ABSTRACTIONS AND TRANSFORMATIONS FOR AUTOMATED DATA NETWORK CONFIGURATION

by

SIMON KNIGHT

Bachelor of Engineering (Telecommunications), 2008
Bachelor of Economics, 2009

A thesis submitted for the degree of
Doctor of Philosophy

School of Information Technology & Mathematical Sciences
IT, Engineering and the Environment

University of South Australia

January 2017

ABSTRACT

Modern computer networks have grown to become fundamental infrastructure, but the engineering process used to design and manage them lacks strong high-level abstractions and systematic design tooling. They are typically managed by network engineers entering device configurations in a low-level vendor-specific syntax. The information in this low-level configuration must be consistent across the entire network, with the engineer responsible for converting of high-level network designs into device configurations. This makes configuration complex, time consuming and error-prone. Recent techniques such as Software-Defined Networking or network programmability relieve some of this burden, but are still in development, and often require upgrades of existing infrastructure and retraining of engineering staff, and can vary between vendors.

In this thesis we present an approach to automate the process of generating low-level device configurations for existing network deployments, from a high-level specification. We present a specification abstraction to capture high-level policy in a format which is compatible with current industry practice, and an approach to transform this to an intermediate network-wide configuration state representation. A second transformation step converts the intermediate representation into the low-level device configuration state appropriate for the target device, which is then assembled using simple templates. We show how this multi-stage compiler approach allows the expression of new high-level policies on different network topologies, variation of network designs and routing protocols, and generation of configurations for different target devices.

We have incorporated this approach into an open-source tool, AutoNetkit, which has been tested on a range of industry-derived network topologies. The test cases show that the approach is extensible to a wide range of protocols and devices, and scalable up to the a size comparable to the core devices of the European academic network. Valid configurations for over a thousand devices can be generated in seconds. AutoNetkit has also been used in peer-reviewed demonstrations and as a component in tools used by network engineers in industry.

DECLARATION

This thesis presents work carried out by myself and does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university; to the best of my knowledge it does not contain any materials previously published or written by another person except where due reference is made in the text; and all substantive contributions by others to the work presented, including jointly authored publications, are clearly acknowledged.

Adelaide, January 2017

[illegible]

Simon Knight

ACKNOWLEDGMENTS

This thesis has been a long road, with assistance from many people along the way. Firstly I'd like to thank my family and close friends— Mum, Dad, Andrew, and Marisa— for their help and support through the ups and downs, of which there have been many.

A huge thanks to Associate Professor David Kearney, for being the near-mythical supervisor who was able to help me transform a big pile of ideas and a prototype into a research thesis. Without his help this thesis would still be just a big pile of ideas. Thank you so much for helping me reach this stage. Dr Grant Wigley was instrumental in co-ordinating my candidature at the University of South Australia, helping me navigate the PhD procedure, and for very valuable feedback on thesis drafts.

I was very fortunate to have a patient and understanding manager at Cisco in Joel Obstfeld. Thank you for your flexibility in switching between part-time and full-time as I navigated the waters of transferring Universities and resuming my candidature, your assistance as a referee, your support as a manager, and your support of the value of the work in making it accessible to customers.

At the University of Adelaide (UofA), I would like to thank my co supervisors Dr Matthew Sorell and Dr Hung Nguyen for their support and assistance in the early days of my candidature. Their academic and administrative support in navigating difficult waters, especially at often urgent times, was a huge help. Without them this work would have stalled long ago. I would also like to thank Ann Madigan, an Education Welfare Officer, for the many hours she spent helping to navigate UofA processes and supporting my candidature transfer. I acknowledge the other two members of my supervisory panel at the UofA: Professor Matthew Roughan and Dr Nick Falkner.

Thanks to all of those who I worked with on the VIRL project at Cisco, including Chris Hartley, Ed Kern, Ian Wells, Tom Bryan, Miroslav Los, Ralph Schmieder, Brian Daugherty, Dominik Kapisinsky, and Scott Anderson for suggestions with building upon the core work for customer use; and to other academics I co-authored or exchanged ideas with, including Iain Phillips, Olaf Maennel, Nicklas Semmler and Xenofontas Dimitropoulos. A special thanks to Massimo Rimondini and the Netkit team at Roma Tre for making Netkit available, and granting permission to use their lab as a working example in this thesis.

This thesis was prepared using the ClassicThesis[1] layout. The multi-layer ANM views are based on the simple graph-model in 3d[2] example from TEXample.net.

1 https://www.ctan.org/pkg/classicthesis
2 http://www.texample.net/tikz/examples/3d-graph-model/

GLOSSARY

ABSTRACT NETWORK MODEL The set of all Network Views, representing the intermediate representation of the network to be configured.

DEVICE COMPILER A function to transform the Abstract Network Model and Intermediate Hardware Model into a device entry in the Intermediate Device Model

DESIGN FUNCTION A function to build a Network View from the Network Whiteboard and/or other Network Views

INTERMEDIATE DEVICE MODEL The model which stores the configuration information for each network device

INTERMEDIATE HARDWARE MODEL The model which stores the hardware information for a network device.

INTERMEDIATE PLATFORM MODEL The model which stores information about the platform configuration, such as for a simulation environment

NETWORK ELEMENT CONNECTION A physical or logical connection between Network Element Interfaces

NETWORK ELEMENT INTERFACE A physical or logical interface on a Network Element

NETWORK ELEMENT An element in the Network View, representing a device to be configured

NETWORK VIEW A *slice* of the Abstract Network Model representing a topology such as for a routing protocol or for IP Addressing

NETWORK WHITEBOARD The High-Level annotated topology describing the network to be configured

PLATFORM COMPILER A function to transform the Network Whiteboard into the Intermediate Hardware Model

PSEUDO NETWORK ELEMENT A Network Element that doesn't correspond to a device to be configured. Typically used to represent intermediate concepts used for subsequent Network Views

CONTENTS

List of Figures viii
List of Tables xiii
List of Algorithms xiv
List of Code Listings xvi

1 INTRODUCTION 2
1.1 Introduction 2
1.2 Motivation 2
1.3 Background 4
1.4 Structure of Thesis 8

i BACKGROUND AND METHODOLOGY 10

2 RELATED WORK 11
2.1 Introduction 11
2.2 Chapter Overview 11
2.3 Scope of Network Configuration Management 12
2.4 General comments about High-Level Network Representation 16
2.5 Existing tools for High-Level Configuration Network Policy 19
2.6 Intermediate Network Representation 31
2.7 Low-level Device Configuration State 39
2.8 Summary 43
2.9 Network Testbeds 44
2.10 Conclusion 45

3 RESEARCH METHODOLOGY AND OVERVIEW 46
3.1 Introduction 46
3.2 Research Questions 47
3.3 Research Methodology 48
3.4 Scope of Thesis 49
3.5 Discussion of Configuration Generation Process 50
3.6 Complete Methodology 54
3.7 Conclusion 55

ii FRAMEWORK AND TOOLCHAIN 56

4 SPECIFICATION AND REPRESENTATION USING NETWORK VIEWS 57
4.1 Introduction 57
4.2 Chapter Overview 58
4.3 Overview of Approach and Toolchain 59

4.4 Network Whiteboard and Network Views Abstractions . . 70
4.5 Simplified Small Internet Example 82
4.6 Design Details 96
4.7 Discussion 98
4.8 Conclusion 100
5 NETWORK VIEW TRANSFORMATION 102
5.1 Introduction 102
5.2 Key Concepts 103
5.3 Low-Level Primitives 107
5.4 High-Level Primitives 111
5.5 Design Functions 122
5.6 Simplified Small Internet Design Functions 124
5.7 Extended Design Function Examples 140
5.8 Transfer Function Approach 152
5.9 Conclusion 153
6 GENERATION OF LOW-LEVEL DEVICE CONFIGURATION STATE 154
6.1 Introduction 154
6.2 Motivation: Device Syntax Variations 158
6.3 Artefacts and Processes 166
6.4 Compilation for Simplified Small Internet Example 171
6.5 Configuration Assembly Using Templates 186
6.6 Transfer Function Approach 189
6.7 Conclusion 190

iii IMPLEMENTATION AND CASE STUDIES 191
7 SYSTEM IMPLEMENTATION AND EXPERIMENTATION SETUP 192
7.1 Introduction 192
7.2 System Implementation 193
7.3 Experimentation Setup 199
7.4 Simplified Example: House Topology 201
7.5 Prior Publications 225
7.6 Conclusion 226
8 CASE STUDIES 227
8.1 Introduction 227
8.2 Chapter Overview 227
8.3 Simplified Netkit Small-Internet Lab 229
8.4 Complete Netkit Small-Internet Lab 241
8.5 VLAN Topology 270
8.6 Large-Scale Example 290
8.7 System Adoption 306
8.8 Conclusion 309

iv FUTURE WORK AND CONCLUSIONS 310
9 FUTURE WORK 311
9.1 Introduction 311
9.2 Specification and Representation Using Network Views . . 311
9.3 Network View Transformation 314
9.4 Generation of Low-Level Configuration State 326
9.5 Expansion and new Applications of System 329
9.6 Conclusion 331
10 CONCLUSIONS 332
10.1 Research Questions 333
10.2 Conclusion 336
v APPENDICES 338
A RFC 3139 REQUIREMENTS 339
B NETKIT SMALL INTERNET LAB 341
C NOTATION FOR SMALL INTERNET EXAMPLE 345
D DESIGN DETAILS AND DEFINITIONS OF PRIMITIVES 350
E DEFINITIONS OF PRIMITIVES 382
F CODE FOR CHAPTER 6 395
G CODE FOR CHAPTER 7 410
H CODE FOR CHAPTER 8 CASE STUDIES 424
I CODE FOR SPECULATIVE FUTURE WORK 487
BIBLIOGRAPHY 489

LIST OF FIGURES

Figure 2.1 Reproduction of Figure 2 from Sanchez *et al.* 14
Figure 2.2 Reproduction of "Intent-based configuration flow" from [38] 14
Figure 2.3 Reproduction of figure 2 from Sanchez *et al.* 31
Figure 2.4 Reproduction of Figure 1.12 *Template based generator* from [4] 42
Figure 3.1 Flowchart of toolchain for configuration generation . . 50
Figure 3.2 Reproduction of Figure 2 from Sanchez *et al.* 51
Figure 3.3 Toolchain flowchart: Network Whiteboard and ANM . . 51
Figure 3.4 Toolchain flowchart: Design Functions 52
Figure 3.5 Toolchain flowchart: Compiler 53
Figure 3.6 Experimentation verification process 53
Figure 3.7 General Implementation and Verification Methodology . 55
Figure 4.1 Simplified toolchain flowchart 59
Figure 4.2 House example topology. 59
Figure 4.3 Input description with labelled devices. 61
Figure 4.4 Physical Network Topology. 61
Figure 4.5 IP Address Network Topology 62
Figure 4.6 OSPF Network Topology. 63
Figure 4.7 eBGP Network Topology. 63
Figure 4.8 iBGP Network Topology. 64
Figure 4.9 Layer 2 Network Topology. 65
Figure 4.10 Network Model consisting of each Network Topology . 65
Figure 4.11 Creating Network Topology from Input Topology . . . 66
Figure 4.12 Compiling Device Model from Network Model 66
Figure 4.13 Configuration Assembly: Device Model and Templates . 68
Figure 4.14 Flowchart of toolchain 69
Figure 4.15 Toolchain: Network Whiteboard and Abstract Network Model 70
Figure 4.16 Network Whiteboard and Network View components . 71
Figure 4.17 Example Network Whiteboard 72
Figure 4.18 Network View example 72
Figure 4.19 Abstract Network Model with multiple Network Views 73
Figure 4.20 Abstract Network Model 74
Figure 4.21 Symbols used to represent Network Element Roles . . . 75
Figure 4.22 Pseudo Network Elements 75
Figure 4.23 Pseudo Network Element Example 76
Figure 4.24 Network Element Connections 77
Figure 4.25 Network Element Interfaces inside a Network Element . 77

Figure 4.26 Network Element Interface Bindings 78
Figure 4.27 Overview of Interface Pointers. 78
Figure 4.28 Logical Network Element Interfaces 79
Figure 4.29 Logical Network Element Interface associations 80
Figure 4.30 NE and NEI Identifiers 81
Figure 4.31 Netkit Lab Small Internet Topology 82
Figure 4.32 Example: Network Whiteboard 85
Figure 4.33 Design Function Flowchart 85
Figure 4.34 Simplified Small Internet Example: Physical Network View 86
Figure 4.35 Example: Layer 2 Network View 87
Figure 4.36 Example: IP Address Network View 88
Figure 4.37 IP Allocation Example, showing Broadcast Domain Pseudo Network Element, and Network Element Interfaces . . 89
Figure 4.38 IP Address Blocks for each ASN for Loopback and Infrastructure Blocks. 89
Figure 4.39 Example: Layer 2 Connectivity Network View 90
Figure 4.40 Example: OSPF Network View 91
Figure 4.41 Example: iBGP Network View 92
Figure 4.42 Example: eBGP Network View 93
Figure 4.43 Cross-Section Abstract Network Model example 95
Figure 4.44 NEI Bindings 97
Figure 5.1 Toolchain highlighting Design Functions 102
Figure 5.2 Flowchart of Design Function Process 103
Figure 5.3 Flowchart showing Design Functions composed of High-Level Primitives and Low-Level Primitives 104
Figure 5.4 Example output for Group of Network Elements in Small Internet example by *asn* attribute 113
Figure 5.5 Example topology for Boundary Network Elements . 114
Figure 5.6 Example topology for Connected Components 115
Figure 5.7 Example of combined transformation High-Level Primitives 116
Figure 5.8 Example of split High-Level Primitive 117
Figure 5.9 Example of merge High-Level Primitive 118
Figure 5.10 Example of explode High-Level Primitive 120
Figure 5.11 Design Function Internals 123
Figure 5.12 Abstract Network Model example 124
Figure 5.13 Physical Network View 125
Figure 5.14 Layer 2 Network View 127
Figure 5.15 IP Address Network View 128
Figure 5.16 Layer 2 Connectivity Network View 133
Figure 5.17 OSPF Network View 134
Figure 5.18 iBGP Network View 135
Figure 5.19 eBGP Network View 137

Figure 5.20 Extended Layer 2 Design Function for Switches 140
Figure 5.21 Design Function example using Switches 143
Figure 5.22 Flowchart of Network View construction for VLANs . . 144
Figure 5.23 Conceptual view of VLANs 145
Figure 5.24 Network Whiteboard for VLAN example 146
Figure 5.25 Step 1 of VLAN example 146
Figure 5.26 Step 2 of VLAN example 147
Figure 5.27 Connectivity for VLAN example 147
Figure 5.28 Workflow for Enhanced iBGP Design Function 148
Figure 5.29 Topology showing basic route reflector hierarchy 149
Figure 5.30 Route Reflector directionality label assignment 151
Figure 6.1 Toolchain highlighting Compilation 154
Figure 6.2 Cross-Section Abstract Network Model example 156
Figure 6.3 Flowchart of Low-Level Configuration Generation Process 157
Figure 6.4 Example OSPF Network View: interface names and costs 158
Figure 6.5 Example OSPF Network View: IP addresses 158
Figure 6.6 Intermediate Hardware Model 167
Figure 6.7 Platform Compilation process 168
Figure 6.8 Device Compilation process 169
Figure 6.9 Configuration Assembly process using templates 169
Figure 6.10 Platform Configuration generation and assembly 170
Figure 6.11 Platform and Device Compiler workflow 171
Figure 6.12 Workflow for Intermediate Hardware Model. 172
Figure 6.13 Mapping of Network Whiteboard interfaces 172
Figure 6.14 Adding management interfaces 173
Figure 6.15 Interface name allocation 173
Figure 6.16 Device Specification 174
Figure 6.17 Workflow for Intermediate Device Model construction . 175
Figure 6.18 Device Model for Network Element *4* 176
Figure 6.19 Intermediate Device Model for Network Element *4* . . . 177
Figure 6.20 Device Information in Device Model 178
Figure 6.21 Physical Network View and Interfaces Device Model . . 178
Figure 6.22 OSPF Network View and Device Model 180
Figure 6.23 iBGP Network View and Device Model 182
Figure 6.24 eBGP Network View and Device Model 184
Figure 7.1 System Implementation and Experiment Setup 193
Figure 7.2 Flowchart of visualisation process 195
Figure 7.3 Screenshot of visualisation system. 196
Figure 7.4 OSPF Network Convergence time-series visualisation . 197
Figure 7.5 House example Physical Topology. 200
Figure 7.6 Netkit emulation system overview for House topology . 200
Figure 7.7 House Methodology 202

Figure 7.8 Visualisation of Network Whiteboard with nodes added 204
Figure 7.9 Visualisation of Network Whiteboard with links added . 204
Figure 7.10 Visualisation of Physical Network View 207
Figure 7.11 Visualisation of Layer 2 Network View 208
Figure 7.12 Visualisation of IP Network View 210
Figure 7.13 Visualisation of Layer 2 Connectivity Network View . . 211
Figure 7.14 Visualisation of OSPF Network View 212
Figure 7.15 Visualisation of iBGP Network View 213
Figure 7.16 Visualisation of eBGP Network View 214
Figure 8.1 Simplified Small Internet Methodology 230
Figure 8.2 Visualisation of Network Whiteboard 231
Figure 8.3 Drawing of Network Whiteboard using yEd editor. . . . 232
Figure 8.4 Visualisation of Physical Network View 233
Figure 8.5 Visualisation of Layer 2 Network View 234
Figure 8.6 Visualisation of IP Network View: interface addresses . 234
Figure 8.7 Visualisation of IP Network View: loopback addresses . 235
Figure 8.8 Visualisation of Layer 2 Connectivity Network View . . 236
Figure 8.9 Visualisation of OSPF Network View 236
Figure 8.10 Visualisation of iBGP Network View 237
Figure 8.11 Visualisation of eBGP Network View 237
Figure 8.12 TRACEROUTE from *as300r1* to *as100r2*. 239
Figure 8.13 Complete Small Internet Methodology 243
Figure 8.14 Visualisation of Network Whiteboard 244
Figure 8.15 Drawing of Network Whiteboard using yEd editor. . . . 245
Figure 8.16 Visualisation of Network Whiteboard showing BGP Roles 247
Figure 8.17 Visualisation of Physical Network View 249
Figure 8.18 Visualisation of Layer 2 Network View 250
Figure 8.19 Visualisation of IP Addressing Network View 252
Figure 8.20 Visualisation of Layer 2 Connectivity Network View . . 253
Figure 8.21 Visualisation of RIP Network View 253
Figure 8.22 Visualisation of iBGP Network View 255
Figure 8.23 Visualisation of eBGP Network View 256
Figure 8.24 eBGP Network View with Default Originate label . . . 259
Figure 8.25 eBGP Network View: Prefix Lists 260
Figure 8.26 Visualisation of three traceroutes across simulated network 265
Figure 8.27 Visualisation of traceroute from *as100r1* to *eth0.as300r3* . 267
Figure 8.28 VLAN Case Study Methodology 272
Figure 8.29 Visualisation of VLAN Network Whiteboard 273
Figure 8.30 Physical Network View for VLAN topology. 274
Figure 8.31 Step 2 of VLAN Network View for VLAN topology. . . 275
Figure 8.32 Step 3 of VLAN Network View for VLAN topology. . . 276
Figure 8.33 Switches Network View of VLAN topology 278

Figure 8.34 Layer 2 Network View for VLAN topology. 278
Figure 8.35 IP Address Network View for VLAN topology. 279
Figure 8.36 Layer 2 Connectivity Network View for VLAN topology. 280
Figure 8.37 OSPF Network View for VLAN topology. 281
Figure 8.38 Interface names allocated by Platform Compiler 282
Figure 8.39 Visualisation of TRACEROUTE from r11 to r7. 287
Figure 8.40 Visualisation of full TRACEROUTE path from r11 to r7 . . 288
Figure 8.41 Geographic plot of NREN1400 network 290
Figure 8.42 Large Scale Case Study Methodology 293
Figure 8.43 Physical Network View for Large-Scale Example 294
Figure 8.44 OSPF Network View for Large-Scale Example 296
Figure 8.45 iBGP Network View for Large-Scale Example 297
Figure 8.46 eBGP Network View for Large-Scale Example 298
Figure 8.47 Example of paths highlighted on Physical Network View. 299
Figure 8.48 Subsection of Physical Network View showing connectivity from *DE.GEANT* 303
Figure 9.1 Network Element Path with Network Elements 312
Figure 9.2 Network Element Path with Network Element Interfaces 312
Figure 9.3 A Network Element Path using NE and NEIs 312
Figure 9.4 Network Element Capabilities 313
Figure 9.5 Higher level grammar to construct Network Whiteboard 314
Figure 9.6 IP Addressing tree for an example AS1. 315
Figure 9.7 IP Addressing tree for an example AS2. 315
Figure 9.8 Workflow for Redistribution Design Function 317
Figure 9.9 Topology showing multiple IGPs 318
Figure 9.10 Topology showing two-level Route Reflector hierarchy . 320
Figure 9.11 Workflow for BGP Policy Design Function 320
Figure 9.12 Routing Policy Workflow 321
Figure 9.13 Application of routing policy 322
Figure 9.14 Workflow for OSPF Traffic Engineering Design Function 322
Figure 9.15 Deriving OSPF Role from OSPF Area labels 325
Figure 9.16 Expanded workflow using suggested Design Functions . 326
Figure 9.17 An example of device inheritance sharing common logic. 327
Figure 9.18 Engineering workflow from analysis to deployment . . 330
Figure B.1 Slide from Di Battista *et al.* [25] showing BGP Policy . . 341
Figure B.2 Netkit Lab Small Internet Topology from Di Battista *et al.* [25] 342
Figure C.1 Example of $E_{whiteboard}$ for Simplified Small Internet Example 347
Figure C.2 Example of T for Simplified Small Internet Example . . 347
Figure C.3 Example of B for Simplified Small Internet Example . . 348
Figure C.4 Example of L_{Π} for Simplified Small Internet Example . 348

Figure C.5 Example of L_B for Simplified Small Internet Example . 349
Figure D.1 Group membership from NE labels 358
Figure D.2 NEC_FOR_NE for Network Element 4 360
Figure D.3 NEIGH_NE for Network Element 4 361
Figure D.4 NEIGH_NEI_NE for Network Element 4 362
Figure D.5 NEI_FOR_NE for Network Element 4 362
Figure D.6 NE_FOR_NEC result highlighted for Network Element 4 . 364
Figure D.7 NEI_FOR_NEC 365
Figure D.8 NEC_OTHER_NE 366
Figure D.9 NEC_OTHER_NEI 366
Figure D.10 Parallel Network Element Connections 367
Figure D.11 NEC_BY_NE_NE 368
Figure D.12 NEI_BY_NEC_NE 370
Figure D.13 NEC_FOR_NEI 371
Figure D.14 NEI_BY_NE 372
Figure D.15 NEIGH_NEI_FOR_NEI 373
Figure D.16 NEIGH_NE_FOR_NEI 373
Figure D.17 NEC_FOR_NEI 374

LIST OF TABLES

Table 4.1	Table of element notation symbols	96
Table 5.1	Summary of *create* and *remove* Low-Level Primitives	108
Table 5.2	Summary of *labels* Low-Level Primitives	109
Table 5.3	Summary of *associative array* Low-Level Primitives	109
Table 5.4	Summary of *properties* Low-Level Primitives	110
Table 5.5	Summary of High-Level Primitives	112
Table 5.6	Route Reflector Roles and Directions	151
Table 6.1	Summary of example code for OSPF example	164
Table 6.2	IDM, Template, and output for OSPF, iBGP and eBGP	187
Table 7.1	Mapping of theory to implementation terms	194
Table 8.1	Translation of names from theoretical example	229
Table 8.2	Large-scale European NREN topology summary statistics	291
Table C.1	Network Element Connections for Simplified Small Internet Example	345
Table C.2	Network Element Interfaces for Simplified Small Internet Example	346
Table C.3	Network Element Interface bindings for Simplified Small Internet Example	346
Table C.4	Loopback IP Address allocations for Simplified Small Internet Example	347
Table D.1	Table of element notation symbols	351
Table D.2	Return values for IS_PARALLEL for topology shown in Figure D.10	368
Table D.3	Summary of *create* Low-Level Primitives	375
Table D.4	Summary of *add* Low-Level Primitives	376
Table D.5	Summary of *remove* Low-Level Primitives	377
Table D.6	Summary of *initialise label* Low-Level Primitives	378
Table D.7	Summary of *remove label* Low-Level Primitives	379

LIST OF ALGORITHMS

Algorithm 5.1	Definition for SPLIT High-Level Primitive	117
Algorithm 5.2	Definition for MERGE High-Level Primitive	119
Algorithm 5.3	Definition for EXPLODE High-Level Primitive	121
Algorithm 5.4	Pseudo-Code for Physical Design Function	126
Algorithm 5.5	Pseudo-Code for Layer 2 Design Function	127
Algorithm 5.6	Pseudo-Code for structure of IP Address Design Function	129
Algorithm 5.7	Pseudo-Code for IP Address Loopback Allocation	130
Algorithm 5.8	Pseudo-Code for IP Address Infrastructure Allocation	132
Algorithm 5.9	Pseudo-Code for Layer 2 Connectivity Design Function	133
Algorithm 5.10	Pseudo-Code for OSPF Design Function	134
Algorithm 5.11	Pseudo-Code for iBGP Design Function	136
Algorithm 5.12	Pseudo-Code for eBGP Design Function	138
Algorithm 6.13	Pseudo-Code for Interface Description utility function	176
Algorithm 6.14	Pseudo-Code for interfaces component of device compiler	179
Algorithm 6.15	Pseudo-Code for OSPF component of device compiler	181
Algorithm 6.16	Pseudo-Code for iBGP component of device compiler	183
Algorithm 6.17	Pseudo-Code for eBGP component of device compiler	185
Algorithm E.18	Definition for E_{Π} Low-level Primitive	382
Algorithm E.19	Definition for DEGREE_Π Low-level Primitive	382
Algorithm E.20	Definition for Π_E Low-level Primitive	382
Algorithm E.21	Definition for Π_u and Π_v Low-level Primitives	383
Algorithm E.22	Definition for $\mathtt{other}_{\Pi}$ Low-level Primitive	383
Algorithm E.23	Definition for $\mathtt{other}_T$ Low-level Primitive	383
Algorithm E.24	Definition for NEC_BY_ENDPOINTS Low-level Primitive	383
Algorithm E.25	Definition for PARALLEL Low-level Primitive	384
Algorithm E.26	Definition for IS_PARALLEL_E Low-level Primitive	384
Algorithm E.27	Definition for E_T Low-level Primitive	384
Algorithm E.28	Definition for E_T Low-level Primitive	384
Algorithm E.29	Definition for T_u and T_v Low-level Primitives	385
Algorithm E.30	Definition for B_{π} Low-level Primitive	385
Algorithm E.31	Definition for $\mathtt{Degree}_T$ Low-level Primitive	385
Algorithm E.32	Definition for T_IS_BOUND Low-level Primitive	385
Algorithm E.33	Definition for CREATEθ Low-Level Primitive	386
Algorithm E.34	Definition for ADD_π Low-Level Primitive	386
Algorithm E.35	Definition for ADD_π Low-Level Primitive	386
Algorithm E.36	Definition for NEXT_NEC_δ Low-Level Primitive	386
Algorithm E.37	Definition for ADD_ϵ Low-Level Primitive	386

Algorithm E.38 Definition for ADD_B Low-Level Primitive 387
Algorithm E.39 Definition for ADD_π_T Low-Level Primitive 387
Algorithm E.40 Definition for ADD_T Low-Level Primitive 387
Algorithm E.41 Definition for SET_BINDING Low-Level Primitive 387
Algorithm E.42 Definition for ADD_NEC_BY_NE_NE Low-Level Primitive . 387
Algorithm E.43 Definition for REMOVE_π Low-Level Primitive 388
Algorithm E.44 Definition for REMOVE_B Low-Level Primitive 388
Algorithm E.45 Definition for REMOVE_τ_π Low-Level Primitive 388
Algorithm E.46 Definition for REMOVE_τ_τ Low-Level Primitive 388
Algorithm E.47 Definition for REMOVE_ϵ Low-Level Primitive 388
Algorithm E.48 Definition for ADD_L_ϵ Low-Level Primitive 389
Algorithm E.49 Definition for ADD_L_π Low-Level Primitive 389
Algorithm E.50 Definition for ADD_L_T_π Low-Level Primitive 389
Algorithm E.51 Definition for ADD_L_T_τ Low-Level Primitive 389
Algorithm E.52 Definition for REMOVE_L_π Low-Level Primitive 389
Algorithm E.53 Definition for REMOVE_L_τ_π Low-Level Primitive . . . 390
Algorithm E.54 Definition for REMOVE_L_τ_τ Low-Level Primitive . . . 390
Algorithm E.55 Definition for REMOVE_L_ϵ Low-Level Primitive 390
Algorithm E.56 Definitions for the SET_λ Low-Level Primitives 391
Algorithm E.57 Definition for GROUPING High-Level Primitive 392
Algorithm E.58 Definition of BOUNDARY_NODES High-Level Primitive . . 392
Algorithm E.59 Definition for E_EQUIVALENT_Π High-level Primitive . . 392
Algorithm E.60 Definition for E_EQUIVALENT_T High-level Primitive . . 393
Algorithm E.61 Definition of CONNECTED_NODES High-Level Primitive . 393
Algorithm E.62 Definition of Role Access Functions High-Level Primitives 393
Algorithm E.63 Definition of TO_GRAPH High-Level Primitive 394
Algorithm E.64 Definition of GRAPH_CONNECTED_NODES Function . . . 394
Algorithm E.65 Definition for SET_LABEL_π High-Level Primitive 394

LIST OF CODE LISTINGS

Listing 2.1 Example template from Enck *et al.* [31] 21
Listing 2.2 Example template from Elbadawi *et al.* [29] 26
Listing 2.3 Example of a template from Gottlieb *et al.* [41] 36
Listing 2.4 Example of a template from PRESTO, taken from Enck *et al.* [31] 39

Listing 4.1 Example device model for *r4* 67
Listing 4.2 Example router configuration for *r4* 67

Listing 6.1 Quagga interface and OSPF configuration 159
Listing 6.2 Example Quagga Device Model 159
Listing 6.3 IOS interface and OSPF configuration 160
Listing 6.4 Example IOS Device Model 160
Listing 6.5 IOS-XR interface and OSPF configuration 161
Listing 6.6 Example IOS-XR Device Model 161
Listing 6.7 NX-OS interface and OSPF configuration 162
Listing 6.8 Example NX-OS Device Model 162
Listing 6.9 Junos configuration 163
Listing 6.10 Example Junos Device Model 163
Listing 6.11 Example device global information 176
Listing 6.12 Interfaces Component of Intermediate Device Model 179
Listing 6.13 OSPF Component of Intermediate Device Model 180
Listing 6.14 iBGP Component of Intermediate Device Model 183
Listing 6.15 eBGP Component of Intermediate Device Model 184
Listing 6.16 Interfaces JSON 187
Listing 6.17 Interfaces Template 187
Listing 6.18 Interfaces Output 187
Listing 6.19 Combined Intermediate Device Model 188
Listing 6.20 Combined Jinja2 Template 188
Listing 6.21 Combined example device configuration 188

Listing 7.1 Running AutoNetkit as a console script 195
Listing 7.2 Installation of *thesis16* branch using the PIP package manager . 198
Listing 7.3 Initialisation 203
Listing 7.4 Create Network Whiteboard 203
Listing 7.5 Update Visualisation 205
Listing 7.6 Set Edges in Network Whiteboard 205
Listing 7.7 Physical Design Function 206
Listing 7.8 Layer 2 Design Function 207
Listing 7.9 Lab output showing *nidb* information 216
Listing 7.10 Example Netkit lab configuration for House topology 216
Listing 7.11 Example Intermediate Device Model 218
Listing 7.12 Example OSPF configuration for *r4* of House topology 219
Listing 7.13 Example BGP configuration for *r4* of House topology 219
Listing 7.14 Compile Platform 220
Listing 7.15 Tree Output for Lab 221
Listing 7.16 Example ospfd.conf output for r4 221

Listing 7.17 Example bgpd.conf output for r4 222
Listing 7.18 Launching Lab 223
Listing 7.19 Logging in and running SHOW IP ROUTE from *r4* 223
Listing 7.20 Post-processed SHOW IP ROUTE from *r3* 224
Listing 7.21 Output of TRACEROUTE from *r3* to *r5* 225
Listing 7.22 SHOW IP BGP SUMMARY from *r4* 225

Listing 8.1 *show ip route* output for *as1r1* 238
Listing 8.2 Output of TRACEROUTE from *as300r1* to *as100r2* 239
Listing 8.3 List of nodes with their attribute data 244
Listing 8.4 List of edges with their attribute data 245
Listing 8.5 Specification of BGP Roles 246
Listing 8.6 Small Internet Infrastructure Address Allocations 251
Listing 8.7 Small Internet Loopback Address Allocations 252
Listing 8.8 Nodes with redistribute connected set 254
Listing 8.9 Prefixes announced by each eBGP router 257
Listing 8.10 Customer prefixes for each eBGP provider router 258
Listing 8.11 Intermediate Device Model for *as20r1* 262
Listing 8.12 Generated *ripd.conf* configuration file for *as20r1* 264
Listing 8.13 Generated *bgpd.conf* configuration file for *as20r1* 264
Listing 8.14 Traceroute from *as100r3* to *eth0.as200r1* 266
Listing 8.15 Traceroute from *as100r3* to *eth1.as300r3* 266
Listing 8.16 Traceroute from *as100r3* to *eth0.as300r3* 266
Listing 8.17 SHOW IP BGP output for *as1r1* 268
Listing 8.18 SHOW IP BGP output for *as100r1* 268
Listing 8.19 SHOW IP ROUTE output for *as300r3* 269
Listing 8.20 VLANs grouped by pseudo-node 276
Listing 8.21 Intermediate Device Model for *r9* 282
Listing 8.22 IOSv router configuration for *r9* 283
Listing 8.23 Intermediate Device Model for *sw4* 284
Listing 8.24 IOSvL2 Switch configuration for *sw4* 284
Listing 8.25 Output of SHOW VLAN on *sw4* 286
Listing 8.26 Output of TRACEROUTE from *r11* to *r7* 287
Listing 8.27 Netkit Compilation Performance 300
Listing 8.28 Extract of Intermediate Platform Model showing eBGP peers . 301
Listing 8.29 C-BGP Compilation Performance 302
Listing 8.30 TRACEROUTE from London.JANET to Roznava.SANET 302
Listing 8.31 Post-processed TRACEROUTE from London to Roznava 302
Listing 8.32 C-BGP BGP SHOW RIB result for DE.GEANT node 303

Listing B.1 BGPd configuration for *as20r1* from Di Battista *et al.* [25] 343
Listing B.2 RIPd configuration for *as20r1* from Di Battista *et al.* [25] 343
Listing B.3 BGPd configuration for *as30r1* from Di Battista *et al.* [25] 344

Listing D.1 Example of grouping for IP Addresses 357

Listing F.1 Source code for setup for OSPF configuration example 395
Listing F.2 Simple Router Compiler for OSPF configuration example . . . 398
Listing F.3 Simple Quagga Compiler for OSPF configuration example . . 398
Listing F.4 Simple IOS Compiler for OSPF configuration example 399
Listing F.5 Simple IOS-XR Compiler for OSPF configuration example . . . 400
Listing F.6 Simple NX-OS Compiler for OSPF configuration example . . . 401
Listing F.7 Simple Junos Compiler for OSPF configuration example 402

Listing F.8 Simple Platform Compiler for OSPF configuration example . . 403
Listing F.9 Template for Quagga for OSPF configuration example 404
Listing F.10 Template for IOS for OSPF configuration example 405
Listing F.11 Template for IOS-XR for OSPF configuration example 405
Listing F.12 Template for NX-OS for OSPF configuration example 406
Listing F.13 Template for Junos for OSPF configuration example 406
Listing F.14 OSPF example compilation and rendering source code 406
Listing F.15 OSPF JSON 408
Listing F.16 OSPF Template 408
Listing F.17 OSPF Output 408
Listing F.18 iBGP JSON 408
Listing F.19 iBGP Template 408
Listing F.20 iBGP Output 408
Listing F.21 eBGP JSON 409
Listing F.22 eBGP Template 409
Listing F.23 eBGP Output 409

Listing G.1 Quagga Template Example 410
Listing G.2 Layer 2 Connectivity Design Function 411
Listing G.3 IP Addressing Design Function 411
Listing G.4 OSPF Design Function 412
Listing G.5 OSPF Advertise Design Function 412
Listing G.6 iBGP Design Function 412
Listing G.7 eBGP Topology component of eBGP Design Function 413
Listing G.8 eBGP Network Advertisement component 413
Listing G.9 Create Platform Compiler 414
Listing G.10 NIDB Shortcuts 415
Listing G.11 Compile Platform 416
Listing G.12 Specify Device Compiler 416
Listing G.13 Setup Environment 418
Listing G.14 BGP Template 418
Listing G.15 Daemons Template 419
Listing G.16 OSPF Template 419
Listing G.17 Zebra Template 419
Listing G.18 Hostname Template 419
Listing G.19 Startup Template 419
Listing G.20 Shadow Template 420
Listing G.21 SSH Config Template 420
Listing G.22 Lab Template 421
Listing G.23 Setup Templates 421
Listing G.24 Render Lab 421
Listing G.25 SHOW IP ROUTE from *r3* 422
Listing G.26 SHOW IP OSPF NEIGHBOR from *r3*. 422
Listing G.27 SHOW IP BGP SUMMARY from *r5* 422
Listing G.28 Output of TRACEROUTE from *r5* to *r1* 423

Listing H.1 Load GraphML 424
Listing H.2 Create Network Whiteboard 424
Listing H.3 Add Edges to Network Whiteboard 425
Listing H.4 Update Visualisation 425
Listing H.5 Create Physical Network View 425
Listing H.6 Create Layer 2 Network View 425
Listing H.7 Create Layer 2 Connectivity Network View 426

Listing H.8 Create IP Network View . . . 426
Listing H.9 Create OSPF Network View . . . 427
Listing H.10 OSPF Networks . . . 427
Listing H.11 Create iBGP Network View . . . 427
Listing H.12 Create eBGP Network View . . . 428
Listing H.13 eBGP Networks . . . 428
Listing H.14 Netkit Platform Compiler . . . 429
Listing H.15 Quagga Device Compiler . . . 429
Listing H.16 Launching Lab . . . 431
Listing H.17 *show ip route* output for *as20r1* . . . 432
Listing H.18 *show ip bgp summary* output for *as1r1* . . . 432
Listing H.19 *show ip bgp summary* output for *as20r1*label . . . 432
Listing H.20 Post-processed output of TRACEROUTE from *as300r1* to *as100r2* 433
Listing H.21 Initialisation . . . 434
Listing H.22 Load Nodes and Edges . . . 434
Listing H.23 Load Network Whiteboard nodes . . . 434
Listing H.24 Load Network Whiteboard edges . . . 435
Listing H.25 Create Physical Network View . . . 435
Listing H.26 Sum Neighbor Averages . . . 435
Listing H.27 Create Layer 2 Network View . . . 435
Listing H.28 Create Layer 2 Connectivity Network View . . . 436
Listing H.29 Create IP Network View . . . 437
Listing H.30 Create RIP Network View . . . 437
Listing H.31 Set RIP Redistribution . . . 438
Listing H.32 Set RIP Advertisement Networks . . . 438
Listing H.33 Create iBGP Network View . . . 438
Listing H.34 Create eBGP Network View . . . 438
Listing H.35 Apply eBGP Policies . . . 439
Listing H.36 Add eBGP Prefix Advertisements . . . 439
Listing H.37 Apply per-node eBGP policies . . . 440
Listing H.38 Create Prefix Lists . . . 442
Listing H.39 Create Route Maps . . . 443
Listing H.40 Set Routing Protocol Redistribution . . . 443
Listing H.41 RIP Template . . . 444
Listing H.42 BGP Template . . . 444
Listing H.43 Netkit Platform Compiler . . . 446
Listing H.44 Quagga Device Compiler . . . 446
Listing H.45 BGPd configuration for *as30r1* . . . 449
Listing H.46 Prepare Network Whiteboard . . . 452
Listing H.47 Create Network Whiteboard . . . 452
Listing H.48 Add Edges to Network Whiteboard . . . 453
Listing H.49 Create Physical Network View . . . 453
Listing H.50 Create VLAN Network View . . . 453
Listing H.51 Aggregate Nodes High-Level Design Primitive . . . 453
Listing H.52 Aggregate VLANs . . . 454
Listing H.53 Form VLAN Broadcast Domains . . . 455
Listing H.54 Create Switches Network View . . . 455
Listing H.55 Create Layer 2 Network View . . . 455
Listing H.56 Create Layer 2 Connectivity Network View . . . 456
Listing H.57 Create IP Network View . . . 456
Listing H.58 Create OSPF Network View . . . 457
Listing H.59 VIRL Platform Compiler . . . 458

Listing H.60 VIRL IOSv Router Compiler 459
Listing H.61 VIRL IOSvL2 Switch Compiler 460
Listing H.62 VIRL IOSv Router Template 461
Listing H.63 VIRL IOSvL2 Switch Template 462
Listing H.64 VIRL XML Packaging 463
Listing H.65 Output of SHOW VLAN for *sw8* 465
Listing H.66 Output of SH IP ROUTE for *r2* 465
Listing H.67 Output of SH IP ROUTE for *r9* 465
Listing H.68 Output of SH IP ROUTE for *r9*, after post-processing 466
Listing H.69 Load Source Graphml 468
Listing H.70 Add Nodes to Network Whiteboard 468
Listing H.71 Add Edges to Network Whiteboard 468
Listing H.72 Normalise Locations 468
Listing H.73 Create Physical Network View 470
Listing H.74 Create Layer 2 Network View 470
Listing H.75 Create Layer 2 Connectivity Network View 470
Listing H.76 Allocate IP Addresses 470
Listing H.77 Create OSPF Network View 471
Listing H.78 Create iBGP Network View 471
Listing H.79 Allocate iBGP Roles 471
Listing H.80 Create iBGP Connectivity 472
Listing H.81 Highlight Route Reflectors in Visualisation 472
Listing H.82 Create eBGP Network View 473
Listing H.83 BGP Network Advertisements 473
Listing H.84 Example of Paths 473
Listing H.85 Example analysis of paths in NREN1400 topology 474
Listing H.86 Quagga Device Compiler 475
Listing H.87 Quagga Device Compiler 475
Listing H.88 C-BGP Platform Compiler 477
Listing H.89 C-BGP Template 479
Listing H.90 Extract of C-BGP configuration file 480
Listing H.91 C-BGP BGP SHOW RIB result for London node 482
Listing H.92 C-BGP NODE SHOW RT result for London node 483
Listing H.93 C-BGP BGP SHOW RIB result for DE.GEANT node 485

Listing I.1 Source code for validation example 487

1 INTRODUCTION

1.1 INTRODUCTION

In this chapter we introduce the main content of the thesis. We first motivate the need for the work. We provide an overview of the deficiencies in current solutions, and how the work of this thesis aims to overcome these. We then provide an overview of the structure of this thesis. Finally, we summarise the research contributions of this thesis.

1.2 MOTIVATION

Computer networks have become fundamental to all aspects of the information age. Computer networks have evolved progressively and much of the infrastructure for their design and management was created in an ad-hoc way to address problems as they arose. Far too many network configuration updates are inserted into existing networks manually at the lowest level command line interface of routers. Whilst networking professionals work with high-level conceptual designs, often called high-level network configuration policies, these are often informal documents that require extensive interpretation before they can be implemented. The consequences of this situation are that errors, related to the large level of details that individual probationers need to be aware of, are common place. This can lead to unplanned outages and very high maintained costs for upgrades.

This situation has not gone unnoticed in the academic community. The most well-known proposal to address these concerns is software defined networking (SDN). However SDN implies a revolution in the way networks are designed, implemented and managed and requires wholesale replacement of low level hardware. This has proved attractive for cloud players who have total control of the infrastructure that they use. However SDN has faced barriers to adoption in the majority of network installations where a complete replacement of hardware and software together with the retraining of staff and engineering new interfaces to internet as a whole are significant.

Yet many of the issues surrounding network management in existing installation could be addressed by the application of methods that are already widely understood in other domains such as computer languages. For example the overall problem has been likened to the programming of a distributed system. This approach has also been pursued by researchers but with only limited success in adoption by practitioners. One reason for these efforts not bearing fruit has been the adoption of an input specification "language" that does not adequately expresses or is sympathetic to the domain knowledge that network engineers already use for design. In addition the internal representation used in these research tools is not based on the apparently natural choice of graphs. In addition research solutions often only addressed a limited part of the design domain (such as the BGP routing protocol), or the solution cannot handle the large range of special cases that common arise in systems that have evolved from low-level manual input.

Introducing a new approach to the industry for the design of networks starting from a high-level policy artefact and automatically transforming this into router configurations that are compatible with existing hardware is still an intensive task. Inevitably there is a need to decide which parts of the process are of increased importance, such that need to be addressed first and which parts can be introduced later. Researchers have acknowledged this by identifying various aspects of the overall task of network configuration management. In particular it has been suggested network management has a static design component and a dynamic update component. The static design component is a good place to start changing process because of the time scales are longer and it is not reliant on non standard APIs and interface for real-time telemetry and monitoring of the dynamic network state. It is argued that a good dynamic update process is totally reliant on good abstractions and these abstractions are better evaluated initially for the static design case. So the although work in this thesis is limited to the static case, it is considered as an essential foundation of a fully dynamic design update tool.

Given the objective stated above it is clear that any tool embodying a new approach to high-level network policy configuration needs to be trialled and evaluated by a wide range of practitioners. One way that network engineers can experiment with a new approach is to make it available in large scale simulation environments and to make the source code available for industry contributions. This approach to validation of tools gives much greater confidence in their scalability and flexibility in handling real life situations.

Whilst the future of network design is impossible to predict precisely it is clear that the future reliability expected of computer networks and the cost reduction expected in their design and maintenance is unlikely to be achieved by the status quo. It seems likely therefore that automated tools will play a much larger role in network management in the future. This automation cannot ignore the current installed based of hardware and software nor the skills and knowledge of existing network engineers. It could be anticipated that islands of automation may develop inside SDN style networks but there is also a need to automate processes across such islands. It is this research opportunity that this thesis aims to fulfil.

In the next section, we will provide a more detailed background to some of these issues.

1.3 BACKGROUND

In 2001, the Internet Engineering Task Force (IETF) held a two day meeting to discuss challenges in network management. This meeting was summarised in RFC 3139 [89]. The Problem Statement from RFC 3139 is as follows:

> Configuring large networks is becoming an increasingly difficult task. The problem intensifies as networks increase their size, not only in terms of number of devices, but also with a greater variety of devices, with each device having increasing functionality and complexity. That is, networks are getting more complex in multiple dimensions simultaneously (number of devices, time scales for configuration, etc.) making the task of configuring these more complex.
>
> In the past, configuring a network device has been a three step process. The network operator, engineer or entity responsible for the network created a model of the network and its expected behavior. Next, this (model + expected behavior) was formalised and recorded in the form of high-level policies. Finally, these policies were then translated into device-local configurations and provisioned into each network device for enforcement.
>
> Any high-level policy changes (changes in the network topology and/or its expected behavior) needed to be translated and provisioned to all network devices affected by the change.
>
> In this model, network operators or engineers functioned as configuration data translators; they translated the high-level policies to device-local configuration data. (Sanchez *et al.* [89])

1.3.1 *Network Management Challenges*

Why do these problems occur? Network configuration is complex.

Configuring a network by manually configuring the individual network devices has been likened to programming in distributed assembly language [60] [34].

[16] identifies key features of the configuration management problem: the high degree of configurability, where routers provide many configuration option; the fact that routers are configured using "complex, low-level" languages, and the rapid rate at which features are added to routers. The process of manual configuration is both expensive and error-prone [16].

Network science lacks the formal rigidity found in other disciplines of Computer Science [95]. The development of high-level abstractions to represent networks will allow more formal methods to be introduced to networks.

Configuring networks device by device reduces thinking to be in terms of implementation, rather than the high-level design or architecture goal. There are a vast array of protocols and protocol configuration options. This is one of the key contributors to the complexity of device configuration [16].

1.3.2 *Network Requirements*

Modern networks are more commonly large in size: a service provider network can contain thousands of devices [21]. These devices each have configuration files ranging from hundreds to thousands of lines of low-level configuration syntax [21]. These configuration files describe not only individual device setup, but the role the device plays in services distributed across the network. This is a critical problem: it is not a problem of just configuring the device in isolation, but the effect that its configuration has on the entire network. A change in a single device could have network-wide impact [21].

Modern networks undergo rapid changes, both in network demands, such as adding new customers or business features, and in new features offered by network vendors, such as new protocols [31]. Further, there are unplanned events: devices and links can fail. Dealing with both planned and unplanned events is complex, yet timely response is critical to deliver high-performance networks [21]. This is further complicated when the network itself is the database: an operator making an emergency fix for an unexpected outage may unknowingly violate a high-level network design requirement. A change in a configuration on one device may violate the assumptions made when a configuration on another device was written, leading to unpredictable network effects [21].

[21] identifies two key requirements for network operators to meet: correctness, where changes to the network should not cause any side-effects in meeting their outcomes; and timeliness, where changes are able to be implemented in a timely fashion. These requirements are difficult to meet using manual systems, and become more difficult as networks grow: manual configuration does not efficiently scale. Production networks are continuously carrying traffic [20], increasing the impact of any outage due to an upgrade or device failure.

As well as network availability being a core requirement, network performance is becoming increasingly important [110]. Network management typically involves a human operator unifying and reconciling disparate information across multiple sources, which complicates network troubleshooting [110]. These changes include adding new customers, introducing new services or features, or upgrading hardware [15].

