http://ode.apache.org/*, May 2008.

[95] The OGSi Alliance. OSGi Service Platform Core Specification, relase 4, version 4.2. *http://www.osgi.org/Main/HomePage*, June 2009.

[96] P. Tripathy and B. Sarikaya. Test generation from LOTOS specifications. *IEEE Transactions on Computers*, 40(4):543–552, Apr. 1991.

[97] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. Open grid services infrastructre (ogsi) version 1.0. Gwd-r, Global Grid Forum, 2003.

[98] K. J. Turner. Exploiting the *m4* macro language. Technical Report CSM-126, Department of Computing Science and Mathematics, University of Stirling, UK, Sept. 1994.

[99] K. J. Turner. Formalising the CHISEL feature notation. In M. H. Calder and E. H. Magill, editors, *Proc. 6th Feature Interactions in Telecommunications and Software Systems*, pages 241–256. IOS Press, Amsterdam, Netherlands, May 2000.

[100] K. J. Turner. Formalising graphical behaviour descriptions. In C. Rattray, S. Maharaj, and C. E. Shankland, editors, *Proc. 10th Int. Conf. on Algebraic Methodology and Software Technology*, number 3116 in Lecture Notes in Computer Science, pages 537–552. Springer, Berlin, Germany, June 2004.

[101] K. J. Turner. Formalising web services. In F. Wang, editor, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XVIII)*, number 3731 in Lecture Notes in Computer Science, pages 473–488. Springer, Berlin, Germany, Oct. 2005.

[102] K. J. Turner. Representing and analysing composed web services using CRESS. *Network and Computer Applications*, 30(2):541–562, Apr. 2007.

[103] K. J. Turner. CHIVE (CRESS Home-grown Interactive Visual Editor). Technical report, Apr. 2009.

[104] K. J. Turner. CRESS reference manual version 4.3. Technical report, University of Stirling, Nov. 2009.

[105] K. J. Turner and K. L. L. Tan. CLOVE project web page. *http://www.cs.stir.ac.uk/~kjt/research/clove.html*.

[106] K. J. Turner and K. L. L. Tan. Graphical composition of grid services. In D. Buchs and N. Guelfi, editors, *Rapid Introduction of Software Engineering Techniques*, number 4401 in Lecture Notes in Computer Science, pages 1–17, Berlin, Germany, May 2007. Springer.

[107] K. J. Turner and K. L. L. Tan. A rigorous approach to orchestrating grid services. *Computer Networks*, 51(15):4421–4441, Oct. 2007.

[108] K. J. Turner and K. L. L. Tan. A rigorous methodology for composing services. In M. Alpuente, B. Cook, and C. Joubert, editors, *Proc. 14th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2009)*, number 5825 in Lecture Notes in Computer Science, Eindhoven, The Netherlands, Nov. 2009.

[109] W. M. P. van der Aalst. Pi calculus versus petri nets: Let us eat humble pie rather than further inflate the pi hype. In *BPTrends*, pages 1–11, May 2005.

[110] VASY. Case studies achieved using the CADP toolset. *http://www.inrialpes.fr/vasy/pub/cadp/case-studies/*, June 2009.

[111] C. A. Vissers. What makes industries believe in formal methods. In A. A. S. Danthine, G. Leduc, and P. Wolper, editors, *Proc. Protocol Specification, Testing and Verification XIII*, pages 3–26. North-Holland, Amsterdam, Netherlands, May 1993.

[112] B. Wassermann, W. Emmerich, B. Butchart, N. Cameron, L. Chen, and J. Patel. Sedna: A BPEL-based environment for visual scientific workflow modelling. In I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 428–449. Springer, 2007.

[113] M. Weiss and B. Esfandari. On feature interactions in web services. In *Proc. IEEE International Conference on Web Services*, pages 88–95, San Diego, California, July 2004.

[114] P. Y. H. Wong and J. Gibbons. A process-algebraic approach to workflow specification and refinement. In *SC'07: Proceedings of the 6th international conference on Software composition*, pages 51–65, Berlin, Heidelberg, 2007. Springer-Verlag.

[115] World Wide Web Consortium. *Document Type Definition (DTD)*. Version 1.2. World Wide Web Consortium, Geneva, Switzerland, 1998.

[116] World Wide Web Consortium. *XML Path Language (XPATH)*. Version 1.0. World Wide Web Consortium, Geneva, Switzerland, Nov. 1999.

[117] World Wide Web Consortium. *XSL Transformations*. Version 1.0. World Wide Web Consortium, Geneva, Switzerland, Nov. 1999.

[118] World Wide Web Consortium. *Web Services Description Language (WSDL)*. Version 1.1. World Wide Web Consortium, Geneva, Switzerland, Mar. 2001.

[119] World Wide Web Consortium. *Simple Object Access Protocol (SOAP)*. Version 1.2. World Wide Web Consortium, Geneva, Switzerland, June 2003.

[120] World Wide Web Consortium. *Web Ontology Language (OWL) – Semantics and Abstract Syntax*. Version 1.0. World Wide Web Consortium, Geneva, Switzerland, Feb. 2004.

[121] World Wide Web Consortium. *Web Services Addressing (WS-Addressing)*. World Wide Web Consortium, Geneva, Switzerland, Aug. 2004.

[122] World Wide Web Consortium. *XML Schema Definition (XSD)*. Version 1.0. World Wide Web Consortium, Geneva, Switzerland, Oct. 2004.

[123] World Wide Web Consortium. *Web Services Choreography Description Language*. Candidate Version 1.0. World Wide Web Consortium, Geneva, Switzerland, Nov. 2005.

[124] World Wide Web Consortium. *Web Services Description Language (WSDL)*. Version 2.0 (Draft). World Wide Web Consortium, Geneva, Switzerland, Aug. 2005.

[125] H. yun Long and J. shi Li. A process algebra approach of bpel4ws. *Journal of Information and Computing Science*, 4(2):93–98, Dec. 2008.

[126] M. Zager. SOA/Web Services - Business Process Orchestration with BPEL. *http://webservices.sys-con.com/read/155631_1.htm*, Dec. 2005.

[127] J. Zhou and G. Zeng. Describing and reasoning on the composition of grid services using pi-calculus. In S. An and D. Wei, editors, *Proc. 6th. International Conference on Computer and Information Technology*. Institution of Electrical and Electronic Engineers Press, New York, USA, Sept. 2006.