Both equipment and protocols evolve — router operating system upgrades are released, and extensions with new options are added to protocols [16]. Additionally the operational goals of networks change with evolving business requirements. [16]. Adding a new customer or service may require co-ordination with many internal organisations and systems within a service provider [31].

1.3.3 *Configuration Files*

Configuration files themselves are large: a 500-router network can have over one million lines of configuration [34], and are implemented in low-level, device-specific languages [31]. While networks and their services are designed network-wide, at a high-level, it is generally up to an operator to translate these high-level design rules into their low-level mechanisms. This is complex, and error-prone. Not surprisingly, problems can occur in this manual translation process: a survey using the rcc [34] BGP analysis tool detected faults in the operator implementation of high-level designs into low-level configuration syntax. These were due to the ambiguity in implementation: there is often more than one way to realise the same policy; and that configuration in a low-level language is often unintuitive [34]. Whereas a high-level language allows design flexibility and expressiveness, implementing using low-level syntax is obscure, complex, and provides many opportunities for mistakes [34].

Configuration through low-level configuration files creates the potential for inconsistencies both within a single device configuration, and across configuration files in the network [35]. The router configurations themselves consist of many "assembly language" commands [16].

Networks are made up of multiple systems, and multiple protocols [64], each of which typically has many tunable parameters [16]. This diversity complicates management tools: one operator's definition of a correctly functioning network may differ to another's, as each network has specific requirements and design goals [64]. An operator must co-ordinate the configuration of such a diverse range of devices and protocols [29]. Additionally vendor-specific languages differ in their implementation of the same feature, and there is often multiple ways to implement a high-level objective [34].

Further, the inter-dependency of configuration parameters means that changing one configuration parameter may lead to a complex chain of subsequent effects — there are often implicit dependencies between configuration parameters [29].

Network outages occur due to network misconfigurations, as described in reports [49] [51], or lack of testing configurations [48]. Research papers have been published studying the occurrence of outages, including [97] [69] [101] [47] [102]. These problems: large-scale, low-level, and diversity, mean that manual configuration is error-prone, time-consuming, and expensive [16]. They are also error-prone: outages can violate contractual obligations, disrupt business workflows, and cause delays in service establishment, delaying revenues [31]. A single misconfiguration can take out a Wide-Area Network's control plane [42].

Once created, these method of procedure documents must be manually applied, by an engineer carefully following them step-by-step. This is a time consuming process, and liable to mis-interpretation: even the most complete document cannot realistically cover all implementation scenarios. Any ambiguities could be mis-interpreted by the operator applying the design guidelines. Not all of these errors are immediately obvious: some may remain latent in the network, and only exposed in the event of an network event such as a device failure, leading to a cascading series of problems. Without a canonical network model it is difficult to verify field configurations.

Configuration data supplied from disparate sources is also problematic, in that it may not be complete, or remain accurate as the network evolves [31].

[31] provides a detailed motivation for an automated configuration management system. Manual configuration is costly, time-consuming and doesn't scale. A lot of time and resources are invested in documenting and interpreting network standards, and learning the device-specific interfaces used to implement configuration. This typically results in a model configuration, which is used to configure subsequent network devices. Creating such a document is very time consuming, taking many months, and expensive.

In this section we have shown the problems faced by network operators, and the need for new approaches and tools that assist in the network configuration management problem.

In the next section we describe how the work in this thesis aims to address these issues.

1.4 STRUCTURE OF THESIS

This thesis relates to the domain of network configuration, which is the task of converting high-level business and technical requirements into their concrete instantiation within a network device, such as a router of switch. As discussed, this process often requires a number of manual steps, introducing complexity and the risk of error.

The primary focus of this thesis is in bridging the gap between network-wide policy specification, and low-level device configuration. This is performed by introducing a set of abstractions that can be composed into a toolchain. This toolchain enables to expression of high-level network configuration policy to be automatically compiled to the individual low-level device configurations.

The structure of the remaining chapters of the thesis is as follows:

Part I provides a background of the problem space and the methodology used in this thesis. Chapter 2 is a survey of related literature in the field, from both the research community and industry. This identifies the research gaps in existing solutions. Chapter 3 identifies five research questions leading from these research gaps, and outlines the methodology used to address these research questions.

Part II introduces the proposed solution with a detailed explanation of the abstractions and transformations. Chapter 4 describes the **Network Whiteboard**, used for describing the high-level policy; and the **Network View** abstractions, which are used as the high-level intermediate form. Chapter 5 describes the **Design Functions** which are used to construct the Network Views. Chapter 6 describes the **Device Compilation** step, which transforms the Network Views into the low-level **Device Configurations**.

Part III presents an implementation and evaluation of the proposed solution. Chapter 7 describes an implementation of the system in Python, which is then used for a number of Case Studies in Chapter 8.

In Part IV we describe future work to expand the approach described further, in Chapter 9. Finally, in Chapter 10 we summarise how the thesis has addressed the research questions identified in Chapter 3.

Part I

BACKGROUND AND METHODOLOGY

2

RELATED WORK

2.1 INTRODUCTION

In this chapter we motivate the need to develop abstractions and tools to improve the network management process, and survey existing literature in this area. In the previous chapter we noted that configuration management was an important function in network design. It was noted that configuration management consumed a large part of the roles of network operators yet at the same time was prone to errors and inconsistencies that caused downtime. We now explore the various concepts and associated tools that have been proposed to resolve the network configuration challenge. We conclude that there remains scope for a comprehensive solution that addresses the significant facets of the network configuration problem. The organisation of this section is described below.

In the first part we explain the scope of network configuration management. In the following sections we elaborate on each stage in the common workflow of network configuration management. These stages are: high-level network policy, high-level topology dependant network description, network wide configuration views, Network View to device configuration translator and low-level configuration deployment. Within the section describing each stage we explain the abstraction and concepts which are covered by the stage and review the various research contributions in the literature that relate to each stage. Finally, we identify research opportunities that have yet to be addressed.

2.2 CHAPTER OVERVIEW

This chapter is organised as follows. We first provide the background and motivation for this thesis. We discuss the significant challenges present in the network configuration management process. We next provide an overview of alternative approaches to configuration management, including the use of centralised controllers with current network hardware, Clean-Slate and SDN approaches, and Autonomic Networking. We then focus on solutions proposed for more effective management of current network hardware and software infrastructure. We look at the need for high-level network management, and survey existing literature in this area. We next identify the need for an abstracted view of the network, and intermediate models, and review work on this in the current literature. We then discuss a template-based approach to transform the intermediate models into low-level device configurations. Finally, we need a way to validate the approach to network configuration management presented in this thesis. To do so, we provide an overview of testbed and simulation platforms that can be used to validate our approach.

2.3 SCOPE OF NETWORK CONFIGURATION MANAGEMENT

The Internet Engineering Task Force (IETF), has defined the network configuration management problem as taking the inputs of high-level (topology independent) network policies, desired and current network topology and data concerning the current network status and performance to manage a set of device configurations. In this thesis we are concerned initially with that part of the network configuration management task that involves the generation of new device configurations from high-level network policy and network topology descriptions. This is in contrast to the dynamic changes made to a network once the new device configurations have been generated and loaded onto the network devices. It should be noted that the current status and performance of the network is a real-time dynamic set of information, whereas the network policies and topology are more static sources of information. Normally in designing new networks or adding to existing networks the designers would only consider the static information. Well-designed abstractions and transformations for static network design will however contribute significantly to structuring any structuring a dynamic network design tool.

In RFC 3139, Sanchez *et al.* have suggested 15 requirements for network configuration management. These are reproduced in full in Appendix A. We reproduce the relevant points 1, 2, 3, and 14 below.

> 1 Provide means by which the behavior of the network can be specified at a level of abstraction (network-wide configuration) higher than a set of configuration information specific to individual devices,

> 2 Be capable of translating network-wide configurations into device-local configuration. The identification of the relevant subset of the network-wide policies to be down-loaded is according to the capabilities of each device,
>
> 3 Be able to interpret device-local configuration, status and monitoring information within the context of network-wide configurations,
>
> 14 Be flexible and extensible to accommodate future needs. Configuration management data models are not fixed for all time and are subject to evolution like any other management data model. It is therefore necessary to anticipate that changes will be needed, but it is not possible to anticipate what those changes might be. Such changes could be to the configuration data model, supporting message types, data types, etc., and to provide mechanisms that can deal with these changes effectively without causing inter-operability problems or having to replace/update large amounts of fielded networking devices, (Sanchez *et al.* [89])

We denote these requirements as *IR:1–15*. The work presented in this thesis only addresses requirements *IR–1*, *IR–2*, *IR–3*, and *IR–14*. The remainder of this literature review will focus on these requirements. Requirements *IR–4* to *IR–13* and *IR–15* are primarily directed at the low-level mechanics of monitoring and provisioning of a network, or dynamic modifications, and are thus not considered in this thesis. We note that recent work such as in NETCONF [32] make contributions to address these requirements.

RFC 3139 proposed a workflow for network configuration management as shown in Figure 2.1. It might be expected that since this workflow was proposed more than 14 years ago that this high-level approach would have been widely adopted and suitable tools had been developed to support it. However as we shall show in this chapter this is not necessarily the case.

There has been more recent research published on the general concept of High-Level Network Configuration Policy, since RFC 3139 was published. In general, this research supports the point of view of RFC 3139.

For example, Oppenheimer *et al.* [80] describe tools to ensure that an operator's mental model of a network configuration matches that of the network itself. They describe the need for widely used generic tools, where an operator can describe their intent in a high-level manner, which can then be checked and appropriate configurations generated. They state that that a GUI wrapper around existing tools is not sufficient: the tools themselves need to be fundamentally advanced.

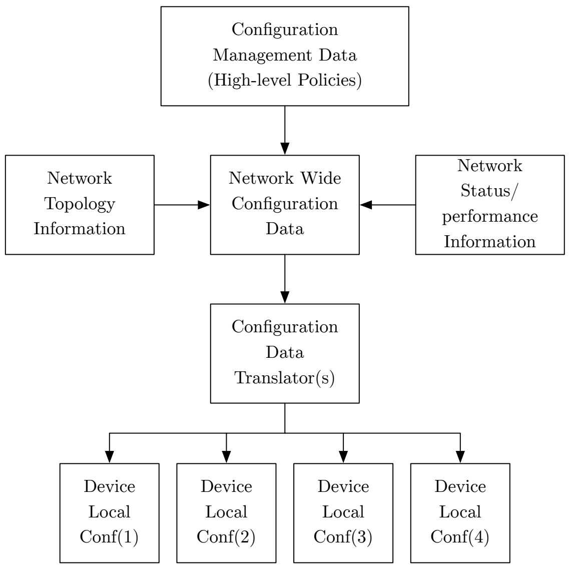

Configuration Management Data (High-level Policies)
Network Topology Information
Network Wide Configuration Data
Network Status/ performance Information
Configuration Data Translator(s)
Device Local Conf(1)
Device Local Conf(2)
Device Local Conf(3)
Device Local Conf(4)

Figure 2.1: Reproduction of Figure 2, *Proposed model for configuring network devices*, from Sanchez *et al.* [89]

Indeed, in their proposal of the OpenConfig [38] project in 2015, Google present "Model-driven network management", with an intent-based workflow shown in Figure 2.2. Their approach consists of three data sources:

1. Topology: the network structure
2. Configuration: the configuration data structure and content
3. Telemetry: the monitoring data structure and attributes (Google [38])

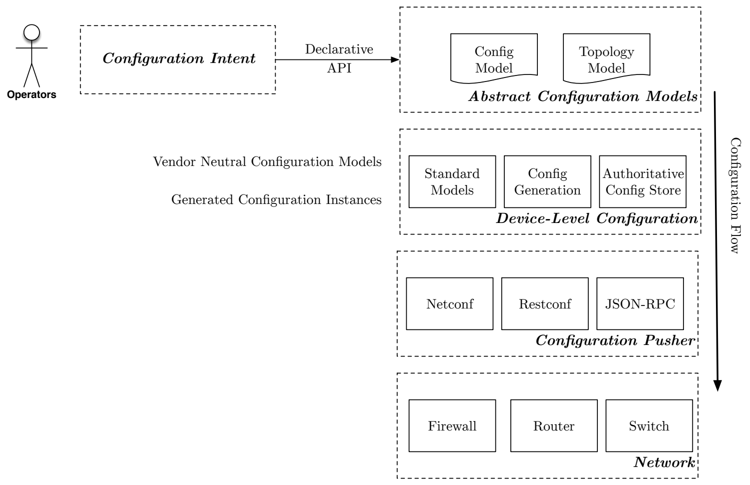


Figure 2.2: Reproduction of "Intent-based configuration flow" from [38]

They note that the management plane has not kept pace with the needs of network management, particularly in the areas of "proprietary CLIs (command line interfaces) requiring scripts"; "imperative and incremental configuration", and a "lack of abstractions" [38].

Other researchers, Enck *et al.* [31] identify a number of requirements for a network management platform. It must support existing configuration languages: care must be taken when defining generalised abstractions, as they can introduce layers of interpretation between the specification and the actual device configuration. Network operators often do not have the time or the inclination to learn new abstractions, and instead enjoy greater efficiency working with the native configuration interfaces of network devices — despite their shortcomings [31].

Enck *et al.* have pointed out that practitioners are unlikely to adopt languages and abstractions that are foreign to their day-to-day work. However, Enck *et al.* suggest that one solution to this problem is a description language that closely adheres to the low-level device configuration language [31]. This could be viewed as the lowest-common denominator solution. It does not solve the complexities of the problem, or addresses the concerns raised in RFC 3139.

There are other researchers who have contributed ideas to the structuring of the High-Level Network Configuration Policy.

Caldwell *et al.* have noted manual configuration is not a sound basis for progress in the field. They note that often the only representation of the network state is the configurations in the devices themselves. Because this configuration is low-level, this inhibits automation and reasoning. Feldmann *et al.* [35] observe that the most important lesson in creating a configuration management tool is the structure of the software. Feldmann *et al.* also advocates a database-driven solution.

Chen *et al.* [22] emphasise that finding the right level of abstraction is one of the biggest challenges in network management. Feamster *et al.* recommend that the network should be configured using a centralised, high-level language that directly represents the policies [34], and low-level mechanisms should be hidden from operators [34]. Banks [6] presents an industry point of view, noting that just providing new APIs to existing low-level functions does solve the network configuration management problem. Rather, we need network-wide abstraction models. Beckett *et al.* [8] note that the semantic mismatch between the high-level objectives and the low-level configurations is a major contributor to network misconfigurations.

In conclusion, there is a consensus amongst research and industry of the need for a change in the way in which network policy is defined and implemented. Some researchers have noted that unless the new abstractions are adopted by industry practitioners, progress will be slow. Any successful progress is likely to require advancements in the representation of the network state.

In the next section, we structure our discussion of research contributions around a layered taxonomy of descriptions, abstractions, and transformations, which lead from the high-level policy to the device configuration. These three layers we used in the taxonomy are the High-Level Network Configuration Policy description language, intermediate network representation, and the low-level device configuration state.

2.4 general comments about high-level network representation

In discussing the remaining detailed research in the field, we have introduced a taxonomy consisting of three layers, that represents the transition from the high-level policy document to the low-level device configurations.

In this section we describe how researchers have addressed the problem of representing high-level policy and their approaches to transforming these representations into a lower-level form. First, we explore how researchers have provided details concerning this top layer of abstraction. Then, we discuss how each of the proposed tools has implemented these ideas, at this level of abstraction.

In this section we gather together various research contributions that help to define what we mean by the top level of abstraction in our taxonomy: the representation of High-Level Network Configuration Policy.

In RFC 3139, Sanchez *et al.* suggest that high-level network (configuration) policy involves "the construction of a model of the network and its expected behaviour" [89, Page 4]. However RFC 3139 does not provide a detailed or prescriptive approach to how this model might be achieved.

Freeman [37] suggests that network configuration management should be divided between purely physical inventory and logical inventory. In the area of logical inventory he suggest that policy is a combination of conditions and actions that lead to router configurations. He calls these "condition action based" abstractions.

Vanbever *et al.* [104, Page 1] refer to the network policy level as high-level configuration objectives which are "design decisions made by the architect about the organisation of the network". We will discuss the details of Vanbever's work in the section on NCGuard that follows.

Chen *et al.* [20] suggest that high-level policies are a type of constraint "expressed independently from the authors of operation transactions". By operation transaction he means the low-level configuration of routers. Chen goes on to state that high-level policies should be "considered declarative in that they describe what should happen as opposed to how to enforce them during each network operation". As Chen's proposed tool is more targeted at dynamic network management than static network design, we do not discuss it further in this review.

In describing high-level policy Narain [73] uses the term "end-to-end requirements". Narain suggests that a specification language should be developed that specifies the policy (the end to end requirements) of a network configuration. He gives as an example of this natural language: "No gateway router in a statement COI (community of interest) has a static route to a destination in a different COI" By community of interest Narain means groups of devices connected by VPNs. By using natural language, this approach may be appealing to practitioners, but may be hard to transform into precise configurations. Once a well-defined set of abstractions are developed to represent network configuration policy, a natural-language based description language could be built on top.

Elbadawi *et al.* [30] hint at high-level network policy by suggesting the need of "building a system that allows network operators to formalize requirements of Network Elements or service using high-level abstractions". We agree that high-level abstractions should be easily understood by network practitioners.

Hinrichs *et al.* [46] point to the need for "a high-level declarative language for expressing network-wide policies about a variety of different management tasks within a single, cohesive framework". They propose the FML language in the context of a software defined network which address the need for an authorisation language. We agree that the objective is well-founded. But in this thesis we do not address the domain of Software-Defined Networking.

Prakash *et al.* [86] describe how computer networks are governed by high-level policies. These policies are derived from network-wide requirements, and apply to many forms of networks, including "ISPs, enterprise, datacenter, campus, and home networks". They describe how "these network policies primarily relate to connectivity, security and performance, and dictate who can have access to what network resources". These authors are correct in focussing on connectivity should be the basis of a high-level abstraction. They also describe

> Simple and intuitive: Anecdotally, we found that many network admins and cloud tenants design their policies by drawing diagrams on whiteboards. We believe a policy abstraction must be as simple as drawing diagrams similar to Fig.1(a), yet expressive enough to capture their intents for diverse and dynamic SDN, cloud and NFV applications with sophisticated service chain requirements. (Prakash *et al.* [86])

While their work focusses on Software-Defined Networking, the high-level goal of making policy abstraction as simple as drawing diagrams could also be applied to conventional (non-SDN) network design and configuration.

In their Metaconfiguration concept, Matuska *et al.* [67] outline an approach that configures the "functions and attributes of the whole network, rather than functions and attributes of individual devices". They describe this approach as resolving issues with inconsistencies, as well as providing a basis for automation of routine tasks. They also describe that working at this higher level avoids redundant information repeated in multiple configurations, and of policy embodied in the "silently assumed relations in the network". By this the authors mean that information described in a configuration is not always explicit about the high-level policy it is realising. These observations tend to support the point of view that the high-level description of the network must be decoupled from the network device configurations.

Maltz *et al.* [66] also emphasises the need to escape from the dominance of low-level device configuration as the foundation for network descriptions:

> Ultimately, we believe that researchers should not need to work at the level of the configs themselves, but with a higher-level representation that abstracts away the idiosyncrasies of particular configuration languages and exposes the critical information. However, developing such a data model is an extremely difficult task, one that must be driven and validated by examples of how configurations are used in real networks. We see our work as the first logical stepping stone to the creation of a high-level representation of configuration data. (Maltz *et al.* [66])

In conclusion, in terms of the literature cited above an ideal high-level network configuration policy is a logical inventory rather than a physical inventory, incorporates decisions made by an architect about the top level organisation of the network, is expressed independently of the low-level configuration of routers, formalises requirements using high-level abstractions in a declarative language that its natural to network architects.

In the next part, we will review how a number of prominent tools have addressed the representation of High-Level Network Configuration Policy, and its transformation into lower-level artefacts.

2.5 existing tools for high-level configuration network policy

There are number of prominent tools that have been presented in research, that aim to show the way towards an improvement in network design methodology. In this section we discuss how each of these tools has approached the top-level abstraction of network policy, and where applicable, have defined its transformation into an intermediate format.

2.5.1 *PRESTO*

PRESTO [31] is a system to automate network device configuration, through the use of configlets — small configuration templates that can be composed together to create a device configuration. These configlets are written in a custom, general-purpose template language, and can extract configuration parameters from either an internal database or external systems. The PRESTO compiler combines these templates and information sources to generate configurations suitable for deployment to network devices. PRESTO has been designed to closely correspond to the low-level device configurations used on the network devices themselves, and generates complete device-native configurations [31]. This contrasts to a number of other configuration automation solutions, which only generate an appropriate configuration model: they don't then take the next step of generating the actual device configurations.

The users of PRESTO can be classified into two groups: domain experts, who setup the configuration services and options into active templates, and provisioners, who combine these templates with data from external sources to create the device configurations [31]. Configuration generation is initiated either by a network upgrade, or a customer order. The input provided in these requests is augmented with relevant supplementary external data, and an application-specific data model that corresponds to the target service or device. This is used to create a short-lived provisioning database, which roughly corresponds to the Device-Local configuration model of [89]. The provisioning generator uses the information in this provisioning database, together with the active templates to create complete device-native configurations [31].

These configlets are able to programmatically import other templates or extract external data. This allows composition and dynamic data importation, which provides flexibility to handle the wide variety of configuration scenarios encountered in network management. One design feature of the PRESTO system is decoupling the design specification and preparation of the configlets from the provisioning step of combining the configlets with external data to generate device configurations [31]. The abstraction used has allowed these tasks to be separated.

The authors describe the PRESTO configuration language as a language extension, build upon native configuration languages such as Cisco IOS. By adopting this approach, the configlet templates can leverage the designer's knowledge of existing configuration syntax and components. This could be further decomposed into areas of expertise, where one designer may be familiar with interface design, and another with BGP configuration. This is similar to modularity in programming languages, where the programming tasks can be allocated based on programmer skill-sets [31]. The use of parametrised templates reduces the learning curve for operators, increases the transparency of the configuration generation process, and simplifies the work required to add support for a new platform or service.

The data model used is a relational database, whose schema, representing the model of the Network Elements, is designed by the same domain experts who define the configlets [31]. This database serves as a higher level of abstraction to represent the network structure and configuration attributes.

While this approach offers flexibility, too much embedded logic can complicate the templates, especially if complex logic is embedded directly with vendor-specific low-level configuration syntax. This reduces readability, and makes it difficult to reason about design intent. There are also limitations to performing logic in this manner, especially for configuration of network-wide components, which require co-ordination across multiple device configurations.

Despite these limitations, the authors identify three key contributions of the configlet-based approach:

> - Breaking down tasks into smaller templates simplifies both creation and maintenance - It allows configuration design to be divided into areas of configuration expertise - Re-usability is encouraged, as configlets can be stored as libraries ([31])

Each of these three points has analogies to computer programming where modular components, made possible by an abstraction, are easier to program and maintain, allow for individual expertise, and encourage code re-use.

The data model is stored as a database schema, where router properties are stored in table fields. This allows router properties to be accessed as variables, and for peer router properties to be selected using database queries. This is used as the provisioning database, the short-lived intermediate database used as the information source for the configuration templates.

This database application dependent, as each network has its own requirements. Separate tables are used to store network properties such as routers, and their interfaces, with database keys used to link the two. The template language allows nesting and iteration, so that interfaces can be configured [31]:

```
[INT:SELECT (*) FROM (WAN_INTERFACE) WHERE
(WAN_INTERFACE.HOSTNAME=<ROUTER.HOSTNAME>)]
interface serial0/<INT.SLOT>/<INT.PORT>
bandwidth <INT.BANDWIDTH>
ip address <INT.IP> <INT.MASK>
! [/INT]
```

Listing 2.1: Example template from Enck *et al.* [31]

Conditional logic is also supported, which enables configuration components to be included based on boolean logic.

Data transformation is also an important component of configuration generated. This can occur in the case of IP addresses, where the network mask could either be expressed as /24 or as 255.255.255.0, depending on the device syntax. PRESTO supports such transformations, along with operations such as simple arithmetic, substrings, and regular expressions. These inline transformations can also complicate the readability of the template. Recognising this, the authors allow for blocks of code to be setup within a *label* scope, where the results are "hidden" from the configuration. This can be used to evaluate variables, ready to substitution into the templates.

Having the network model represented in a database however leads to complexity. An alternative is to introduce an intermediate model, which would allow logic operations to be performed in a standard programming language, with the results propagated into the provisioning database. The templates could then consist of simple decision blocks (such as the presence of absence of a service), or iteration (such as for configuring multiple interfaces). This would allow the design logic to be decoupled from the low-level configuration generation.

The templates used in PRESTO are unable to capture complex operational tasks, especially those involving dynamic changes to networks, and are unable to be composed from other templates [22]. Additionally PRESTO has limited support for checking network status, and verification against high-level constraints [20].

The contribution of PRESTO to to High-Level Network Configuration Policy is achieved through the data model or schema used in its relational database. However the schemas are not declarative of network policy and it is not clear how the schemas come into being. They would appear to need manual construction from a network policy document that is not part of the PRESTO tool. PRESTO papers make reference to drawing information from external sources but these are not well defined.

2.5.2 *Metaconfiguration*

Metaconfiguration [67] describes a network using metadata. It is used to describe a network in order to automate configuration generation. The metadata is broken into four categories: topology and addressing, routing, filtering/firewalling, and other features.

This data is described using XML, according to a schema defined using RelaxNG. These blocks of data are described as "templates". It should be noted that this definition of a template is different to that used as the final step of a configuration generation process, where templates are used to interpolate data into an output structure.

We briefly summarise these categories below. The Topology and Addressing data describes the network topology, and the IP addressing prefixes. The topology describes the groups of nodes in the network, and the links between them. Inheritance is used, so a property set on a group of nodes will apply to all nodes in that group. This helps to avoid redundancy. The IP addressing data describes how the IP address space is divided up, and is used in the system to allocate the specified IP address prefix to the links in the network. The routing properties mirror those of the Netopeer [17] data schema, which is an XML-based schema for describing router configuration data.

In our previous definition Section 2.4 of high-level network policy we make reference to a logical inventory rather than a physical inventory, decisions made by an architect about the top level organisation of the network, expressed independently of the low level configuration of routers, formalised requirements using high-level abstractions in a declarative language that its natural to network architects.

Metaconfiguration is a logical inventory, features decisions made about top level organisation expressed independently of low level router configuration. Thus Metaconfiguration has aspects of high-level network policy but by using XML, it lacks a declarative language that is natural to network architects.

2.5.3 *COOLAID*

COOLAID [21] uses a relational database schema as the basic definition language for the network. The choice of a database to represent the topology and policy intent of a network means that complicated queries are required to reason about the network. To make matters more unnatural for network architects, the policies are expressed in the Datalog rule based language. Much of the work on COOLAID relates to the incremental dynamic changes to an existing network, which solves a different problem to that of transforming High-Level Network Configuration Policy into individual device configurations.

2.5.4 *NCGuard*

NCGuard [104] aims to introduce software engineering practices to network engineering. It allows an operator to describe formal objects of their network, which are validated for network designs in the tool. It allows high-level network objectives to be composed of smaller, low-level network objectives. This models real-world configurations, where a high-level network objective is realised through the implementation of lower-level configuration statements across multiple network devices, and across multiple protocols. NCGuard breaks network design into two key components: a high-level description of the network configuration, and a set of design rules to be automatically tested. The network is represented in a high-level XML document. The high-level description allows network features to be described once, avoiding repetition, and removing the chance of repetition errors. Evaluation is performed on the XML model, using the design patterns described in the following validation section. To generate router configurations, the high-level XML model is transformed to a low-level device model, suitable for transformation into device configurations using XSLT stylesheets.

NCGuard achieves many of the requirements of a High-Level Network Configuration Policy. It has a declarative language which expresses the high-level network architecture, and this language is transformable at high-level. NCGuard also provides an extensive set of validation tools. NCGuard needs additional efforts to make its XML-based rules easily understood by the current network engineering workforce. As we will show in Section 2.6 and Section 2.7, NCGuard is less flexible when it comes to dealing with full variety of router platforms currently implemented in industry. Adding new routers involves both defining new transformations in XSLT, and domain knowledge. It is not certain that these two skills are widely shared in the engineering workforce.

NCGuard thus fulfils the role of a high-level network policy document, although we again note that XML is not a language naturally used and reasoned upon by network architects.

2.5.5 *Model Finding and Constraint Programming*

One approach to configuration is through model finding, where the network and requirements are described, using a formal language, with a solver finding a configuration model that satisfies the requirements. This allows high-level network requirements to be specified. A requirement solver performs translation of high-level requirements into a network model.

Using a declarative language to express requirements contrasts to a policy-based language, where the transformation from specification to configuration model must be programmed, with limited support for verification or error diagnosis [72]. Procedural languages, such as Python or Perl also require process of reasoning from high-level requirements to be programmed [72]. If a configuration platform is based on either a policy-based or procedural approach, then significant effort must be invested to provide a framework for this transformation and reasoning process.

While a requirement solver approach simplifies the transformation from high-level specification to a configuration model, the formal language used to express such requirements can become challenging to specify and read, especially by current network practitioners. Additionally the requirement solver itself can be a computationally intensive process, if all possible configuration combinations are to be enumerated. For instance, it required several hours, on a modern PC, to find the configuration for an 8-site VPN network example presented [72]. This is faster and reduces the likelihood of errors over a manual approach, but may present scalability problems for large networks. This approach preclude a real-time response to a dynamic response to a network event, such as a failure, outage, or even a planned upgrade.

[72] presents a Requirement Solver, which uses the Alloy logical system to perform the reasoning, which allows the specification of object types, their attributes, and first-order logic constraints. Allow performs the tasks of finding appropriate objects and attributes to satisfy the logic constraints, which capture high-level network design goals. This configuration synthesis provides an abstract network model of objects and their attributes. This can then be converted into device-specific syntax (this step, and the deployment to network devices is not covered in the paper). The author provides a number of typical requirements for the example of setting up a fault-tolerant wide area virtual private network (VPN), including the IP addressing, routing, access, and the VPN itself. These components that are difficult and error-prone to manually configure [72]. As well as configuration synthesis, the Requirement Solver approach can also be used for Requirement Verification. The authors provide an example discussing packet reachability.

ConfigAssure [74] synthesises network configurations from a configuration database of variables, and requirements, expressed as first-order logic constraints. These are processed using a Requirement Solver, generating a configuration model that is provably correct. One problem with synthesis using constrained logic is the complexity of model finding. This project improves on the Requirement Solver previous work by one of the authors [72].

Declarative language approaches are also well-suited to configuring existing networks, not just "greenfields" networks. A requirement solver approach allows the existing configuration to be expressed as a requirement, where the new configurations must preserve certain features. For instance physical links between routers must remain the same in the both the existing network, and the new model generated by the requirement solver [72].

Model finding constraint based programming is able to express high-level architecture and is strongly declarative. While declarative languages offer provably correct solutions, the syntax to express network requirements deviates from standard network operator implementation techniques. This may prove to be too steep a learning curve: unless the user has prior experience with formal languages, it may be time-consuming, or even error-prone to express network configuration requirements in a formal language. Using this approach it would be very difficult to derive alternative high-level views of the network using transformation.

It is important to make such languages as user friendly as possible, to encourage adoption. One approach mentioned in [72] is to use

Alloy from a traditional programming language, however the author does not provide details on the form that the configuration specification language would take.

2.5.6 *Configuration Semantic Model*

Configuration Semantic Model (CSM) [29] provides a middle-level abstraction to complement the NETCONF protocol [32]. NETCONF provides a vendor-independent interface to install manipulate and delete the configuration of network devices [32]. This allows a solution to work across a variety of network devices, but provides limited support for high-level design and management [29].

CSM provides both a global view of configuration, and co-ordination of configuring the inter-dependencies in network devices. It provides a multiple layer model, where each layer corresponds to an abstraction of a network management task, and provides a high-level specification language for network device, service, and system management. The framework allows multiple users to configure a network, with designated management roles to limit access. By using the NETCONF protocol, CSM can operate across network devices from a number of vendors. CSM has been implemented in Erlang as a system called NSConf.

CSM breaks automated configuration into two categories: static configuration, which is the initial configuration if a new service is provisioned, and dynamic configuration, where configuration changes are performed in response to network events such as outages from device failures. CSM focusses on automation of static configuration, and provides a bridge between high-level specification and low-level configurations — something that NETCONF does not provide [29].

CSM consists of a number of concepts, which is a set of attributes and characterisations. CSM models network topology using device and service concepts. The device concept represents a Network Element to be configured: this could be a physical device, or a logical element, such as an Autonomous System, and is represented using a name, a parent, and classifications. CSM uses a high-level language, Structured Configuration Language (SCL), a tree-structured language, which allows formal specification of configurations as a set of predicates. An example of the syntax to set the cost of the eth0 interface to 100 for three routers is reproduced from [29] below:

```
manage [r1, r2, r3]:ospf (.parent == as100) {
   config default {
       set interface {
         set name "eth0"; set metric 100;
} }}
```

Listing 2.2: Example template from Elbadawi *et al.* [29]

CSM describes the network architecture at a high-level. It is declarative, in that it uses a version of first-order logic predicates to express policies. There is no evidence provided that CSM can easily be transformed to produce different high-level view of the network.

Formal logical descriptions of a network are unlikely to be suitable for the current network engineering workforce. While the topology description is more accessible, the formalisation of requirements requires an understanding of formal logic, which is beyond the experience of most network operators.

2.5.7 *Netopeer*

Netopeer [62] Is a system that is used to configure routers in a manner that is independent of the individual devices or vendors. It uses an internal representation that is based on XML.

2.5.8 *YANG Description Models*

YANG is a data modelling language for the configuration and manipulation of state data. This is used to manage router configurations using the NETCONF protocol.

Layer 3 VPN Model

An example of the use of YANG is a data model describing a Layer 3 Provisioned VPN Service [63].

The interest in this model from the perspective of research into High-Level Network Configuration management is that it confirms the current interest (published December 2015) in the problem domain. Although the authors identify the need for transformation from a high-level declarative model and low-level configuration, they do not describe such a transformation. They also identify that the low-level configuration may be performed using the command-line interface (CLI).

This YANG model does not provide any implementation, as noted:

> This model is not a configuration model to be used directly on Network Elements. This model provides an abstracted view of the Layer 3 IPVPN service configuration components. It will be up to a management system to take this as an input and use specific configurations models to configure the different

> Network Elements to deliver the service. How configuration of Network Elements is done is out of scope of the document. (Litkowski *et al.* [63])

The authors provide a more detailed summary of the aims of their work as follows:

> A typical usage is to use this model as an input for an orchestration layer who will be responsible to translate it to orchestrated configuration of Network Elements who will be part of the service. The Network Elements can be routers, but also servers (like AAA), and not limited to these examples. The configuration of Network Elements MAY be done by CLI, or by NetConf/RestConf coupled with specific configuration YANG data models (BGP, VRF, BFD ...) or any other way. (Litkowski *et al.* [63])

We also note that this approach has components which are declarative. The authors create the concept of a site, which consists of a number of Network Elements. These sites can have roles, specified by the network architect. The paper then goes on to describe how such sites should be handled. This describes a design policy based on this high-level site abstraction. The handling of these sites, include establishing the VPNs, allocating resources, such as router identifiers, and considerations for IP addressing and routing protocols.

The idealised network management system outlined by the authors of these paper, therefore has a component that allows the user to describe their network topology at an abstract level, and specify the role of components of that network topology, such as sites. The network management system then follows the set of prescribed rules described in the document, to realise this policy, into a set of configuration attributes. These configuration attributes are then transferred to the individual network devices, using a variety of approaches, such as command-line interface, or newer methods such as NETCONF/ YANG.

While YANG brings benefits in approaching the modelling of network intent, it does not provide a framework to implement these models.

2.5.9 *Other Tools*

There are others tools which specific aspects of High-Level Network Configuration Policy. These focus on specific protocols such as BGP, which differs to the broader challenge of representing High-Level Network Configuration Policy.

Nettle

Nettle [107] is a tool for BGP configuration. This is a domain-specific embedded language for configuring BGP networks, embedded inside the Haskell programming language. Nettle allows an operator to describe BGP policies at a high-level, network-wide level of abstraction, that compiles down to existing platforms: the underlying network infrastructure does not need to change. By raising the level of abstraction, logical routing design is able to be separated from implementation and hardware specific configurations, allowing the operator to express policy in a platform independent manner.

As well as a policy description language, Nettle provides a compiler that translates BGP policies into router configurations. An implementation is provided that supports the XORP routing platform.

Nettle is embedded within Haskell, a functional programming language. This allows greater expressiveness, but requires familiarity with Haskell syntax: like other less-common languages, this may prove a significant barrier for operator adoption. Additionally, Nettle does not provide support for policy verification. Finally, Nettle only supports BGP policy. While BGP policy is a complex configuration task, network configuration involves multiple protocols, notably IGPs such as OSPF and IS-IS, which Nettle does not currently support. To be a viable part of a management platform, the policy expressed using the Nettle description language would need to be integrated with BGP verification tools, and inter-operate with tools to configure other services. Despite these shortcomings, BGP policy is an important part of network configuration, and Nettle offers many insights into BGP policy description languages.

Since Nettle focuses on configuration of the BGP protocol, and given this restriction it realises many of the desired properties of a High-Level Network Configuration Policy. However it uses Haskell, which is not widely used by network practitioners, and cannot be transformed to other high-level Network Views, as it only deals with one protocol.

Since it only has an implementation for the XORP platform it lacks vendor-independence. Nettle thus is a specific tool to configure BGP Policy for the XORP platform. This differs to the broader challenge of capturing High-Level Network Configuration Policy.

Boehm

Boehm *et al.* [10] present a system for an autonomous system to specify its routing policy for the BGP protocol, and then generate the relevant device configurations. To do this they use a series of policy "atoms", which are combined with a "network specification". Each entry is represented in XML. Like Nettle, the focus of this work is on BGP policy specification, rather than a generalised approach to network design and configuration.

Propane

Propane [8] is a language and a compiler to translate network-wide objectives into low-level device configurations. It also has an emphasis on configuration of the BGP routing protocol. High-level objectives are expressed using regular expressions to specify network paths. These are then translated into graph-based intermediate representations, which are used to calculate paths through the network topology that match the user constraints.

This work also focusses on the area of BGP policy, rather than the broader task of network configuration. By narrowing the domain of focus, a more specific expression language can be developed, which allows deeper analysis than a generalised solution. This addresses a different problem space than the compilation of generic high-level network configuration designs into low-level configuration, and can be viewed as a complementary approach to the work presented in this thesis.

2.5.10 *Conclusion*

As a result of reviewing the various proposals for existing tools, we note that the existing tools either do not provide a comprehensive high-level declarative description language of the network, or do so in a language that is not readily accessible to current network practitioners.

Current tool proposals contain many creative ideas for improving the state of the art of the network design experience, and include pointers to methods which could be adopted. However none of the currently existing tools have been able to draw these concepts together into a simple coherent system.

In the next section we explore research contributions that address the transformation of High-Level Network Configuration Policy into Intermediate Forms.

2.6 INTERMEDIATE NETWORK REPRESENTATION

In this section we explore research that suggests the possibility of intermediate formats for Network Configuration, that are lower level in abstraction than the High-Level Network Configuration Management Policy representation, but higher level than the individual device-specific configurations.

An analogy exists between such Intermediate Forms for network configuration and the intermediate languages used in programming language compiler technologies. Where possible we highlight research contributions into the transformations from the high-level description of the network into these intermediate forms.

This intermediate step concerns the "Network-Wide Configuration Data" section of Figure 2.3. Intermediate Network Forms are the data structure used at this step. The input to the intermediate stage of the compilation is the high-level network policy artefact. This corresponds to the "Configuration Management Data (High-Level Policies)" of Figure 2.3.

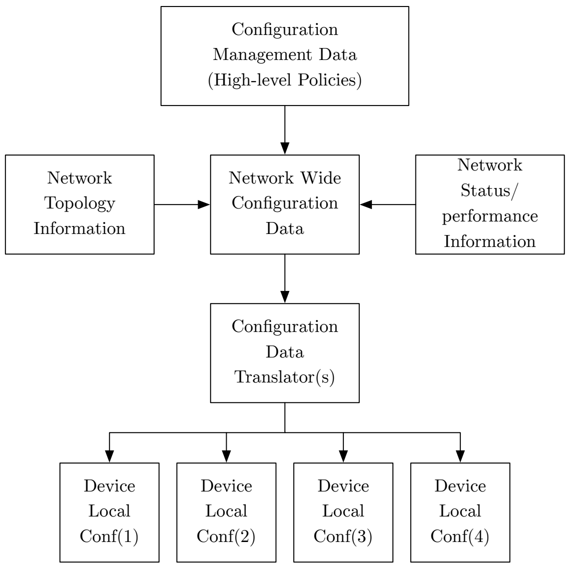

Configuration Management Data (High-level Policies)
Network Topology Information
Network Wide Configuration Data
Network Status/ performance Information
Configuration Data Translator(s)
Device Local Conf(1)
Device Local Conf(2)
Device Local Conf(3)
Device Local Conf(4)

Figure 2.3: Reproduction of Figure 2, *Proposed model for configuring network devices*, from Sanchez *et al.* [89]

This section is organised as follows. We first make some general comments about what constitutes the intermediate network form. We then review existing research tools, with a focus on how they propose to handle the intermediate forms, and relevant transformations.

2.6.1 *General Comments about Intermediate Forms*

For many approaches, the Intermediate Forms are not explicitly separated.

There are many different methods through which Intermediate Forms can be represented. Forms that have been used by the tools referred to in this section include:

- XML [104]
- an SQL database [9] [10] [21] [31]
- Comma Separated Value text files [91] [92]
- The YAML markup language [96]
- Perl hash tables [35]
- internal formats used in constraint-based programming [72].

In addition to the wide variability of languages used for representation, there is no consistent approach to the underlying schema and associated transformations of intermediate forms.

2.6.2 *PRESTO*

PRESTO [31] is a system to automate network device configuration, through the use of configlets — small configuration templates that can be composed together to create a device configuration. Despite their name, PRESTO templates are general and perform the role of both intermediate representation and code generation functions at the same time.

The PRESTO schemas are reflective of network policy, but do not directly represent it. They can thus be described as an intermediate format. In terms of the structure shown in Figure 2.3, PRESTO emphasises the "Configuration Data Translator", as compared to the "Network Wide Configuration Data". Rather than performing transformations on the Intermediate Representation, PRESTO uses low-level configlets to directly generate device configurations. This does not address the analysis and reasoning of the Intermediate Representations.

2.6.3 *Metaconfiguration*

Metaconfiguration [67] describes a network using metadata. It is used to describe a network in order to automate configuration generation. As previously noted, this metadata is broken into four categories: topology and addressing, routing, filtering/firewalling, and other features. We interpret these categories to be representations of an intermediate form. This is one of the positive contribution that Metaconfiguration has made. However, these intermediate forms in Metaconfiguration are expressed as XML documents, which are tree structured. This is at odds with the natural structuring of networks using graphs. We have previously noted that Metaconfiguration lacks an over-arching declarative description of the network, at a level of abstraction above these categories. This limits the reasoning and transformation of the intermediate forms.

2.6.4 *COOLAID*

COOLAID [21] does not reveal how it converts the database representing the high-level network policy into device configurations. The authors hint at progressive refinement of a minimally configured set of routers but the details are now explained. Hence it is not possible to compare the intermediate or low level steps of COOLAID with the structure suggested in Figure 2.3, specifically as it addresses the management of dynamic changes to run-time networks.

2.6.5 *NCGuard*

NCGuard [104] aims to introduce software engineering practices to network engineering. It allows an operator to describe formal objects of their network, which are validated for network designs in the tool.

NCGuard compiles its high-level representation into manufacturers configurations using a two step process with a single intermediate representation. NCGuard has separate XML-based descriptions of the network configuration, and as an intermediate format. This allows the use of XSLT to define transformations. A major contribution of NCGuard is the recognition of the need for a formal intermediate form in the process of developing a network configuration tool. The deficiency of XML as a language for describing a graph-based network in a tree-based structure also applies to NCGuard.

2.6.6 *Model Finding and Constraint Programming*

Model finding [72] uses first-order logic to define high-level network policy requirements. The authors claim that this enables high-level network tasks to be performed including synthesis. By synthesis the authors really mean composing partial formal models of parts of the network into a more complete model of the network. However the authors note that the SAT solver used to service the synthesised device configurations is NP-complete and whether a solution is found may depend on the way the original requirements are written.

The authors have spend some effort in explaining how to write requirements that can be synthesised. In common with all first-order logic formalisms there is no way to be sure that the requirements are complete. The authors do not attempt "synthesise" right down to device configurations and thus have no way to cope with variations between manufacturers devices. Despite their power, formal methods are not the natural language of network architects. The authors acknowledge this to some extent because they often have to use topology diagrams to explain the construction of these requirements.

If model finding and constraint programming have an intermediate form, it is hidden inside their tool framework. This is reflected in the fact that these approaches do not allow any synthesis of device configurations.

2.6.7 *Configuration Semantic Model*

Configuration Semantic Model (CSM) [29] expresses high-level network policies using formal logic. CSM has no clearly defined intermediate form. This reflects the fact that CSM is a one-step translation from a high-level model to NETCONF. The transformations use algorithms based on Binary Decision Diagrams. This is not a natural approach used by network operators, and would require translation steps to be accessible to network operators.

2.6.8 *Other Approaches*

Industry

Industry approaches tend to be pragmatic and built from the bottom up. One example of an approach used in industry is Ansible, as presented by Lewis *et al.* [61]. This approach describes using the Ansible framework to automatically generate configurations using templates to be pushed out to the target devices.

In the presentation by Lewis *et al.*, some key aspects of Ansible are outlined. These include:

- Playbooks, which specify a series of actions to perform, including the generation of configurations using templates. Sets of actions can be broken up into separate play books.
- Inventory, which is a set of specific data They can be passed to a playbook.
- Roles, which allow breaking functions of the network into common categories. The example provided is a set of roles for Routers is set of roles switches.

We note that in their presentation, the authors describe that there are lots of different ways to structure the intermediate data which is used for the configurations generation step. They note that one approach could be using roles, another using inventory, or another using global group and host variables. The recommendation is that the best approach depends on the individual situation.

We note there may not be a common industry approach to the intermediate representation of the network.

Oswalt emphases the importance of intermediate forms:

> So what's the missing link with all of this material? It comes back around to the DevOps methodology. Configurations should not only be created as templates, but their very existence should be version controlled and be the point of authority. Having the data structured separately [i.e an intermediate form] allows for all sorts of external systems to participate in the workflow. From provisioning new systems to modifying existing ones, the data can be independently verified. Keeping both the templates and data in source control also create unparalleled accountability. The process of tracking changes over time becomes trivial — and the dreaded middle of the night troubleshooting session becomes that much easier. (Oswalt [81])

Nettle

Nettle [107] for BGP configuration. This is a domain-specific embedded language for configuring BGP networks, embedded inside the Haskell programming language. Nettle allows an operator to describe BGP policies at a high-level, network-wide level of abstraction, that compiles down to existing platforms: the underlying network infrastructure does not need to change. By raising the level of abstraction, logical routing design is able to be separated from implementation and hardware specific configurations, allowing the operator to express policy in a platform independent manner.

Boehm

Boehm *et al.* [10] does not provide details of an intermediate model between the XML specifications and the "configurator" to generate the route configlets, by briefly describing the use of an SQL database.

Automated Provisioning of BGP Customers

In their work on Automated Provisioning of BGP Customers, Gottlieb *et al.* [41] describe an approach for BGP configuration based on SQL tables for the high-level specification and intermediate models. They make use of SQL queries to extract the information from these tables to build the individual device configurations using templates. We note the complexity of such an approach for the typical network practitioner. An example of such an SQL query is shown in Listing 2.3.

```
SELECT Customer.AS_number,
 Customer.Description,
 Customer.Geographic_location,
 Inventory.Router,
 Inventory.Location,
 Link.Interface,
 SUBSTR(IP_prefixes.prefix, 1, INSTR(IP_prefixes.prefix, '.', 1, 3) ---1)
      || '.' ||
  TO_CHAR(TO_NUMBER(SUBSTR(IP_prefixes.prefix,
   INSTR(IP_prefixes.prefix, '.', 1, 3) + 1)) + 1),
 Link.Packet_filter_ID,
 DECODE(BGP_session.Loopback_IP_address, ' ',
  SUBSTR(IP_prefixes.prefix, 1, INSTR(IP_prefixes.prefix, '.', 1, 3) --
      -1) || '.' ||
  TO_CHAR(TO_NUMBER(SUBSTR(IP_prefixes.prefix,
   INSTR(IP_prefixes.prefix, '.', 1, 3) + 1)) + 2),
  BGP_session.Loopback_IP_address),
 BGP_session.Route_filter_ID,
 BGP_session.Inbound_route_map,
 BGP_session.Outbound_route_map,
 BGP_session.#_of_intermediate_devices,
 DECODE(BGP_session.#_of_intermediate_devices, '0', ' ', 'Loopback0'),
  BGP_session.Default_originate,
  BGP_session.BGP_keep_alive_timer,
  BGP_session.BGP_holder_timer
FROM BGP_session, IP_prefixes, Customer, Inventory, Link, Assignment
WHERE Customer.AS_number = '6431'
  AND Customer.Customer_ID = Assignment.Customer_ID
  AND Assignment.Session_ID = BGP_session.Session_ID
  AND Assignment.Router = Inventory.Router
  AND Assignment.Router = Link.Router
  AND Assignment.Interface = Link.Interface
  AND Link.Prefix_ID = IP_Prefixes.Prefix_ID
```

Listing 2.3: Example of a template from Gottlieb *et al.* [41]

2.6.9 *Graph Representations*

An important consideration at the intermediate level is what intermediate representation or form should be used. Surprisingly, the natural intermediate form for networking, has not been widely investigated as a possible intermediate form in tools. In this section we review the graph theory that is relevant to its use in network, in particular in intermediate representations.

Graphs have previously been used in specific solutions for network configuration, including in rcc [34], representing iBGP relationships, where nodes represented routers in a graph, and edges the sessions, and in shadowed subgraphs. Lauer *et al.* point out that:

> People naturally treat networks as graphs, whether they are writing software to implement network-based services, provisioning VPNs, or trying to figure out how to implement BGP policies. (Lauer *et al.* [59])

Zegura *et al.* support this point of view:

> Graphs are commonly used to model the structure of internetworks, for the study of problems ranging from routing to resource reservation. (Zegura *et al.* [112])

McClurg *et al.* have also emphasised the use of graphs:

> a network can be thought of as a graph with switches as nodes and links as edges (McClurg *et al.* [68])

Finally, Sung *et al.* illustrate how useful graphs can be in the abstraction of network topology:

> Network topology: We abstract the network topology as a graph G. H denotes the the set of end-hosts. S denotes the set of switches, i.e., devices that are capable of performing layer-2 switching. We let R denote the set of routers, i.e., the subset of switches that are also capable of performing layer-3 routing. E denotes the set of edges. Two vertices are connected by an edge in if they are physically connected in the underlying network. (Sung *et al.* [98])

The use of graphs in industry is discussed by Caldwell *et al.* [16], in their summary of the non-disclosed EDGE system. In their work they discuss how graphs can be used for visualising a network in particular grouping by various attributes such as the AS number the BGP, or the area for OSPF. They also discuss how graph compression agreements can be used to reduce the amount of information presented by grouping by common features such as geographical co-located devices in a Point-of-Presence (PoP).

All of the above quotations refer to tools or approaches that target a specific subdomain of networking. They have not applied graphs to the general problem of High-Level Network Configuration Management.

In the talk "The Future of Networking, and the Past of Protocols" given at the Open Networking Summit in 2011, Shenker [95] discusses the fundamental ideas behind Software Defined Networking. He notes that SDN requires three key abstractions: a Distributed State Abstraction, a Specification Abstraction, and a Forwarding Abstraction. As our work focusses on existing network equipment and protocols, we cannot redefine the forwarding abstraction. However Shenker notes that the Distributed State Abstraction, an annotated network graph is a natural choice.

2.6.10 *Conclusion*

From our discussion and review above, we can identify that whilst intermediate formats are crucial to a systematic structuring of network configuration tools, they have not been treated as first-class entities in most of the published tools. In fact, current tools could be as contributing to fragmentation of the intermediate forms, as suggesting approaches in many different formats, such as XML, CSV, and SQL databases.

Graphs have been identified as a natural fit for intermediate forms, however there are no comprehensive tools that use graphs in this way. A graph-based tool would offer many advantages in presenting a natural abstraction and leverage the formality of graph theory for the transformation steps that are used in the network design process.

2.7 LOW-LEVEL DEVICE CONFIGURATION STATE

In this section we discuss the literature relating to the lowest level of the IETF workflow. This is the "Configuration Data Translator(s)" component of Figure 2.1 to produce the Device Local Configurations. This section is structured as follows. We first review templates which are widely used in practice, and have been adopted by many of the research tools. Then we describe how transformations have been used to translate templates into device configurations.

2.7.1 *Templates*

Many network engineers are focused on the final commands that must be entered into a particular router to achieve the intent of the high-level network configuration. However there are many issues surround this command line interface that lead to network failures and misconfigurations. These have been previously discussed in the motivation chapter. The reasons for difficulties with the final commands is that they are device specific, often have inconsistent, confusing, and position-dependent syntax. A commonly adopted strategy to reduce the difficulty of creating commands is templating [9].

Through a transformation templates can be used to create the device-specific commands for router configuration. The nature and complexity of the template itself and inputs to the templating process varies widely amongst published tools. For example in PRESTO a template is a script which accesses a database of network inventory, performs filtering, processing, and verification operations before generating the final output commands. A sample PRESTO template is shown in Listing 2.4.

```
%% Configure Back-to-Back Interface to Peer Router
[INCLUDE FROM (B2B) WHERE (B2B.TYPE=<ROUTER.B2B_TYPE>)] %% Define the BGP
     configuration
router bgp <ROUTER.LOCAL_ASN>
network <ROUTER.LOOPBACK0> mask 255.255.255.255
[PEER:SELECT FROM (ROUTER) WHERE (ROUTER.HOSTNAME=<ROUTER.PEER>)]
%% Ensure the peer is in the same network
[EVAL B2B_CHECK noprint]
[COND WRONGNET ("<ROUTER.B2BNET:computeOffsetMaskIP(<ROUTER.B2B_IP>,0)>"
    \
ne "<PEER.B2BNET:computeOffsetMaskIP(<PEER.B2B_IP>,0)>" ) ] <ROUTER.
    WARNING:templateWarning(<ROUTER.HOSTNAME> and <PEER.HOSTNAME>
    different B2B Net)> [/WRONGNET]
[/B2B_CHECK]
network <PEER.NETIP:computeOffsetMaskIP(<PEER.B2B_IP>,0)> mask
    255.255.255.252 neighbor <PEER.B2B_IP> remote-as <ROUTER.LOCAL_ASN>
neighbor <PEER.B2B_IP> next-hop-self
[/PEER]
[WAN:SELECT FROM (WAN) WHERE (WAN.HOSTNAME=<ROUTER.HOSTNAME>)]
```

```
network <WAN.NETIP:computeOffsetMaskIP(<WAN.IP>,0)> mask <WAN.MASK:
    computeMask(<WAN.IP>)> %% The gateway is the second IP in the subnet
     (for this example)
neighbor <WAN.GW:computeOffsetMaskIP(<WAN.IP>,1)> remote-as <WAN.REMOTE_
    ASN> [/WAN]
 no auto-summary
!
```

Listing 2.4: Example of a template from PRESTO, taken from Enck *et al.* [31]

There is a very wide diversity in the way that templates have been conceived and implemented. Google uses a simple templating process. "Configurations ["Commands"] are templates with [just] variable substitution" [39]. Microsoft have a concept of "configlets" which appear to be midway in complexity between the PRESTO and Google approach [90]. NTT's approach is also intermediate between these two extremes [70]

Arnoldus *et al.* [4] note that the literature does not provide a formal definition of a template. They cite an informal definition from, which defines a template as "an output document with embedded actions which are evaluated when rendering the template".

Template-based solutions have been presented at industry conferences, such as the North American Network Operators Group (NANOG). In these presentations templates are designated for code generation within a Development Operations (DevOps) framework such as Ansible [3]. The Netomata Configuration Generation system, an open-source framework, combines a network inventory model and templates to generate device and service configurations [18]. Templates are also used in industry to build XML configurations for NETCONF, such as in the Kipper project [106]. Thus is could be argued that templates are the only abstract concept that both researchers and practitioners in the field of network configuration management agree on. What is not always agreed on is the level of abstraction and complexity for which the template is designed.

Templates have been widely studied in the computer science domain. Parr [82] has promoted principles that which aim to enhance the separation of concerns in templating:

> Unrestricted templates are extremely powerful, but there is a direct relationship between a template's power and its ability to entangle model and view (Parr [82])

Parr has listed five constraints that he considers are essential to maintaining separation of concerns in templates. In Parr's explanation of constraints, the view is the template, and the model is a data source, from which the template draws information.

1. the view (template) cannot modify the model either by directly altering model data objects or by invoking methods on the model that cause side-effects. That is, a template can access data from the model and invoke methods, but such references must be side-effect free.
2. the view (template) cannot perform computations upon dependent data values because the computations may change in the future and they should be neatly encapsulated in the model in any case.
3. the view (template) cannot compare dependent data values, but can test the properties of data such as presence/absence or length of a multi-valued data value.
4. the view (template) cannot make data type assumptions.
5. data from the model must not contain display or layout information. (Parr [82])

In the context of networking, the model could be viewed as an inventory of network assets. To illustrate these constraints in the context of networking will now the use of templates in PRESTO in relation to these constraints.

PRESTO performs complex transformations in the template on data from the (network) model. This breaches constraint number 2. This is fundamental to the construction of PRESTO. The approach used by NCGuard [104] uses XSLT templates to generate device command configurations, does not breach these constraints.

In summary, templates have been shown to be a widely useful approach by both practitioners and researchers. However to better promote a systematic approach to High-Level Network Configuration Management, it would be advisable for tool designers to pay closer attention to the way in which templates are used, in particular to ensure that they contribute to the separation of concerns between intermediate forms and device configurations.

2.7.2 *Transformation*

The final step in the low-level device configurations process is transformation of intermediate forms and templates to generate device configurations. In this section we review the different approaches that have been suggested for this step.

If templates are defined according to the principles outlined by Parr, then the transformation step could be considered to become almost trivial. The major challenge for the transformation stage is the wide diversity and irregularity of device configuration syntax and semantics. Therefore it is important that the transformation association with template itself is kept simple, as it is unlikely that the final code generation step will ever be very simple.

Arnoldus *et al.* [4] provide an overview of a typical transformation, shown below as figure Figure 2.4, where input data refers to some type of intermediate form.

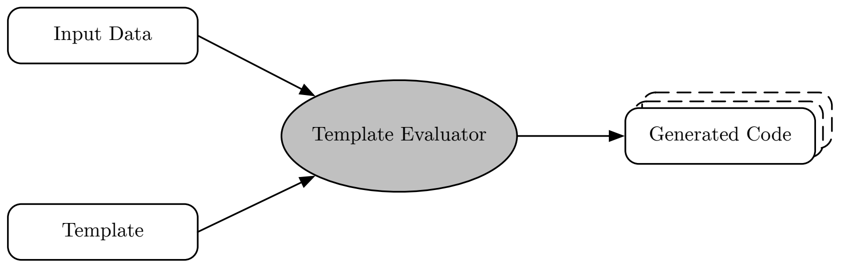


Figure 2.4: Reproduction of Figure 1.12 *Template based generator* from [4]

With NCGuard, Vanbever *et al.* [104] use XSLT style sheets to transform an XML model of the device configurations into. The relative complexity of XSLT could be a concern for adoption by practitioners.

By increasing the rigour on the intermediate forms, it should be possible to reduce the pressure on the templating system to do too many different and disparate tasks. This frees the templating transformation to be mostly syntactic. No one in the literature has explored this direction in detail.

DFA/Petri-Net

The DFA/Petri-Net approach is intended to formally define the workflow of network configuration in response to changes in the configuration. This makes no contribution to high-level architecture. This follows on from the Business Process/Method of Procedure approach.

2.7.3 *Conclusion*

In this section we have reviewed the lower levels of a proposed compilation hierarchy between High Level Network Configuration Policy and device configurations.

We have shown that templating is one of the most widely accepted approaches between researchers and practice. However we have noted that because of the deficiencies of intermediate forms used by many tools, much complexity has been transferred to the templating step. When added to the complexity of syntax and semantics of the device configuration languages, it has made many approaches unwieldy. No tool has effectively proposed a separation of concerns between a well-constructed intermediate form and a simplified template. It seems likely that this approach could simplify both the intermediate form and the template. This would leave the templating step to follow well-established computer science principles.

2.8 SUMMARY

Based on our survey of related work, we identify the following research gaps:

1. There is insufficient separation of concerns between low level configurations and high-level policy design objectives.
2. Whilst networks are inherently graph based the internal representations of current research and industry approaches are not strongly based on graph theory.
3. Where compilation is attempted from high-level policy to low level configuration the process is not well structured. Well established principles derived from other domains where compilation is required are not widely adopted in the networking domain.
4. The languages used for the input of policy design objectives is not expressed in language that would be readily adopted by the current practitioners
5. Many tools published do not provide a comprehensive solution for the implementation of high-level network configuration policy.

We will address these gaps in the next chapter, where we will present our Research Methodology.

2.9 NETWORK TESTBEDS

In this section we briefly review the infrastructure that has been created to enable networking tools and concepts to be prototyped and evaluated in non-production environments. There are three ways to construct such testbeds. These are simulation, emulation, and hardware testbeds. Here we provide a brief overview of some of the well-known testbeds.

By hardware testbeds, we mean using physical routers as the basis for the experiments. Three examples of hardware testbeds are Routerfarm [1], shadow networks [19], PlanetLab [23]. The main appeal of a hardware testbed is their fidelity. The main disadvantage is the cost and likely access restrictions.

An alternative is simulation. Simulation typically focuses on one specific aspect of the routing process, such as C-BGP [88]. However simulation usually avoids the detail of individual device configurations, yet these configurations are the endpoint for many tools. There are many network simulation tools in existence.

The third alternative is emulation, where the operating system of a physical network device is run in a virtual environment. This ensures device-level realism, but allows for a more flexible deployment in terms of scale and wiring of the devices. Because this approach uses the same operating system as production devices, it also uses the same configuration syntax. This affords a useful environment to test configuration generation tools. Examples of emulations include Emulab [45], Mininet [58], Junosphere [50], VIRL [79] and Netkit [84].

Mininet is focussed on OpenFlow-based SDN experimentation. Junosphere is a commercial product from Juniper. Junosphere has been used for research [7]. VIRL (Virtual Internet Routing Lab) is comparable product from Cisco. Netkit is an open-source project.

Netkit is an attractive choice as it is open-source, and is respected in the research community as it has been used for both teaching and research. It is based on the open-source Quagga [87] router. Quagga is a well-established project that provides functionality comparable to commercial router operating systems. A disadvantage of Netkit is the lack of diversity of use of commercial operating systems such as those from vendors such as Cisco or Juniper. A comprehensive evaluation of such a network configuration generation tool would require evaluation against commercial network operating systems.

2.10 CONCLUSION

In this chapter we have reviewed the research relating to the process of transforming High-Level Network Configuration Policy descriptions into low-level device configurations.

When considering High-Level Network Configuration Policy descriptions, we noted that no tool provides a declarative description language for policy which is also readily accessible to current computer network professionals.

Most tools propose some sort of new abstractions, however the systematic layering of these abstractions is not well-handled by most of the proposals in the literature. In particular, there has been insufficient attention applied to formalising the intermediate forms. This has had serious consequences for many of the tools proposed in the literature, and has led to fragmentation of the abstractions used.

For example, some intermediate forms have attempted to express high-level policy, and other intermediate forms have lacked generality, forcing added complexity into the templating step.

Whilst graphs are a natural fit for networking, they have not been seriously considered as the basis for a generalised intermediate form.

While templating is the only methodology widely accepted by both researchers and practitioners, in many cases deficiencies in other parts of the tools have meant that too much has been asked of the templating step, complicating its transformations.

In summary, whilst current tool proposals contain many interesting ideas for improving the way network policy is transformed into device configurations, no current proposal has drawn these concepts together in a compelling way, that may have real impact on the way that data networks are designed and managed.

In the next chapter we present a research methodology which aims to address this deficiency.

3

RESEARCH METHODOLOGY AND OVERVIEW

3.1 INTRODUCTION

In the previous chapter it was shown that there are significant gaps in the research literature and industry practice reading the expression and use of high-level network configuration policy to systematically generate low-level device configurations. A summary of these gaps are stated below and in this thesis research contributions are made to address them.

1. There is insufficient separation of concerns between low level configurations and high-level policy design objectives.
2. Whilst networks are inherently graph based the internal representations of current research and industry approaches are not strongly based on graph theory.
3. Where compilation is attempted from high-level policy to low level configuration the process is not well structured. Well established principles derived from other domains where compilation is required are not widely adopted in the networking domain.
4. The languages used for the input of policy design objectives is not expressed in language that would be readily adopted by the current practitioners
5. Many tools published do not provide a comprehensive solution for the implementation of high-level network configuration policy.

3.2 RESEARCH QUESTIONS

The following research questions are based on the work of Shaw [94], which presents a characterisation of research strategies for conducting research in software engineering. These identify a series of types of questions, and a series of validation methodologies that can be used to validate the questions. We use these categorisations to classify our question type, strategy, and validation methods below.

The research questions addressed in this thesis are follows:

Question 1	Is there a declarative representation of High-Level Network Policy that is also likely to be widely adoptable by current network practitioners?
Question Type	Feasibility
Strategy	Technique
Validation	Implementation

Question 2	What is a better intermediate representation of network configuration that is based on graph theory, supports a well structured compilation process and provides clear separation of concerns?
Question Type	What is a better way to do X?
Strategy	System
Validation	Evaluation

Question 3	How can declarative High-Level Network Policy descriptions be transformed into a graph-based intermediate format using a compiler that is extensible in terms of new types of policies and new protocols.
Question Type	What is a better way to do X?
Strategy	System
Validation	Evaluation

Question 4	How can the configurations for a diversity of network devices be generated systematically from a graph based intermediate representation?
Question Type	Method means. How can we do X?
Strategy	Technique
Validation	Implementation

Question 5	What are the scalability and extensibility characteristics for the compilation of High-Level Network Configuration Policy to device configurations in terms of network size and diversity of network protocols and target devices?
Question Type	Characterisation
Strategy	Empirical
Validation	Experience/Evaluation

3.3 RESEARCH METHODOLOGY

For each research question we have identified a strategy for answering it and a means of validating the result.

Question 1: The strategy for answering this question is to construct a system which incorporates the new proposed declarative high-level policy representation. The validation involves making a tool based on the proposed representation widely available to industry practitioners and observe adoption rates.

Question 2: The strategy for answering this question is to construct a system (tool) which incorporates the proposed intermediate format. The validation will be to establish if the format can support a wide range of high level network policies and low level devices at scale.

Question 3: The strategy for answering this question is to construct an instance of a compiler that performs the transformation. The validation will be a successful implementation.

Question 4: The strategy for answering this question is to construct an instance of a code generator that is capable of configuring a wide range of network devices. The validation is to be obtained through emulation of significant sized networks with a significant range of devices from a significant range of protocol expressed in the intermediate representation.

Question 5: The strategy for answering this question is to use the complete tool instance incorporating all the design concepts and abstraction to build a range of networks as case studies. The case studies would encompass scale in both number of network nodes and range of policies and protocols. The validation is carried out by checking for correct packet routing and inspection of each devices routing tables. The tool has also been validated through testing performed by a large cohort of practitioners who will provide informal experience reports and testimonials.

3.4 SCOPE OF THESIS

This thesis is concerned with network configuration, with an emphasis on a methdology to express high-level network-wide configuration requirements which can then be transformed into low-level device configurations.

One of the main contributions of the thesis is in developing these generalised abstractions and transformations, which can be used as building blocks for common network configuration tasks. While we provide a number of examples of network configuration tasks, the contribution is in the abstractions and transformations rather than the artefact produced. In this thesis we will discuss the process of generating device configurations for a greenfields network. In the future work section we discuss how the work of this thesis could be further developed with existing research, to address an existing brownfields network, and to handle on-going run-time management. The automated configuration of a greenfields network is a precursor to on-going network management of an existing network.

The methodology developed in this thesis can be implemented using a modular toolchain. While our toolchain incorporates the generation of low-level device configurations using an intermediate representation, the modular architecture allows for other approaches to be substituted. This allows our high-level approach to specification and transformation to be incorporated with existing research for generation device-level configurations. Our approach could be used to model the inter-device configuration parameters, which are then passed to the intra-device configuration tools. In this thesis we use a case-study of a Small Internet topology, taken from a teaching scenario, which reflects many of the configuration tasks involved in real-world network configuration. This is used as an illustrative example. We expand this example with a series of case studies, to demonstrate the flexibility of our approach. Finally, in the Future

Work chapter we present a series of possible future directions to further develop our approach.

3.5 DISCUSSION OF CONFIGURATION GENERATION PROCESS

We will aim to build the toolchain shown in Figure 3.1. To do this we first need to build the theory for each of the components of the toolchain. From there we can construct a reference implementation and use this for our experimentation process.

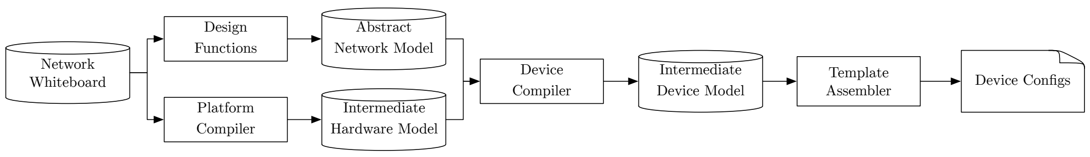


Figure 3.1: Flowchart of toolchain for configuration generation

We will aim to reproduce the Netkit topology described in [84] which provides a realistic network topology based on a number of inter-connected networks. We will reproduce this for the Netkit testbed platform, allowing us to verify the correctness of the generated configurations. Our approach should provide the network and device abstractions described by Shenker [95], and have the property that $f(model) = configurations$ described in industry talks such as Google [39] or Shenker [95]. We would like our approach to fit within the workflow of network practitioners, providing transparency into the internal operations, such as by visualisation of the network model, and presentation of the device model in a format that is readily understandable by a network practitioner.

We will use the general approach outlined in RFC 3139 [89] and reproduced in Figure 3.2. We will seek to combine this with language compiler theory, in particular its use of intermediate representations and transformations of these intermediate representations.

3.5.1 *Abstractions*

We first need to develop the appropriate abstractions to represent the input specification and the various network topologies. We denote these as the Network Whiteboard and the Abstract Network Model. The Abstract Network Model consists of Network Views which represent network topologies, such as for routing protocol designs. These are shown in Figure 3.3.

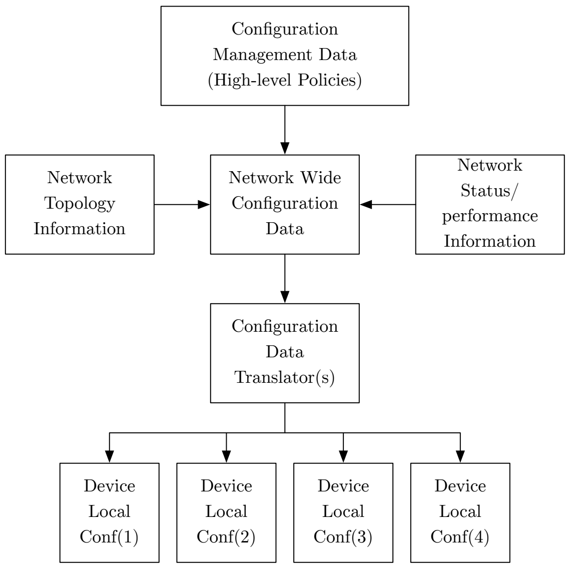

Configuration Management Data (High-level Policies)
Network Topology Information
Network Wide Configuration Data
Network Status/ performance Information
Configuration Data Translator(s)
Device Local Conf(1)
Device Local Conf(2)
Device Local Conf(3)
Device Local Conf(4)

Figure 3.2: Reproduction of Figure 2, *Proposed model for configuring network devices*, from Sanchez *et al.* [89]

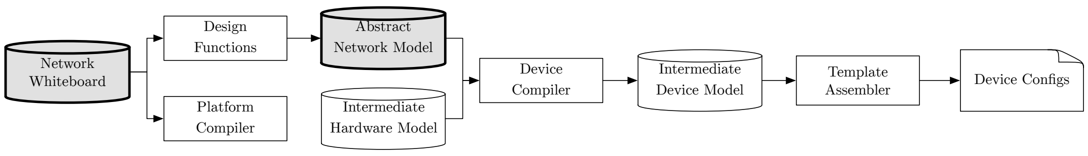


Figure 3.3: Flowchart of configuration generation toolchain, showing the role of the Network Whiteboard and Abstract Network Model abstractions

We would like the input specification to be as simple as the common practice of drawing policy requirements on whiteboards as described by Prakash *et al.* [86].

As discussed in the literature survey, there are hints in literature that graphs are a natural abstraction. However we will need to determine the appropriate level of representation: what do the nodes and edges represent? Graphs consist of nodes and edges: how can we represent interfaces? Finally, networks typically consist of multiple different protocols and topologies: what is the appropriate level of abstraction to capture this different topologies in a systematic manner?

Finally, how can we systematically capture network information such as IP addresses, interface names, or routing protocol metrics such as OSPF costs?

3.5.2 *Transformations*

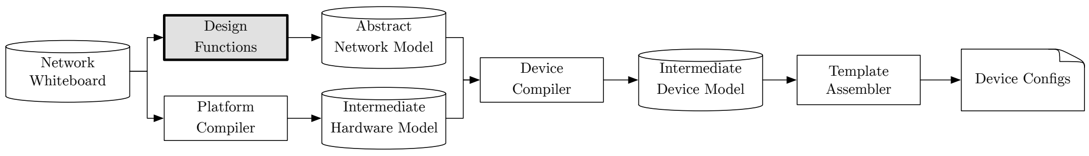


Figure 3.4: Flowchart of configuration generation toolchain, showing the role of the Design Functions

We then need to develop the appropriate transformations that can transform the Network Whiteboard input specification into the Network Views that comprise the Abstract Network Model. These are denoted as Design Functions in our toolchain, as shown in Figure 3.4.

We would like these transformations to be readily understood by network practitioners, and implementable in a common programming language. While the graph theory details are not often taught to network practitioners, we can build a series of primitives that capture common elements of network designs. These primitives can be composed to build functions for the various network design tasks. These functions operate on the abstractions, these steps can be decoupled, enabling one network practitioner can write the transformations and another to provide the Network Whiteboard as the input. This separation of concerns allows individual expertise in specific domains, and independent extensibility of the design functions.

3.5.3 *Low-Level Configuration Generation*

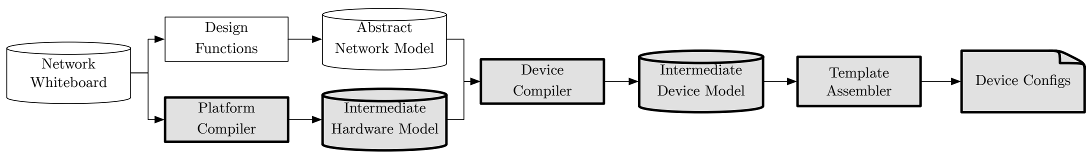


Figure 3.5: Flowchart of configuration generation toolchain, showing the role of the Compiler

Finally we need to develop the workflow for taking the Abstract Network Model intermediate representation, and building the low-level device configurations. This is done through the use of two compilers: a Platform Compiler and a Device Compiler. The Platform Compiler is responsible for mapping the hardware inventory, such as interface naming and hardware-specific information, into the Intermediate Hardware Model. The Device Compiler combines the logical inventory information from the Abstract Network Model with the Intermediate Hardware Model, to produce the Intermediate Device Model. This contains all of the configuration information needed for the specific device. There is an entry in the Intermediate Device Model for each device in the network that is to be configured. The final step is to assemble the device entry in the Intermediate Device Model with one or more templates to generate the final device configuration in the appropriate low-level syntax. These are shown in Figure 3.5. This step concludes the toolchain.

3.5.4 *Implementation*

We will verify the correctness of the configurations generated by our approach using an experiment-based approach as shown in the extended toolchain in Figure 3.6.

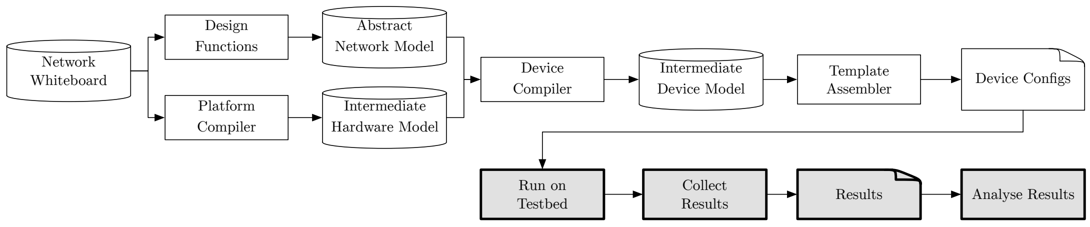


Figure 3.6: Experimentation verification process

We will first implement our theoretical approach in a widely used programming language. For assistance in debugging and to provide insight into the design process, we will develop an automated visualisation system to display the intermediate network-wide representations of the Network Views. We will use this implementation to reproduce the Netkit Small Internet topology. To test extensibility we reproduce an enterprise network with vlans and managed switches, and to test scalability we will use our implementation to configure a large scale network with over one thousand devices. We will use our abstractions and toolchain to assist in analysis, such as post processing raw data to map the IP Addresses to their corresponding network devices and interfaces. We will use the implementation to generate configurations for devices run on different testbed environments, further demonstrating the extensibility of the approach.

3.6 COMPLETE METHODOLOGY

Our full methodology is shown in Figure 3.7. The High-Level Network Configuration policy is first expressed on the Network Whiteboard. A series of Design Functions transform the Network Whiteboard into a series of Network Views to form the Abstract Network Model, which contains the network logical inventory. The Network Views can be visualised with an automated visualisation engine.

The Platform Compiler uses the Network Whiteboard to build the Intermediate Hardware Model, which contains the physical inventory of the network devices, such as interface names. The Platform Compiler can also produce the Intermediate Platform Model, containing information on how to configure the testbed, such as setting up virtual devices and connecting them together. The Device Compiler takes the Abstract Network Model and Intermediate Hardware Model, and builds the Intermediate Device model, which contains the configuration information for each device in the network. The Template Assembler combines this Intermediate Device Model with simple templates to produce the final Device Configurations. If necessary a template can be used to produce the Platform Configuration.

We then launch the simulation testbed with the Device Configurations and Platform Configurations. This provides a running network using the specification provided on the Network Whiteboard. We collect diagnostic information from this running network to enable us to verify the network has been correctly configured.

We demonstrate this methodology for our implementation in Chapter 7 and will vary components, such as the Design Functions, Compilers, or Testbed for the case studies in Chapter 8. This allows us to demonstrate the extensibility and scalability of our approach and toolchain, and to provide verification of these variations.

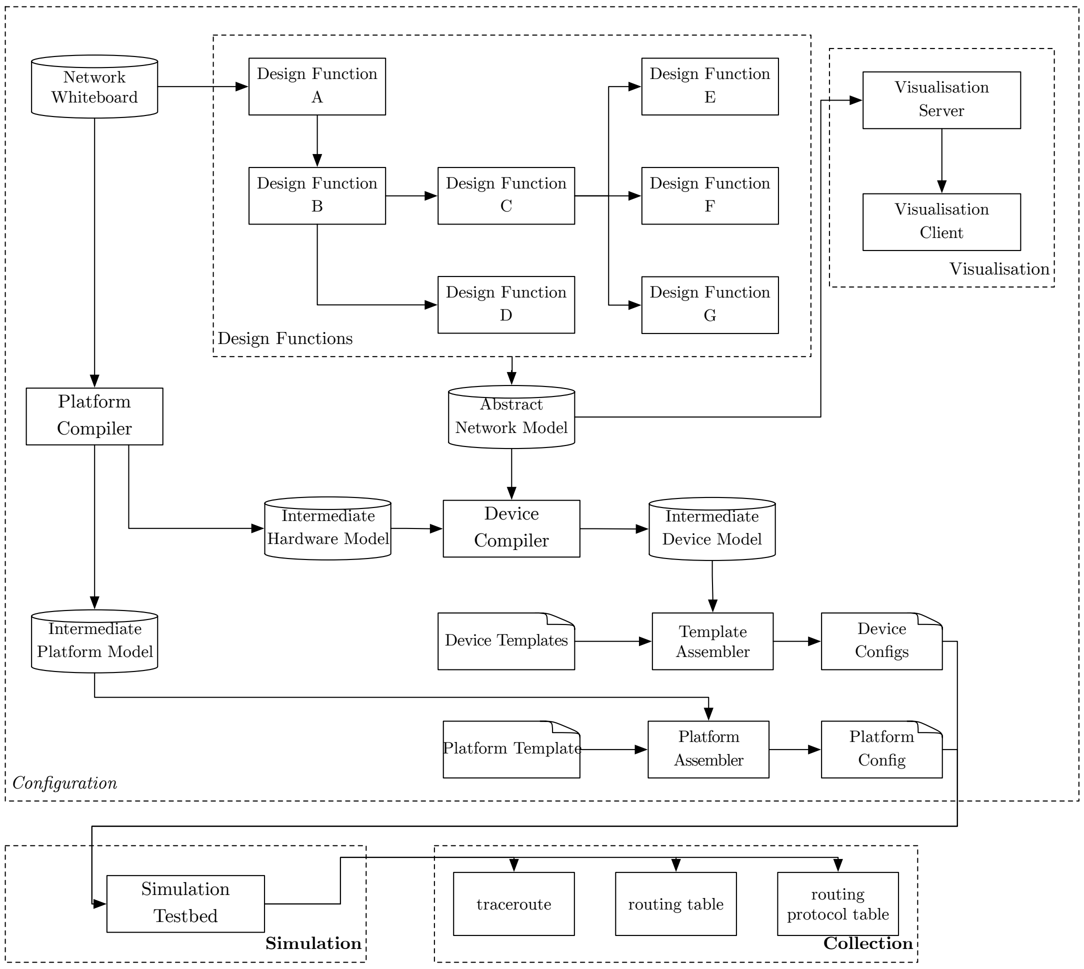


Figure 3.7: Implementation and Verification Methodology used in this thesis

3.7 CONCLUSION

In this chapter we have presented the five research questions that are addressed in this thesis. We have also provided a strategy for answering these question and identified how the result of the research strategy will be validated. In the next chapter we present a more detailed example of the toolchain, and then a detailed description of our key abstractions: the declarative high-level network policy representation, called the Network Whiteboard and the intermediate representation, the Abstract Network Model, comprised of Network Views.

Part II

FRAMEWORK AND TOOLCHAIN

4

SPECIFICATION AND REPRESENTATION USING NETWORK VIEWS

4.1 INTRODUCTION

In the previous chapter we provided an overview of our research methodology and our toolchain. In this chapter we present the fundamental abstractions from which our network modelling language is constructed. These abstractions are used for the high-level specification language, and for the network-wide design transformations.

Network configuration involves configuring individual network devices. This configuration on each device has network-wide context: the configuration state of an individual device in the network is dependent of the configuration of other devices in the network. As discussed in previous chapters, the task of network configuration is one of distributing the configuration state to each device in the network, to describe the functionality of the device in the network context.

In traditional networks, the configuration state of each device is represented in low-level configuration files, using a syntax and semantics that vary depending on the vendor of the network device. Together, each of the device configurations form the network. These device configurations are used to implement a desired network-wide state. This network-wide state in turn is used to implement high-level business or technical goals. In this chapter we will develop a series of abstractions which can help to automate this process: the capturing of high-level goals and the transformation of these to a network-wide model. In this chapter we present these abstractions, with the next chapter discussion the transformations of the Network Whiteboard abstraction into the Network Views, which can then be subsequently transformed into the target device configurations.

In this chapter we will show how the Network Whiteboard abstraction addresses Research Question 1: *Is there a declarative representation of High-Level Network Policy that is also likely to be widely adoptable by* *current network practitioners?*, and the Network Views abstraction addresses Research Question 2: *What is a better intermediate representation of network configuration that is based on graph theory, supports a well structured compilation process and provides clear separation of concerns?*

4.2 CHAPTER OVERVIEW

This chapter describes the overall toolchain used in our approach, and then focusses on the **Network Whiteboard** abstraction used for describing the high-level policy; and the **Network View** abstractions which are used as the high-level intermediate form. This chapter is organised as follows. We first provide a simplified example of the overall toolchain. We then introduce the key concepts involved in the Network Whiteboard and Network View abstractions.

From there we expand this example to show how the Network Whiteboard and Network View abstractions can be used to represent a more complicated topology of fourteen routers organised into seven autonomous systems. We show the use of the Network View abstraction to represent the relationships of the Physical and Layer 2 connectivity, IP addressing, and the OSPF, iBGP and eBGP routing protocols.

We then discuss the implementation details of the Network Whiteboard and Network View abstractions. We provide an overview of the key components of the implementation, with additional detail provided in the appendices.

4.3 OVERVIEW OF APPROACH AND TOOLCHAIN

In this section we provide an informal overview of our approach to introduce the key components in the tool chain. A simplified view of the toolchain is shown in Figure 4.1. We use this simplified view to explain the general concepts. At the end of this section, we will revisit the complete toolchain which we have shown in previous chapters.

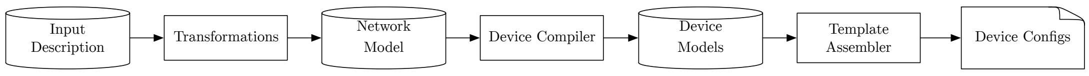


Figure 4.1: Simplified toolchain flowchart

4.3.1 *House Topology*

The topology that we use for this example, is based on a five router "house" network topology, as shown in Figure 4.2. We will use this topology to illustrate a number of aspects of the network design process, including various routing protocols, and IP address allocation. This topology contains two autonomous systems, which represent different network entities, such as different companies or Internet Service Providers. The four routers on the left belong to autonomous system *1*, and the right-most router belongs to autonomous system number 2. We use this classification when we design the routing protocols configuration.

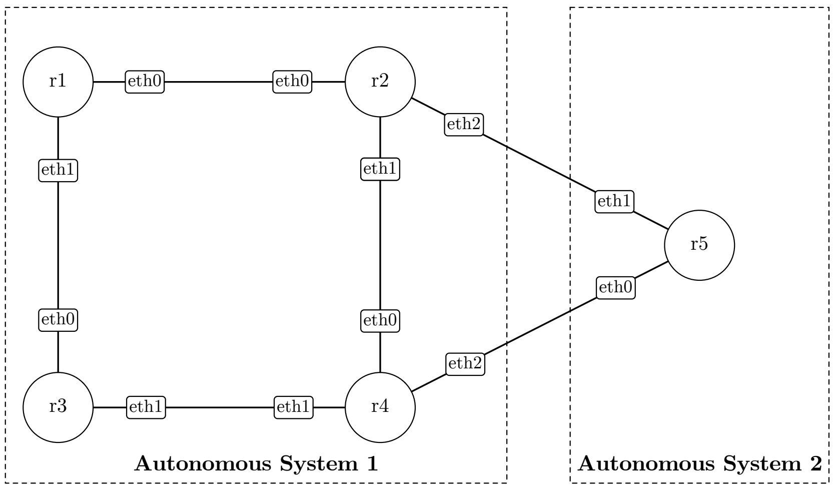


Figure 4.2: House example topology.

4.3.2 *Network Topologies*

We now provide an overview of the various "views" of the network configurations can arise from this house topology example. This includes the physical wiring of the devices, the IP addresses allocated to the interfaces of the devices, and the various designs for the routing protocols. These "views" could be constructed by a network practitioner, based on a description of the network topology and a set of design rules. The views correspond to the diagrams used to describe networks, such as in network textbooks, or in diagrams constructed by network operators in diagramming tools such as Microsoft Visio. Our views are based on graphs, where the nodes correspond to the network devices, and the links between these nodes correspond to the connectivity implied by the view. For instance on the physical view, the links would represent physical connectivity, such as a wire. On the OSPF view, a link would represent an adjacency configured in the OSPF routing protocol. We also allow for interfaces to be specified at each end of the link. Finally, we allow for nodes, links, and interfaces, to be labelled, which allows us to specify information such as a node or interface name, or an interface IP address.

These views are decoupled from the specific implementation in the configuration of the network device. A network practitioner could take the information contained in these network views, and manually create the configuration for each device on the network.

4.3.3 *Input Topology*

The first step is to describe the network topology and key information, such as the autonomous system numbers. We do this by labelling the devices in the input topology. This labelled input description is then used to build the various network topologies. An example of the original topology, with the devices labelled with their autonomous system number, is shown in Figure 4.3.

4.3.4 *Physical Network Topology*

The Physical Network view shows the devices and the wiring between them, and is shown in Figure 4.4. It can be derived simply from the input topology specification.

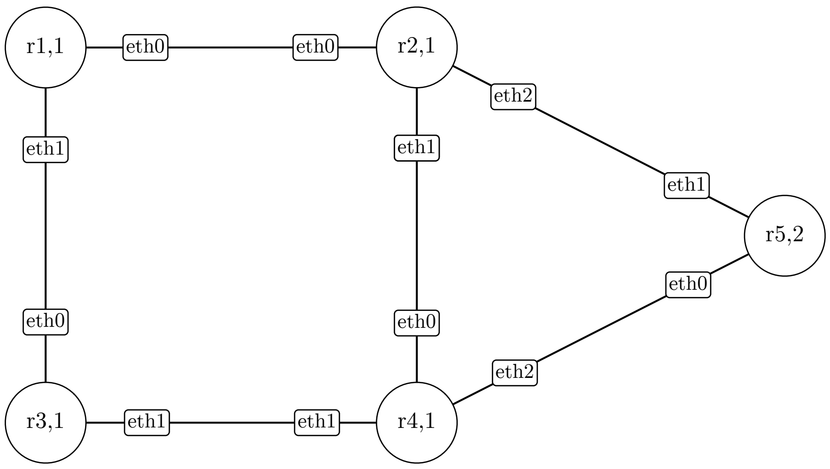


Figure 4.3: Input description with labelled devices.

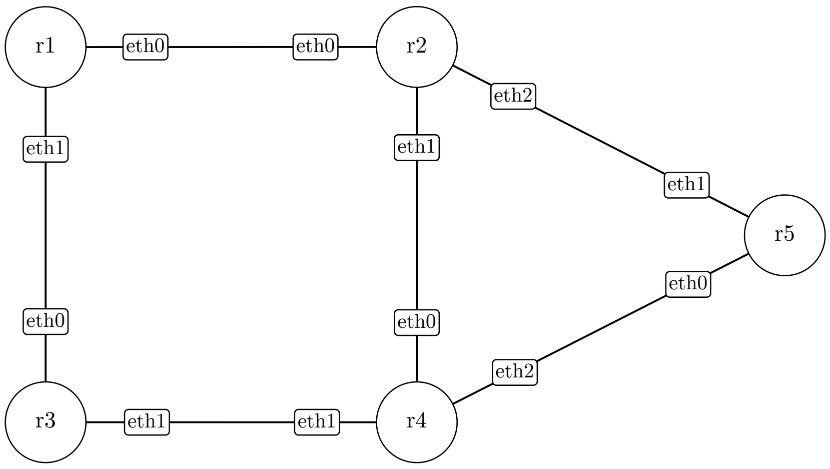


Figure 4.4: Physical Network Topology.

4.3.5 *IP Address Network Topology*

The IP address topology is shown in Figure 4.5. Here, we have labelled each of the interfaces with an IP address, and each of the nodes with a loopback IP address.

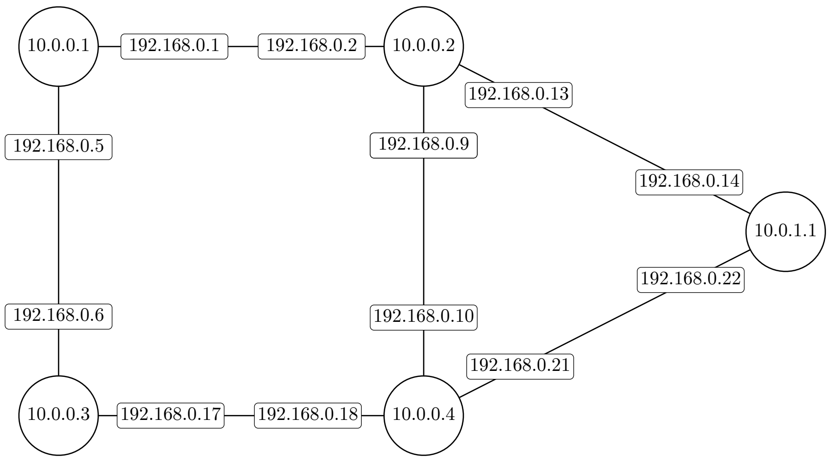


Figure 4.5: IP Address Network Topology with loopback addresses on devices and infrastructure addresses on interfaces.

4.3.6 *OSPF Network Topology*

OSPF is a protocol that is used to share network information within an autonomous system. Therefore, the links in the topology shown in Figure 4.6 are only between devices in the same autonomous system. This can be constructed from the input topology, by simply removing the links between routers in different autonomous systems.

In this example we have extended the idea of labelling interfaces, and use this to carry further specifications about the OSPF network design. OSPF allows for a network to be broken up into smaller regions, known as "areas". It also allows for a metric to be specified in an interface, which is used to influence the shortest path algorithm used when finding the optimal route to a destination IP address. This metric is known as a "cost". In the topology shown in Figure 4.6, the interface labels are a tuple of ("area", "cost"). This allows more detailed information to be provided to the configuration process. We also mark the area for the loopback interface of the router in the tuple on the router. We use the area 0 for all of the physical and loopback interfaces in this example.

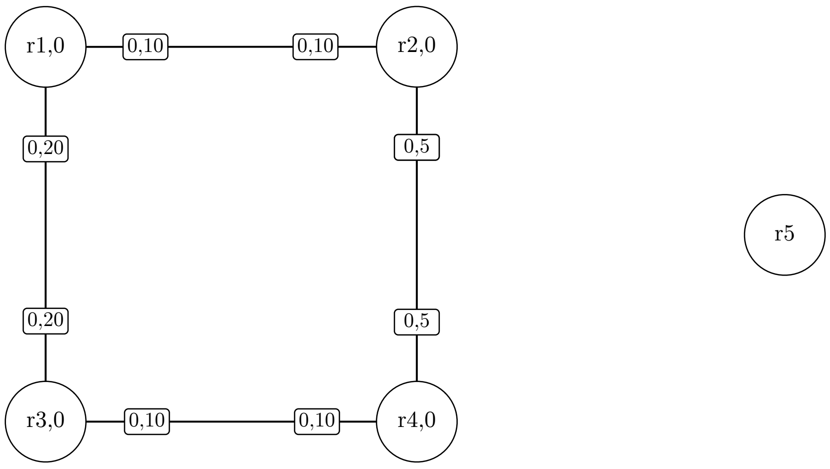


Figure 4.6: OSPF Network Topology.

4.3.7 *eBGP Network Topology*

BGP is the Border Gateway Protocol, and is used to share routing information between autonomous systems. We first look at the topology for eBGP, the exterior Border Gateway Protocol. eBGP is used to establish BGP sessions between routers that belong to different autonomous systems. For our house example, this can be derived by retaining the links that connect routers in different autonomous systems. This is essentially the inverse of the step used to construct the OSPF network topology. This topology is shown in Figure 4.7.

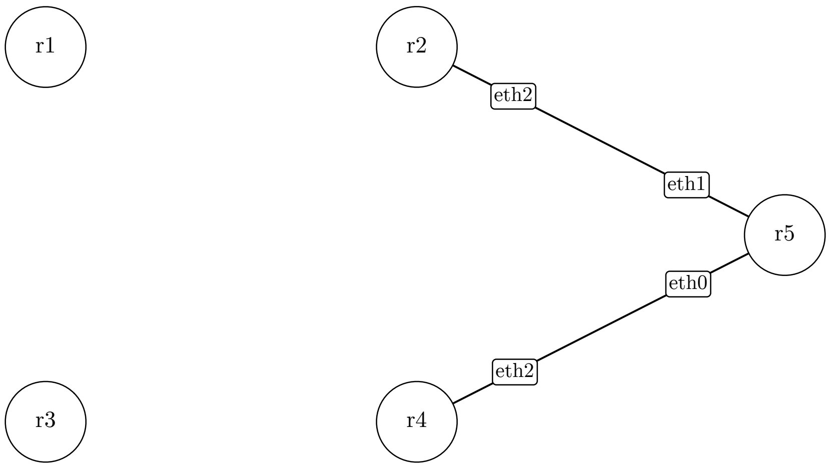


Figure 4.7: eBGP Network Topology.

4.3.8 *iBGP Network Topology*

iBGP is interior BGP, and is used to share eBGP information into an autonomous system. For our example, this will allow the routers *r1* and *r3*, on the left of the diagram, to hear the information learnt by *r4* and *r2* over their eBGP session to *r5*. This would allow us to be able to successfully *ping* from *r1* or *r3* to *r5*. There are many methods to construct the iBGP topology, which we discuss Here we use a simple full-mesh, where each router iBGP peers to the other routers in the same autonomous system, as shown in Figure 4.8. This simple full-mesh design means we do not consider the physical links from the input specification, and only considers the node labels.

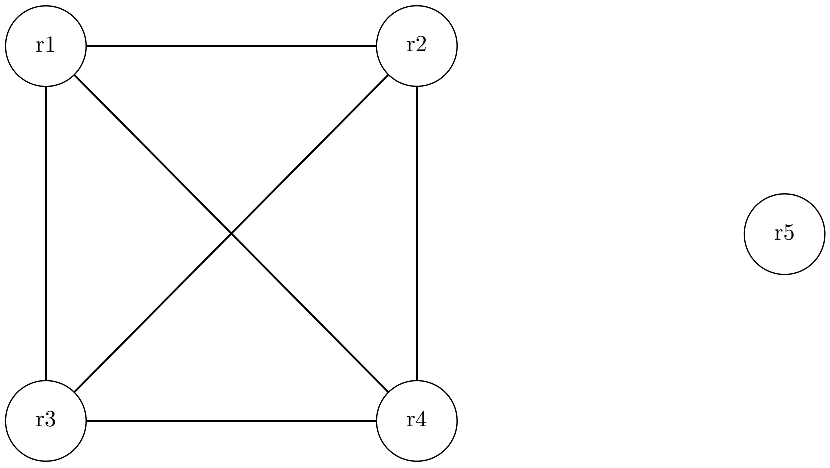


Figure 4.8: iBGP Network Topology.

4.3.9 *Layer 2 Network Topology*

The final network topology which we look at is for Layer 2. By Layer 2 we mean the Data Link Layer of the OSI reference model, and can be useful in modelling hub and switch network devices. As we will show in this thesis, this can be used to derive connectivity to build the other topologies. Our house input topology simply consists of direct point-to-point links between routers, so the layer topology shown in Figure 4.9 reflects the physical topology, with a virtual "device" introduced to represent the broadcast domain of each point-to-point link. This broadcast domain concept is represented by the diamond symbol. It provides an abstraction which we develop further in this thesis to incorporate network devices at different levels of functionality.

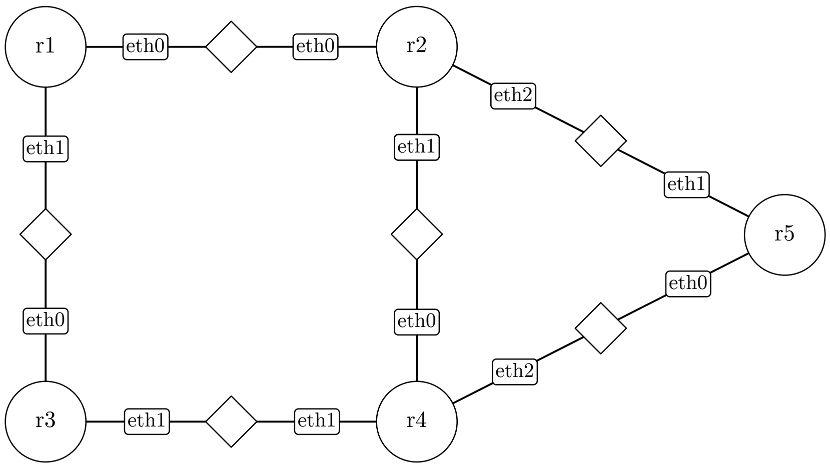


Figure 4.9: Layer 2 Network Topology.

4.3.10 *Network Model*

We can then construct a network model, which is the set of all of these network topologies. An example of such a network model is shown in Figure 4.10. This is used as the input to the configuration generation process.

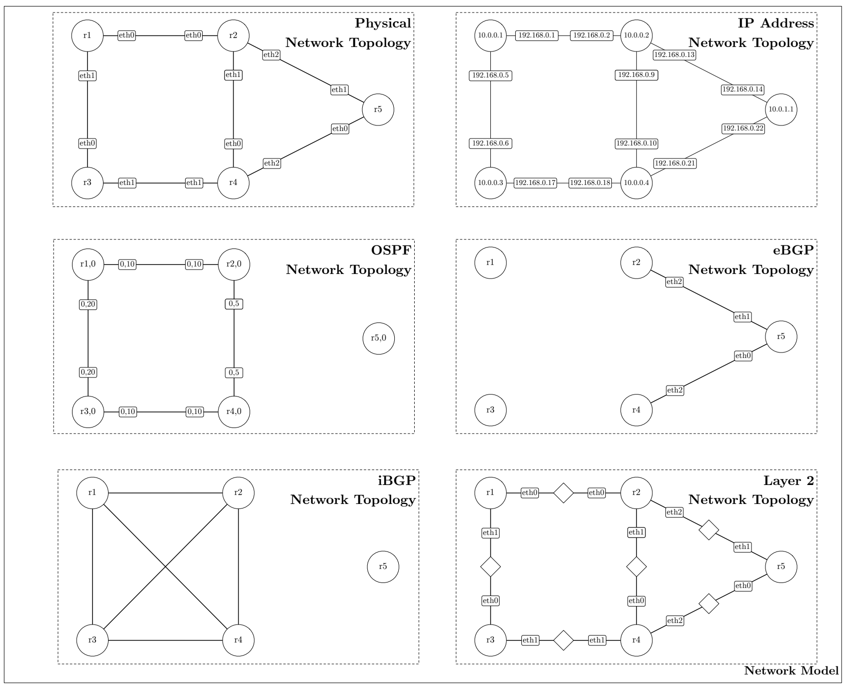


Figure 4.10: Network Model consisting of each Network Topology

4.3.11 *Design Process*

We now provide a brief overview of the design process, which transforms the high-level representation into each of these various network topologies. The input description topology is transformed into each of the network topologies. An overview of this transformation step is shown in Figure 4.11. We will discuss these transformations in detail in Chapter 5.

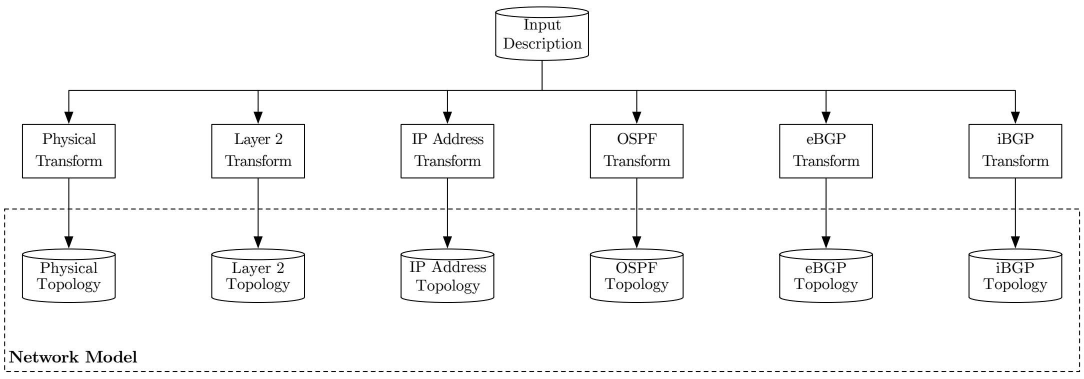


Figure 4.11: Creating Network Topology from Input Topology

4.3.12 *Device Compilation Process*

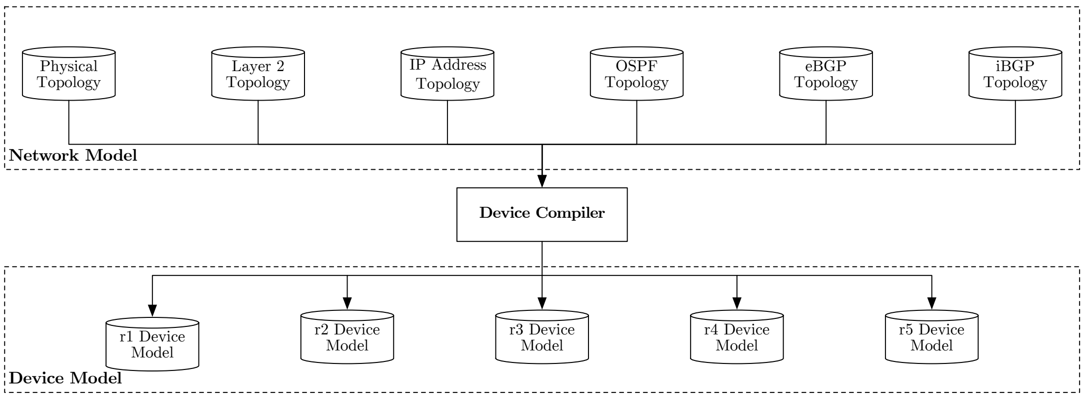


Figure 4.12: Compiling Device Model from Network Model

The next step is to build the individual device configurations. We do this using a second intermediate representation. As shown in Figure 4.12 we first apply a device compiler which maps the relevant information about each device, from each of the network topologies, into a device model. This device model contains the key information required to generate the device configurations. An example of such a device model for an idealised target device for *r4* is shown in Listing 4.1. While network operator could transcribe the information in this device model into low-level device configurations, we automate this process using templates.

```
{
  "asn": 1,
  "bgp": {
    "ebgp_neighbors": [
      { "asn": 2, "ip": "192.168.0.22" }
    ],
    "ibgp_neighbors": [
      { "asn": 1, "ip": "10.0.0.1" },
      { "asn": 1, "ip": "10.0.0.2" },
      { "asn": 1, "ip": "10.0.0.3", }
    ],
    "networks": ["10.0.0.0/24", "192.168.0.0/24"]
  },
  "hostname": "r4",
  "interfaces": [
    { "id": "eth0", "ip": "192.168.0.10/30" },
    { "id": "eth1", "ip": "192.168.0.18/30" },
    { "id": "eth2", "ip": "192.168.0.21/30" },
    { "id": "lo", "ip": "10.0.0.4/32", }
  ],
  "ospf": {
    "interfaces": [
      { "cost": 5, "id": "eth0" },
      { "cost": 10, "id": "eth1" }
    ],
    "networks": [
      { "area": 0, "network": "10.0.0.4/32" },
      { "area": 0, "network": "192.168.0.8/30" },
      { "area": 0, "network": "192.168.0.16/30" }
    ]
  }
}
```

Listing 4.1: Example device model for *r4*

4.3.13 *Configuration Assembly Process*

We automate the process of assembling device models into configurations by using templates. The Template Assembler combines the device model with simple templates to produce low-level device configuration syntax for the target devices. This process is shown in Figure 4.13. We provide more details of this process in Chapter 6. An example of the output configuration for *r4* for a simplified device configuration is shown in Listing 4.2.

```
!
hostname r4
!
interface eth0
  ip address 192.168.0.10/30
  ip ospf cost 5
interface eth1
  ip address 192.168.0.18/30
  ip ospf cost 10
interface eth2
  ip address 192.168.0.21/30
```

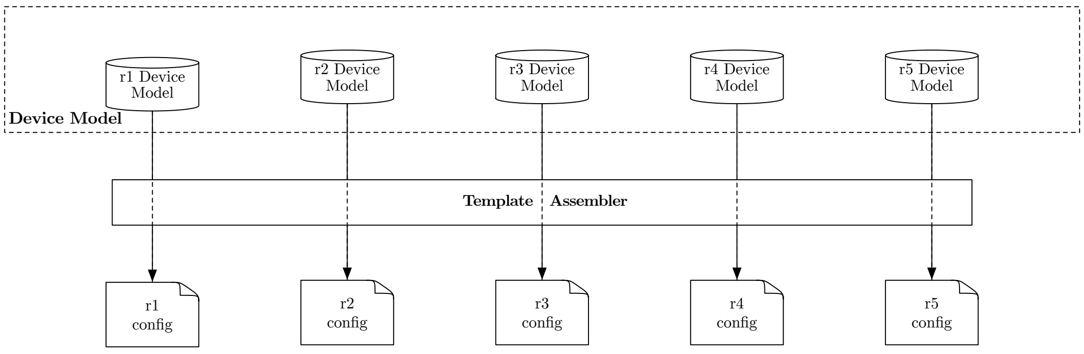


Figure 4.13: Configuration Assembly: Device Model and Templates

```
interface lo
  ip address 10.0.0.4/32
!
router ospf
  network 10.0.0.4/32 area 0
  network 192.168.0.8/30 area 0
  network 192.168.0.16/30 area 1
!
router bgp 1
  network 10.0.0.0/24
  network 192.168.0.0/24
  !
  neighbor 10.0.0.1 remote-as 1
  neighbor 10.0.0.2 remote-as 1
  neighbor 10.0.0.3 remote-as 1
  !
  neighbor 192.168.0.22 remote-as 2
!
```

Listing 4.2: Example router configuration for *r4*

4.3.14 *Complete Toolchain*

In Figure 4.14 we show the complete toolchain for our approach. This uses the terminology used in this thesis. The *Network Whiteboard* corresponds to the input topology discussed in this section. The *Abstract Network Model* corresponds to the Network Model. This consists of *Network Views* which represent the individual topologies such as for IP Addressing, OSPF, or eBGP.

The *Design Functions* are used to transform the *Network Whiteboard* into the *Network Views* of the *Abstract Network Model*. The *Device Compiler* compiles the *Abstract Network Model* into the *Intermediate Device Model* for each device. These are then assembled with templates by the *Template Assembler* to produce the *Device Configurations*.

To allow for the configurations to be verified using the various testbed environments discussed in Section 2.9, our toolchain allows for a *Platform Compiler* which maps the physical connectivity and any relevant labels from the Network Whiteboard into an *Intermediate Hardware Model*. This can include device wiring information, interface naming schemes, and out-of-band management information. This Intermediate Hardware Model can be used as a supplementary input to the Device Compiler.

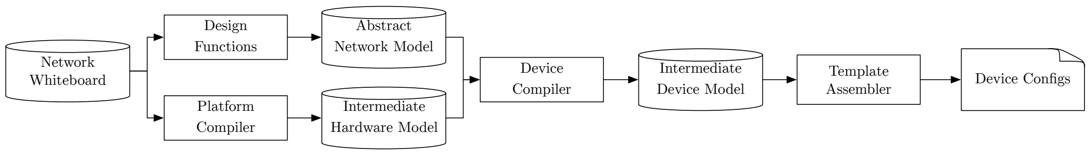


Figure 4.14: Flowchart of toolchain

4.3.15 *Conclusion*

In this section we have provided a general overview of the key aspects of the toolchain used in our approach. We have shown how an input topology, consisting of a labelled graph, can be used to derive a number of network design related topologies through the use of a transformation step. These network design related to policies can then be compiled down to an abstract version of the syntax for a target device, which can then be assembled using templates, to produce a device configuration.

We now formalise the key elements of the labelled graph abstraction.

4.4 NETWORK WHITEBOARD AND NETWORK VIEWS ABSTRACTIONS

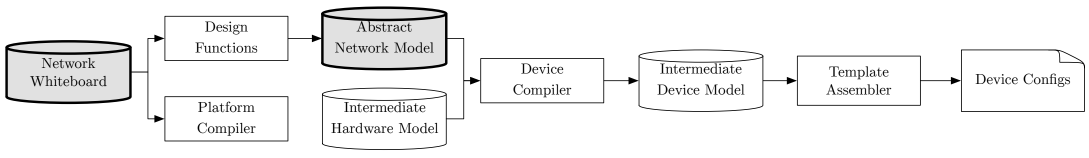


Figure 4.15: Overview of toolchain highlighting the Network Whiteboard and Abstract Network Model

This section presents the key elements of our model to capture a high-level description of the network design goals, and the intermediate network-wide abstraction. This is the Network Whiteboard and Abstract Network Model components of the toolchain as shown in Figure 4.15.

4.4.1 *Specification and Representation*

In this chapter we develop a set of network-level abstractions which can be used for the specification and representation of network topologies. The specification allows the high-level network requirements to be described in a declarative manner. This allows the capture of the High-Level Network Configuration Policy, discussed in Section 2.4. In our approach this specification model is referred to as the Network Whiteboard. The abstractions also form the basis of intermediate network models, which allows the manipulation and transformations of the High-Level Network Configuration Policy into the lower-level configurations. These intermediate representations are known as Network Views in our approach. This forms the Intermediate Representation discussed in Section 2.6.

We now present the key concepts in the network-level abstraction, and then show how these form the Network Whiteboard and the Network Views.

4.4.2 *Abstracted Network Topology Models*

In this section we discuss the abstracted network topology models which are used in our approach. These models are used for Network Whiteboard, which is where the user provides the policy description, and is also used for each of the Network Views, which represent the various routing protocols to be configured.

The topology model contains the Network Elements, Network Element Connections, and Network Element Interfaces.

In addition, each of these elements are able to be labelled. A system revolves around the construction Network Views based on the Network Elements, and the connections, and the label set on these and other Network Element Interfaces. By manipulating these elements through successive filtering operations, and the creation of connections based on the labels, we are able to capture the core information required to describe the configuration of various routing protocols in the network.

As will show in subsequent chapters, this description can then be used to generate individual device configurations, which are logically consistent across a network wide view. In addition, working with individual network protocols, this allows separation of concerns and simplifies the construction and verification of individual network protocol configurations. The core components of the Network Whiteboard and Network Views are shown in Figure 4.16.

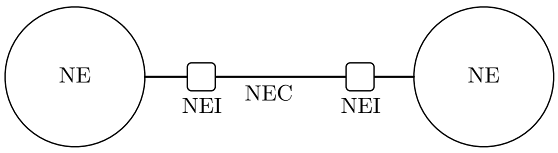


Figure 4.16: Components used in both the Network Whiteboard and Network Views: Network Elements (NE), Network Element Interfaces (NEI), and Network Element Connections (NEC).

Network Whiteboard

The network operator describes their specific network at a high-level, capturing the network devices, their interconnections, and the roles the devices play in the network. This is captured using a **Network Whiteboard** abstraction. The Network Whiteboard consists of Network Elements, which contain Network Element Interfaces. Network Element Interfaces are connected by Network Element Connections.

Our Network Whiteboard abstraction aims to facilitate the goal of "policy abstraction must be as simple as drawing diagrams" described by Prakash *et al.* [86], and discussed in the Literature Survey in Section 2.4. The Network Whiteboard is the highest level abstraction in our design process. It is the starting point for the design flow shown in Figure 4.17.

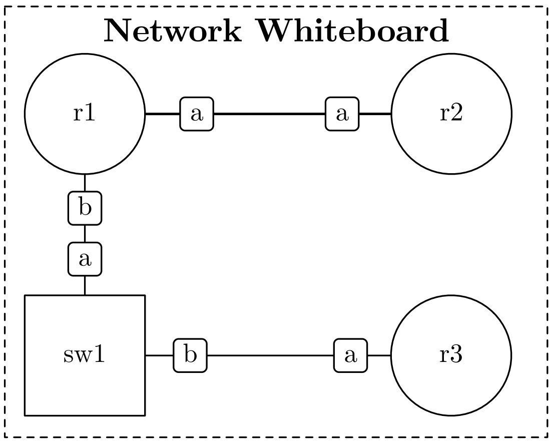


Figure 4.17: Network Whiteboard, showing four Network Elements and their Network Element Interfaces, and three Network Element Connections between them. The Network Elements represent three routers (r1, r2 and r3) and a switch (sw1).

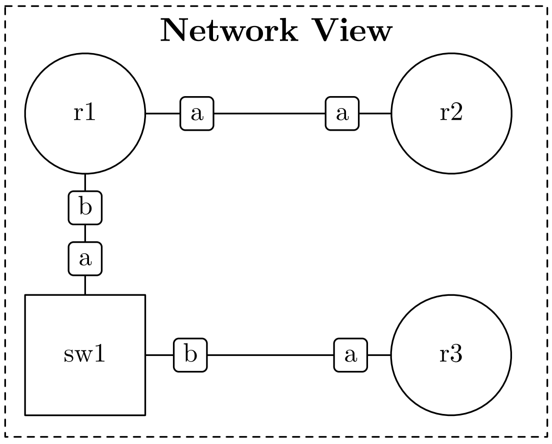


Figure 4.18: Network View, with the same elements as the Network Whiteboard shown in Figure 4.17

Network Views

The **Network Views** enable the use of topologies to represent relationships between Network Elements, at various aspects of the configuration process. A number of Design Functions transform the Network Whiteboard into a series of Network Views. Network Views are network-design focussed: each view represents a layer or "slice" of the network design process. The Network Views capture the information about the role of a Network Element in the network. They do not capture the internal structure of a device: this information is provided to the Device Configuration process. An example of a Network View is shown in Figure 4.18.

Abstract Network Model

The set of all Network Views is the **Abstract Network Model**, as shown in Figure 4.19. The Network Whiteboard is not part of the Abstract Network Model, but is used to derive the Network Views in the Abstract Network Model.

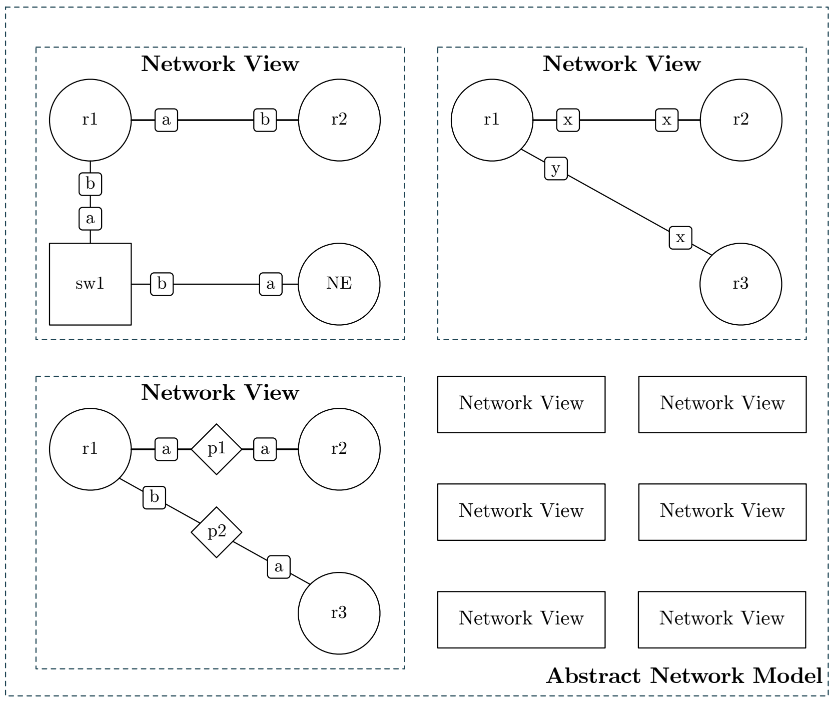


Figure 4.19: An example Abstract Network Model containing multiple Network Views.

4.4.3 *Network Devices*

A Network Device is a component that receives and sends a data packet, with or without processing. Examples of devices include switches, routers, firewalls, and end-hosts such as servers. The approach can be extended to support other devices as required. A Network Device is captured in two aspects in our approach. The first is the high-level functionality, such as the type of device (router, switch, server, etc), which is captured on the Network Whiteboard and used in the Design Functions to create the Abstract Network Model. This corresponds the logical inventory of a device, such as for routing protocol configuration. The second aspect is specific device parameters, such as the vendor, model number, and operating system version, as well as hardware capabilities such as interface numbering structure. These are captured as input to the device compilation step to build the Intermediate Hardware Model, which corresponds to the physical inventory of a device.

4.4.4 *Network Elements*

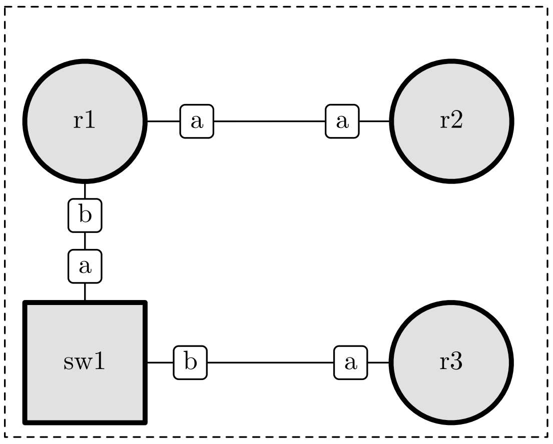


Figure 4.20: an Abstract Network Model showing Network Elements.

A Network Element is an abstraction of a Network Device, and are shown in Figure 4.20. Network Elements also have a unique identifier, which can be used to identify the Network Element across different Network Views. This allows that topologies to be constructed for various protocols, due to the relationship between the same set of Network Elements. Network Elements can be annotated with labels used to construct the Network Views. These labels can capture the role and other information about of the Network Element.

Role

The device type of a Network Element represents the role of the Network Element, and is represented by a symbol, used on the Network Whiteboard. An example of common symbols used is shown in Figure 4.21. These shapes are a pseudo-label: they could be represented by a label chosen from an alphabet of available device types: $\{\mathtt{router}, \mathtt{switch}, \mathtt{server}, \mathtt{firewall}, ...\}$.

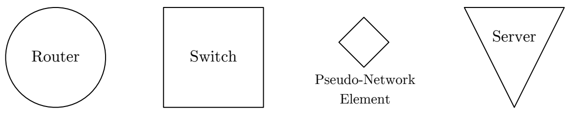


Figure 4.21: Symbols used to represent Network Element Roles

Network Element Labels

Labels can be applied to Network Element by the network designer. These could be used to indicate membership of a group, such as an autonomous system; a role within a topology such as an iBGP route reflector; an IP address; or an OSPF area or cost. As we will show in this thesis, labels provide a flexible approach to capture High-Level Network Policy.

Pseudo Network Elements

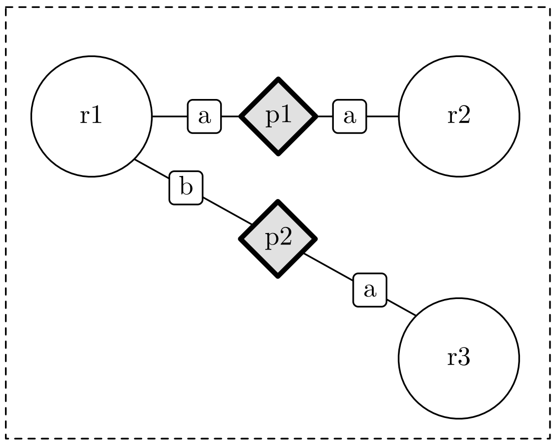


Figure 4.22: An Abstract Network Model containing Network Elements and Pseudo Network Elements, with the Pseudo Network Elements highlighted.

We previously introduced the concept and motivation for the Pseudo Network Element in our Layer 2 Network Topology in the House example. Pseudo Network Elements are Network Elements which do not correspond to a target physical device, and therefore do not have a configuration generated for them. An example is shown in Figure 4.22. "Normal" (physical) Network Elements are specified on the Network Whiteboard, and are present in the Intermediate Hardware Model. Pseudo Network Elements can be created in the Network Views as part of the network design process. This was demonstrated in the Layer 2 Network Topology of the House example. They can also be used on the Network Whiteboard to capture virtual or external connectivity, such as an aggregate of end hosts, or a third-party network which is connected over a peering link.

The example in Figure 4.23 shows Pseudo Network Elements being used to represent the broadcast domains in a Layer 2 Network View. This simplifies the creation of the OSPF topology, where connections are made if two Network Elements have a connection in the Layer 2 Network View. This extends the motivation presented in the Layer 2 Network View in the House topology example. Pseudo Network Elements can also be used on the Network Whiteboard to represent external devices which may not be necessarily under the specific configuration of the configuration system, but may represent a part of a configuration, such as an external network and its IP address, that we wish to establish a connection to, as we will show in Chapter 8.

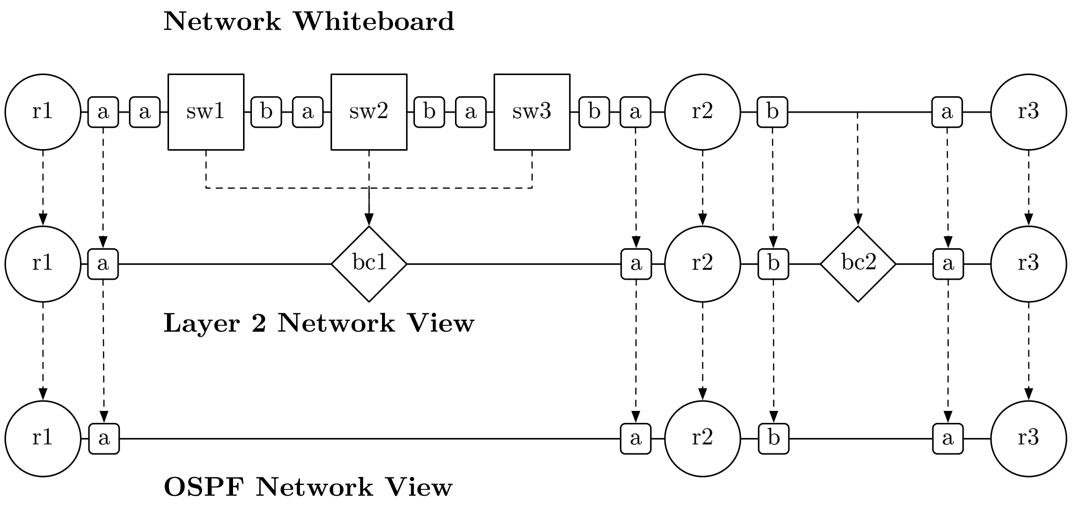


Figure 4.23: Pseudo Network Element Example

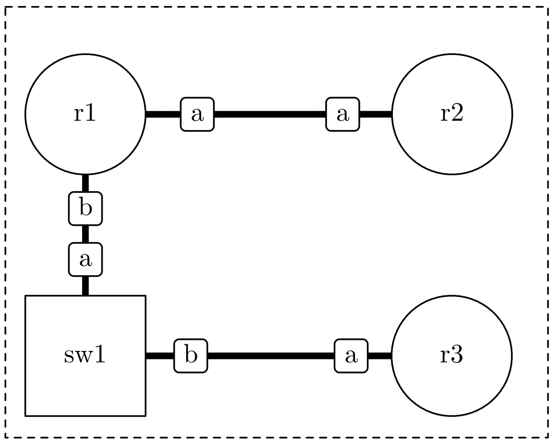


Figure 4.24: Abstract Network Model showing Network Element Connections.

4.4.5 *Network Element Connections*

Associations can be made between Network Elements, called Network Element Connections. These are shown in Figure 4.24. These can be bound to Network Element Interfaces. Network Element Connections can also have labels associated with them.

4.4.6 *Network Element Interfaces*

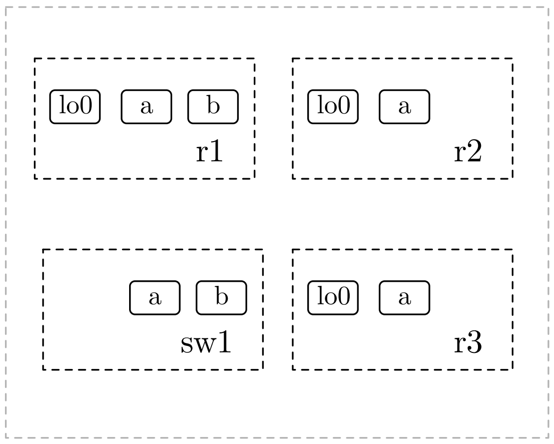


Figure 4.25: A conceptual representation of Network Element Interfaces shown inside each Network Element.

A Network Element can have Network Element Interfaces. These have labels such as whether or it is a physical or logical interface, or whether it is management or data. A conceptual representation is shown in Figure 4.25. These also can have labels associated with them.

Network Element Interface Bindings

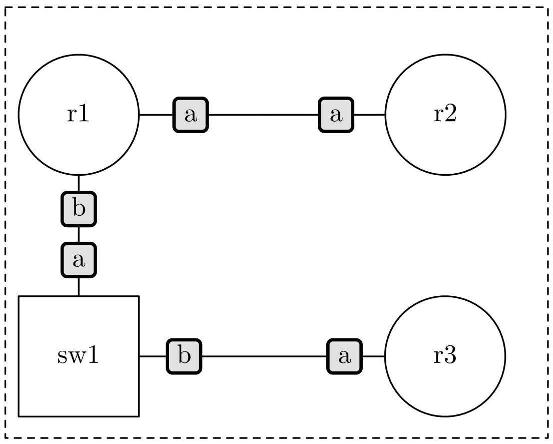
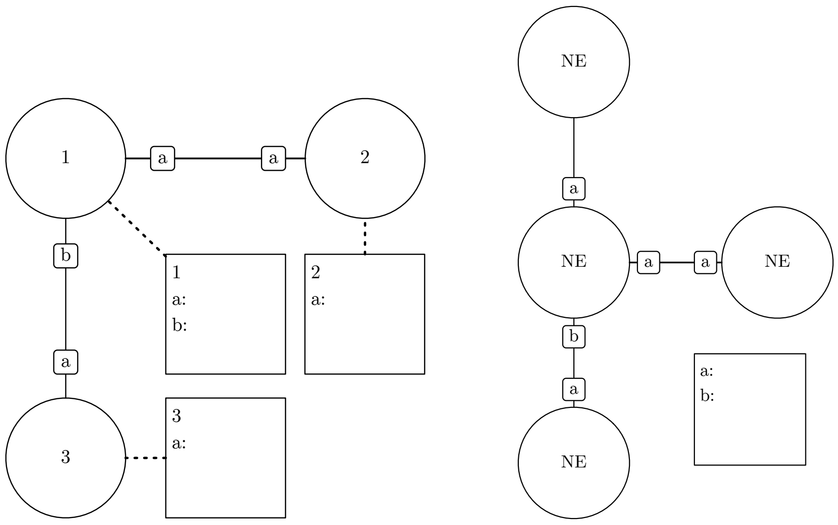


Figure 4.26: Network Element Interface Bindings



(a) Network Element Interface Pointers: Binding a Network Element Connection to a Network Element Interface

(b) Network Element Interface Bindings: Binding multiple Network Element Connections to a single Network Element Interface.

Figure 4.27: Overview of Interface Pointers.

Network Element Interface bindings are shown in Figure 4.26, and in more detail in Figure 4.27.

We use bindings allows us to be free of storing attribute information for the configuration itself, on the edges in our model. By reducing the information stored on the edges to only connectivity or information about the edge, we simplified working with Network Element connections across different overlays.

As we will show in the Chapter 5, this simplifies constructing Network Views from other Network Views, while retaining connectivity. An example is a hub which connects multiple devices such as routers together: at the Physical Network View each router is directly wired to the hub. However in another Network View, such as IP connectivity, each of these routers will act as if it has a direct connection to the other routers connected to the hub. The IP connectivity Network View would then have the routers directly connected to each other. However we want to retain the interface on which they connect, which is derived from their connectivity to the hub on the Physical Network View. By storing a reference to the interface on the Network Element Connection, we are able to retain this in advanced design steps such as explode, split, and merge, which we will present in the next chapter. The alternative approach is to store configuration information on the Network Element Connection itself. This approach becomes complex to reconcile information across Network Views, especially when Network Element Connections have been split or exploded using Design Functions.

Logical Network Element Interfaces

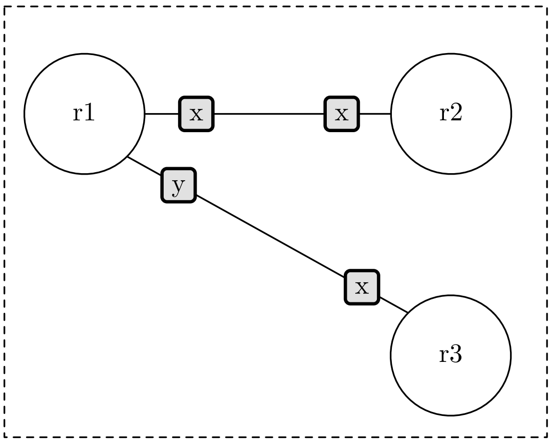


Figure 4.28: Logical Network Element Interfaces

Physical Network Element Interfaces are added from the Network Whiteboard, and used in the Intermediate Hardware Model. They cannot be added after the Network Whiteboard has been read. Logical Network Element Interfaces can be added at any time during the design process. These can correspond to interfaces such as loopbacks or tunnel endpoints, and are shown in Figure 4.28.

Association of Logical Interfaces

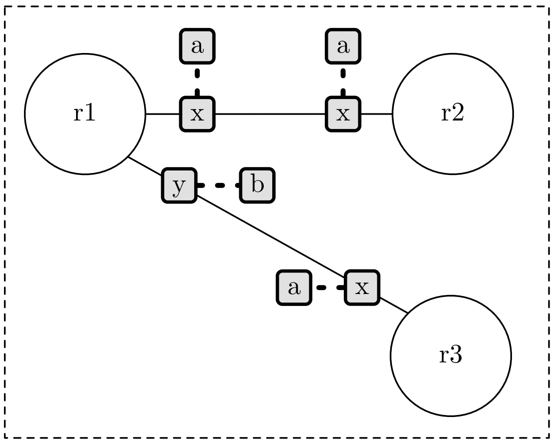


Figure 4.29: Association of Logical Network Element Interfaces to Network Element Interfaces

Logical interfaces do not map to a physical interface on the actual physical device. These can be generalised to represent termination points, such as for BGP sessions. For example, Logical Interface Associations can be used to map a BGP session to a loopback or physical interface. An example is shown in Figure 4.29.

4.4.7 *Discussion*

Identifiers

Network Element Interfaces have an identifier that is also unique across all Network Views, as shown in Figure 4.30. Network Elements are unique across all views; Network Element Interfaces are unique within a Network Element across all views; and Network Element Connections have scope within a Network View.

Consistency

As well as label verification using a system such as a schema, we can also do structural consistency checks on a network. For instance you may not be allowed to connect to network devices of the same role together, or interfaces of different roles may not be used to establish a connection between two Network Elements.

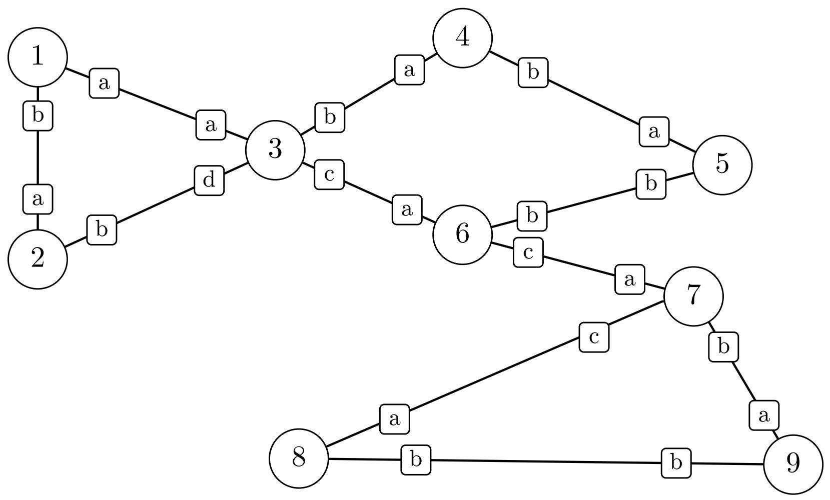


Figure 4.30: Identifiers: Network Elements and Network Element Interfaces. Network Element Identifiers are globally unique, and Network Element Interface Identifiers have local scope to the Network Element. A Network Element Interface can be uniquely identified globally by the tuple of (Network Element, Network Element Interface).

However for the Network Views the schema and consistency checks are used to verify that the Design Rule is correct. The scope of the Network Whiteboard is to provide information that can be used in a sensible manner by the Design Functions, and so was bounded by the Design Functions themselves, both for consistency, but also for the values of the labels that are set on the elements. However, the output of the Design Functions, which is the Network Views, is dependent on what is consistent and logically correct from a networking body of knowledge point of view. It is important the Network Views expressed by the Design Functions, are able to be translated into configurations that makes sense by the compilers. As compilers, which we discuss in Chapter 6, do not perform network consistency checks, the verification of the semantics of the network views, is performed as part of the Design Function stage.

4.4.8 *Conclusion*

In this section we have presented the Network Whiteboard abstraction for the High-Level Network Configuration Policy, and the Network Views abstraction as the intermediate network-wide abstraction. We will now demonstrate their use in a more complete example.

4.5 SIMPLIFIED SMALL INTERNET EXAMPLE

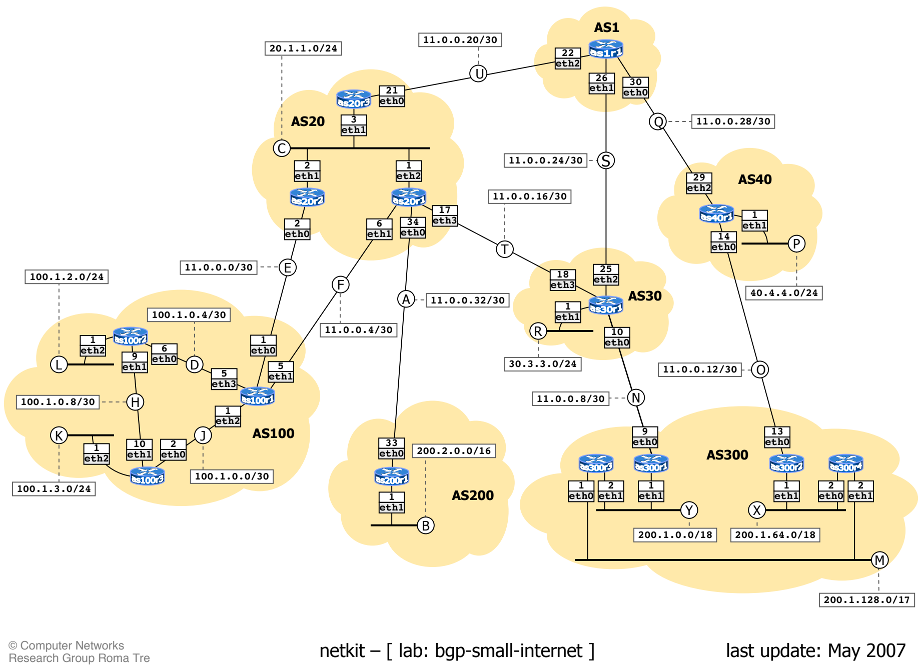


Figure 4.31: Netkit Lab Small Internet Topology from Di Battista *et al.* [25]

We now present a more encompassing example of multiple Network Views that are derived from a Network Whiteboard. The process of generating these Network Views will be described in Section 5.6. This example is a simplified version of the "BGP Lab: Small Internet" network topology described in Di Battista *et al.* [25]. This topology contains fourteen routers which are are grouped into seven autonomous systems. This lab is provided as part of the Netkit [84] simulation platform, and is described as:

> An example of complex customer-provider hierarchy resembling the typical structure of the Internet.

The topology diagram for this lab is shown in Figure 4.31, and is reproduced at full size in Figure B.2. This shows the seven autonomous systems and the connections between them. These autonomous systems are designated with roles reflecting the business relationships common in an Internet scenario. AS1 acts as the "backbone", AS20, AS30 and AS40 act as "providers" and AS100, AS200 and AS300 acts as "customers". The interface names and IP addresses are shown on the interfaces. In addition to the physical topology, there are routing policies configured to implement the backbone/provider/customer business relationships. We use this topology to demonstrate how the approach of Network Views can capture a more realistic network design than the simplified House example we showed previously. To better explain our methodology and toolchain, we first make some simplifying assumptions to the topology which allows us to focus on the key details of our approach. We remove these simplifications in Section 8.4.2 to demonstrate how our approach and toolchain can be used to automate the configuration of this topology. In the case studies in Chapter 8 we provide a number of other topologies which demonstrate the flexibility of our approach, including an enterprise network using VLANs, and a large-scale example. We now explain the simplifications made for the Simplified Small Internet example.

4.5.1 *Simplifying Assumptions*

This example is an important mechanism used to communicate the content of this research. It is introduced here and continued in Chapter 5 and Chapter 6. The implementation and case studies of Chapter 7 and Chapter 8 also use this example. We refer to this example topology as the Simplified Small Internet Example. This is example is based on Netkit BGP lab Small Internet of Di Battista *et al.* [25]. We make a number of simplifications in order to better introduce and communicate the concepts at this stage of the thesis. The simplifications are not due to limitations of our approach, and has no material impact of the intent of the example cited. We remove these simplifications in Section 8.4, where we revisit the Netkit Small Internet Lab. In Section 8.4.2 we discuss how each of these simplifications can be removed, allowing us to accurately reproduce the "BGP Lab: Small Internet" network topology described in Di Battista *et al.* [25].

The example policy and topology without simplifications, taken from Di Battista *et al.* [25] is shown in Figure B.2 and Figure B.1. The simplifications we have made are as follows:

1. There are no end-host subnets. The autonomous systems *20*, *30*, *40*, *100*, *200*, and *300* each have a */16*, */17* or */24* prefix to represent end-hosts.

2. Autonomous systems are not multi-homed. We remove the eBGP peering link from *as100r1* to *as20r1*, and from *as300r1* to *as20r1*. These correspond to *9* to *4*, and *11* to *6* in the Network Whiteboard shown in Figure 4.32.

3. IP addresses are automatically allocated. This simplifies the specification of information on the Network Whiteboard. Figure B.2 specifies each IP address in the network.
4. OSPF is used as the IGP within each autonomous system. Di Battista *et al.* [25] uses RIP as the IGP within the autonomous systems.
5. All iBGP peerings are full-mesh, and routes are not redistributed into the IGP. This simplifies the routing protocol configuration. Di Battista *et al.* [25] uses a variety of methods to announce eBGP routes to the other routers within an autonomous system.
6. There is no eBGP policy.
7. All links are point-to-point.

We now introduce the Network Whiteboard and Network Views involved in this complete example.

4.5.2 *Network Whiteboard*

The Network Whiteboard shown in Figure 4.32 represents a high-level description of the Network to be configured. This is based on the overview diagram provided in Di Battista *et al.* [25]. The labels specified on the Network Whiteboard are used by the Design Functions discussed in Chapter 5.

4.5.3 *Network Views*

We now present each of the Network Views. These are constructed in the following order as shown in Figure 4.33. We discuss each of these Network Views in detail. These Network Views capture the key information about the network design that we can use as an input to the later stages of our toolchain to generate the low-level device configurations. We will explain the Design Functions used to create these Network Views in Section 5.6.

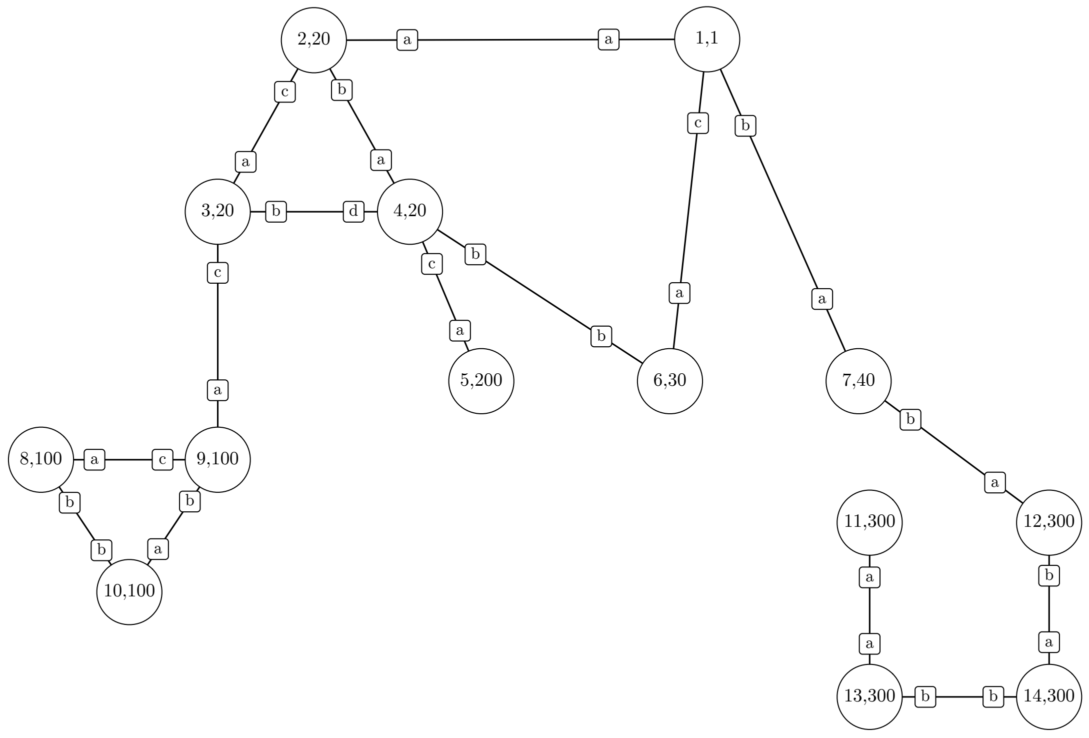


Figure 4.32: Network Whiteboard description for Simplified Small Internet Example. Network Elements are labelled with a tuple of *(identifier, asn)*

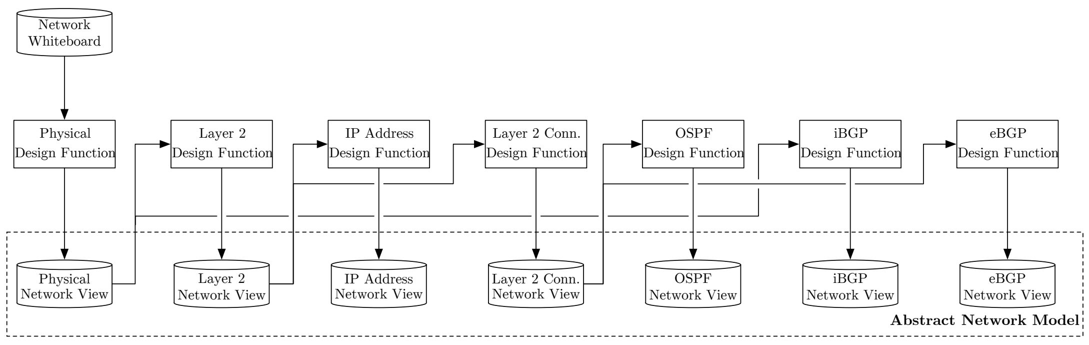
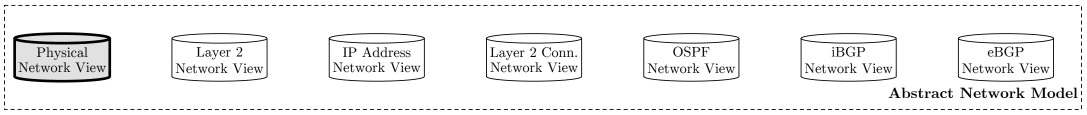


Figure 4.33: Flowchart of Network View construction using Design Functions for Simplified Small Internet Model

Physical Network View



The Physical Network View, shown in Figure 4.34 represents the Physical structure of the Network.

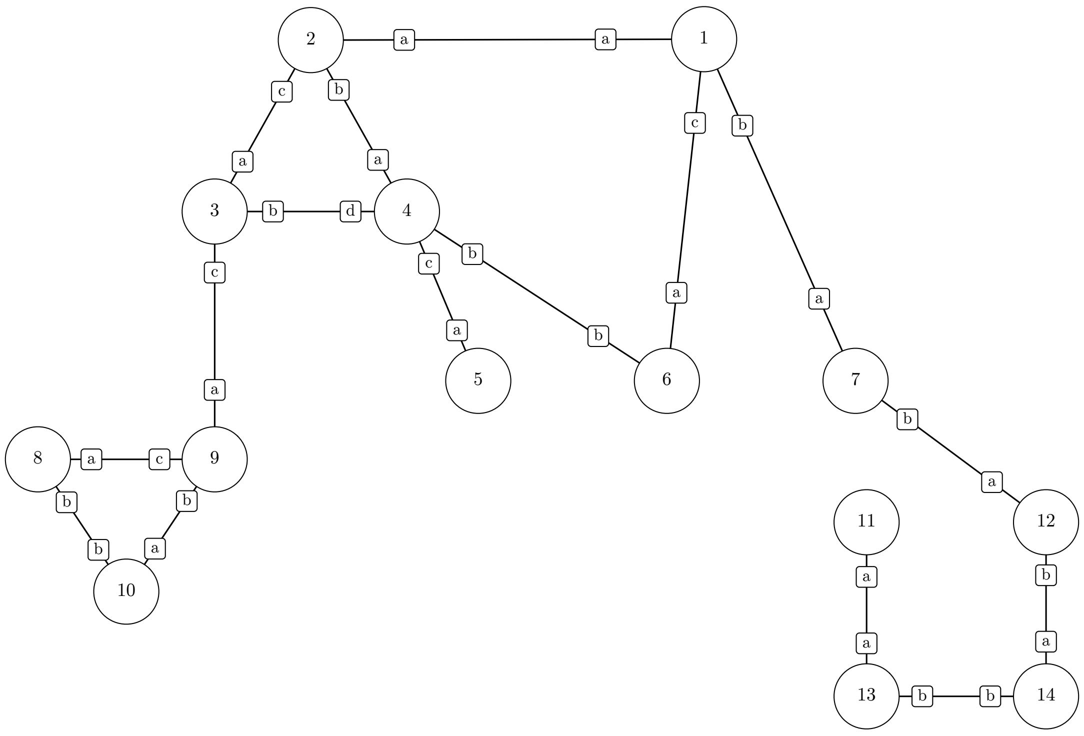
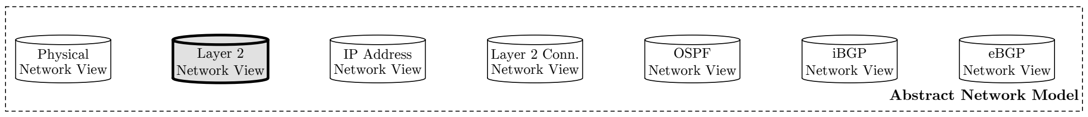


Figure 4.34: Physical Network View, where Network Elements represent routers, and Network Element Connections show physical connections.

The Physical Network View consists of the following: The set of Network Elements which are not Pseudo Network Elements. The set of Network Element Interfaces which have the Physical Role. The set of Network Element Connections which have the Physical Role.

The Physical View is therefore: all Network Elements which correspond to a physical network device; all Physical Network Element Interfaces; all Physical Network Element Connections.

Layer 2 Network View



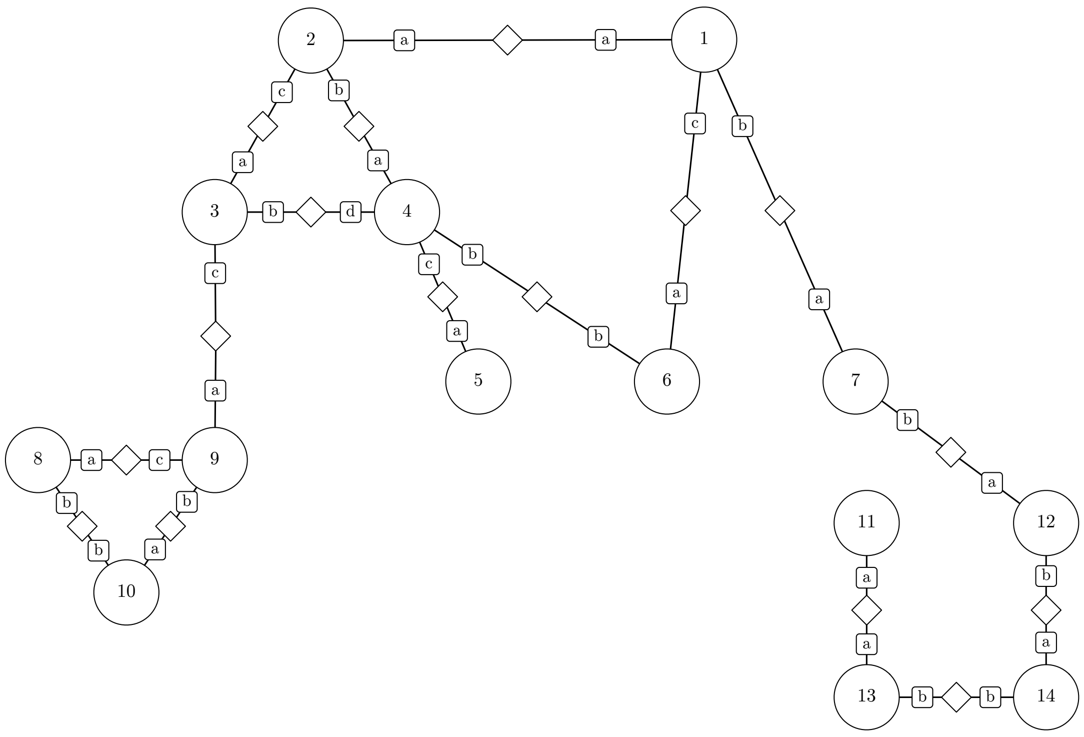
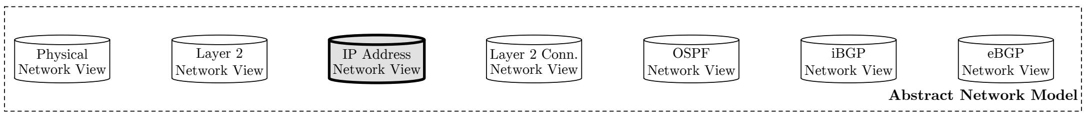


Figure 4.35: Layer 2 Network View. Network Elements show routers and Pseudo Network Elements show Broadcast Domains. Network Element Connections show Layer 2 connectivity.

The Layer 2 Network View shown in Figure 4.35 is used to derive the IP Addressing and Layer 2 Connectivity Network Views. It is based on the Physical View, and consists of the Network Elements from the Physical Network View and their Network Element Interfaces. Pseudo Network Elements are used to represent the Broadcast Domains. Since this example only contains point-to-point links between routers, the Broadcast Domain Pseudo Network Elements are created by splitting the Network Element Connections from the Physical View. The Network Element Connections in this Network View represent membership of the Broadcast Domain to which they are connected. For simplicity Broadcast Domain Network Element Interfaces are not shown in this example.

IP Address Network View



The IP Address Network View shown in Figure 4.36, represents the IP Addressing allocated to the network. Labels are allocated to the IP Address View to represent IP addresses.

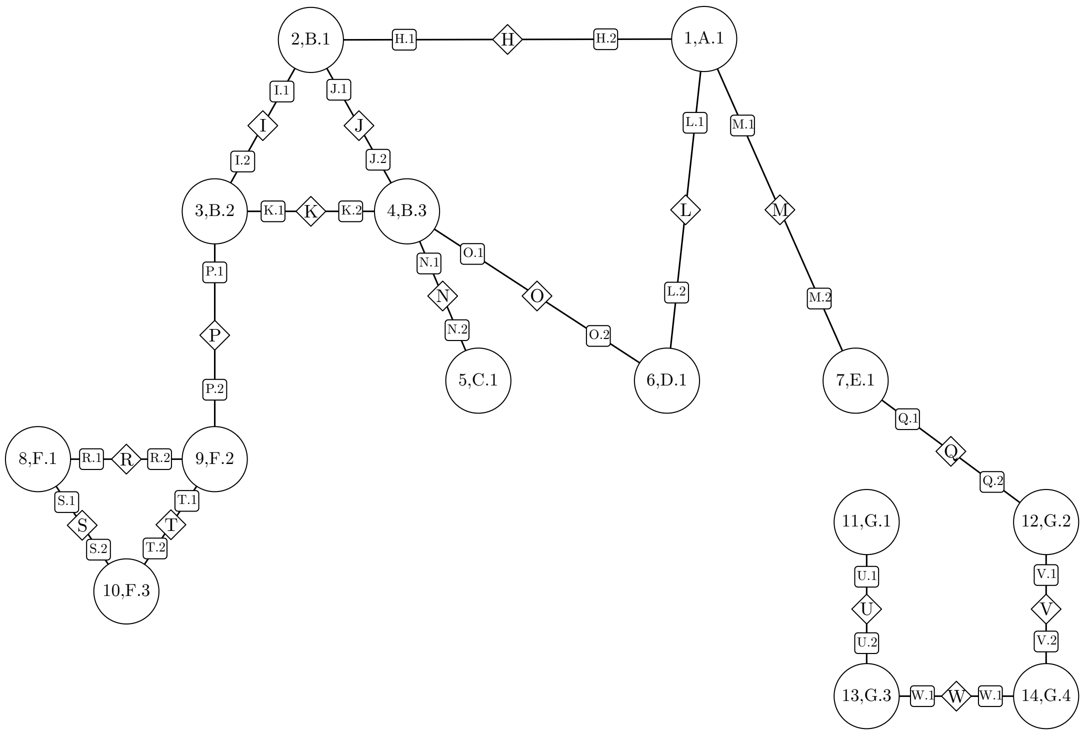


Figure 4.36: IP Address Network View, with Network Elements representing routers, and Pseudo Network Elements representing Broadcast Domains. Broadcast Domains are shown with their Subnet Label. Network Element Interfaces are labelled with their IP address, allocated from the Subnet to which they are connected.

These IP address label allocations are as follows. All routers are allocated a loopback address, from a subnet that has been allocated to each Autonomous System. All broadcast domain Pseudo Network Elements are allocated a subnet, from which individual addresses are allocated to each Network Element Interface of non-Broadcast Domain elements. All routers devices are allocated an individual IP address from the subnet to which they are connected. An example of the IP allocation process to Network Element Interfaces is shown in Figure 4.37. The IP Address blocks allocated in each autonomous system are summarised in Figure 4.38.

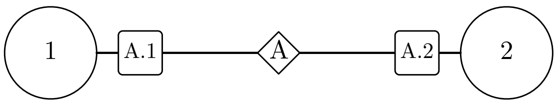


Figure 4.37: IP Allocation Example, showing Broadcast Domain Pseudo Network Element, and Network Element Interfaces

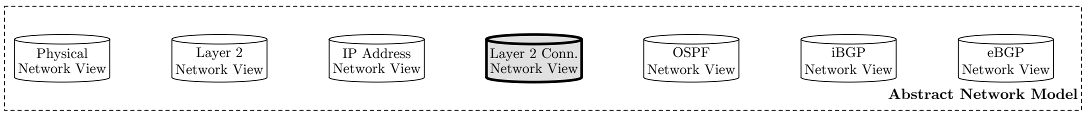
Loopback	Infrastructure
1: A	1:
20: B	20: I, J, K
30: D	30:
40: E	40:
100: F	100: R, S, T
200: C	200:
300: G	300: U, V, W

Figure 4.38: IP Address Blocks for each ASN for Loopback and Infrastructure Blocks.

The IP Address Network View is therefore:

- All Network Elements from the Layer 2 Network View.
  - Network Elements which have the Role of Router have a label used to represent their loopback IP address.
  - Network Elements which are Pseudo Network Elements with the Role of Broadcast Domain have a label used to represent the Subnet for which their neighboring Network Element Interface IPs belong.
- All Network Element Interfaces from the Layer 2 Network View, with a label to represent the IP address allocated.
- All Network Element Connections from the Layer 2 Network View. The Network Element Connections are used in the mapping of IP addresses to Network Element Interfaces, but do not hold any specific meaning for the compilation process.

Layer 2 Connectivity Network View



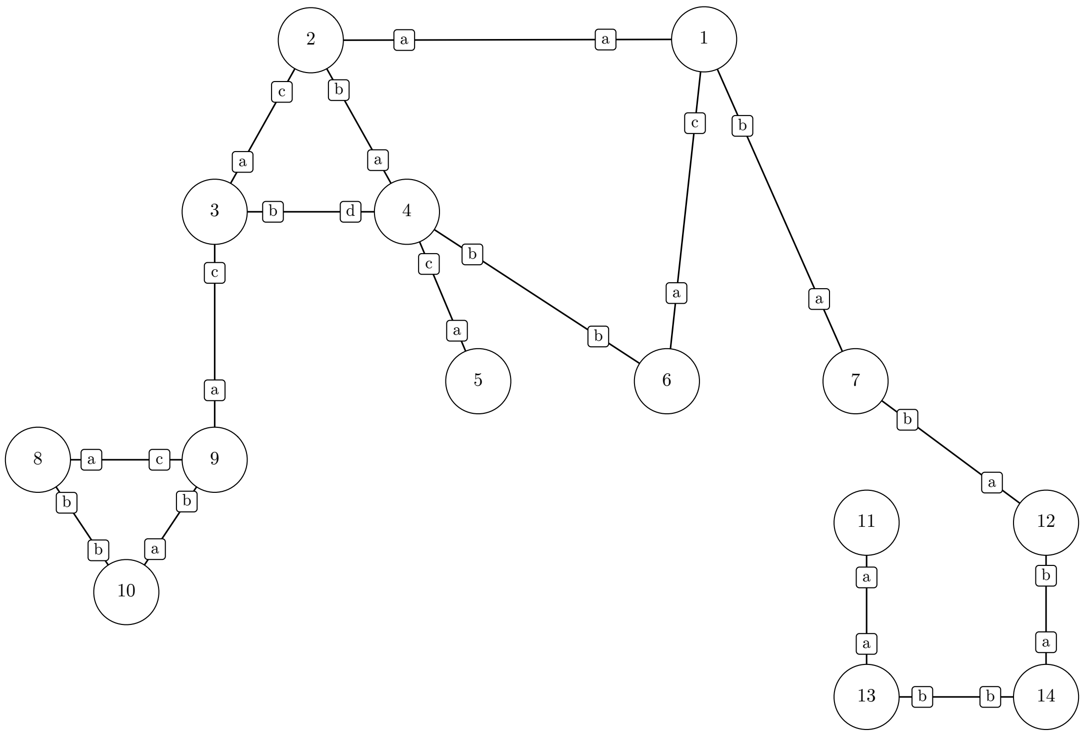
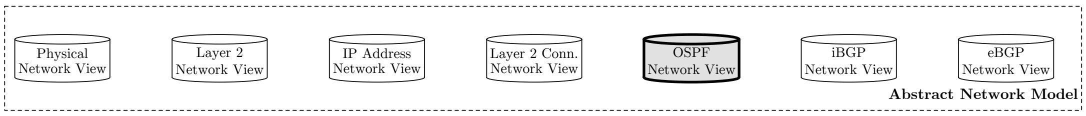


Figure 4.39: Layer 2 Connectivity Network View, where Network Elements represent routers, and Network Element Connections show layer 2 connections.

The Layer 2 Connectivity Network View, shown in Figure 4.39, shows the connectivity at Layer 2. It is formed by expanding the Layer 2 Pseudo Network Elements which represent Broadcast Domains. Since these Broadcast Domain Pseudo Network Elements were created from point-to-point Network Element Connections, this Network View will look the same as the Physical Network View. For more complicated topologies, such as with hubs or switches, the Layer 2 Connectivity Network View will not have a 1:1 relationship to the Physical Network View. We discuss such topologies in Section 5.7.1.

OSPF Network View



The OSPF Network View shown in Figure 4.40 represents OSPF adjacency relationships between routers in the network. The OSPF Network View contains the Network Elements which have the role of router. Informally, links are added in the OSPF Network View where there is a link in the Layer 2 Connectivity Network View which connects two routers that belong to the same ASs. More formally, Network Element Connections in the OSPF Network View exist where Network Element Connection in the Layer 2 Connectivity Network View and the Network Element Connection connects two Network Elements which have the same value for the *ASN* label.

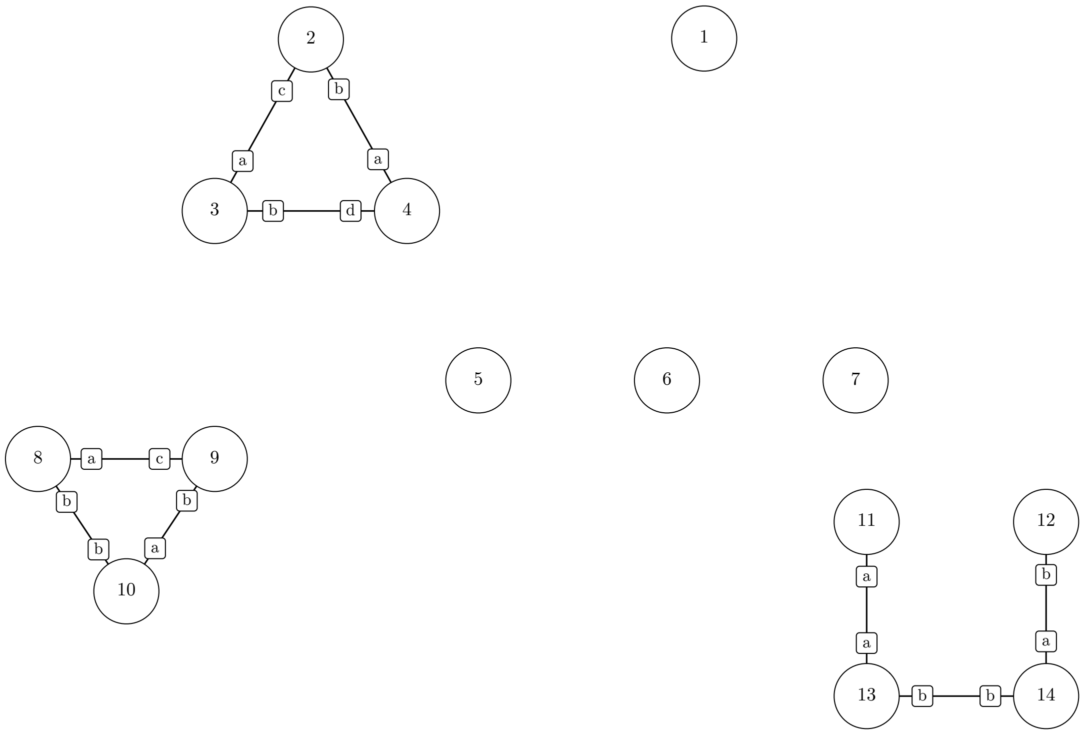
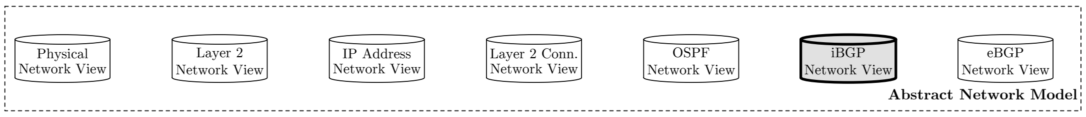


Figure 4.40: OSPF Network View. Network Elements represent routers, and Network Element Connections represent OSPF adjacencies.

iBGP Network View



The iBGP Network View shown in Figure 4.41 represents iBGP peering relationships between routers in the network. Informally, links are added in the iBGP Network View between two routers if they belong to the same AS. The iBGP Network View consists of the set of Network Elements which have the role of *router*, a set of Network Element Interfaces for these Network Elements, which iBGP session termination points. The set of Network Element Connections which represent iBGP peering sessions, bound to the iBGP session termination point Network Element Interfaces.

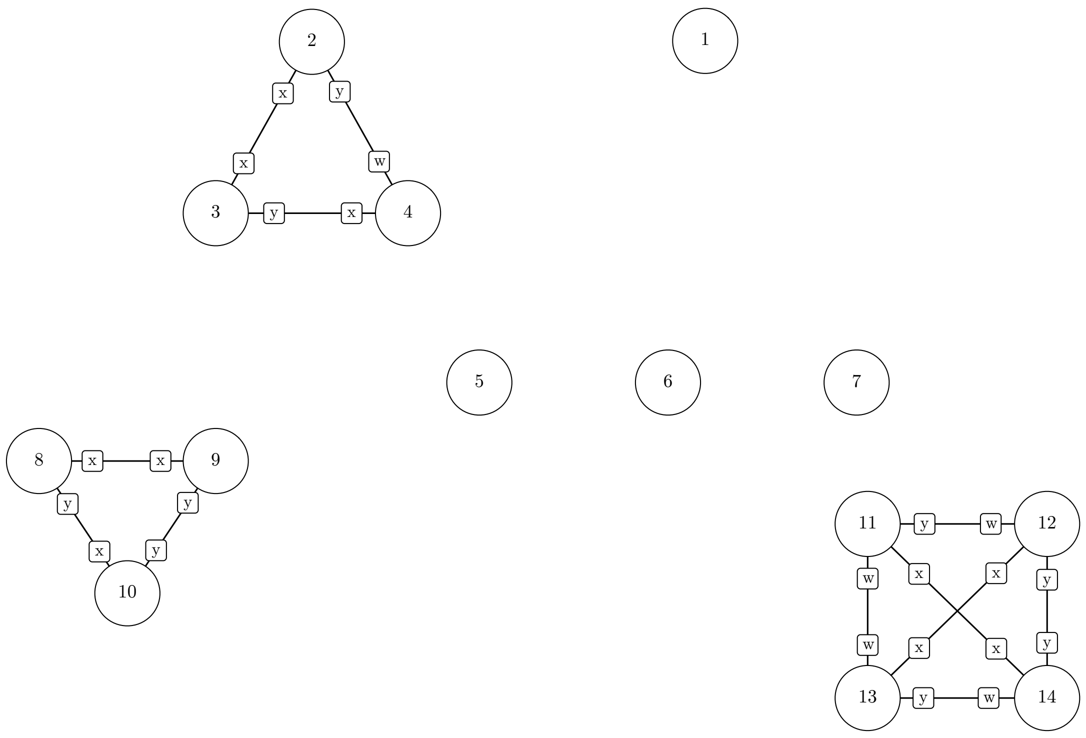
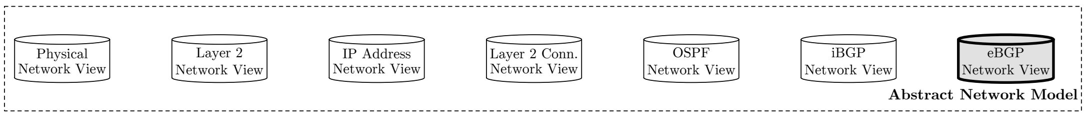


Figure 4.41: iBGP Network View. Network Elements represent routers, and Network Element Connections represent iBGP peerings. Each Logical Network Element Interface is associated to the Loopback Zero Network Element Interface of that Network Element (not shown).

eBGP Network View



The eBGP Network View shown in Figure 4.42 represents eBGP peering relationships between routers in the network. The eBGP Network View consists of Network Elements which have the role of router and their Network Element Interfaces. The Network Element Connections represent eBGP peerings. The Network Element Connections in the eBGP Network View exist where there is a Network Element Connection in the Layer 2 Connectivity Network View and the Network Element Connection connects two *router* Network Elements which have different values for their *ASN* label. This is the inverse of the links from the OSPF Network View.

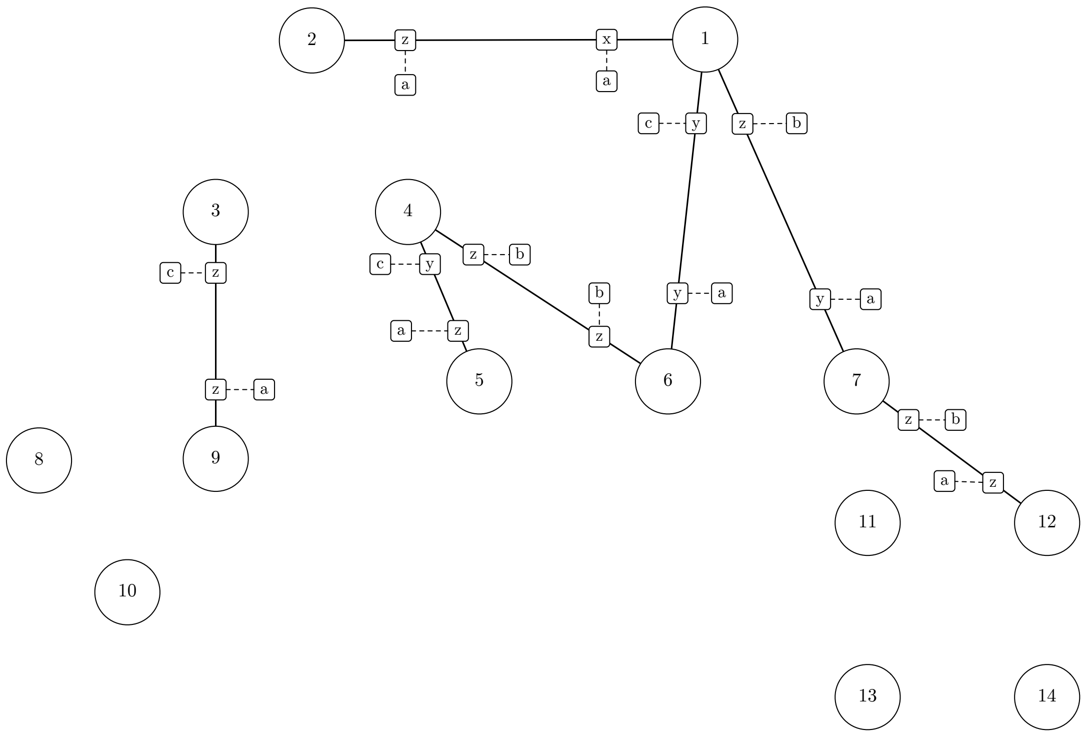
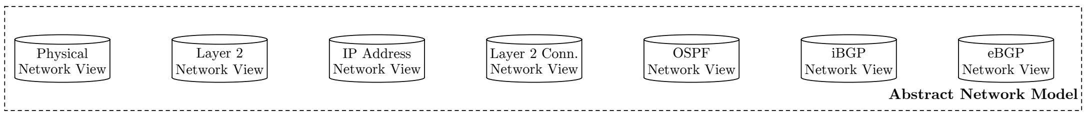


Figure 4.42: eBGP Network View. Network Elements represent routers and Network Element Connections represent eBGP peerings. Network Element Interfaces represent eBGP session endpoints, dashed lines show associations to physical Network Element Interfaces

4.5.4 *Abstract Network Model*



In Figure 4.43 we show how these Network Views in the Abstract Network Model are related conceptually.

4.5.5 *Summary*

This example has demonstrated how the approach of Network Whiteboard and Network Views can be used to capture the information required in network design. We have shown how a network topology consisting of multiple autonomous systems can be represented at a high-level using the Network Whiteboard. We then derived the Physical Network View, and the resulting Layer 2 and Layer 2 Connectivity Network Views. The Layer 2 Network View demonstrated the use of Pseudo Network Elements. It was used to derive the IP Addressing Network View, which showed labels on Network Elements and Network Element Interfaces to represent loopback and infrastructure IP Addresses. We also showed labels on the Network View itself to store the IP address blocks allocated to each autonomous system. The Layer 2 Connectivity Network View was used to derive the OSPF and eBGP Network Views. The eBGP Network View demonstrated how Logical Network Element Interfaces can be used to represent protocol endpoints. These were then bound to the Physical Network Element Interface that the BGP session was to be established on. Finally, the iBGP Network View was built from the autonomous system label on the Network Elements, and showed an algorithmic approach to establishing Network Element Connections. The iBGP Network View also used the Logical Network Element Interfaces. These were bound to the Loopback Logical Network Element Interface for the device. These Network Views show how Network Element Connections can be used to represent connectivity relationships between Network Elements depending on the context of the Network View.

This has shown how Network Views and their constituent elements can be used to formally capture key aspects of the network design process. We have shown a number of methods in which the Network Views can be derived, which we formalise in Chapter 5 using Design Functions.

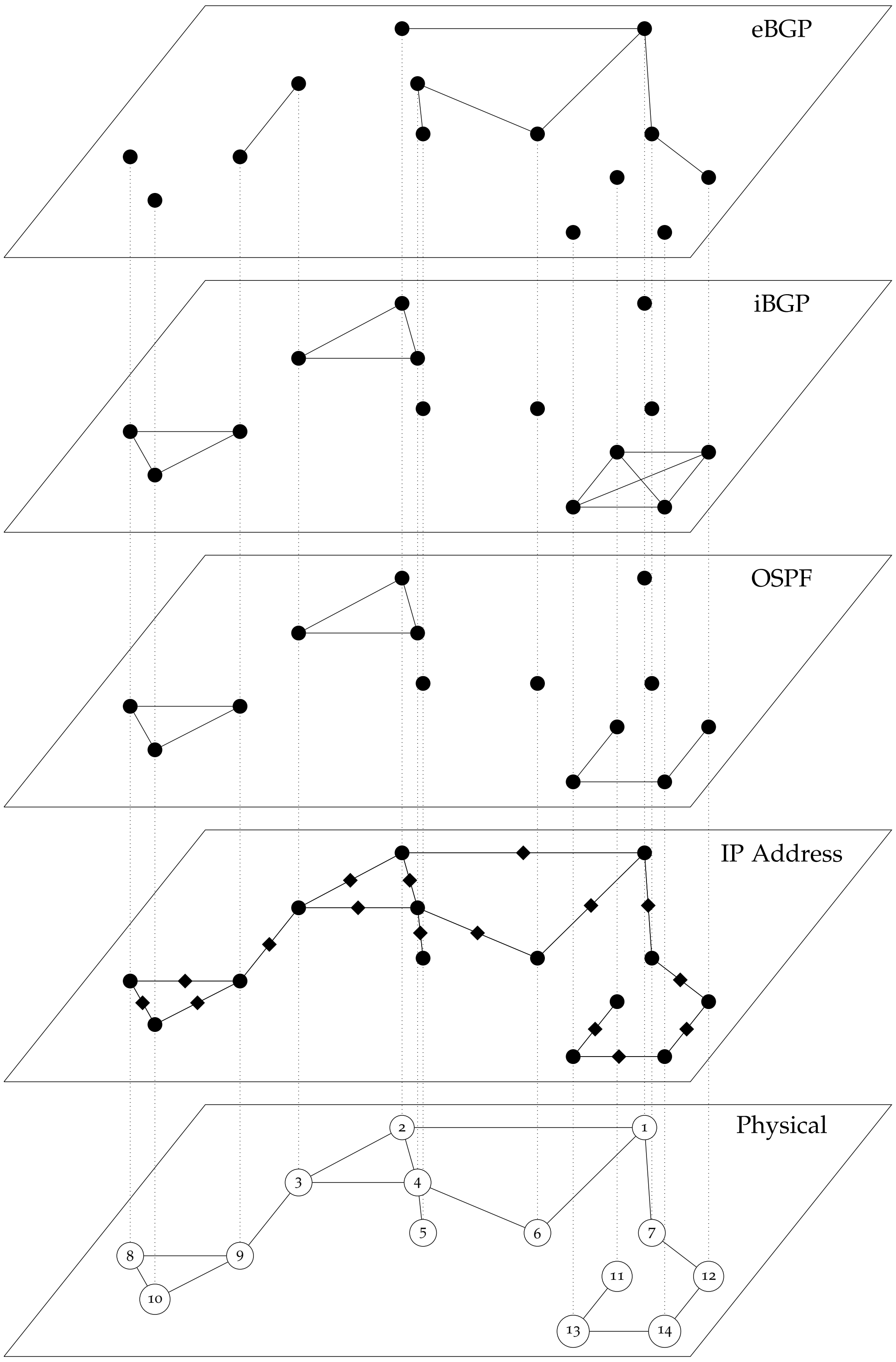


Figure 4.43: Cross-section illustration of the set of Network Views in the Abstract Network Model for the Simplified Small Internet Example. Vertical lines represent the association of Network Elements across Network Views. Each horizontal plane corresponds to a Network View. The Layer 2 Network Views have been omitted for visual simplicity.

4.6 DESIGN DETAILS

In this section we provide a brief overview of the design details describing how the approach of network views can be implemented using a graph theory based approach. For brevity we provide a general overview here, with are much more detailed discussion of the notation used, and the various functions defined, provided in Section D.1.

The core abstraction of our approach is the Network View. These can be represented using graphs, which are a set of nodes and edges, and can be denoted as $G = (N, E)$. We represent Network Views as such a graph, where the nodes correspond to Network Elements, and the edges correspond to Network Element Connections. We represent Network Element Interfaces by storing them on the Network Element, and using a pointer reference from the edge to the Network Element Interface on the node. This simplifies the task of accessing and manipulating network element interfaces across network views and simplifies the use of Network Element Connections to being used to represent the connectivity between Network Elements.

We therefore define a Network View θ to be $\theta = (\Pi, E, T, B, L)$, where the symbols are defined in Table 4.1.

Table 4.1: Table of element notation symbols

Element Type	Set Symbol	Element Symbol
Network Whiteboard	–	ω_i
Network View	Θ	θ_i
Network Element	Π	π_i
Network Element Connection	E	ϵ_i
Network Element Interface	T	τ_i
Network Element Interface Binding	B	b
Network Element Connection Distinguisher	δ	-

The Network Whiteboard, Network View, Network Element, Network Element Connection, and Network Element Interface are as previously defined in Section 4.4. The Network Element Interface Binding B has the form $(\epsilon_i, \{(\pi_j, \tau_j), (\pi_k, \tau_k)\})$ where ϵ_i is the Network Element Connection, τ_j is a binding of this Network Element Connection to Network Element Interface τ_j on Network Element π_j, and τ_k is a binding of this Network Element Connection to Network Element Interface τ_k for Network Element π_k. This is shown in Figure 4.44. The Network Element Connection Distinguisher is used to distinguish between multiple parallel Network Element Connections between the same pair of Network Elements. The set of labels, L is defined as $L = (L_\Theta, L_\Pi, L_E, L_T)$, and consists of the labels for the Network View itself, the Network Elements, the Network Element Connections, and the Network Element Interfaces. Some Network Element labels have scope across multiple Network Views, such as the role of the Network Element (such as router or switch), or the ASN to which it belongs. These could be stored on the Abstract Network Model itself. We define role access functions $\rho_\Pi(\Theta, \pi_i)$, $\rho_E(\Theta, \epsilon_i)$, and $\rho_T(\Theta, \tau_i)$ to access the role of a given Network Element, Network Element Connection, or Network Element Interface. We also define $\lambda_{ASN}(\Theta, \pi_i)$ to access the ASN of a given Network Element.

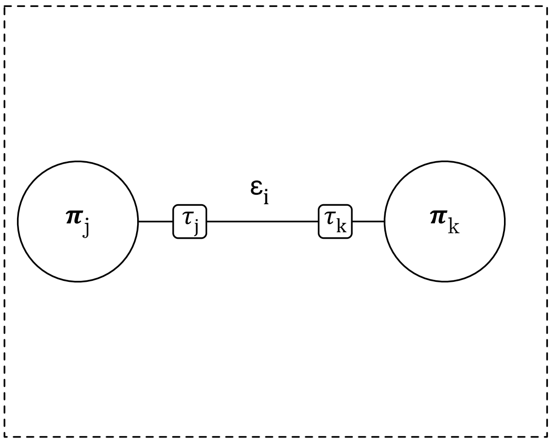


Figure 4.44: Symbols for the components of Network Views

The Abstract Network Model is the set of all Network Views. We can denote a generalised Network View as $\theta_i = (\Pi_i, E_i, T_i, B_i, L_i)$, and the Abstract Network Model is then denoted as $\Theta = \theta_1, \theta_2, \theta_3, \ldots$. A Network View can be named using subscript notation, such as θ_{phy}, θ_{ospf}, *etc.* An example of the use of this notation is provided in Appendix C for the Simplified Small Internet example topology.

Using this notation and structure, we can define Low-Level Primitives which access properties and modify these structures. These Low-Level Primitives can be composed with graph theory algorithms to form High-Level Primitives. We can then use these Low-Level Primitives and High-Level Primitives to form Design Functions, which provide a systematic approach to transforming the Network Whiteboard into the Network Views. We will discuss Low-Level Primitives, High-Level Primitives, and Design Functions in Chapter 5.

4.7 DISCUSSION

We now discuss some alternative ways of implementing the design choices made in the previous sections. We also acknowledge where design choices presented in this chapter have been influenced by previous work.

4.7.1 *Labels*

A role-based approach has been advocated by Buchmann *et al.* [13]. Our label use achieves many of the benefits advocated in this paper.

In using a graph-based representation, there are alternative choices as to what the nodes and edges represent. Tozal *et al.* [100] list some of these choices in their presentation on Network Layer Internet Topology Construction:

> Nodes: Autonomous Systems (ASes), Routers, Router Triangles, Interfaces, Subnetworks (Subnets)
>
> Edges: Policy-based connections, Subnets, Routers

In our approach, we have decided to make nodes represent network devices, referred to as Network Elements. The edges represent the connectivity between these devices. The role of the edge depends on the context of the Network View. We believe the choices made in this research are justified based on simplicity of conception and ease of implementation, as we will demonstrate in the following chapters.

4.7.2 *Interface Representation*

Interfaces are an important network construct, however they do not fit directly into graph theory. A model in which nodes represent devices presents a challenge for capturing interface configuration attributes.

One approach is to represent interfaces as nodes in the graph, where a device consists of a *device* node, and multiple *interface* nodes. The problem with this approach is the size of the graph grows dramatically as new devices are added, and limits the ability to express design policy as sets of nodes and edges: policy will need to distinguish the node type.

Another approach is storing the interface information on the edges, such as IP addresses or interface speed. This presents a problem with non-symmetric attributes such as individual IP addresses. A solution could be to use directed edges, but this also complicates expression: the protocol relationships may not be naturally expressed as a directed graph, and require an extra mapping step.

The approach we adopt is inspired by the suggestion made by [99] and has two components. The interfaces are stored inside the nodes, as a set. Similarly to nodes in the graph, we required a unique labelling for each interface within a node. The edges then contain set of pairs, which relate to the interface mapped. This approach offers a number of advantages. The information is stored on the node itself, signifying the transformation from the network model to the device level state. The task of mapping interfaces across layers in is greatly simplified. This approach allows for node level properties such as degree or neighbours, to also apply to interfaces, by filtering based on the $(node, interface)$ pair present on edges. This simplifies the design and verification steps which discuss in later chapters.

4.7.3 *Multiple Network Views*

Finally, we need to represent multiple topologies, such as physical, layer 2, OSPF, and BGP. While there may be a relationship between the topologies, their configuration is often independent.

Our approach represents each topology on a separate graph, with functions to work across multiple layers. This simplifies design policy: the presence of an edge in a graph can indicate adjacency, and also analysis, in that shortest paths and clustering algorithms can be applied directly, rather than needing to filter nodes and edges based on their layer.

These layers can be bound through uniqueness requirements. We require each node, representing a device, to have a globally unique label. This allows the node to be identified across layers, much like a primary and foreign key in a database. We also require interfaces to have a unique identifier within a node. This addresses the following point from Hares *et al.*:

> VT-TDM-REQ6: The topology model should allow association between components of different layers. For example, Layer 2 port may have several IPv4/IPv6 interfaces. The Layer-2 port and the IPv4/ IPv6 interfaces would have an association. (Hares *et al.* [44])

We place no such restriction on edges: their relationship across topologies can be found through the use of the nodes and interfaces they connect. This simplifies design policy which can form new relationships, such as a clique at layer 3, based on layer 2 properties, such as a switch. The relationship between these adjacencies can be found from the interface, which at layer 2 would be connected to the switch, but at layer 3 having an edge to each layer 3 host connected to that switch.

Other approaches such Neo4j [75], which has been used in network management represent the network with a single graph, in which edges are labelled to mark the role in which they play, such as a physical link or a BGP session. This can add overhead to each algorithm. We believe representing each topology independently provides greater clarity, and simplifies policy expression.

4.8 CONCLUSION

In this chapter we have presented an overview of our approach to automated network configuration generation from a high-level policy abstraction.

We introduced two key concepts of our approach, the Network Whiteboard, and Network Views. Both of these concepts build on the idea of a graph of Network Elements and Network Element Connections. Network Elements correspond to the individual devices which are to be configured. Network Element Connections correspond to the links between these devices. The meaning of the Network Element Connection depends on the topology view that we are looking at. Network Element Connections terminate at a Network Element Interface, which is an interface on a Network Element. Each of Network Elements, Network Element Connections, and Network Element Interfaces can be annotated through the use of labels. Our approach is based on using Network Element Connections and labels to represent configuration information on Network Elements and Network Element Interfaces. A network design can be expressed through use of a Network Whiteboard. This Network Whiteboard consists of Network Elements, Network Element Connections, and Network Element Interfaces, and labels, to capture a high-level network design policy.

This chapter also introduced the Simplified Small Internet Example, and gave examples of a number of Network Views, for the physical, layer 2, IP addressing, and for the routing protocols.

In this chapter we have shown how the Network Whiteboard abstraction addresses Research Question 1: *Is there a declarative representation of High-Level Network Policy that is also likely to be widely adoptable by current network practitioners?*, and the Network Views abstraction addresses Research Question 2: *What is a better intermediate representa-*

tion of network configuration that is based on graph theory, supports a well structured compilation process and provides clear separation of concerns?

In the next chapter we will explain how the Network Whiteboard is transformed into Network Views using Design Functions.

5

NETWORK VIEW TRANSFORMATION

5.1 INTRODUCTION

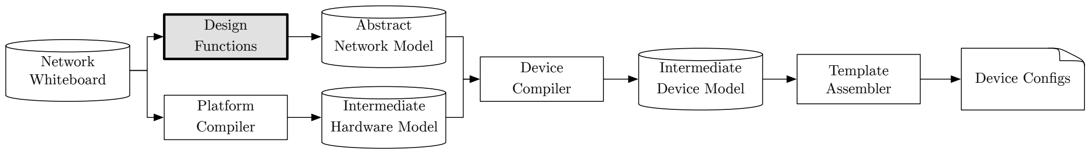


Figure 5.1: Toolchain highlighting Design Functions

In the previous chapter we introduced our toolchain and approach, and presented the details of our Network Whiteboard and Network Views abstractions. In this chapter we present the Design Functions which are used to derive the Network Views intermediate representations from the high-level Network Whiteboard input description. This aims to address Research Question 3: *How can declarative High-Level Network Policy descriptions be transformed into a graph-based intermediate format using a compiler that is extensible in terms of new types of policies and new protocols?*

Design Functions are used to transform the Network Whiteboard description into Network Views. These Network Views capture network designs at the network level of abstraction. This is shown in the toolchain in Figure 5.1. The flowchart in Figure 5.2 illustrates the relationship between Design Functions, the Network Whiteboard, and Network Views. This chapter describes the development of Design Functions, through Low-Level Primitives and High-Level Primitives, and provides a number of examples of the use of Design Functions.

This chapter is organised as follows. We first outline the key concepts involved in the construction of Network Views, including Design Functions, and the Primitives from which the Design Functions are composed. We then provide an overview of the Low-Level Primitives, and how they can be composed to form High-Level Primitives. Additional detail on these Low-Level and High-Level primitives is

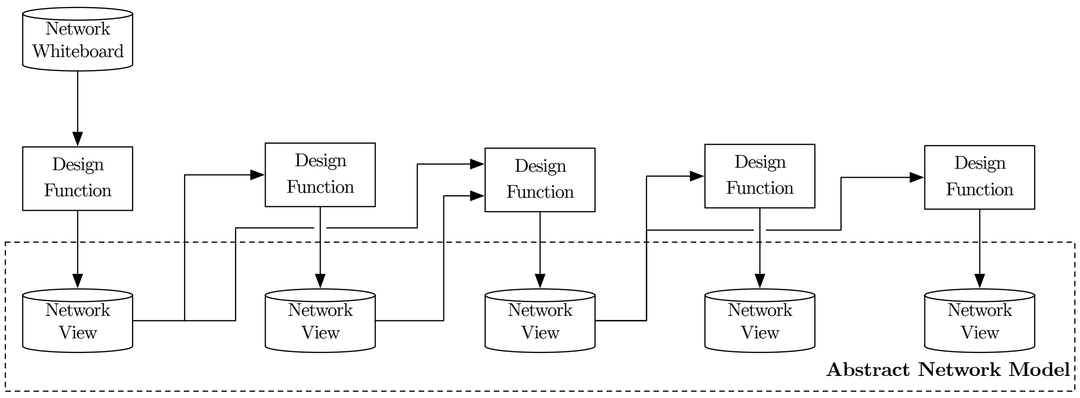


Figure 5.2: Overview Flowchart of Network View construction from Network Whiteboard using Design Functions.

provided in the appendices. From there, we provide a definition of the Design Function abstraction, and present common Design Function patterns, using the Low-Level and High-Level Primitives.

We then revisit the Small Internet example from Chapter 4, and provide a worked example of how the Network Views presented in Chapter 4 can be constructed through the use of Design Functions. We next examine a series of expanded Design Functions, demonstrating how they can be used to capture more complicated network design tasks.

Finally we discuss how each of these key components can be composed to provide an end-to-end solution that transforms the Network Whiteboard specification into the Network Views of the Abstract Network Model, suitable for compilation into Device Models. This compilation step is presented in Chapter 6.

5.2 KEY CONCEPTS

We first outline the key concepts used in the Network View Transformation process. Figure 5.3 presents a high-level overview of these concepts.

5.2.1 *Network View Construction*

In Chapter 4 we defined and showed examples of Network Views. This section will explain how these Network Views can be derived from the Network Whiteboard, using **Design Functions**.

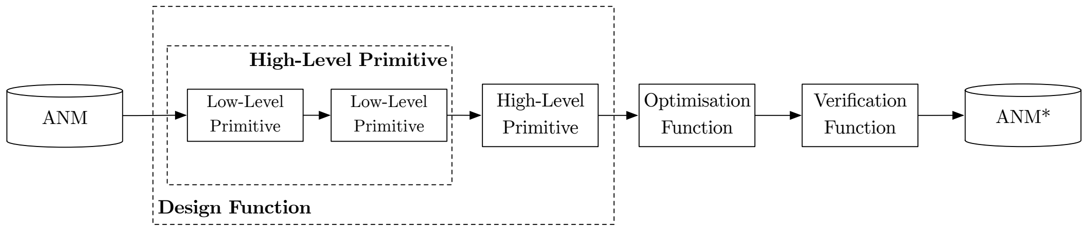


Figure 5.3: Flowchart showing Design Functions composed of High-Level Primitives and Low-Level Primitives

An important task in the Network Design stage is the construction of Network Views, which represent the topologies used in the configuration process. These Network Views are constructed using Design Functions, which apply Low-Level and High-Level Primitives. Design Functions describe how a Network View can be created by the transformation of either the Network Whiteboard or other Network Views.

5.2.2 *Primitives*

Primitives are a set of low-level operations on the Network Views.

Low-Level Primitives describe the most basic operations that can be performed on the elements (Network Elements, Network Element Connections, and Network Element Interfaces). These operations include addition and deletion, and the setting of labels.

For commonly repeated tasks, the Low-Level Primitives can be grouped into **High-Level Primitives**. High-Level Primitives also allow network configuration domain-specific concepts to be realised through the use of graph theory. This can simplify the application of graph theory fundamental concepts by network designers. These Low-Level and High-Level Primitives are then composed to build the **Design Functions**. These Design Functions are used to implement network design goals.

5.2.3 *Design Functions*

The Design Functions specify the rules to transform the Network Whiteboard into the Network Views, or to successively build Network Views from other Network Views. They decouple the Whiteboard specification of an individual network from the design patterns used to construct a network. Design Functions can be used to configure routing protocols topologies, generate IP addresses, or to run verification checks. Once the Design Functions have been run, the Network Views represent the desired network-wide state for the specific network to be configured. This state specification is independent of the syntax and semantics of the specific target device.

Together, the Design Functions and Network Views form the high-level network design component of our approach. Design Functions operate on the Network Whiteboard, and Network Views to create or modify Network Views. Design Functions can take four general forms: a Consistency Check, Construction Design, Verification, or Optimisation. In this chapter we discuss Construction Design Functions, which we will refer to as Design Functions for brevity. We discuss optimisation and verification in Chapter 9.

Consistency Check Functions operate on a Network View and are run before the Design Function, to check feasibility. They can also be incorporated as part of a Design Function, to apply a consistency check during the design process, such as after a transformation has been applied.

Construction Design Functions are the most common form of Design Function. Design Functions in this category are responsible for creating and modifying Network Views. These implement high-level designs to realise design policy, and their output is used in the device configuration generation process described in Chapter 6.

Optimisation Functions are a special case of Construction Design function, that are used to optimise the functionality of a single Network View, rather than constructing a new Network View. This optimisation could involve setting labels, or simplifying the connectivity that is described using the Network Element Connections.

Verification Functions operate on a Network View, and are used to verify the Network View is consistent, according to a set of consistency rules defined in the Verification Function. They do not create or modify Network Views. Their output can be a boolean to indicate an invalid Network View, with optional logging messages.

5.2.4 *Intermediate Network Views*

The purpose of Network Views is to provide a way to capture configuration topologies, which are then used in the configuration generation step. Most Network Views are directly used to capture configuration information. These Network Views are used to generate the Intermediate Device Models discussed in Chapter 6. However not all Network Views need to be used in the configuration generation step. **Intermediate Network Views** are constructed to assist in the formation of the final Network Views.

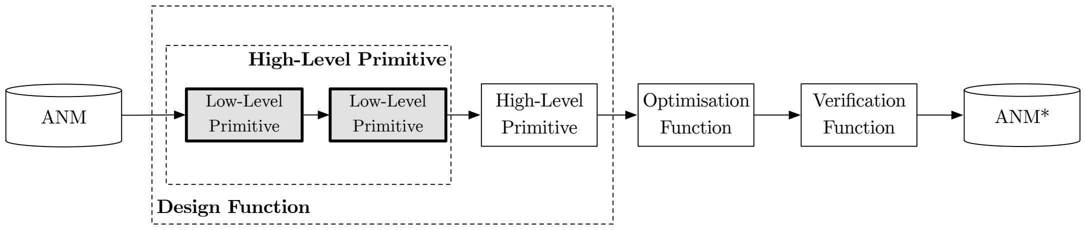
Intermediate Network Views are useful to capture network design goals or concepts, and to avoid repeated logic in the Design Functions. For example multiple IGP routing protocols may form adjacencies from element labels in the Layer 2 connectivity Network View. In this case, capturing the Layer 2 connectivity as an Intermediate Network View avoids repeated the same logic in each of the IGP Design Functions: each of the IGPs can build from the same common Layer 2 view.

This also allows a separation of concerns. By avoiding logic repetition, the Design Functions used to form the Intermediate Network View can be modified without needing to modify the Design Function of the subsequent Network Views. For instance, the Design Functions used to create the Layer 2 Network View could be expanded to handle VLANs and managed switches. If the Design Function produces a valid Layer 2 Network View, the IGP routing protocol Design Functions do not need to be altered. This allows expansion of the Layer 2 network functionality without requiring other Design Functions to be modified. Finally, this presents a clearer conceptual model to the network designer. As it is not used for the Device Compilation step, the Layer 2 Network View could be considered an Intermediate View.

5.2.5 *Conclusion*

This section has presented the key concepts in the Network View Transformation process, which is used to transform the Network Whiteboard specification into the Network Views in the Abstract Network Model. The Network Views are produced using Design Functions, which are composed of Low-Level Primitives and High-Level Primitives. We next examine the Low-Level Primitives in detail.

5.3 LOW-LEVEL PRIMITIVES



In this section we provide an overview of the Low-Level Primitives. These are the most fundamental functions in our approach, and directly manipulate the set of elements described previously in Section 4.6. A more detailed discussion of the Low-Level Primitives is presented in Section D.2. The *create* and *remove* Low-Level Primitives are responsible for adding or removing the elements from the appropriate set. These are shown in Table 5.1. The *label* Low-Level Primitives are summarised in in Table 5.2, which can be set on the various elements of the Network Whiteboard and Network Views. The *associative array* Low-Level Primitives, shown in Table 5.3, are used to store key-value pairs in associative arrays, such as for IP Address allocations in an autonomous system. Finally, the *properties* Low-Level Primitives are used to access the elements, such as all Network Elements in a given Network View, and their properties, such as the degree or testing if a Network Element Connection is parallel. These are shown in Table 5.4.

Table 5.1: Summary of *create* and *remove* Low-Level Primitives

	Creation
$\mathtt{create\_}\theta(\Theta)$	Creates a Network View θ_i within set Abstract Network Model Θ
$\mathtt{create\_}\pi(\Pi)$	Creates a Network Element π_i within set of Network Elements Π
$\mathtt{create\_}\epsilon(E)$	Creates a Network Element Connection ϵ_i within set of Network Element Connections E
$\mathtt{create\_}\tau\mathtt{\_}\pi(T)$	Creates a Network Element Interface within set of Network Element Interfaces T
	Addition
ADD_NE(θ_i, π_j)	Adds Network Element π_j to Network View θ_i
ADD_NEC(θ_i, ϵ_j)	Adds Network Element Connection ϵ_j to Network View θ_i
ADD_LNEI(θ_i, π_j)	Adds Network Element Connection ϵ_j to Network View θ_i
	Removal
REMOVE_NE(θ_i, π_j)	Removes Network Element π_j from Network View θ_i
REMOVE_NEC(θ_i, ϵ_j)	Removes Network Element Connection ϵ_j from Network View θ_i

Table 5.2: Summary of *labels* Low-Level Primitives

	Get Label
$\lambda_{\Theta}(\theta_i, label)$	Returns value of label $label$ for Network View θ_i
$\lambda_{\Pi}(\theta_i, \pi_j, label)$	Returns value of label $label$ for Network Element π_j in Network View θ_i
$\lambda_{T}(\theta_i, \pi_j, \tau_k, label)$	Returns value of label $label$ for Network Element Interface τ_k on Network Element π_j in Network View θ_i
$\lambda_{ASN}(\Theta, \pi_i)$	Returns ASN label for Network Element Pi_i in Abstract Network Model Θ
$\rho_{\Pi}(\Theta, \pi_i)$	Returns $role$ label for Network Element π_i in Abstract Network Model Θ
$\rho_{E}(\Theta, \epsilon_i)$	Returns $role$ label for Network Element Connection ϵ_i in Abstract Network Model Θ.
$\rho_{T}(\Theta, \tau_i)$	Returns $role$ label for Network Element Interface τ_i in Abstract Network Model Θ.
$\lambda_{M}(M, \pi_i, \tau_j, id)$	Returns label $label$ for Network Element Interface τ_j on Network Element pi_i in Intermediate Hardware Model M.
	Set Label
SET_LABEL_$\theta(\theta_i, k, v)$	Sets label k on Network View θ_i to value v
SET_LABEL_$\Pi(\theta_i, \pi_j, k, v)$	Sets label k on Network Element π_j in Network View θ_i to value v
SET_LABEL_T$(\theta_i, \pi_j, \tau_k, k, v)$	Sets label k on Network Element Interface τ_k of Network Element π_j in Network View θ_i to value v

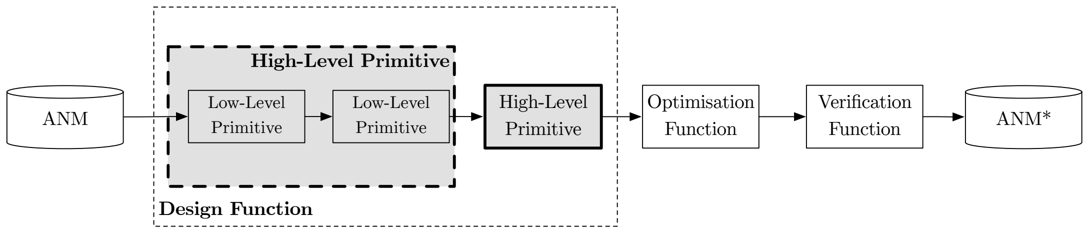
Table 5.3: Summary of *associative array* Low-Level Primitives

GET_KEY_VAL$(array, key)$	Returns the value stored by key key in associative array $array$.
SET_KEY$(array, label, value)$	Sets the value $value$ for key key in associative array $array$.

Table 5.4: Summary of *properties* Low-Level Primitives

	Network View
$NV(\Theta, name)$	Returns Network View θ_{name} in Abstract Network Model Θ
	Network Element
$\Pi(\theta_i)$	Network Elements for Network View θ_i
$\Pi_u(\theta_i, \epsilon_j)$	First (by sorting) Network Element connected by Network Element Connection ϵ_j in Network View θ_i
$\Pi_v(\theta_i, \epsilon_j)$	Second (by sorting) Network Element connected by Network Element Connection ϵ_j in Network View θ_i
$\Pi_{routers}(\theta_i)$	Network Elements with role of *router* for Network View θ_i
NEIGH_NE$(\theta_i, \pi_j, \tau_k)$	Returns neighbor Network Element for Network Interface τ_k on Network Element π_j in Network View θ_i
	Network Element Interface
$Degree_T(\theta_i, \pi_j, \tau_k)$	Returns degree of Network Element Interface τ_k on Network Element π_j in Network View θ_i
GET_LO_ZERO(θ_i, π_j)	Returns loopbacko Network Element Interface for Network Element π_j in Network View θ_i
NEC_NEI$(\theta_i, \epsilon_j, \pi_k$	Network Element Interface bound to Network Element Connection ϵ_j for Network Element π_k in Network View θ_i
$T(\theta_i, \epsilon_j, \pi_k)$	Alias for NEC_NEI function
$T_u(\theta_i, \epsilon_j)$	First (by sorting) Network Element Interface bound to Network Element Connection ϵ_j in Network View θ_i
$T_v(\theta_i, \epsilon_j)$	Second (by sorting) Network Element Interface bound to Network Element Connection ϵ_j in Network View θ_i
	Network Element Connection
$NEC(\theta_i, \pi_j)$	Network Element Connections connected to Network Element π_j in Network View θ_i
$E(\theta_i, \pi_j)$	Alias for NEC function
NEC_OTHER_NE$(\theta_i, \epsilon_j, \pi_k)$	Network Element Interface bound to Network Element Connection ϵ_j for remote connection Network Element π_k in Network View θ_i
PARALLEL(θ_i, ϵ_j)	Boolean result for whether Network Element Connection ϵ_j has parallel Network Element Connections (connects same Network Element pairs) in Network View θ_i

5.4 HIGH-LEVEL PRIMITIVES



High-Level Primitives combine the Low-Level Primitives and graph theory algorithms to provide higher-level functions, which can be used for common steps of network design tasks. These can be used together with the Low-Level Primitives to form the Design Functions which create Network Views.

5.4.1 *Summary*

We now present a brief summary of the High-Level Primitives used in the Design Functions of this chapter. These are shown in Table 5.5. They include *creation* High-Level Primitives, used to create Network Views, Pseudo Network Elements, or Logical Network Element Interfaces. They can also be used for associating Network Element Interfaces to Logical Network Element Interfaces, such as for iBGP and eBGP sessions. We also summarise the advanced modification and graph-theory based High-Level Primitives which we will present in this section. We present a more detailed discussion of High-Level Primitives in Section D.3.

Table 5.5: Summary of High-Level Primitives

Primitive	Description
	Creation and Modification
CREATE$\theta(\Theta, phy)$	Creates Network View phy in ANM Θ
CREATE_PNE(θ_i)	Creates a Pseudo Network Element in Network View θ_i
CREATE_RETURN_LNEI(θ_i, π_j)	Creates a Logical Network Element Interface on Network Element π_j in Network View θ_i
ASSOCIATE_NEI$(\theta_i, \pi_j, \tau_k, \tau_l)$	Associates a Network Element Interface τ_l to Logical Network Element Interface τ_k for Network Element π_j on Network View θ_i.
	Advanced Modification
GROUP	Groups Network Elements
SPLIT	Splits Network Element Connection
MERGE	Merges Network Elements
EXPLODE	Explodes Network Elements
	Graph-Theory Based
CONNECTED_COMPONENTS(G)	Returns a set of subgraphs of each connected component of the graph or subgraph G
SUBGRAPH(θ_i, X)	Subgraph of Network View θ_i contains Network Elements X, their Network Element Interfaces, and the Network Element Connections contained within X

5.4.2 *Graph-Theory Based High-Level Primitives*

As part of the Network View construction process, it is often necessary to establish certain properties between Network Views. An example is testing if two Network Elements in different Network Views, are equivalent. This test is used in both the Design Function and Compilation steps. We can also define High-Level Primitives to test if a label is the same for both Network Elements, or Network Element Interfaces, for a given Network Element Connection. These tests are defined in Algorithm E.59 and Algorithm E.60. Another class of High-Level Primitive queries properties of the network, such as $\Pi_{\mathrm{routers}}(\theta_i)$ which returns the elements Π, the set of Network Elements which have the role of *router*, for a given Network View θ_i. This is defined as Algorithm E.62.

Grouping

An important type of query used to construct Network Views is to establish the Network Elements fulfil a particular set of criteria. We refer to the output of such a query as a grouping. The GROUP High-Level Primitive returns a set Network Element sets which meet a set of grouping criteria. An example definition is shown in Algorithm E.57. The function signature is $\mathrm{grouping}(\theta_i, X, \mathrm{label})$ where θ_i is the Network View, X is the set of Network Elements to group, and label is the label to group by, such as *asn*. An example of the grouping output for the ASN value in the Small Internet Model discussed in Chapter 4 is shown in Figure 5.4.

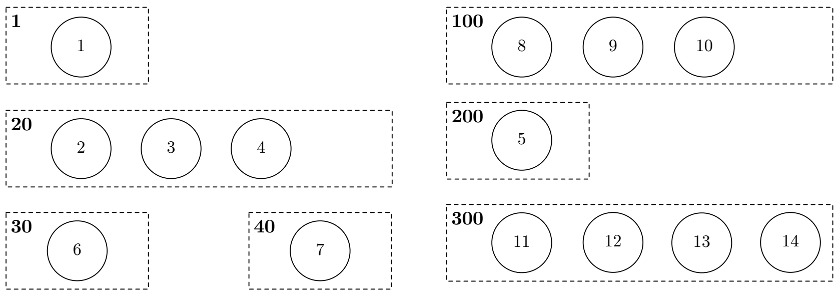


Figure 5.4: Example output for GROUP of Network Elements in Small Internet example by *asn* attribute

Boundary Network Elements

The BOUNDARY NETWORK ELEMENTS High-level Primitive is based off the *node_boundary* function from NetworkX [93]. For a given set of Network Elements, the Boundary Network Elements are those that have a Network Element Connection to a Network Element outside of the set. An example pseudo-code definition is shown in Algorithm E.58. The function signature is $\mathrm{boundary\_nodes}(\theta_i, X)$ where θ_i is the Network View, and X is the set of Network Elements to find the boundary Network Elements of.

For the topology shown in Figure 5.5, consider the set of Network Elements shown in the dashed box, (*r3*, *r4*, *r5*, *r6*). The Boundary Network Elements for these Network Elements would be *r3* and *r6*, as they have a Network Element Connection to a Network Element outside of the set (*r3* connects to *r1* and *r2*, and *r6* connects to *r7*).

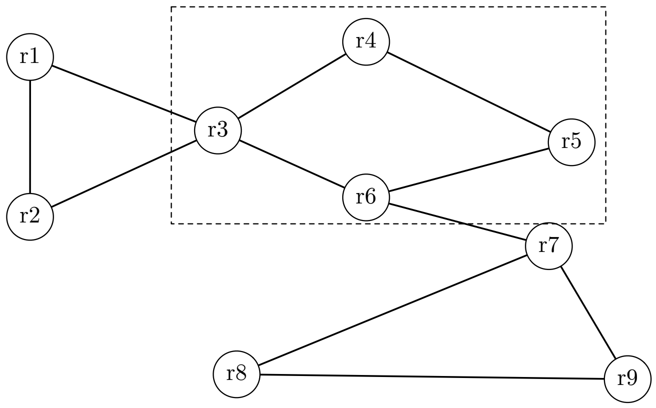


Figure 5.5: Example topology for BOUNDARY NETWORK ELEMENTS High-level Primitive. For clarity the Network Element Interfaces are not shown.

Connected Components

The CONNECTED COMPONENTS High-Level Primitive is useful in both Design Functions and Verification Functions. For Design Functions it can be used to break a Network View up into domains based on connectivity, such as determining switching domains when building the connection domains of managed switches for VLANs. This is discussed further in Section 5.7.2. It can also be used in Verification Functions, which we discuss in Section 9.3.3, to confirm full connectivity within a set of Network Elements. For instance in the OSPF Network View, we can verify that there is full connectivity between the nodes in each autonomous system.

This High-Level Primitive can be implemented using a Depth-First Search (DFS) or Breadth-First Search (BFS) algorithm. A node in a graph (Network Element in a Network View) is first selected, and a DFS/BFS performed from the node. This then gives the set of nodes that form the first connected component. We then consider the set of nodes which have not been visited, and select a node from this set, and then perform the DFS/BFS algorithm, giving the set of nodes forming the second connected component. This process is repeated until all nodes have been visited. A pseudo-code definition is shown in Algorithm E.64.

For the example topology shown in Figure 5.6, there would be two sets of connected Network Elements. The first set would contain (*r1*, *r2*, *r3*, *r4*, *r5*, *r6*), and the second set would contain (*r7*, *r8*, *r9*).

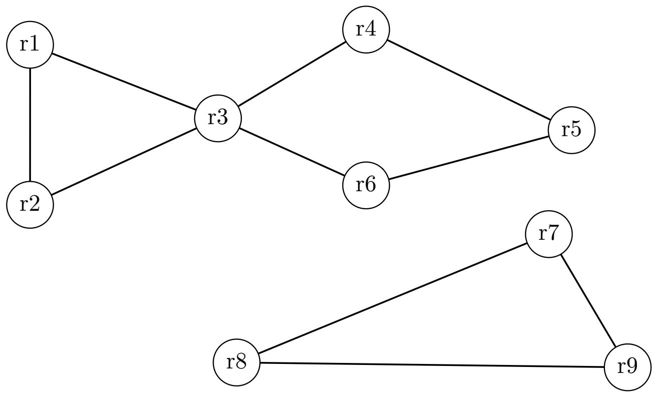


Figure 5.6: Example topology for CONNECTED COMPONENTS High-level Primitive. This topology has two connected components. For clarity the Network Element Interfaces are not shown.

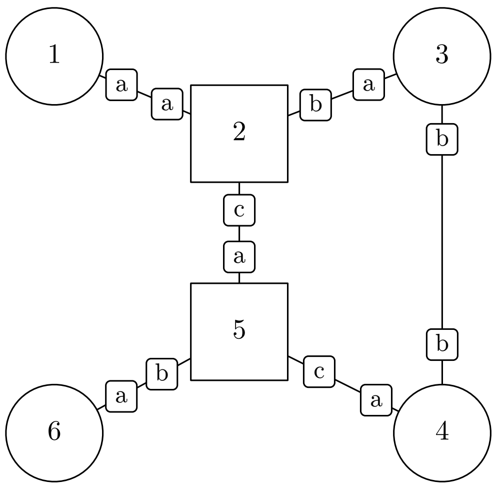
Subgraphs

In graph theory, a *subgraph* can be defined as follows. Let H be a subgraph of a graph G. Then the set of nodes in H is a subset of the nodes in G, and the set of edges in H is a subset of the edges in G. The subgraph concept can be used in Design Functions to break a Network View into a smaller graph whilst retaining connectivity. They can be used as an extension of the grouping function, where the subgraph provides both a set of nodes and the connectivity between these nodes. This is useful to be able to use Low-Level Primitives such as NEIGHBOURS or DEGREE on the Network Elements of the subgraph.

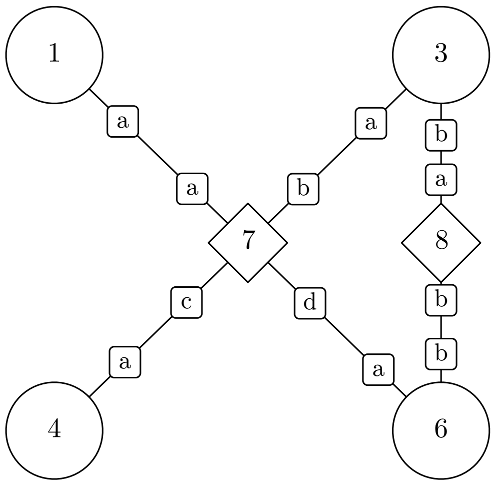
5.4.3 *Advanced Topology Modification High-Level Primitives*

Introduction

In network design, it is common to want to modify the structure of a Network View. For example, when creating a Layer 2 Network View, point-to-point links are split to place a broadcast domain Pseudo Network Element between them. The merging function is also used when constructing the Layer 2 Network view. We need to merge multiple connected switches together forming a single broadcast domain. We also need to explode Network Elements when generating a connectivity view, such as building the Layer 2 connectivity Network View, from the Layer 2 Network view. In this case, the Layer 2 Broadcast Domain Pseudo Network Elements, are exploded to form the Layer 2 Connectivity Network View. These functions can be combined to transform a the Network View shown in Figure 5.7a to the Network View shown in Figure 5.7b.



(a) Combined transform: before



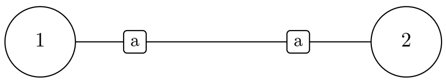
(b) Combined transform: after

Figure 5.7: Example of combined transformation High-Level Primitives

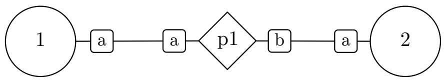
The High-Level Primitives defined in this section are SPLIT, GROUP, and EXPLODE.

Split

The SPLIT High-level Primitive takes a Network Element Connection, and splits it by placing a new Pseudo Network Element in the middle. This Pseudo Network Element automatically has new Network Element Interfaces created for it. Importantly, the Network Element interface bindings from the two original Network Elements are retained. An example of the use of this Primitive is in creating the Layer 2 Network View, where point-to-point links are split to place a broadcast domain Pseudo Network Element between them. We provide further details of this operation in Section 5.6.2. A pseudo-code definition of the SPLIT High-Level Primitive is provided in Algorithm 5.1, and an example in Figure 5.8.



(a) SPLIT: before



(b) SPLIT: after

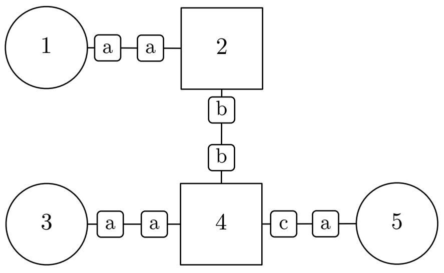
Figure 5.8: Example of SPLIT High-Level Primitive

Algorithm 5.1 Definition for SPLIT High-Level Primitive

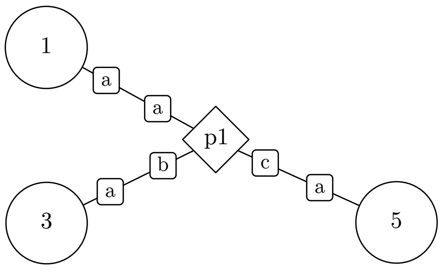
```
function SPLIT($\theta_i, X$)
    $\theta'_i \leftarrow \theta_i$
    for $\epsilon_i \in X$ do
        $(\theta'_i, p) \leftarrow$ CREATE_PNE($\theta'_i$)    ▷ Create Pseudo NE
        $(\theta'_i, \tau_{p1}) =$ CREATE_RETURN_LNEI($\theta'_i, p$)    ▷ Add logical NEI to p
        $(\theta'_i, \tau_{p2}) =$ CREATE_RETURN_LNEI($\theta'_i, p$)    ▷ Add logical NEI to p
        $\pi_u = \Pi_u(\theta_i, \epsilon_i)$
        $\pi_v = \Pi_v(\theta_i, \epsilon_i)$
        $\tau_u = T(\theta_i, \epsilon_i, \pi_u)$
        $\tau_v = T(\theta_i, \epsilon_i, \pi_v)$
        $\theta'_i \leftarrow$ ADD_NEC($\pi_u, \tau_u, p, \tau_{p1}$) ▷ Add NEC from u to p, keep u NEI binding
        $\theta'_i \leftarrow$ ADD_NEC($\pi_v, \tau_v, p, \tau_{p2}$) ▷ Add NEC from v to p, keep v NEI binding
        $\theta'_i \leftarrow$ REMOVE_NEC($\theta'_i, \epsilon_i$)    ▷ Remove the original NEC
    end for
    return $\theta'_i$
end function
```

Merge

The MERGE High-level Primitive takes two Network Elements and merges them into a single Pseudo Network Element. Importantly, the bindings of the neighbouring Network Element Interfaces for these merge Network Elements are retained. This High-Level Primitive could be applied multiple times to merge multiple Network Elements. In Figure 5.9, the binding of Network Element Interface *a* on Network Element 1, Network Element Interface *a* on Network Element 3, and Network Element Interface *a* on Network Element 5, have been retained, as is shown in Figure 5.9b.



(a) MERGE: before



(b) MERGE: after

Figure 5.9: Example of MERGE High-Level Primitive

This High-Level Primitive is also used when constructing the Layer 2 Network view. It is used to merge multiple connected switches together forming a single broadcast domain. This operation is described in further detail in Section 5.6.2. A pseudo-code definition of the MERGE High-Level Primitive is provided in Algorithm 5.2, and is similar to the graph-theory concept of edge contraction described in [108, page 23]

Algorithm 5.2 Definition for MERGE High-Level Primitive

```
function MERGE($\theta_i, X$)
    $\theta'_i \leftarrow \theta_i$
    $G \leftarrow$ SUBGRAPH($\theta_i, X$)
    $C =$ CONNECTED_COMPONENTS($G$)
    for $c \in C$ do                                        ▷ Each set of connected components
        $(\theta'_i, p) \leftarrow$ CREATE_PNE($\theta'_i$)                    ▷ Create Pseudo NE
        for $x \in c$ do                                         ▷ Each NE in component
            for $\epsilon \in E(x)$ do                                 ▷ Each NEC for NE x
                $\pi_{remote} =$ NEC_OTHER_NE($\theta_i, \epsilon_j, x$)         ▷ Far-end NE
                if $\pi_{remote} \notin c$ then        ▷ Far-end NE not in component
                    $(\theta'_i, \tau_p) =$ CREATE_RETURN_LNEI($\theta'_i, p$)       ▷ Add logical NEI to p
                    $\theta'_i \leftarrow$ RECONNECT_NEC($\epsilon, \pi_{remote}, p, \tau_p$)       ▷ Reconnect NEC to p
                else
                    $\theta'_i \leftarrow$ REMOVE_NEC($\theta'_i, \epsilon$)   ▷ Internal NEC, remove
                end if
            end for
            $\theta'_i \leftarrow$ REMOVE_NE($\theta'_i, x$)                      ▷ Remove the NE
        end for
    end for
    return $\theta'_i$
end function
```

Explode

The EXPLODE High-level Primitive takes a Network Element, and explodes this, to form a full mesh of connectivity between all of the Network Elements connected to this exploded element. Importantly, the Network Element Interface bindings of the Network Elements that originally connected to the exploded element, are retained.

If the original exploded Network Element is connected to more than two Network Elements, then the retained Network Element Interface bindings, will result in there being more than one Network Element Connection for each Network Element Interface. The example shown in Figure 5.10 illustrates this, where Network Element *1* is connected to Network Element *3* with the Network Element Interface binding *a*. Network elements 2 and 4 also connect with their Network Element Interface binding *a*. Network Element *5* connects with Network Element Interface binding *b*.

We then apply the EXPLODE High-level Primitive to Network Element 3. The result is Network Element Connections being established

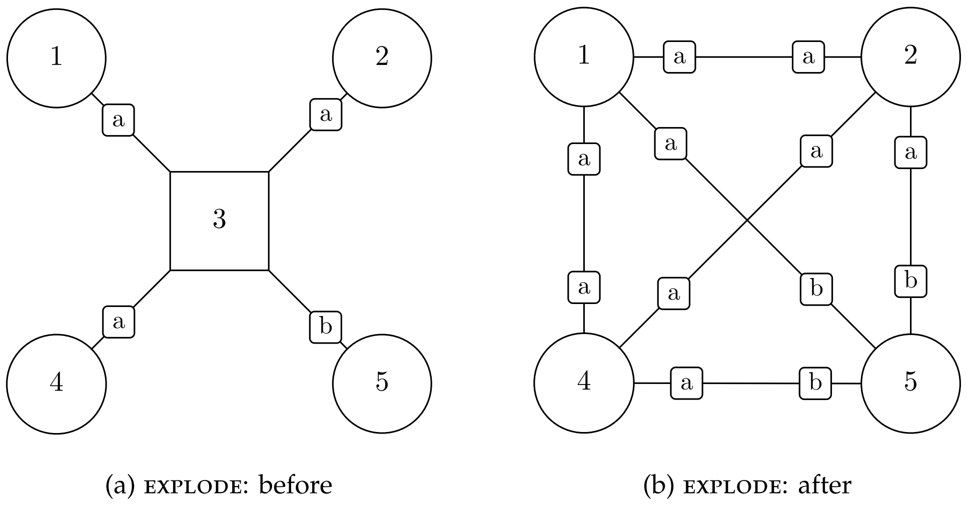


Figure 5.10: Example of EXPLODE High-Level Primitive. Network Element Interface bindings have been retained, such as for *b* from Network Element 5.

between each of Network Elements 1, 2, 4, and 5. We can see that the Network Element Interface bindings have been retained. Network Element 1 connects to2, 4 and 5, all through the Network Element Interface binding *a*, which was the Network Element interface binding of the original connection between Network Element 1 and Network Element 3. Similarly, Network Elements 2 and 4 are also connected through Network Element Interface binding *a*.

Network element interface 5 was originally connected through Network Element Interface binding *b*, so the connection from Network Element 5 to Network Elements 1, 2 and 4 are through Network Element Interface binding *b*.

This High-level Primitive is used when generating a connectivity view, such as building the Layer 2 Connectivity Network View from the Layer 2 Network View. In this case, the Layer 2 Broadcast Domain Pseudo Network Elements are exploded to form the Layer 2 Connectivity Network View. We show this in use in Section 5.6.4.

Some protocol require a special low-level configuration setting, such as if a session, or adjacency is multipoint. Such a label can be set on the Network Element Interface bindings created by the EXPLODE function, by applying this label, if the degree Low-Level Primitive of a Network Element Interface is greater than one. In the example shown in Figure 5.10, each of the Network Element Interfaces are of degree one. However, once we have applied the explode function, the Network Element Interface binding of Network Elements 1, 2 and 4, and the Network Element Interface binding *b* of Network

Element 5, are all of degree three. Therefore we could place a label on these Network Element Interfaces indicating that they are multipoint, which could then be used in setting the appropriate configuration parameters in the compilation process discussed in Chapter 6. This approach allows for the variety of Layer 2 connectivity methods to be handled in a way that is consistent when constructing the various routing protocol Network Views. A pseudo-code definition of the EXPLODE High-Level Primitive is provided in Algorithm 5.3.

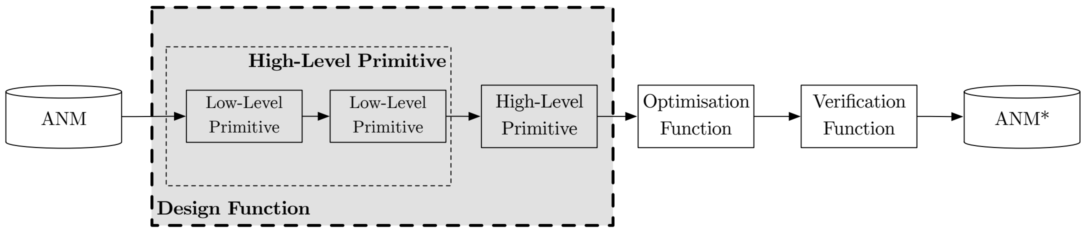
Algorithm 5.3 Definition for EXPLODE High-Level Primitive

```
function EXPLODE($\theta_i = (\Pi, E, T, B, L), X$)
    $\theta_i' \leftarrow \theta_i$
    for $x \in X$ do
        $P \leftarrow \varnothing$                                              ▷ Initialise set of pairs
        for $\epsilon_j \in E(\theta_i, x)$ do                                 ▷ All NECs for this NE
            $\pi_{remote} =$ NEC_OTHER_NE$(\theta_i, \epsilon_j, x)$               ▷ Far-end NE
            $\tau_{remote} =$ NEC_NEI$(\theta_i, \epsilon_j, \pi_{remote})$        ▷ Far-end NEC
            $p \leftarrow (\pi_{remote}, \tau_{remote})$                          ▷ Store this pair
            $P \leftarrow P \cup p$
        end for
        while $|P| > 1$ do                                   ▷ While still at least one pair
            $(\pi_i, \tau_i) \leftarrow P.pop()$                                 ▷ Choose next pair
            for $(\pi_j, \tau_j) \in P$ do                  ▷ For all other remaining pairs
                if $\pi_i \neq \pi_k$ then                                     ▷ No self-loops
                    $\theta_i' \leftarrow$ ADD_NEC$(\pi_i, \tau_i, \pi_j, \tau_j)$     ▷ Add NEC, retain
NEI bindings
                end if
            end for
        end while
    end for
     return $\theta_i'$
end function
```

5.4.4 *Conclusion*

In this section we have described the High-Level Primitives used to create, modify, query and remove elements from Network Views. In the next section we show how these Primitives are assembled into the Design Functions which are used to create Network Views.

5.5 DESIGN FUNCTIONS



A Design Function is an assembly of Primitives which are used to construct a Network View. In this section we describe seven Design Functions to construct views for common network design problems. As part of the extensible framework for this tool, users may construct further Design Functions depending on their design requirements.

We introduce these Design Functions through an example. The number of Design Functions a user will require is determine by the capabilities of the network design for which they are constructing a Network View. For example, to configure the local area network without IP Addressing or routing, only the Physical Design Function would be required.

5.5.1 *Example of Design Function Composition*

An example of a Design Function composed of Low-Level Primitives and High-Level Primitives is shown in Figure 5.11. This hypothetical example illustrates how the Primitives can be combined to form a Design Function. We provide a more realistic example of Design Functions in the next section, when we explain how the Network Views for the Simplified Small Internet Example can be derived.

For this example, we create a *Network View 2* (θ_2) based on the structure and labels of *Network View 1* (θ_1). The first step is to create the new Network View. We then select and filter the Network Elements from (θ_1). For instance this could be only selecting the router Network Elements from a mixed device Network View. We then add the filtered Network Elements to (θ_2). This also adds the Network Element Interfaces of these Network Elements. Next we select and filter the Network Element Connections from (θ_1). This could filter out connections that are labelled with a special designated role such as a VPN, leaving only the physical connectivity. We then add these filtered Network Element Connections to (θ_2). The next step in this example is to use the SPLIT High-Level Primitive on these Network

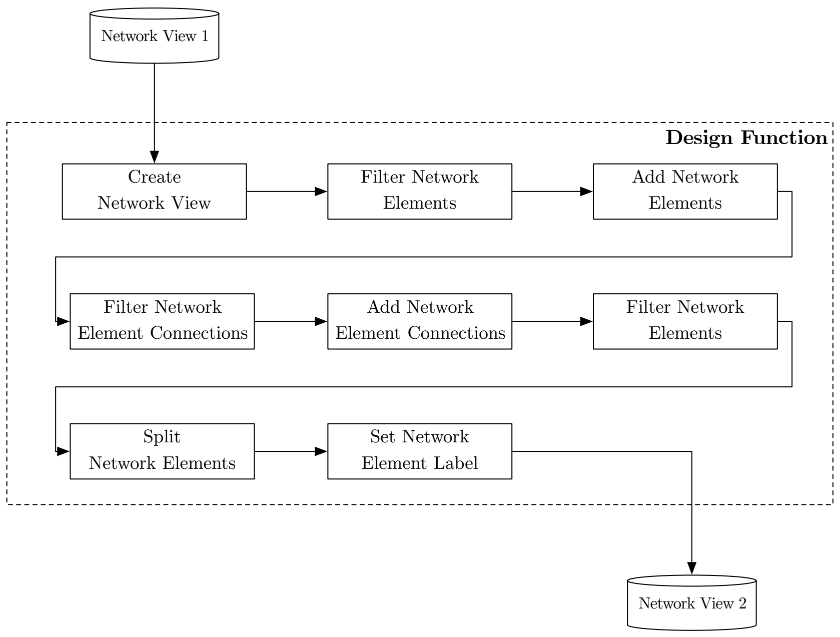
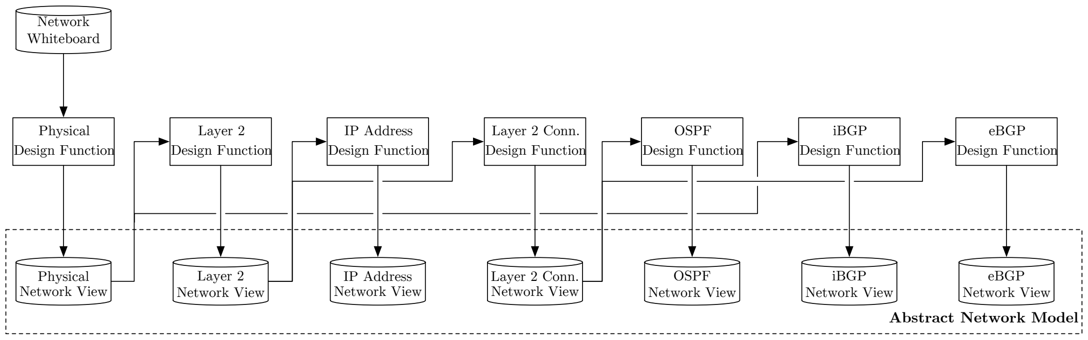


Figure 5.11: Composition of Low-Level and High-Level Primitives to build a Design Function

Element Connections, to create Pseudo Network Elements. The final step is to apply a label to a set Network Elements, such as to mark the Pseudo Network Elements as having a special role, such as being a broadcast domain. This completes the network design steps to create the *Network View 2* (θ_2).

This simplified example shows how the Low-Level and High-Level Primitives can be composed to form a Design Function, which is used to transform *Network View 1* to create *Network View 2* according to network design rules. We now demonstrate how this approach can be used in our toolchain to create the Network Views for the Simplified Small Internet Model.

5.6 SIMPLIFIED SMALL INTERNET DESIGN FUNCTIONS



This example shows how Design Functions that are required to create the Network Views discussed in Section 4.5, and reproduced below in Figure 5.12.

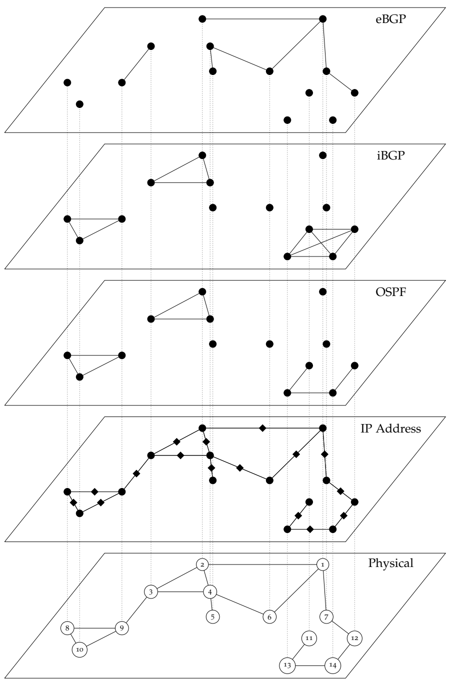
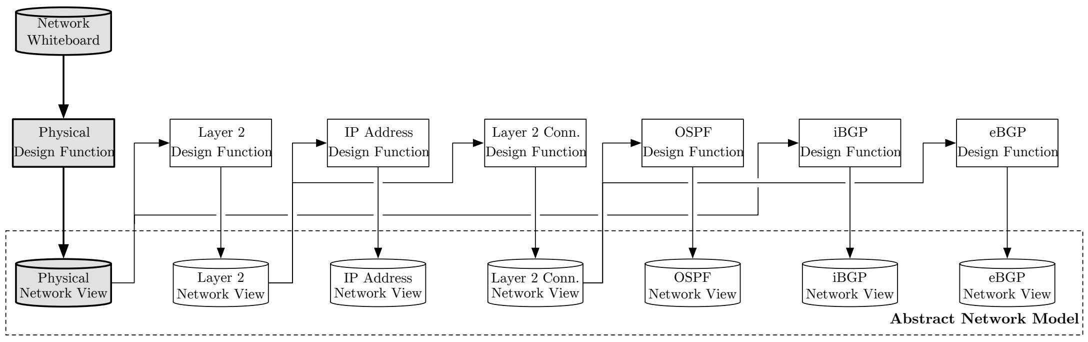


Figure 5.12: Cross-section illustration of the set of Network Views in the Abstract Network Model for the Simplified Small Internet Example, as shown in Figure 4.43. Vertical lines represent the association of Network Elements across Network Views. Each horizontal plane corresponds to a Network View. The Layer 2 Network Views have been omitted for visual simplicity.

5.6.1 *Physical Design Function*



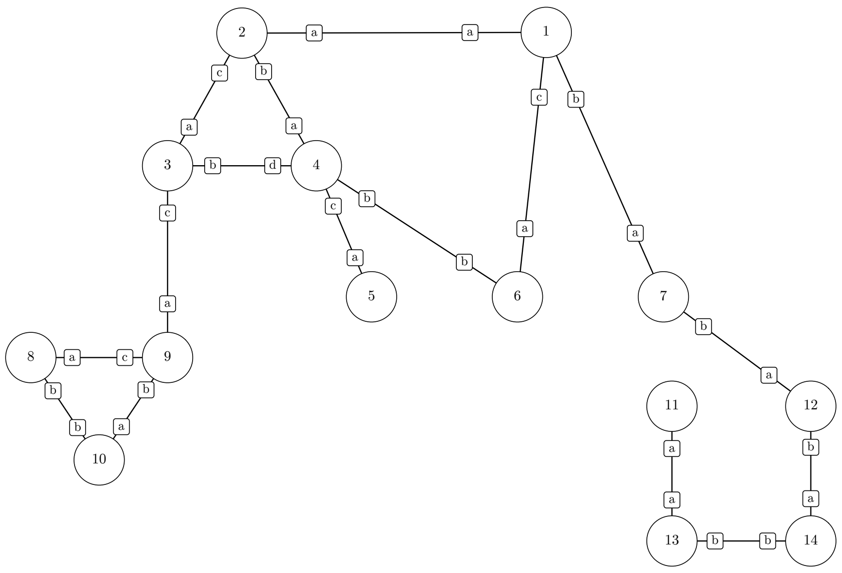


Figure 5.13: Physical Network View

The Physical Design Function is creates the Physical Network View. This is performed by looking at the Network Whiteboard and collecting all Network Elements which represent a physical network device. This is the set of all Network Elements which are not Pseudo Network Elements.

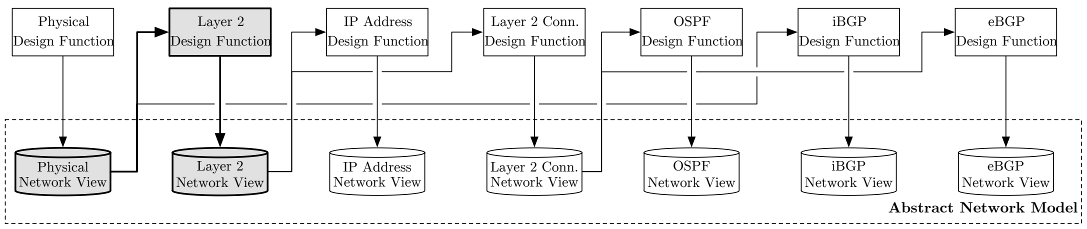
Here, we provide a discussion of Low-Level and High-Level Primitives can be used to construct the Physical Network View presented in Figure 5.13. The first step is to create the physical Network View in the Abstract Network Model. The next step is to add the physical Network Elements. This can be performed by selecting the physical Network Elements from the Network Whiteboard, and then adding them using the adding Network Elements High-Level primitive. This can also be performed using the *Add Specific Network Elements* High-

Level primitive. This constructs a new Network View containing the Physical Network Elements. The next step is to create the physical Network Element Connections. For this example we assume that all Network Element Connections in the Network Whiteboard have the physical role. An example pseudo-code definition is shown if Algorithm 5.4. The notation and functions used in the pseudo-code are shown in the appendices.

Algorithm 5.4 Pseudo-Code for Physical Design Function

```
function PHYSICAL DESIGN FUNCTION($\Theta, \omega$)
    $\theta_{phy} \leftarrow$ CREATE$\theta(\Theta, phy)$
    $X \leftarrow (\Pi_{routers}(\omega))$                          ▷ NE from Network Whiteboard
    $\theta_{phy} \leftarrow$ ADD_NE$(\theta_{phy}, X)$
    $Y \leftarrow (E(\omega))$                                      ▷ NEC from NW
    $Y' \leftarrow \{\epsilon \in Y \mid \rho_E(\Theta, \epsilon_i) = physical\}$    ▷ Filter physical edges
    $\theta_{phy} \leftarrow$ ADD_NEC$(\theta_{phy}, Y')$
end function
```

5.6.2 *Layer 2 Design Function*



The Layer 2 Design Function captures the broadcast domains within the network. This example contains only routers with point-to-point links between them. Therefore the broadcast domains in the network can be created by simply splitting each point-to-point link between the routers, with a Pseudo Network Element that represents a broadcast domain. The Layer 2 Network View is shown in Figure 5.14.

If switches were present in the network, then an extended approach would used to create Layer 2 Network View. We discuss this in Section 5.7.1. The process is then to create the Network View, and then add the Network Elements. We construct the Layer 2 Network View from the Network Elements in the physical Network View. This can be performed using the add specific Network Elements high-level primitive, similar to how the physical Network View was constructed from the Network Whiteboard. The next step is to add Network

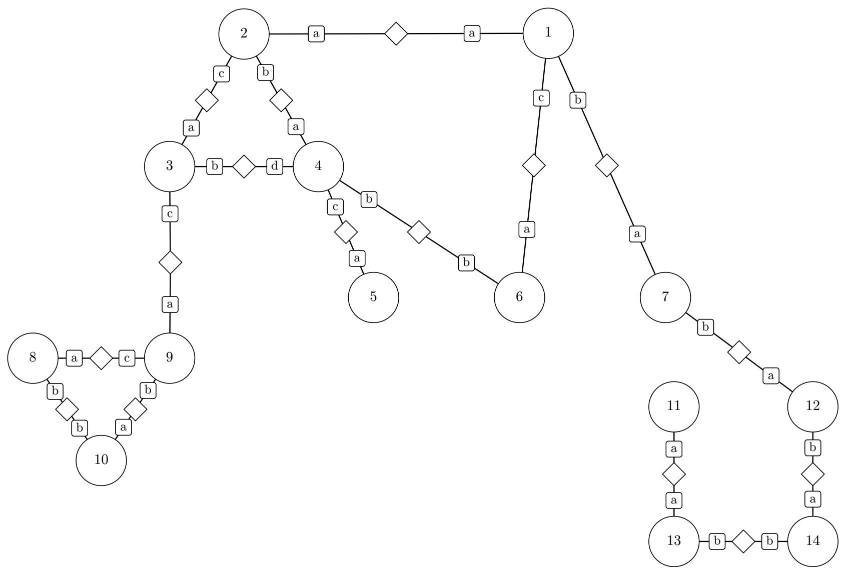


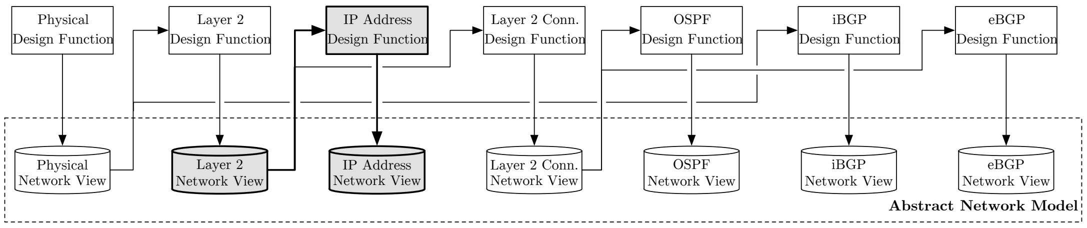
Figure 5.14: Layer 2 Network View

Element Connections. These are also formed from the connections of the physical Network View. We then construct the broadcast domains, by splitting the point-to-point links, using the split high-level primitive. This produces a Pseudo Network Element between each pair of Routers that have a Network Element collection in the physical Network View. These Pseudo Network Elements represent the broadcast domains, and can be used for the IP address allocation Design Function. We mark each of these newly added to the Network Element with a broadcast domain label, which simplifies their selection in Network Element filtering operations. A pseudo-code example of the Layer 2 Design Function is shown in Algorithm 5.5.

Algorithm 5.5 Pseudo-Code for Layer 2 Design Function

```
function LAYER 2 DESIGN FUNCTION($\Theta$)
    $\theta_{layer\_2} \leftarrow \text{CREATE}\theta(\Theta, layer\_2)$
    $X \leftarrow (\Pi_{routers}(\theta_{phy})$    ▷ Routers from Physical NV
    $\theta_{layer\_2} \leftarrow \text{ADD\_NE}(\theta_{layer\_2}, X)$
    $Y \leftarrow (E(\theta_{phy}))$
    $\theta_{layer\_2} \leftarrow \text{ADD\_NEC}(\theta_{layer\_2}, Y)$
    $\theta_{layer\_2} \leftarrow \text{SPLIT}(\theta_{layer\_2}, Y)$    ▷ Split all p-to-p connections
    $P = \Pi(\theta_{layer\_2}) - X$    ▷ Set of newly added Pseudo-NEs
    $\theta_{layer\_2} \leftarrow \text{SET\_LABEL\_}\Pi(\theta_{layer\_2}, P, broadcast\_domain, True)$
▷ Mark as Broadcast Domain
end function
```

5.6.3 *IP Address Design Function*



We now describe the construction of the IP address Network View, which contains the IP addressing used in the network. The Design Function used to create this view is an example of resource allocation, where a sequence is mapped to a set of elements. A sequence representing loopback addresses is mapped onto the router Network Elements, and a sequence representing infrastructure addresses is mapped onto the router Network Element Interfaces. They can be used to produce the Network View shown in Figure 5.15.

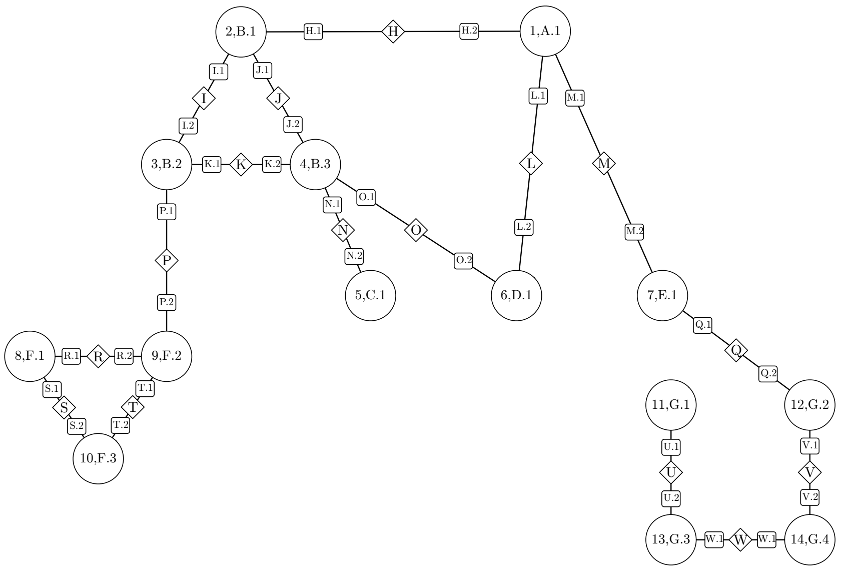


Figure 5.15: IP Address Network View

To simplify this example we use a place-holder denoted by a letter to represent specific IPv4 network address . We provide a more thorough treatment of IP addressing including IPv4 prefixes, in the case studies in Chapter 7 and Chapter 8. There are two types of IP addresses that we allocate in this Network View: the loopback address allocated to each router, and the infrastructure IP address allocated to the physical Network Element Interfaces. In addition to this, we also store the address blocks which have been allocated onto the Network View itself, for use in routing protocol prefix advertisement.

Base Structure

We first create the IP Address Network View, following the same pattern as for the Physical and Layer 2 Network Views, and then add the Network Elements. We use the Layer 2 Network View created in the previous section, which consists of Router Network Elements, and a series of Pseudo Network Elements representing broadcast domains. We copy across the Network Element Connections and Network Element Interfaces directly from the Layer 2 Network View. This captures the connectivity of the Routers to the broadcast domain Pseudo Network Elements, and the interfaces on which the routers connect to the broadcast domains. A pseudo-code description is shown in Algorithm 5.6.

Algorithm 5.6 Pseudo-Code for structure of IP Address Design Function

```
function IP BASE STRUCTURE COMPONENT($\Theta$)
    $\theta_{ip} \leftarrow \text{CREATE}\theta(\Theta, ip)$
    $X \leftarrow (\Pi(\theta_{layer\_2})$                ▷ NEs and PNEs from Layer 2 NV
    $\theta_{ip} \leftarrow \text{ADD\_NE}(\theta_{ip}, X)$
    $Y \leftarrow (E(\theta_{layer\_2}))$
    $\theta_{layer_2} \leftarrow \text{ADD\_NEC}(\theta_{layer_2}, Y)$
end function
```

Loopback Address Allocation

We now look at loopback address allocation. We allocate a prefix block to each autonomous system in the networ, which we denote by the letters *A*, *B*, *C*, *D*, *E*, *F*, and *G*. We allocate loopback addresses to the router Network Elements, but do not allocate loopback addresses to the broadcast domain Pseudo Network Elements. Within each autonomous system, we allocate the individual IP addresses from this block. For instance AS1 has a single router, which is allocated the prefix A.1. AS20, contains three devices, which are allocated IP addresses from the prefix range B: B.1 B.2 and B.3. In a real-world example these prefixes would correspond to actual IPv4 addresses. For instance the prefix A may represent 192.168.0.x with router 1 allocated 192.168.0.1. B may be 192.168.1.x with router 2 allocated 192.168.1.1, router 3 allocated 192.168.1.2 and router 4 allocated 192.168.1.3. The requirement for the loopback addresses is that they are globally unique within the network. A pseudo-code example is shown in Algorithm 5.7.

Algorithm 5.7 Pseudo-Code for IP Address Loopback Allocation component of IP Address Design Function

```
function IP LOOPBACK COMPONENT($\Theta$)
    $\theta_{ip} = NV(\Theta, ip)$
    $G = \text{GROUP}(\Pi(\theta_{ip}), asn)$
    $allocations = \text{NEW\_ASSOCIATIVE\_ARRAY}()$
    for $(asn, X) \in G$ do
        $I \leftarrow$ next unallocated IP block
        SET_KEY(allocations, asn, I)
        for $x \in X$ do
            $i \leftarrow$ next unallocated IP from I
            $\tau_{loopback0} \leftarrow \text{GET\_LO\_ZERO}(\theta_i, x)$
            SET_LABEL_T($\tau_{loopback0}$, loopback, i)
            SET_LABEL_T($\tau_{loopback0}$, prefix, 32)
        end for
    end for
    SET_LABEL_θ($\theta_{ip}$, loopback_allocations, allocations)
end function
```

Infrastructure Address Allocation

The other set of network addresses allocated in the network are the infrastructure addresses, allocated to the physical interfaces. As with loopback IP addresses, infrastructure IP addresses must be globally unique across the network. We allocate the infrastructure IP addresses such that they are grouped by the autonomous system to which the Network Element belongs. This allows us to aggregate and advertise a block prefix, over the eBGP exterior routing protocol. The infrastructure IP addresses are allocated in two stages. The first stage allocates a subnet block to each Pseudo Network Element that represents a broadcast domain. The second stage iterates over the interfaces connected to this broadcast domain, and allocates an individual IP address from the subnet block of that broadcast domain Pseudo Network Element.

This example contains only point-to-point network links between routers, so each broadcast domain connects exactly two Network Elements. This is due to the simplifying assumption made in Chapter 4, which simplifies the allocation of subnets. A more generalised approach, discussed in Section 9.3.1 looks at the degree of the broadcast domain Pseudo Network Element to determine the appropriate size subnet block to be allocated to that broadcast domain. Here the degree is always exactly two.

We follow a similar approach to the loopback address allocation example, where we allocate a label to represent the subnet. Here we use a simplified allocation algorithm which maps a unique letter to each broadcast domain Pseudo Network Element. This is a generalised form of an IP address allocation algorithm. An extension would consider the network structure in allocating these labels. For instance in autonomous system 2, the point-to-point subnets have been allocated the labels I, J, and K. In order to signify the advertisement of network prefixes through the routing protocols, these labels would be allocated from the same parent subnet block.

Regardless of the specific labels are allocated, the rest of the approach continues as follows. Once the subnet label has been allocated to a broadcast domain, the next step is to map the individual IP addresses onto the appropriate Network Element Interfaces. These interfaces are the physical Network Element Interfaces of the router Network Elements. This can be performed by iterating over each broadcast domain Pseudo Network Element in the Network View. These can be selected using the filtering and selection primitives. The next step is to iterate over each of the remote Network Element Interfaces, of the router Network Element, there is connected to this broadcast domain Pseudo Network Element. These remote Network Element Interfaces can be obtained using the Neighbor Network Element Interfaces Low-Level Primitive presented in Figure D.2.4.

The subnet labels allocated to the broadcast domains of the Network Elements present a sequence of IP addresses, that belong to that specific subnet. We allocate the first IP address from the sequence to the first network interface obtained using the neighbour Network Element Interfaces Low-level primitive. We allocate the second IP address from the sequence to the second Network Element Interface. This would continue on, if there were more Network Element Interfaces on the broadcast domain. This demonstrates how such an approach of allocation and then iteration can scale to larger broadcast domains, such as those created from the use of a switch, as we will see in Section 5.7.1. Once the loopback and infrastructure IP addresses have been allocated, they can be used like any other label in subsequent stages of the design and configuration generation process. A pseudo-code example of the IP Address infrastructure allocation component of the Design Function is shown in Algorithm 5.8.

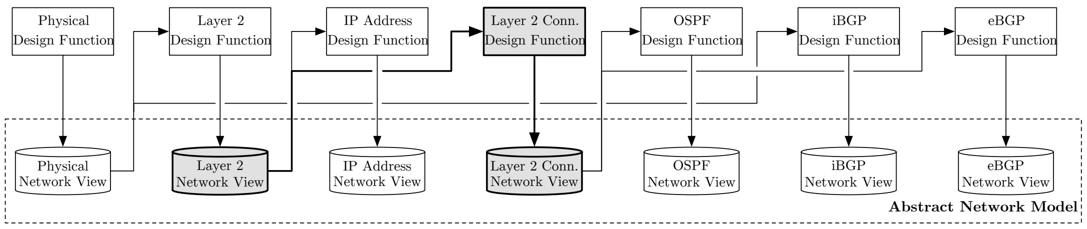
Algorithm 5.8 Pseudo-Code for Infrastructure Allocation component of IP Address Design Function

```
function IP INFRASTRUCTURE COMPONENT($\Theta$)
    $allocations = \text{NEW\_ASSOCIATIVE\_ARRAY}()$
    $X = \{\pi_i \forall \pi_i \in \Pi(\theta_{ip}) \mid \lambda_\Pi(\theta_{ip}, \pi_i, \text{broadcast\_domain}) = \text{True}\}$
    for $x \in X$ do
        $I \leftarrow$ next unallocated IP block
        SET_KEY(allocations, asn, I)
        SET_LABEL_Π(x, ip, i)
        SET_LABEL_Π(x, prefix, 24)
        for $\tau_{neigh} \in \text{NEIGH\_NEI\_NE}(x)$ do
            $i \leftarrow$ next unallocated IP from I
            SET_LABEL_T($\tau_{neigh}$, ip, i)
            SET_LABEL_T($\tau_{neigh}$, prefix, 24)
        end for
    end for
    SET_LABEL_θ($\theta_{ip}$, infrastructure_allocations, allocations)
end function
```

Summary

In this section we have demonstrated the Design Functions used to perform IP Addressing for both Loopback and Infrastructure address allocations. These can be combined to form the complete IP Address Design Function. We show an implementation of the IP Address Design Function in Figure 7.4.3, and discuss how this could be extended in Section 9.3.1.

5.6.4 *Layer 2 Connectivity Design Function*



This Design Function explodes all broadcast domains from the Layer 2 Network View. A pseudo-code example of this Design Function is shown in Algorithm 5.9, and the resulting topology shown in Figure 5.16.

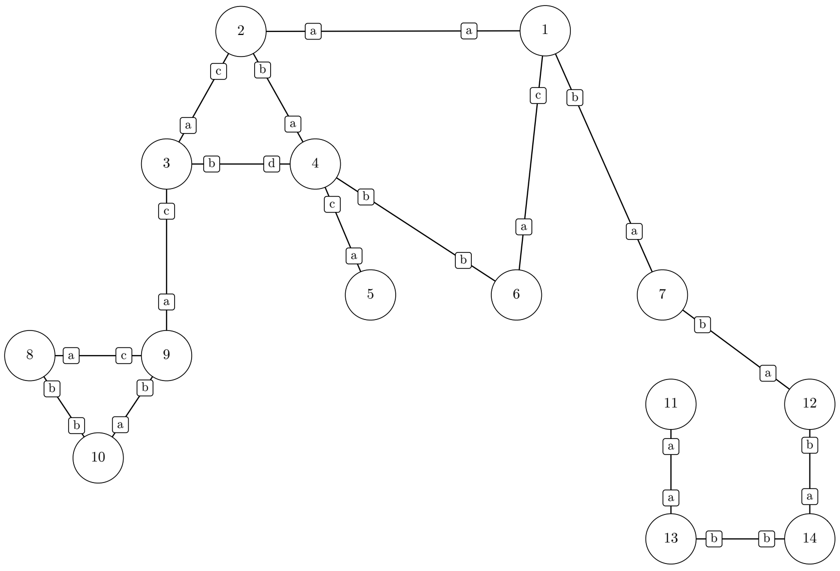
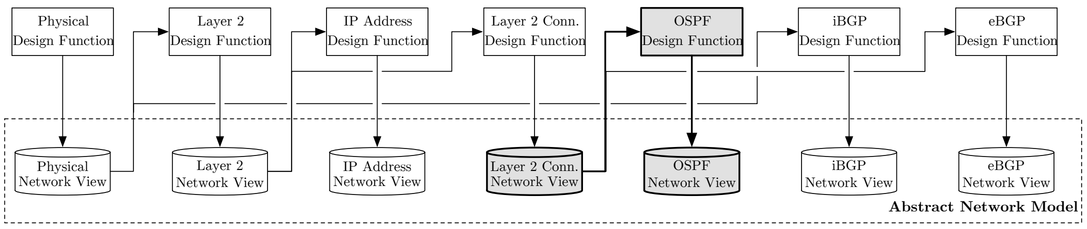


Figure 5.16: Layer 2 Connectivity Network View

Algorithm 5.9 Pseudo-Code for Layer 2 Connectivity Design Function

```
function LAYER 2 CONNECTIVITY DESIGN FUNCTION($\Theta$)
    $\theta_{layer\_2} \leftarrow \text{CREATE}\theta(\Theta, layer\_2)$
    $\theta_{layer\_2\_broadcast} \leftarrow \text{ADD\_NE}(\theta_{layer\_2})$
    $X \leftarrow \{\pi_i \forall \pi_i \in \Pi(\theta_{layer\_2}) \mid \lambda_{\Pi}(\theta_{layer\_2}, \pi_i, broadcast\_domain) = True\}$
    $\theta_{layer\_2\_broadcast} \leftarrow \text{EXPLODE}(\theta_{layer\_2\_broadcast}, X)$
end function
```

5.6.5 *OSPF Design Function*



The OSPF Design Function constructs the OSPF Network View. It produces the Network View shown in Figure 5.17.

The OSPF Network View is used for the configuration of the OSPF routing protocol. We add all router Network Elements from the Layer 2 Connectivity Network View. The Network Element Connections in

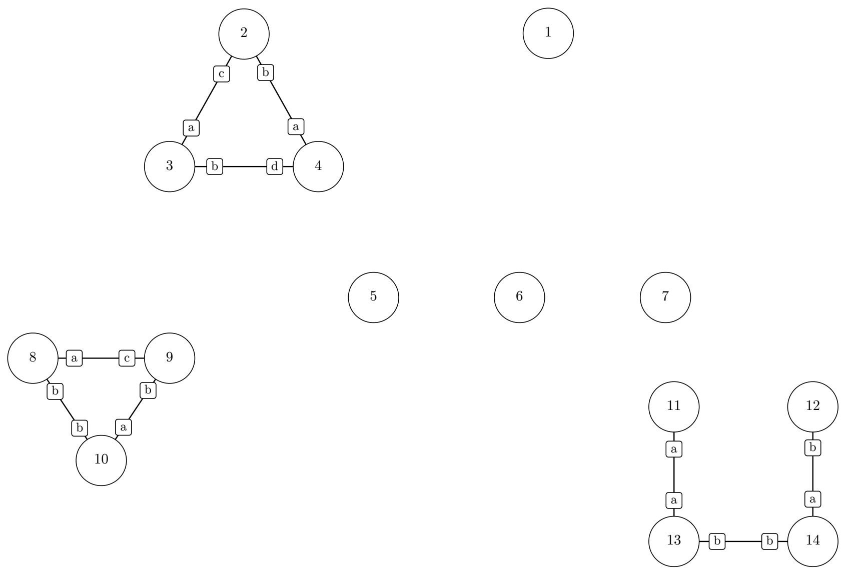


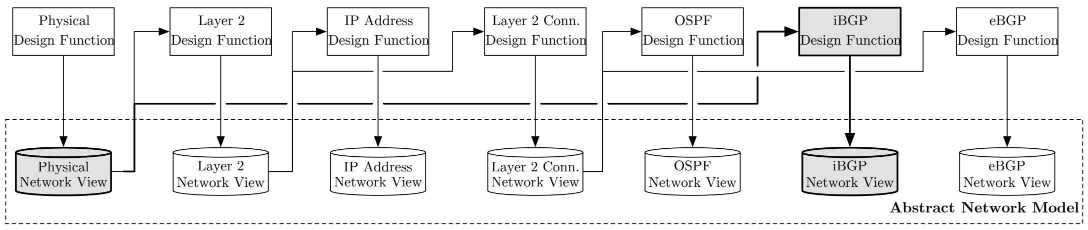
Figure 5.17: OSPF Network View

the OSPF Network View represent the adjacencies between routers in the OSPF routing protocol. We then add the Network Element Connections from the Layer 2 Connectivity Network View, but first apply a filter to retain only those which connect two Network Elements *in the same autonomous system*. This step retains the Network Element Interface bindings on the Network Element Connections from the Layer 2 Connectivity View. A pseudo-code example of the OSPF Design Function is shown in Algorithm 5.10.

Algorithm 5.10 Pseudo-Code for OSPF Design Function

```
**function** OSPF DESIGN FUNCTION($\Theta$)
    $\theta_{ospf} \leftarrow \text{CREATE}\theta(\Theta, ospf)$
    $X \leftarrow (\Pi_{routers}(\theta_{layer\_2\_conn})$
    $\theta_{ospf} \leftarrow \text{ADD\_NE}(\theta_{ospf}, X)$
    $Y \leftarrow (E(\theta_{layer\_2\_conn}))$
    $Y' \leftarrow \{\epsilon \in Y \mid \lambda_{ASN}(\Pi_v(\epsilon)) = \lambda_{ASN}(\Pi_u(\epsilon))\}$ ▷ Filter to same ASN
    $\theta_{ospf} \leftarrow \text{ADD\_NEC}(\theta_{ospf}, Y')$
**end function**
```

5.6.6 *iBGP Design Function*



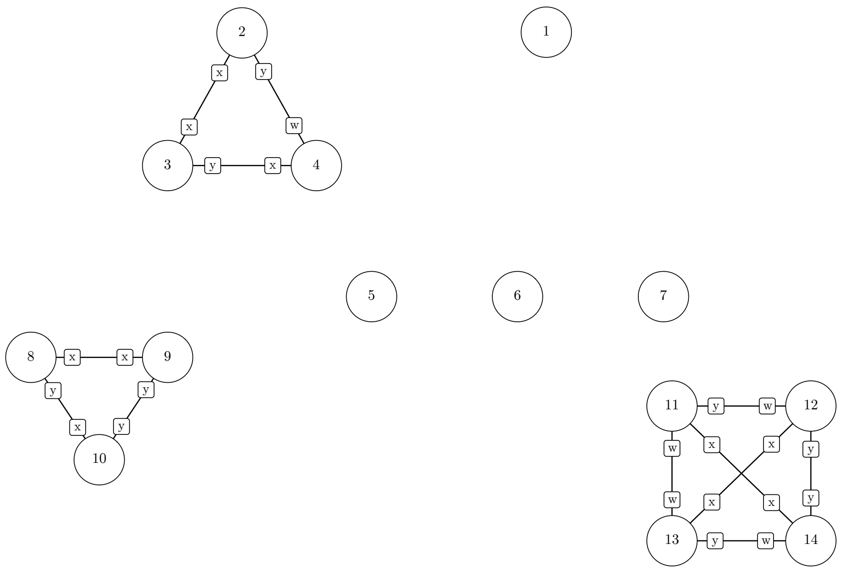


Figure 5.18: iBGP Network View

The iBGP Network View represents the configuration for the iBGP routing protocol. iBGP is responsible for sharing the routes learnt from other autonomous systems over eBGP into the autonomous system. The iBGP Design Function is used to produce the Network View shown in Figure 5.18. A number of different design approaches are available for how an iBGP topology is constructed. The simplest topology is a full mesh, where each router establishes an iBGP session to every other router in the same autonomous system. While this approach has scalability limitations it is suitable for this example given the small size of each autonomous system. We discuss the scalability of iBGP in Section 5.7.3, and in the case studies in Chapter 8.

The iBGP Design Function first creates the Network View and adds the routers from the Physical Network View. This Design Function differs to that of the OSPF and eBGP, as we create a full-mesh independent underlying physical connectivity. For simplicity we assume that there is physical connectivity between the Network Elements in each autonomous system (*i.e.* that the physical network is not partitioned). There are multiple approaches to create a full-mesh for each autonomous system. One approach is to create the Cartesian product of all Network Elements in the Network View. This set is then filtered to remove pairs of repeated Network Elements, *e.g* (*u*,*v*) where *u* = *v*. We then filter this set again to retain only Network Element pairs with the same *autonomous system* label.

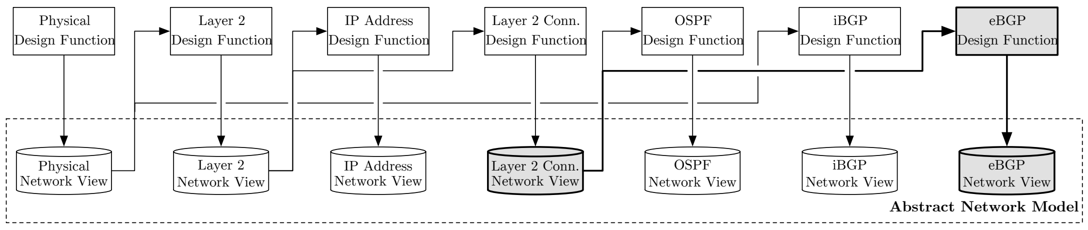
The Network Element Connections in the iBGP Network View represent BGP sessions. The Network Element Interfaces in the iBGP Network View represent the session termination point of the session. They are associated to an interface which the session is established on. The iBGP Sessions are not bound to a physical interface, so we construct a logical Network Element Interface on each Network Element, and then bind the Network Element Connection to these newly created logical network interfaces. A pseudo-code example of the iBGP Design Function is shown in Algorithm 5.11.

Algorithm 5.11 Pseudo-Code for iBGP Design Function

```
function iBGP DESIGN FUNCTION($\Theta$)
    $\theta_{ibgp} \leftarrow$ CREATE$\theta(\Theta, ibgp)$
    $X \leftarrow \Pi_{routers}(\theta_{phy})$    ▷ Routers from Physical NetworkView
    $\theta_{ibgp} \leftarrow$ ADD_NE$(\theta_{ibgp}, X)$
    $\{(\pi_i,\pi_j)\forall(\pi_i,\pi_j) \in X \times X \quad | \quad \pi_i \neq \pi_j \wedge \lambda_{asn}(\pi_i) = \lambda_{asn}(\pi_j)\}$ ▷
Non-self-loop same-ASN NE pairs in cartesian product of X
    for $\{\pi_u, \pi_v\} \in Y$ do
        $\tau_u \leftarrow T_u(\theta_{phy}, \epsilon_i)$
        $\theta_{ibgp}, \tau_{session\_u} \leftarrow$ ADD_LNEI$(\theta_{ibgp}, pi_u)$
        $\tau_{loopback\_u} \leftarrow$ GET_LO_ZERO$(\theta_i, \pi_u)$
        $\theta_{ibgp} \leftarrow$ ASSOCIATE_NEI$(\theta_{ibgp}, \pi_u, \tau_{session\_u}, \tau_{loopback\_u})$
        $\tau_v \leftarrow T_v(\theta_{ibgp}, \epsilon_i)$
        $\theta_{ibgp}, \pi_v, \tau_{session\_v} \leftarrow$ ADD_LNEI$(\theta_{ibgp,p}\ i_v)$
        $\tau_{loopback\_v} \leftarrow$ GET_LO_ZERO$(\theta_i, \pi_u)$
        $\theta_{ibgp} \leftarrow$ ASSOCIATE_NEI$(\theta_{ibgp}, \tau_{session\_v}, \tau_{loopback\_v})$
        $\theta_{ibgp} \leftarrow$ ADD_NEC$(\theta_{ibgp}, \pi_u, \tau_{session\_u}, \pi_v, \tau_{session\_v})$
    end for
end function
```

5.6.7 *eBGP Design Function*

Finally, we consider the eBGP Design Functions used to construct the eBGP Network View. These are conceptually the inverse of the OSPF



Design Function. It can be used to produce the Network View shown in Figure 5.19.

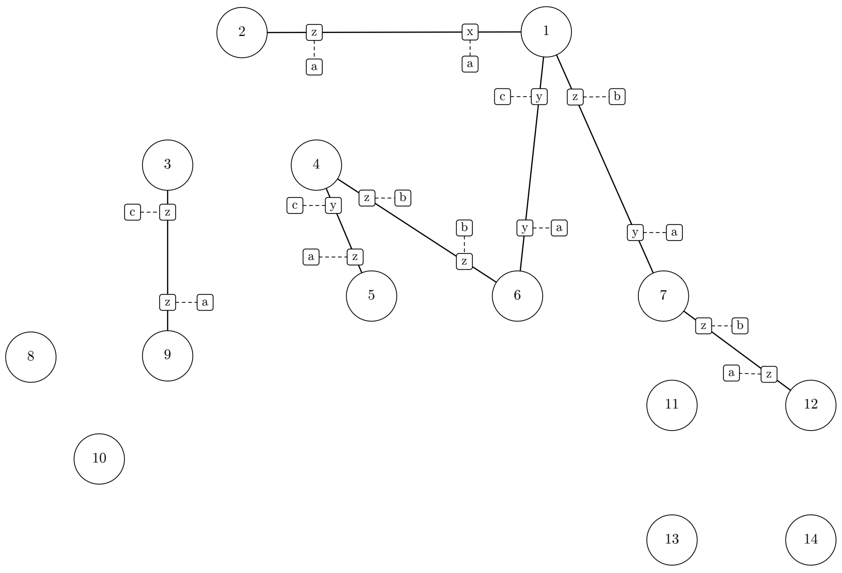


Figure 5.19: eBGP Network View

The Network Element Connections in the eBGP Network View represent BGP sessions. The Network Element Interfaces in the eBGP Network View represent the session termination point of the session, and are associated to the interface which the session is established on. The eBGP Design Function is constructed by first creating the Network View, and adding the routers from the Layer 2 Connectivity Network View. We then add the appropriate Network Element Connections: where Network Element Connection exists in the Layer 2 Connectivity Network View, and that Network Element Connection connects Network Elements that belong to *different* autonomous systems.

This can be implemented in the Design Function similar to the OSPF Design Function. We first iterate over Network Element Connections from the Layer 2 Connectivity Network View. We then filter this set of Network Element Connections, according to a predicate which returns true if the autonomous system labels of the Network

Elements differ. The Network Element Interfaces are used to indicate the interface that the session is established from. These eBGP Network Element Interfaces can also be used to capture routing policy: a set of labels can be added to represent the ingress routing policy applied to routes learnt over the session, or a set of labels added to represent the egress routing policy applied to routes advertised over the session.

This demonstrates how our approach of Network Element Interface bindings on the Network Element Connections, both allows us to maintain consistency when adding Network Element Connections from different Network Views, but also to store additional information such as policy in the eBGP Network View. A pseudo-code example of the eBGP Design Function in shown in Algorithm 5.12.

Algorithm 5.12 Pseudo-Code for eBGP Design Function

```
function eBGP Design Function($\Theta$)
    $\theta_{ebgp} \leftarrow$ CREATE$\theta(\Theta, ebgp)$
    $X \leftarrow (\Pi_{routers}(\theta_{layer\_2\_conn})$
    $\theta_{ebgp} \leftarrow$ ADD_NE$(\theta_{ebgp}, X)$
    $Y \leftarrow (E(\theta_{layer\_2\_conn}))$
    $Y' \leftarrow \{\epsilon \in Y \mid \lambda_{ASN}(\Pi_u(\epsilon)) \neq \lambda_{ASN}(\Theta, \Pi_u(\epsilon))\}$
    for $\epsilon_i \in Y$ do
        $\pi_u \leftarrow \Pi_u(\theta_{layer_2}, \epsilon_i)$
        $\tau_u \leftarrow T_u(\theta_{layer_2}, \epsilon_i)$
        $\theta_{ebgp}, \tau_{session\_u} \leftarrow$ ADD_LNEI$(\theta_{ebgp}, pi_u)$
        $\theta_{ebgp} \leftarrow$ ASSOCIATE_NEI$(\theta_{ebgp}, \tau_{session\_u}, \tau_u)$
        $\pi_v \leftarrow \Pi_v(\theta_{ebgp}, \epsilon_i)$
        $\tau_v \leftarrow T_v(\theta_{ebgp}, \epsilon_i)$
        $\theta_{ebgp}, \tau_{session\_v} \leftarrow$ ADD_LNEI$(\theta_{ebgp}, \pi_v)$
        $\theta_{ebgp} \leftarrow$ ASSOCIATE_NEI$(\theta_{ebgp}, \pi_v, \tau_{session\_v}, \tau_v)$
        $\theta_{ebgp} \leftarrow$ ADD_NEC$(\theta_{ebgp}, \pi_u, \tau_{session\_u}, \pi_v, \tau_{session\_v})$
    end for
end function
```

5.6.8 *Conclusion*

In this section we have shown how the use of the Low-level and High-level Primitives described in Section D.2 and Section D.3 can be used to compose Design Functions. These Design Functions can be used to transform the Network Whiteboard into a series of Network Views, which represent various aspects of the network configuration process. The Design Functions allow the declarative high-level policy outlined on the Network Whiteboard to be transformed into Network Views, representing an abstract representation of these aspects of the configuration process. As will show in the next chapter, these design views can be used in the code generation step to produce real target output device configurations. Before we proceed to code generation we now demonstrate how the framework is extensible to allow for the addition of new Design Functions.

5.7 EXTENDED DESIGN FUNCTION EXAMPLES

In this section we provide a number of examples showing how the Design Functions approach can be extended to perform more advanced network design operations. We use the techniques discussed here in the case studies in Chapter 8.

We first look at example of how network topology with switches can be used to form the resulting Layer 2 broadcast domains and Layer 2 connectivity. These can then be used for the configuration of higher-level IP address allocation and routing protocols. We then look at using VLANs to create virtual LANs, which extends the previous example to cater for virtual network devices. The result of these Design Functions is a Layer 2 broadcast domain and Layer 2 Connectivity Network View that can be used by the high-level layers, in the same way as if virtual LANs were not used. This shows the extensibility of our approach by breaking the design steps into a separation of concerns. Finally, we look at extending the scalability of the iBGP protocol using route-reflection, rather than a full-mesh of all iBGP routers.

5.7.1 *Introduction of Layer 2 Switches*

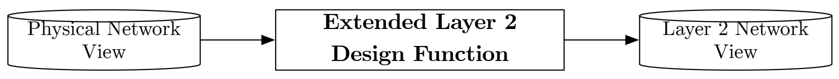


Figure 5.20: Workflow for extended Layer 2 Design Function to handle Switches

This section presents an example of a topology design with Network Elements that have the Switch Role. The workflow for this is shown in Figure 5.20. Switches operate at layer 2 of the OSI reference model. The connectivity of switches influences the connectivity at layer 3, the IP layer. All devices connected to the same switch belong to what is known as the Broadcast domain. At the IP layer, these devices are connected directly to each other. In order to accurately generate the configurations at the IP layer, such as for IP addressing, or for the routing protocols, we need to form the IP connectivity, the results from the switches. To do this, we need to consider the interfaces which connected to the switches, and then connect these together. We do this by forming a Layer Two Network View, based on the physical connectivity of the physical Network View, and the roles of the devices. For this example, we consider two types of device roles: routers and switches. This can be generalised further to other layer 3 devices, such as firewalls, and end hosts such as servers.

We use the EXPLODE High-Level primitive which explodes out the switch, to form a clique of the interfaces connected to that switch. We note that in this scenario, an interface may have more than one network connection. This highlights the value of the approach discussed in Chapter 4, where we use an interface and node centric approach, to store configuration data, rather than storing the information on the Network Element Connections themselves. By using the approach of interface bindings, we are able to retain this binding across the Network Views, and in particular, allow multiple network connections to be bound to the same Network Element Interface. The simplest case is a single switch connected to multiple routers. For this case we can use the EXPLODE function directly, where we explode any Network Element that has the role of switch. A more advanced scenario is where multiple switches are connected together. We then would need to first use the merge function to merge multiple switches that are directly connected. This forms the resulting interconnected switch, which can be exploded to form the resulting layer 2 connectivity.

Example

The example in Figure 5.21 demonstrates these steps. This example has a Network Whiteboard consisting of six routers and two switches, as shown in Figure 5.21a. These switches are directly connected to each other, and so any device connected to the switches will be in the same broadcast domain.

The Network Elements labelled *2*, *5*, *6*, and *8*, are directly connected to the switches, and so are connected at layer 2. Network elements labelled *1* and *3* are not connected to the switches, and therefore their connectivity at layer 2 will be the same as in the Network Whiteboard and Physical Network Views.

The first step is to select the Network Elements that have the *role* of switch, using the selection and filtering primitives. This will select Network Elements *4* and *7*. In this example, these two devices are directly connected to each other, and therefore will be merged. A generalisation would be to first select the Network Elements with the role of switch and then perform an analysis of connected components, using the Connected Components High-Level Primitive. By first selecting the switches, and then looking at the connected components we can determine which switches are to be merged together.

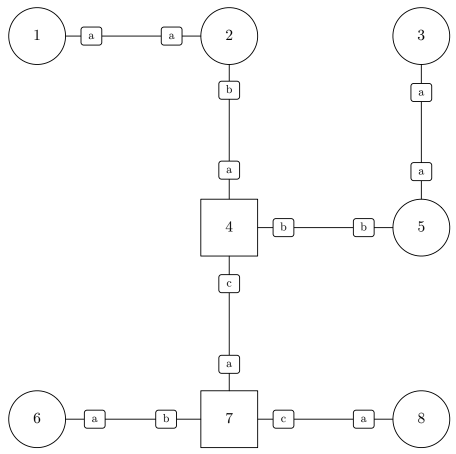
For this example, we only have one set of connected components, which is due to the Network Element Connection between Network

Elements *4* and *7*. Now that we have determined this broadcast domain, can merge these two switches, using the MERGE high-level primitive. This creates a new Pseudo Network Element, which is labelled as *P1* as shown in Figure 5.21b. We note here that the interface bindings of devices 2, 6, 5 and 8 have been retained, in the connectivity to the Pseudo Network Element. This allows the relationship of the Network Element Interfaces across Network Views to be used in the subsequent Design Functions, and in the configuration generation. In order to model the broadcast domains in the Layer 2 Network view, we also need to create broadcast domains for the point-to-point links, between Network Elements 1 and 2, Network Elements 3 and 5. This can be done by selecting all of the Network Elements in the network, which connect to devices that have a role that is determined to be layer 3, such as a router. We then apply the SPLIT function to these Network Elements giving Pseudo Network Elements *p2* and *p3*. The Network Element Interface bindings from these Network Elements *1*, *2*, *3*, and *5* are retained.

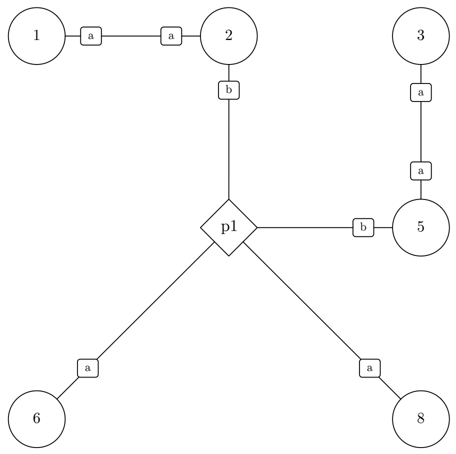
The result of these operations is shown in Figure 5.21c. This Network View shows the broadcast domains in the network, and consists of Network Elements of the role of router, which is Network Elements *1*, *3*, *4*, *5*, *6*, and *8*; and the Network Elements which represent the broadcast domain connectivity, using Pseudo Network Elements *p1*, *p2*, and *p3*. This resulting Network View is the Layer 2 Network View, and can be used for IP address allocation as discussed in Section 9.3.1.

As well as the Layer 2 Network View, some network Design Functions operate on the Layer 2 Connectivity, which arises from the broadcast domains. To avoid repetition, we can form a Layer 2 Connectivity View by exploding each of the Pseudo Network Elements that represent broadcast domains of the Layer 2 Network View. The result of this operation is shown in Figure 5.21d. This has an effect on the Layer 2 Network View, the IP addressing view, and OSPF and eBGP connectivity. We introduce this as the Layer 2 Connectivity Network View.

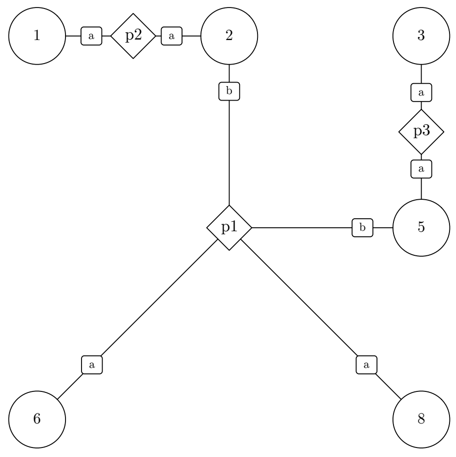
We first aggregate switches to form the broadcast domains, which are then used for IP allocation. We then explode the broadcast domain pseudo nodes to form the Layer 2 connectivity. This can then be used by the routing protocols (OSPF, eBGP) to determine connectivity, instead of the physical Network View. A worked example of this process can be see in Figure 5.21.



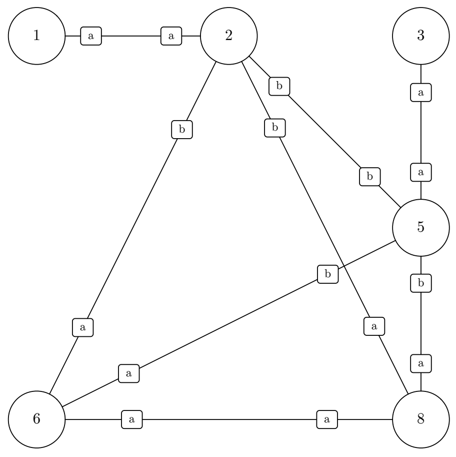
(a) Routers and Switches from Network Whiteboard.



(b) Switches combined using MERGE to form Pseudo Network Element *p1*.



(c) Point-to-Point Network Connections SPLIT to form Pseudo Network Elements *p2* and *p3*. This is the *Layer 2 Network View*.



(d) Pseudo Network Elements can be exploded using EXPLODE to form the *Layer 2 Connectivity Network View*.

Figure 5.21: Example of Design Functions using Switches to transform Network Whiteboard into Layer 2 Connectivity.

5.7.2 *Managed Switches and VLANs*

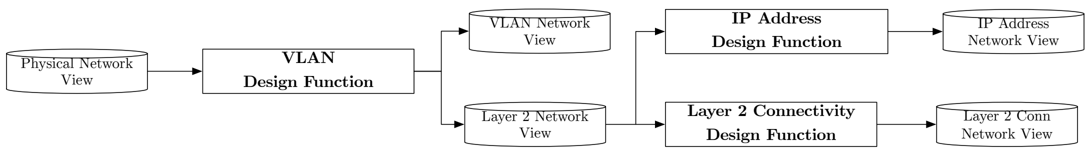


Figure 5.22: Flowchart of Network View construction for VLANs

This section expands on the switch example in Section 5.7.1 by allowing managed switches, which can be partitioned according to the VLAN labels set on an interface. Virtual LANs provide an ability to separate a single managed switch into multiple virtual switches. An interface on a switch can be tagged with a VLAN attribute, which represents the virtual switch that that interface belongs to. Multiple many switches can be connected together, with a protocol such as the VLA0.85N trunking protocol being used to exchange information about the VLANs configured on the managed switches.

For network design, the resulting virtual LAN segments created by the VLANs have an effect on the higher-level protocols, including IP address allocation and IP connectivity. The network topology resulting from the VLAN configuration therefore needs to be considered when allocating IP addresses, and when setting up the routing protocols. An example of the use of VLANs to divide a managed switch into multiple virtual LAN segments is shown in Figure 5.23.

This consideration can be handled by building the IP addressing and routing protocol topologies from the connectivity of the VLAN configuration. Rather than building the IP addressing and routing protocol configurations directly from physical connectivity, we instead build it from the connectivity resulting from the VLAN configuration. The VLANs themselves can be configured by setting a label on the appropriate interface of a managed switch in the Network Whiteboard. This information can be used with the management connectivity to form the VLAN domains.

In this section we provide an example of how the high-level primitives discussed previously such as the SPLIT, MERGE, and EXPLODE functions, can be used together with the VLAN labels of the Network Whiteboard, to build the layer 2 connectivity resulting from the VLANs. This can then be used as the input Network View to the Design Functions for the IP addressing and routing Network Views. An example is shown in Figure 5.22 and example output shown in Section 5.7.2.

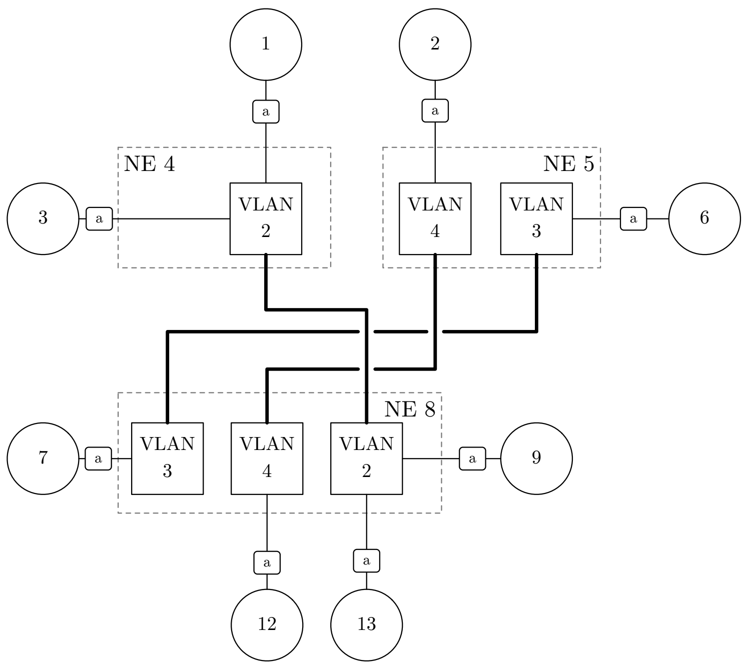


Figure 5.23: Conceptual view of VLANs. Managed switches *4*, *5* and *8* are divided using VLANs *2*, *3* and *4*. Trunk links between virtual LAN segments are shown as thicker lines.

The Network Whiteboard is shown in Figure 5.24, where the switches form two domains, one with switches *4*, *5*, and *8*, and one with switches *10* and *14*. These are merged in Figure 5.25 to form the Pseudo Network Elements *p1* and *p2*. These are then expanded in Figure 5.26 based on the VLAN labels. *p1* is expanded to *p1*, *p2* and *p3*, representing VLANs *1*, *2* and *3*. *p2* from the previous step is expanded to *p4* and *p5* representing VLANs *1* and *2*. Routers are connected to the appropriate VLAN Pseudo Network Element depending on their connectivity in the previous step. Finally, the layer 2 connectivity is formed by exploding the VLAN Pseudo Network Elements as shown in Figure 5.27.

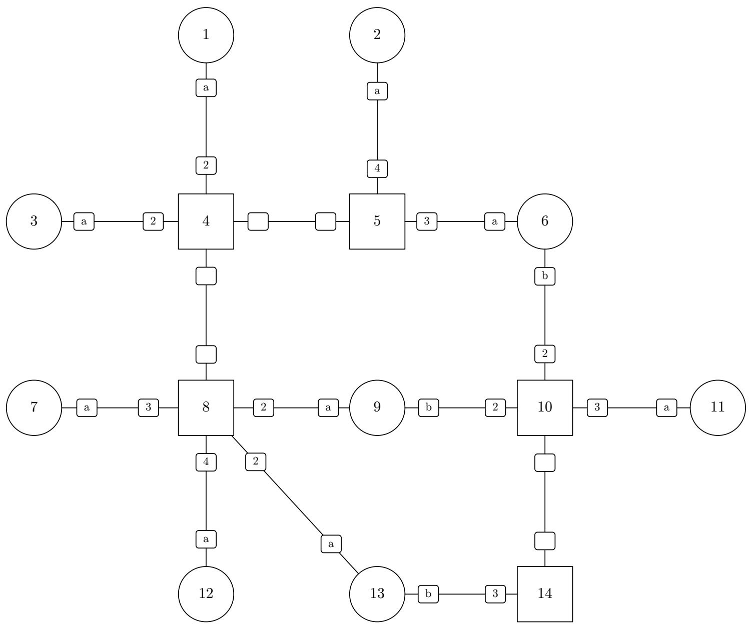


Figure 5.24: Network Whiteboard containing routers and switches. VLANs are shown on switch interfaces, and omitted on switch-to-switch interfaces. Interface bindings are shown on router interfaces.

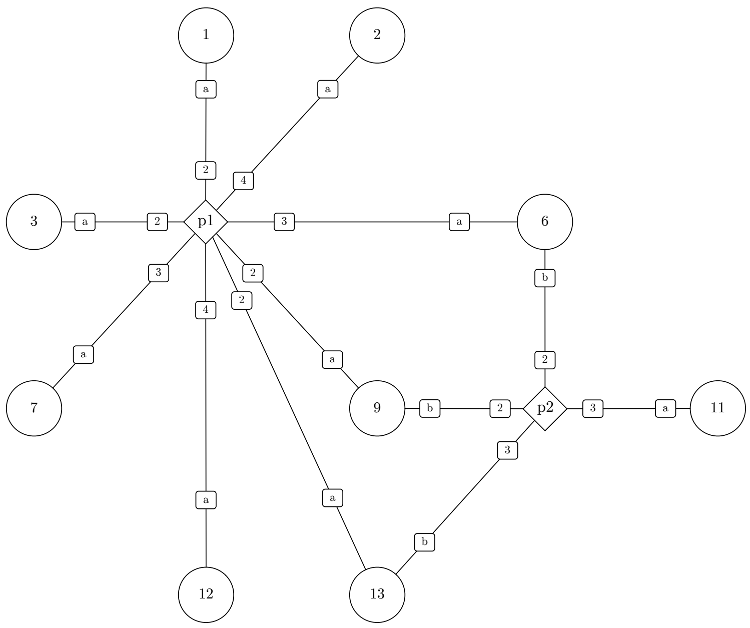


Figure 5.25: Directly connected switches are combined using MERGE to form Pseudo Network Elements *p1* and *p2*.

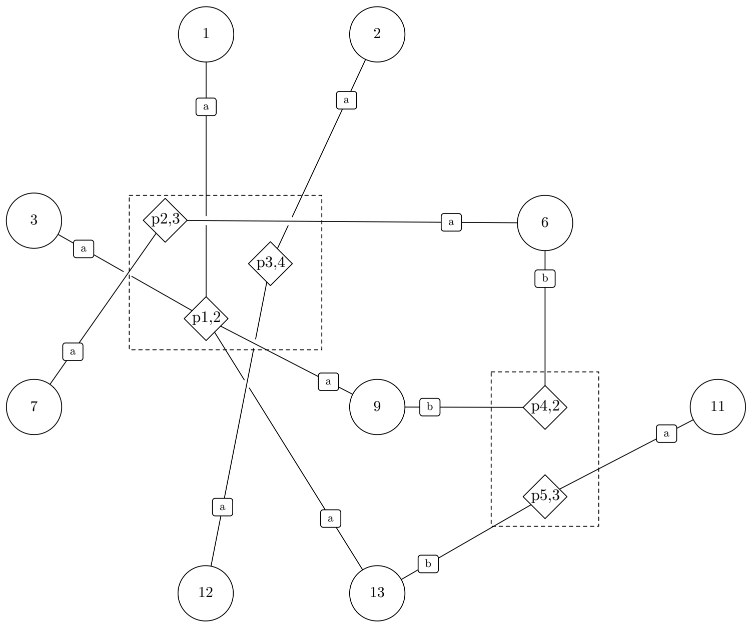


Figure 5.26: Pseudo Network Elements are expanded according to the VLANs connected to them.

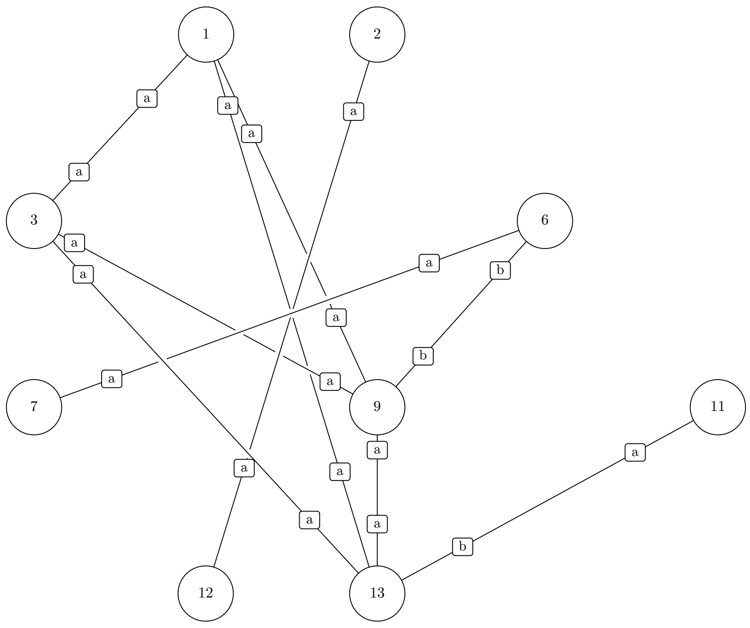


Figure 5.27: Pseudo Network Elements representing the VLANs can be exploded using EXPLODE to form the Layer 2 connectivity view, like in Figure 5.21d.

5.7.3 *iBGP Scalability*

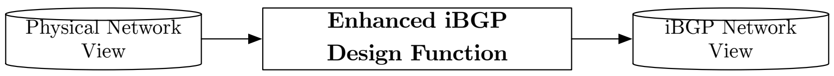


Figure 5.28: Workflow for Enhanced iBGP Design Function

In this section we describe how the iBGP Network View can be extended for scalability. The workflow for this is shown in Figure 5.28.

The default iBGP topology that we have constructed is been a full-mesh, where every router in an autonomous system, connects to every other router in the same autonomous system. This is a scalability issue for larger networks, as it produces $O(N^2)$ connections. Adding a new router would require not only creating N new iBGP sessions on the router, but also needing to touch every other N routers in that autonomous system to establish the peering to the new router. There are two commonly used methods to increase scalability. One of these is the use of a route reflector, which acts as a central point to which the other routers establish a peering. This introduces hierarchy, which improves scalability. Another option is the use of confederations. While we do not discuss confederations here, we note that our approach could be adapted to configure confederations.

We now discuss how the Design Function approach can be extended to configure route reflection. We will consider two cases of route reflection. The first is where the user annotate the Network Whiteboard using labels to indicate the router(s) that will act as route reflectors. The second is an algorithmic based approach, which automatically selects the most central routers as the route reflectors.

iBGP Topology

In our iBGP Network View, we represent a peering using a Network Element Connection. A Network Element Connection in the iBGP Network View between Network Elements A and B implies represents a BGP session established from A to B and also from B to A. Our approach could also allow a single session to be established by adding a label to the Network Element Interface on a Network Element Connection.

Additionally, for a hierarchical topology, we need to consider annotating these sessions. Consider the diagram below, shown as Figure 5.29. In this, we have a number of routers, forming an iBGP hierarchical topology.

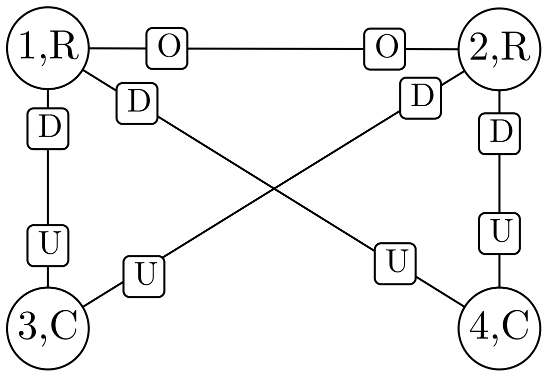


Figure 5.29: Topology showing basic route reflector hierarchy

We then establish iBGP sessions using Network Element Connections. In this case we have two types of sessions: a peering session between two routers acting in the same role, such as both as route reflectors; and a client/server relation session between a route reflector and a route reflector client.

These relation types are important in configuring the session in the device configuration. As an example, on Cisco IOS the session from a Route Reflector to a Route Reflector Client has the keyword "route-reflector-client" specified in the session configuration syntax. Denoting these client/server relationships also allows for the application of verification rules, such as checking for potential oscillation scenarios. We will discuss possible verification approaches in Section 9.3.3.

Role-Based Assignment

The first approach makes use of the Network Whiteboard, where we introduce a new *route-reflector* label. This label is set to True on the Network Elements which we wish to have the role of route reflector, and False otherwise. If unset we assume it is False.

To construct the enhanced iBGP Network View, we follow the same basic steps as for the previous iBGP Network View. We first construct the Network View, then step add the router Network Elements. We then introduce a new step, that copies the *route-reflector* label from the Network Whiteboard to the enhanced iBGP Network View.

We then create the Network Element Connections, using a different approach to the simple case. We first examine each set of Network Elements, grouping them by their autonomous system label.

Within each set of Network Elements we then divide the Network Elements into two subsets. The first is the set of route reflectors (the Network elements where the the *route reflector* label is True) from the

Network Whiteboard. We refer to this as *Subset A*. The second subset is the set of Route Reflector Clients. This is a set of Network Elements in the autonomous system group where the *route reflector* label is set to False. It can also be found as the set difference of the set of Network Elements in an autonomous system, minus the set of route reflector Network Elements. We refer to this as *Subset B*. We then establish Network Element Connections according to the following rules:

- We establish a Network Element Connection between each pair of Route Reflectors which belong to the same autonomous system. This can be performed by creating a full-mesh of the Network Elements in *Subset A*.
- We establish Network Element Connections between Network Elements in *Subset A* and Network Elements in *Subset B*.

Allocating Session Directionality Labels

As discussed in the previous section, we have three types of sessions. These are *over* sessions between two route reflectors, *up* sessions, from a route reflector client to a route reflector, and *down* sessions from a route reflector to a route reflector client.

We can allocate this directionality label to the Network Element Interfaces of a Network Element Connection in the iBGP Network View. We iterate over the Network Element Connections in this Network View, and look at the *route reflector* Role label of the Network Elements the Network Element Connection connects. We can allocate a directionality label to the Network Element Interface, according to the role of the Network Element and neighbouring Network Elements.

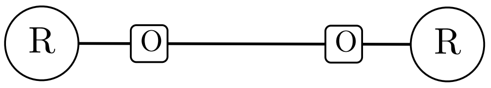
The relationship of the roles and the resulting Network Element Interface label, are shown in Table 5.6. The first column shows the Route Reflector Role label of Network Element A, and the second column shows the Route Reflector Role label of Network Element B. The third column shows the resulting Route Reflector Direction label on Network Element Interface A. The fourth column shows Route Reflector Direction label on Network Element Interface B. The different session labels are shown in Figure 5.30.

Automatic Assignment

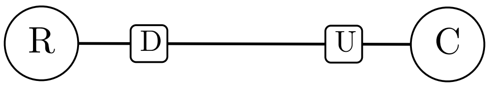
An extension of this approach where we use roles, would be to allocate the iBGP Route Reflector Role labels automatically. This could be done using a graph algorithm. This makes use of the fact that we represent our topologies using a network graph. This could use an algorithm such as a centrality algorithm, to find the Network Elements in an autonomous system according to a set of criteria, such

Table 5.6: Route Reflector Roles and Directions

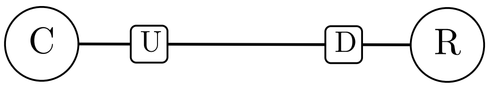
NE A RR Role	NE B RR Role	NEI B RR Direction	NEI B RR Direction
R	R	O	O
R	C	D	U
C	R	U	D
C	C	-	-



(a) Route Reflector to Route Reflector session, with Network Element Interfaces A and B labelled with O for Over.



(b) Route Reflector Client to Route Reflector session, with Network Element Interface A labelled U for Up, and Network Element Interface B labelled D for Down.



(c) Route Reflector to Route Reflector Client session, with Network Element Interface A labelled D for Down, and Network Element Interface B labelled U for Up.

Figure 5.30: Assignment of Route Reflector Directionality labels to Network Element Interfaces based on the Route Reflector label of their Network Elements

as the most central Network Element(s) from the connectivity in the Physical Network View.

This example shows the flexibility in using labels. These labels could either be presented manually by the user on the Network Whiteboard, or allocated algorithmically. Regardless of which is used, the rules in the Design Function, for establishing the Network Element Connections, and denoting the type of session on the Network Element Interfaces, are identical for both the manual, and the automatically allocated case.

5.7.4 *Conclusion*

In this section we have shown that the Design Function approach allows the approach to be extended to accommodate different network design requirements.

5.8 TRANSFER FUNCTION APPROACH

In this section we provide an overview of how the Design Function approach described in this Chapter to produce the Network Views in the Abstract Network can be viewed as a transfer function applied to the Network Whiteboard. For simplicity we assume the Design Functions return a Network View θ_i, as this is a more functional approach than adding each $theta_i$ to the Network Whiteboard Θ as performed in the Design Functions shown in Section 5.6.

$$\begin{aligned} f_{phy}(\omega_i) &= \theta_{phy} \\ f_{l2}(\theta_{phy}) &= \theta_{l2} \\ f_{l2conn}(\theta_{l2}) &= \theta_{l2conn} \\ f_{ip}(\theta_{l2}) &= \theta_{ip} \\ f_{ospf}(\theta_{layer\_2conn}) &= \theta_{ospf} \\ f_{ibgp}(\theta_{phy}) &= \theta_{ibgp} \\ f_{ebgp}(\theta_{layer\_2conn}) &= \theta_{ebgp} \end{aligned} \tag{5.1}$$

$$\begin{aligned} f_{phy}(\omega_i) &= \theta_{phy} \\ f_{layer\_2}(f_{phy}(\omega_i)) &= \theta_{layer\_2} \\ f_{layer\_2conn}(f_{l2}(f_{phy}(\omega_i))) &= \theta_{l2conn} \\ f_{ip}(f_{l2}(f_{phy}(\omega_i))) &= \theta_{ip} \\ f_{ospf}(f_{l2conn}(f_{l2}(f_{phy}(\omega_i)))) &= \theta_{ospf} \\ f_{ibgp}(f_{phy}(\omega_i)) &= \theta_{ibgp} \\ f_{ebgp}(f_{l2conn}(f_{l2}(f_{phy}(\omega_i)))) &= \theta_{ebgp} \end{aligned} \tag{5.2}$$

For each function f we define a modified version f' which takes the complete Abstract Network Model Θ rather than an individual Network View θ_i. This can be performed by a selection function to obtain θ_i from Θ,

$$\begin{aligned} f'_{phy}(\omega_i) &= \theta_{phy} \\ f'_{l2}(\omega_i) &= \theta_{l2} \\ f'_{l2conn}(\omega_i) &= \theta_{l2conn} \\ f'_{ip}(\omega_i) &= \theta_{ip} \\ f'_{ospf}(\omega_i) &= \theta_{ospf} \\ f'_{ibgp}(\omega_i) &= \theta_{ibgp} \\ f'_{ebgp}(\omega_i) &= \theta_{ebgp} \end{aligned} \tag{5.3}$$

$$\begin{aligned}\Theta &= (\theta_{phy}, \theta_{l2}, \theta_{l2conn}, \theta_{ip}, \theta_{ospf}, \theta_{ibgp}, \theta_{ebgp}) \\ &= (f_{phy'}(\omega_i), f_{l2'}(\omega_i), f_{l2conn'}(\omega_i), \\ &\quad f_{ip'}(\omega_i), f_{ospf'}(\omega_i), f_{ibgp'}(\omega_i), f_{ebgp'}(\omega_i)) \\ &= F(\omega_i)\end{aligned} \tag{5.4}$$

Therefore the Abstract Network Model, Θ can be produced by applying the transfer function F to the Network Whiteboard ω_i.

We continue this discussion in Section 6.6 to demonstrate how the set of low-level device configurations can be built from the Network Whiteboard.

5.9 CONCLUSION

Our approach provides a workflow for the systematic construction of Network Views. Network Views can have a dependence on other Network Views. A simple example is the IP addressing Network View being dependent on the Layer 2 Network View. The Design Process allows for this modular flow to be incorporated, by an ordered construction of the Design Functions to construct the Network Views. The approach of Design Functions has addressed Research Question 3: *How can declarative High-Level Network Policy descriptions be transformed into a graph-based intermediate format using a compiler that is extensible in terms of new types of policies and new protocols?*

6 GENERATION OF LOW-LEVEL DEVICE CONFIGURATION STATE

6.1 INTRODUCTION

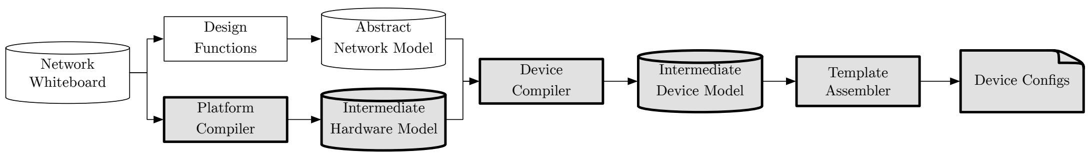


Figure 6.1: Toolchain highlighting Compilation process

In the previous chapter we have described how Network Views are generated from transformations of the Network Whiteboard using Design Functions. From the point of view of the network engineer, and their ability to make use of this information, the intermediate format in which the Network Views are represented is central. This intermediate format provides a clean, device-independent description of the network. However to make the network design practical, this intermediate format needs to be translated into the language of network devices. Device configuration languages are diverse, complex and sometimes inconsistent.

Device configurations use a low-level syntax, which varies between network device vendors, or even within devices produced by the same vendor. However, the network configuration state needs to be consistent across the network, as discussed in Chapter 4 and Chapter 5. In this chapter we discuss how the globally consistent network state, expressed using Network Views can be transformed into the low-level vendor-specific device configurations suitable for individual network device configurations. This is shown in the toolchain steps highlighted in Figure 6.1.

This transformation process, which we denote as Low-Level Configuration Generation Process consists of two phases: a high-level phase which transforms the Network Views into Intermediate Device

Models, and a low-level assembly phase, where the Intermediate Device Models are used to generate device configuration syntax using templates. The high-level phase handles the more complex aspects of Low-Level Configuration Generation Process. The leads to a simplification of the low-level phase, allowing the pure template-driven code generation approach outlined by Arnoldus *et al.* [4].

In this chapter we will show how this approach addresses Research Question 4: *How can the configurations for a diversity of network devices be generated systematically from a graph-based intermediate representation?*

Consider the Network Views in the Abstract Network Model shown in Figure 6.2. Each horizontal plane represents a Network View, and each vertical dashed line corresponds to a single device across each of the Network Views to which it belongs. To generate each individual device configuration we take a *vertical* slice across the Abstract Network Model.

6.1.1 *Chapter Overview*

This chapter is organised as follows. We first provide an overview of the key concepts in the two main phases of the Low-Level Configuration Generation process. This is shown in Figure 6.3. This includes the artefacts created, such as the Intermediate Hardware Model and the process that creates these artefacts, such as the Platform and Device Compilers.

We then examine the artefacts and processes, and provide a detailed discussion of the Platform Compiler and Device Compilers, which transform the Abstract Network Model discussed in Chapter 4 into the Intermediate Hardware Model and Intermediate Device Model. We next provide an example of the first phase compiler process, looking at the steps performed to generate the relevant configuration structure for the interfaces, and the OSPF, iBGP and eBGP routing protocols, for a single device in the Small Internet Case Study from Chapter 4 and Chapter 5. Finally, we discuss phase two which generates the final device configurations through the use of templates.

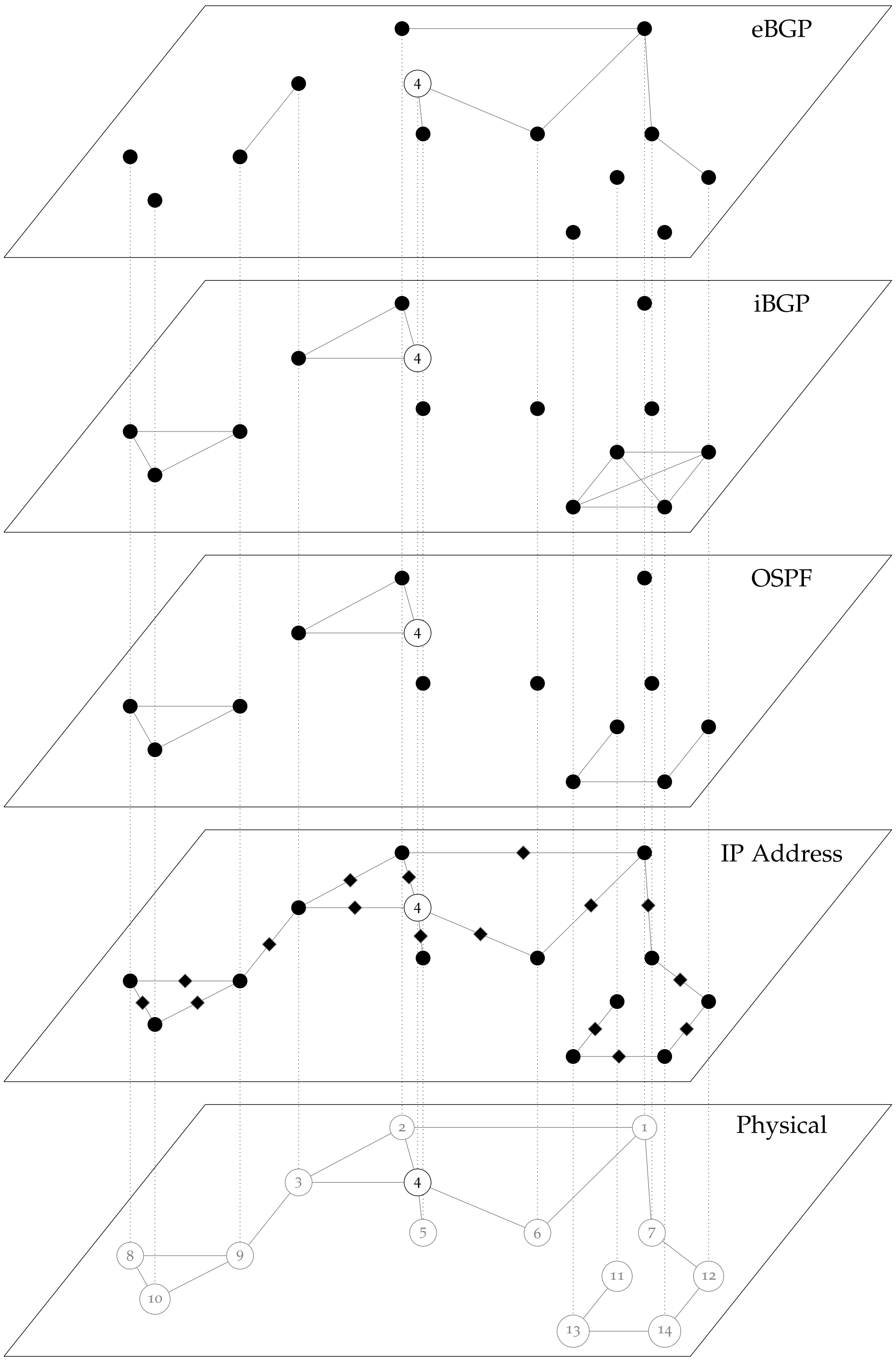


Figure 6.2: Cross-section illustration of the set of Network Views in the Abstract Network Model for the Simplified Small Internet Example, as shown in Figure 4.43. Vertical lines represent the association of Network Elements across Network Views. Each horizontal plane corresponds to a Network View. The Layer 2 Network Views have been omitted for visual simplicity. Conceptually, a device can be *compiled* by vertical traversal of a Network Element along a vertical dashed line between the planes.

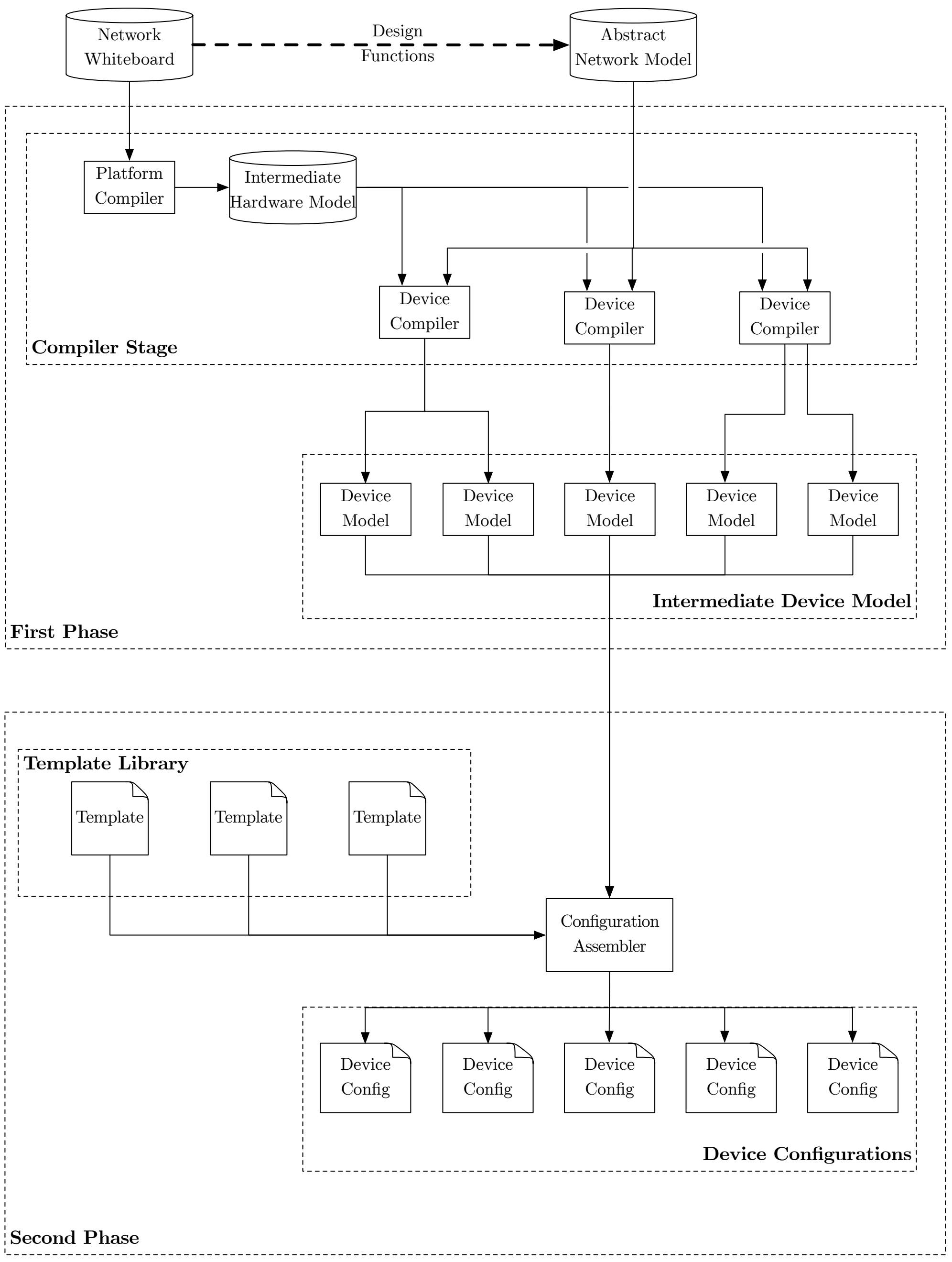


Figure 6.3: Overview Flowchart of Low-Level Configuration Generation Process. The Abstract Network Model is the intermediate format containing Network Views. Note that the Network Whiteboard may be used as input to the process as it may contain the types of network device physical properties.

6.2 MOTIVATION: DEVICE SYNTAX VARIATIONS

We first present a motivating case study of the differences between the low-level device configuration syntax used in different router operating systems. We look at an OSPF example with the interface names, OSPF areas, and OSPF costs shown in Figure 6.4. IP addresses are allocated as shown in Figure 6.5. We look at how the interface and Network Element *r3* would be configured on Quagga, Cisco IOS, Cisco IOS-XR, Cisco NX-OS, and Juniper Junos. The interface naming varies by target platform, and is based on the prefix of interface name shown in Figure 6.4.

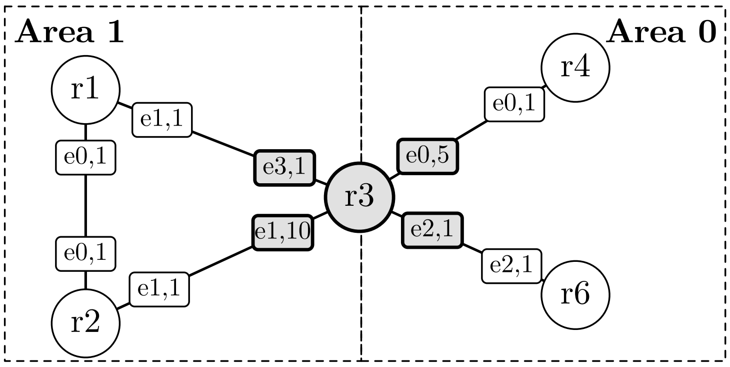


Figure 6.4: Example OSPF Network View. Network Element Interface labels show an example interface *name* and an OSPF *cost*.

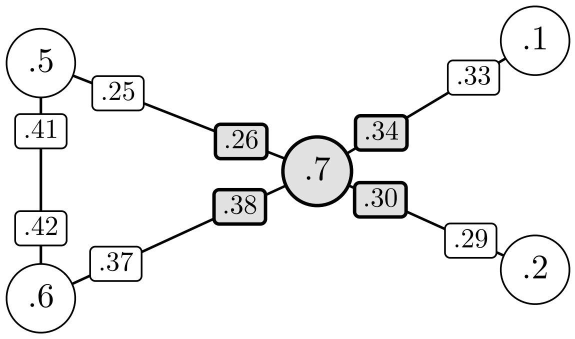


Figure 6.5: Example OSPF Network View, where Network Elements show the final octet of the *10.0.0.x/32* allocation, and Network Element Interface labels show final octet of the *192.168.0.x/30* allocation.

6.2.1 *Quagga*

Quagga has two configuration files, one for the routing process (zebra.conf) and one for the OSPF process (ospfd.conf). An example of the interface and OSPF configuration for Quagga is shown in Listing 6.1. The zebra.conf is responsible for specifying the IP address to use on each interface. The ospfd.conf sets the cost on each interface, and the set of networks to advertise with their relevant area attribute. The loopback interface is configured as passive. An example reference Intermediate Device Model that reflects this structure is shown in Listing 6.2. This has an *interface* block and an *ospf* block.

```
!
! zebra.conf
!
hostname r3
interface eth0
  ip address 192.168.0.34/30
interface eth1
  ip address 192.168.0.38/30
interface eth2
  ip address 192.168.0.30/30
interface eth3
  ip address 192.168.0.26/30
interface lo:1
  ip address 10.0.0.7/32
!
!
!
! ospfd.conf
!
hostname r3
interface eth0
  ip ospf cost 5
interface eth1
  ip ospf cost 10
interface eth2
  ip ospf cost 1
interface eth3
  ip ospf cost 1
!
passive-interface lo:1
!
router ospf
  network 10.0.0.7/32 area 0
  network 192.168.0.32/30 area 0
  network 192.168.0.36/30 area 1
  network 192.168.0.28/30 area 0
  network 192.168.0.24/30 area 1
!
!
```

Listing 6.1: Quagga interface and OSPF configuration

```
{ "hostname": "r3",
 "interfaces": [
  {
   "id": "eth0",
   "cidr": "192.168.0.34/30"
  }, {
   "id": "eth1",
   "cidr": "192.168.0.38/30"
  }, {
   "id": "eth2",
   "cidr": "192.168.0.30/30"
  }, {
   "id": "eth3",
   "cidr": "192.168.0.26/30"
  }, {
   "broadcast": null,
   "id": "lo:1",
   "cidr": "10.0.0.7/32"
  }
 ],
 "ospf": {
  "interfaces": [
   {"cost": 5,  "id": "eth0" },
   {"cost": 10, "id": "eth1" },
   {"cost": 1,  "id": "eth2" },
   {"cost": 1,  "id": "eth3" }
  ],
  "networks": [
   { "area": 0,
    "network": "10.0.0.7/32"},
   { "area": 0,
    "network": "192.168.0.32/30" },
   { "area": 1,
    "network": "192.168.0.36/30" },
   { "area": 0,
    "network": "192.168.0.28/30" },
   { "area": 1,
    "network": "192.168.0.24/30" }
  ],
  "passive_interfaces": [
   { "id": "lo:1" }
] } }
```

Listing 6.2: Example Quagga Device Model

6.2.2 *Cisco IOS*

An example of the interface and OSPF configuration is shown in Listing 6.3. We can see similarities to the configuration of Quagga. The key difference is the single set of interface listings, with the OSPF cost set on the interface along with the IP Address. The networks to be advertised follow a similar format to Quagga, listing the prefix and areas. An example reference Intermediate Device Model shown in Listing 6.4 reflects these similarities and differences between IOS and Quagga.

```
!
hostname r3
!
interface Ethernet0/0
  ip address 192.168.0.34 255.255.255.252
  ip ospf cost 5
  no shutdown
interface Ethernet0/1
  ip address 192.168.0.38 255.255.255.252
  ip ospf cost 10
  no shutdown
interface Ethernet0/2
  ip address 192.168.0.30 255.255.255.252
  ip ospf cost 1
  no shutdown
interface Ethernet0/3
  ip address 192.168.0.26 255.255.255.252
  ip ospf cost 1
  no shutdown
interface Loopback0
  ip address 10.0.0.7 255.255.255.255
!
router ospf
  network 192.168.0.32 0.0.0.3 area 0
  network 192.168.0.36 0.0.0.3 area 1
  network 192.168.0.28 0.0.0.3 area 0
  network 192.168.0.24 0.0.0.3 area 1
  network 10.0.0.7 0.0.0.0 area 0
  passive-interface Loopback0
```

Listing 6.3: IOS interface and OSPF configuration

```
{ "hostname": "r3",
 "interfaces": [
  { "id": "Ethernet0/0",
   "ip": "192.168.0.34",
   "netmask": "255.255.255.252",
   "ospf_cost": 5,
   "shutdown": false },
   { "id": "Ethernet0/1",
   "ip": "192.168.0.38",
   "netmask": "255.255.255.252",
   "ospf_cost": 10,
   "shutdown": false },
   { "id": "Ethernet0/2",
   "ip": "192.168.0.30",
   "netmask": "255.255.255.252",
   "ospf_cost": 1,
   "shutdown": false },
   { "id": "Ethernet0/3",
   "ip": "192.168.0.26",
   "netmask": "255.255.255.252",
   "ospf_cost": 1,
   "shutdown": false },
   { "id": "Loopback0",
   "ip": "10.0.0.7",
   "netmask": "255.255.255.255" }
 ],
 "ospf": {
  "networks": [
   {"area": 0,
    "hostmask": "0.0.0.3",
    "prefix": "192.168.0.32"},
   {"area": 1,
    "hostmask": "0.0.0.3",
    "prefix": "192.168.0.36"},
   {"area": 0,
    "hostmask": "0.0.0.3",
    "prefix": "192.168.0.28"},
   {"area": 1,
    "hostmask": "0.0.0.3",
    "prefix": "192.168.0.24"},
   {"area": 0,
    "hostmask": "0.0.0.0",
    "prefix": "10.0.0.7"}
  ],
  "passive_interfaces": [
   {"id": "Loopback0"}
] } }
```

Listing 6.4: Example IOS Device Model

6.2.3 *Cisco IOS-XR*

An example of the interface and OSPF configuration for Cisco IOS-XR is shown in Listing 6.5. Here we can see that only the IP address is configured directly on the interface. The OSPF block is configured in a different structure, grouped by OSPF area. Within each area, the interfaces in that area are listed together with their cost. The passive loopback is configured in the relevant area to which it belongs. An example reference Intermediate Device Model shown in Listing 6.6, where the interfaces are grouped by their area.

```
!
hostname r3
!
interface Ethernet0/0
  ipv4 address 192.168.0.34 255.255.255.252
  no shutdown
interface Ethernet0/1
  ipv4 address 192.168.0.38 255.255.255.252
  no shutdown
interface Ethernet0/2
  ipv4 address 192.168.0.30 255.255.255.252
  no shutdown
interface Ethernet0/3
  ipv4 address 192.168.0.26 255.255.255.252
  no shutdown
interface Loopback0
  ipv4 address 10.0.0.7 255.255.255.255
!
router ospf
  area 0
    interface Ethernet0/0
      cost 5
    !
    interface Ethernet0/2
      cost 1
    !
    interface Loopback0
      passive enable
    !
  !
  area 1
    interface Ethernet0/1
      cost 10
    !
    interface Ethernet0/3
      cost 1
    !
  !
```

Listing 6.5: IOS-XR interface and OSPF configuration

```
{ "hostname": "r3",
 "interfaces": [
  { "id": "Ethernet0/0",
   "ip": "192.168.0.34",
   "netmask": "255.255.255.252",
   "shutdown": false
  }, {
   "id": "Ethernet0/1",
   "ip": "192.168.0.38",
   "netmask": "255.255.255.252",
   "shutdown": false
  }, {
   "id": "Ethernet0/2",
   "ip": "192.168.0.30",
   "netmask": "255.255.255.252",
   "shutdown": false
  }, {
   "id": "Ethernet0/3",
   "ip": "192.168.0.26",
   "netmask": "255.255.255.252",
   "shutdown": false
  }, {
   "id": "Loopback0",
   "ip": "10.0.0.7",
   "netmask": "255.255.255.255"
  }
 ],
 "ospf": {
  "interfaces_by_area": {
   "0": [
    {"cost": 5,
     "id": "Ethernet0/0"},
    {"cost": 1,
     "id": "Ethernet0/2"},
    {"id": "Loopback0",
     "passive": true}
   ],
   "1": [
    {"cost": 10,
     "id": "Ethernet0/1"},
    {"cost": 1,
     "id": "Ethernet0/3"}
 ] } } }
```

Listing 6.6: Example IOS-XR Device Model

6.2.4 *Cisco NX-OS*

An example of the interface and OSPF configuration for Cisco NX-OS is shown in Listing 6.7. Here we can see that the IP address is specified in the CIDR format of *a.b.c.d/x*. The OSPF configuration is done directly on the interface, with OSPF process id listed with the area, and the cost set directly on the interface. The example reference Intermediate Device Model shown in Listing 6.8 reflects this flat structure.

```
!
hostname r3
!
feature ospf
!
interface Ethernet0/0
  ip address 192.168.0.34/30
  ip router ospf 1 area 0
  ip ospf cost 5
  no shutdown
interface Ethernet0/1
  ip address 192.168.0.38/30
  ip router ospf 1 area 1
  ip ospf cost 10
  no shutdown
interface Ethernet0/2
  ip address 192.168.0.30/30
  ip router ospf 1 area 0
  ip ospf cost 1
  no shutdown
interface Ethernet0/3
  ip address 192.168.0.26/30
  ip router ospf 1 area 1
  ip ospf cost 1
  no shutdown
interface Loopback0
  ip address 10.0.0.7/32
  ip router ospf 1 area 0
  ip ospf cost
!
router ospf
  router-id 10.0.0.7
```

Listing 6.7: NX-OS interface and OSPF configuration

```
{ "hostname": "r3",
 "interfaces": [
  {
   "cidr": "192.168.0.34/30",
   "id": "Ethernet0/0",
   "ospf_area": 0,
   "ospf_cost": 5,
   "ospf_process_id": 1,
   "shutdown": false
  }, {
   "cidr": "192.168.0.38/30",
   "id": "Ethernet0/1",
   "ospf_area": 1,
   "ospf_cost": 10,
   "ospf_process_id": 1,
   "shutdown": false
  }, {
   "cidr": "192.168.0.30/30",
   "id": "Ethernet0/2",
   "ospf_area": 0,
   "ospf_cost": 1,
   "ospf_process_id": 1,
   "shutdown": false
  }, {
   "cidr": "192.168.0.26/30",
   "id": "Ethernet0/3",
   "ospf_area": 1,
   "ospf_cost": 1,
   "ospf_process_id": 1,
   "shutdown": false
  }, {
   "cidr": "10.0.0.7/32",
   "id": "Loopback0",
   "ospf_area": 0,
   "ospf_process_id": 1
  }
 ],
 "ospf": {"router_id": "10.0.0.7"}
}
```

Listing 6.8: Example NX-OS Device Model

6.2.5 *Juniper Junos*

An example Junos configuration is shown in Listing 6.9. We can see a braces-based structure rather than indentation. Interfaces are configured with the CIDR address format, within a *unito* block. OSPF interfaces are grouped by area, and specified with their cost. The Intermediate Device Model is shown in Listing 6.10.

```
interfaces {
 Ethernet0/0 {
  unit 0 {
    family inet {
      address 192.168.0.34/30;
    }
  }
}
 Ethernet0/1 {
  unit 0 {
    family inet {
      address 192.168.0.38/30;
    }
  }
}
 Ethernet0/2 {
  unit 0 {
    family inet {
      address 192.168.0.30/30;
    }
  }
}
 Ethernet0/3 {
  unit 0 {
    family inet {
      address 192.168.0.26/30;
    }
  }
}
 lo0 {
  unit 0 {
    family inet {
      address 10.0.0.7/32;
    }
  }
}
protocols {
  ospf {
    area 0 {
      Ethernet0/0 {
        metric 5;
      }
      Ethernet0/2 {
        metric 1;
      }
      lo0 {
        passive;
      }
    }
    area 1 {
      Ethernet0/1 {
        metric 10;
      }
      Ethernet0/3 {
        metric 1;
      }
    }
  }
}
```

Listing 6.9: Junos configuration

```
{ "hostname": "r3",
  "interfaces": [
    {
      "cidr": "192.168.0.34/30",
      "id": "Ethernet0/0"
    }, {
      "cidr": "192.168.0.38/30",
      "id": "Ethernet0/1"
    }, {
      "cidr": "192.168.0.30/30",
      "id": "Ethernet0/2"
    }, {
      "cidr": "192.168.0.26/30",
      "id": "Ethernet0/3"
    }, {
      "cidr": "10.0.0.7/32",
      "id": "lo0"
    }
  ],
  "ospf": {
    "interfaces_by_area": {
      "0": [
        {
          "id": "Ethernet0/0",
          "metric": 5
        }, {
          "id": "Ethernet0/2",
          "metric": 1
        }, {
          "id": "lo0",
          "passive": true
        }
      ],
      "1": [
        {
          "id": "Ethernet0/1",
          "metric": 10
        }, {
          "id": "Ethernet0/3",
          "metric": 1
} ] } } }
```

Listing 6.10: Example Junos Device Model

6.2.6 *Example Code*

The code to setup this example is shown in Listing F.1 and uses AutoNetkit, a Python implementation of our approach. We introduce AutoNetkit in Chapter 6. The base device compiler is shown in Listing F.2. The other compilers and templates are as shown in Table 6.1. Finally, the code to compile and render the configurations is shown in Listing F.14.

Table 6.1: Summary of example code for OSPF example

Target Device	Compiler	Template
Quagga	Listing F.3	Listing F.9
IOS	Listing F.4	Listing F.10
IOS-XR	Listing F.5	Listing F.11
NX-OS	Listing F.6	Listing F.12
Junos	Listing F.7	Listing F.13

6.2.7 *Discussion*

In this example we have shown how expressing the same interface and OSPF configuration can vary between different low-level device syntaxes., with some expressing OSPF configuration on the interface, some in the OSPF block per-interface, and others grouped by the OSPF area.

We have seen that the difference between the braces and indentation formatting in Junos is abstracted away by the templates, with the Intermediate Device Model similar to that of IOS-XR. We have also seen similarities between the configurations for IOS and Quagga. These similarities hint at patterns, which may allow the re-use of logic in the Device Compilers.

As we have discussed, there are subtle differences in how different target devices represent the same set of configuration objectives. We have also seen common patterns between different devices. These patterns can differ between the devices, so that one pattern may apply to device *a* and device *b*, whereas a different pattern may apply to device *b* and device *c*. This motivates two aspects of our approach.

The first is to be able to represent the information expressed in each device configuration, separate from the syntax used to describe this information. This intermediate representation was shown alongside each device configuration as the example device model. This is shown in our toolchain as the Intermediate Device Model.

The second aspect of our approach is to formalise the transform that generates these device models. This formalisation allows us to share logic for common patterns, and to use specific logic where the configuration of the device differs to other devices. This transform step is shown in our toolchain as the Device Compiler.

This motivating example has also shown us that the interface naming can differ between target devices. This naming depends on the specific configuration of the target device, and relates to the physical inventory, as compared to the routing protocol configurations which form the logical inventory. As we represent these routing protocols and related network design information using Network Views in the Abstract Network Model, we introduce the Intermediate Hardware Model, which represents the physical set up of a target device. This Intermediate Hardware Model can be used by the Device Compiler, along with the Abstract Network Model, to produce the Intermediate Device Model. This allows the physical inventory and logical inventory to be combined to produce the final set of information to be configured on the target device. This intermediate hardware model can be produced in the Platform Compiler which takes into account the target platform on which the device is intended, including a physical testbed network, or a simulation network.

We will present these artefacts and processes in the next section.

6.3 ARTEFACTS AND PROCESSES

In this section we describe the key artefacts that are manipulated in the generation of Low-Level Configuration state at the processes that use and manipulate these artefacts. The artefacts discussed in this section are the Intermediate Hardware Model, the Intermediate Device Model, and Templates. The processes discussed are the Platform Compiler, the Device Compiler, and the Configuration Assembler.

6.3.1 *Artefacts*

In this section we provide further details of the artefacts produced in the intermediate or final steps of the two phase configuration generation process.

Intermediate Hardware Model

The **Intermediate Hardware Model** contains the physical characteristics of the device that will be used to implement the configuration. The Intermediate Hardware Model specifies the physical inventory of the network. This contrasts with the Abstract Network Model which contains the logical inventory of the network.

An example of a physical characteristic is the interface naming allocations for a device, such as *eth0* or *FastEthernet0/2*, which depend on the hardware inventory of a device. The Intermediate Hardware Model can also include information relating to out-of-band management access to the device, such as the *tap* management interface in the Netkit platform simulations. An example of an Intermediate Hardware Model is shown in Figure 6.6 for Network Element *4* of the Simplified Small Internet Example.

Intermediate Device Model

The **Intermediate Device Model** contains code for each device in a format that is complaint with the abstract syntax tree and semantics of the device. For every device in the network, there will be a block of code that represents the network state for the device. This code is called a *Device Model*.

Each device has an entry in the Intermediate Device Model which combines all of the information from the Network Views, together with hardware-specific information from the Intermediate Hardware Model, that are required to configure that device. Despite the fact that devices have different concrete syntax and semantics, at the Device

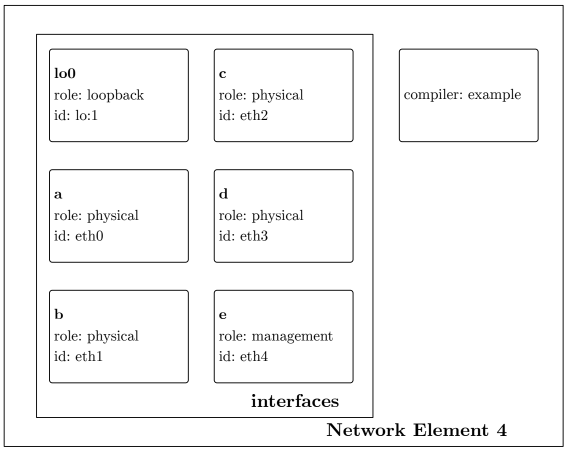


Figure 6.6: Example of Intermediate Hardware Model for Network Element *4* of Simplified Small Internet Example.

Model level we have a more generalised representation, which we show here using JSON.

The Device Model is analogous to the Abstract Syntax Tree for a specific target device. The aim of the Device Model is to capture the *semantics* of the target device platform, to simplify the final template assembly step. Conceptually, each Device Model could be represented in simple JSON format, as the entries consist of key-value pairs organised into lists, such as a list of interfaces.

Due to the differences in the target device semantics, the information format in the Intermediate Device Model is specific to the target platform. This contrasts to traditional compiler construction theory where the intermediate model is fully device independent. The Intermediate Device Model for a Cisco IOS target device would differ from that of a Juniper Junos device, for the same Network Element in the Abstract Network Model. We show examples of Intermediate Device Models alongside extracts from device configuration syntax in We show a motivating example with the variations between the device syntax of a number of common devices in Section 6.2.

Intermediate Platform Model

The **Intermediate Platform Model** is analogous to the Intermediate Device Model, but contains the information required for the platform. For a testbed this could include the operating systems to run on the virtual network devices, and the virtual links to establish between the virtual devices. For a physical hardware testbed, this may be an inventory of the physical hardware to use, and a wiring diagram of how the devices are to be connected. The Intermediate Platform

Model is particularly useful in automating the experimentation process by allowing the experiment setup and device configurations to be generated by the same process. We use this to evaluate our approach in Chapter 7 and Chapter 8.

Templates

Templates are used to translate the Device Models into concrete syntax for a particular target device. We show an example with the device syntax variations for a number of common devices in Section 6.2.

6.3.2 *Processes*

In this section we discuss the processes which generate the artefacts discussed in the previous section.

Platform Compiler

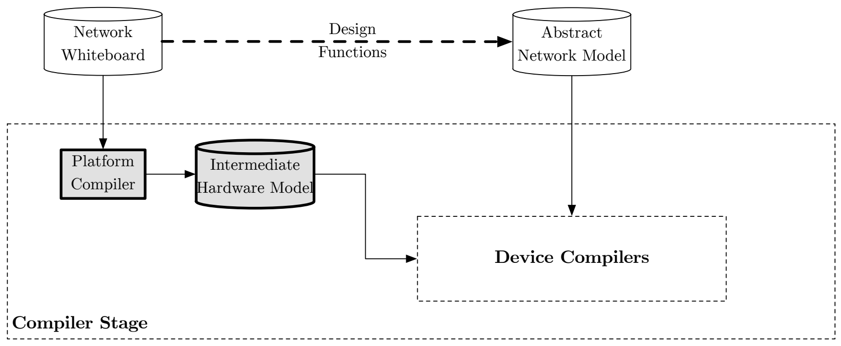


Figure 6.7: Platform Compilation process

The Platform Compiler takes the physical device information from the Network Whiteboard and generates the Intermediate Hardware Model. This process is shown in Figure 6.7. We show an example in Section 6.4.1.

Device Compiler

The Device Compiler merges information from the Intermediate Hardware Model and the Abstract Network Model, as shown in Figure 6.8. We show an example in Section 6.4.2. Conceptually, the Device Compiler vertically traverses the Network Views of the Abstract Network Model shown in Figure 6.2.

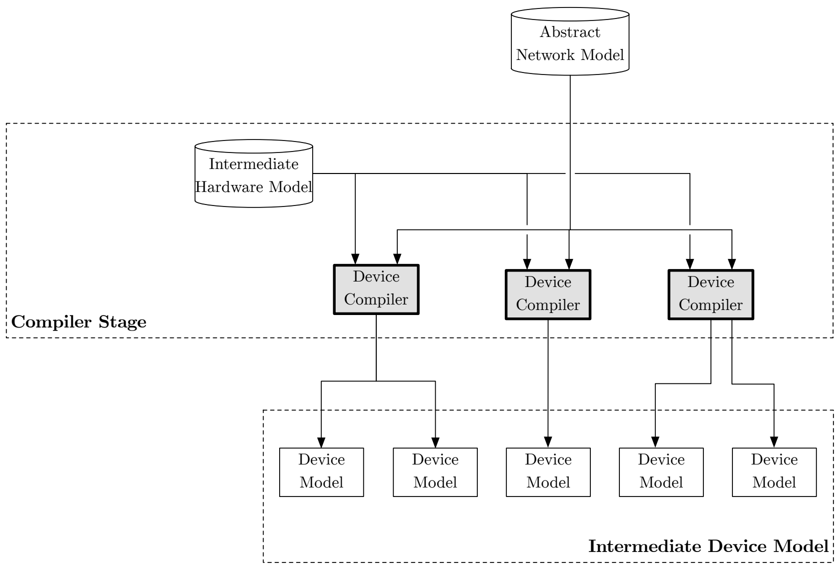


Figure 6.8: Device Compilation process

Configuration Assembler

The Configuration Assembler converts the Device Model entries into concrete Device Configurations. This is performed through the use of templates. This assembly process is shown in Figure 6.9.

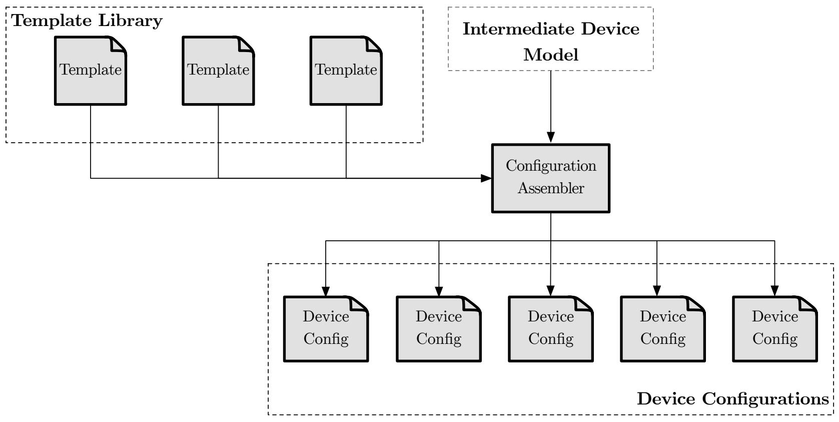


Figure 6.9: Configuration Assembly process using templates

Platform Configuration

The Platform Compiler can also be used to produce the Intermediate Platform Model in addition to the Intermediate Device Model. This can then be rendered using a Platform Assembler, which uses a Platform Template to generate the Platform Configuration. This process is shown in Figure 6.10.

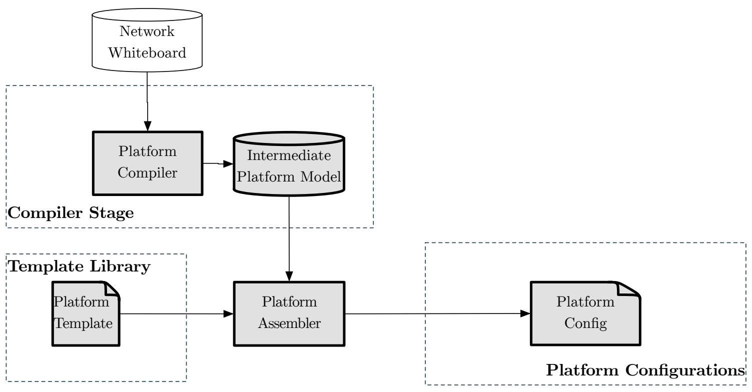
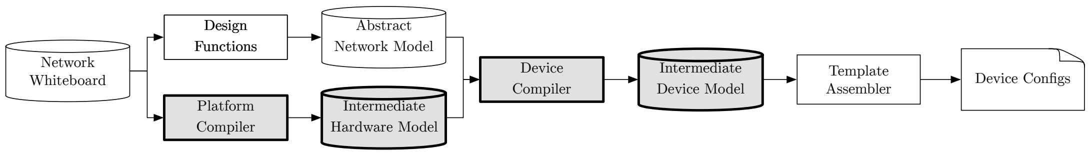


Figure 6.10: Platform Configuration generation and assembly

6.3.3 *Conclusion*

In this section we have introduced the major artefacts and processes for the generation of low-level configuration state. In the next section we provide an example of the variations representations in low-level device configurations. We then give a step-by-step example for a particular Network Element in the Simplified Small Internet Example.

6.4 Compilation for Simplified Small Internet Example



We now present a walkthrough of the compilation process for a specific Network Element in the Simplified Small Internet Example from Chapter 4 and Chapter 5.

In Figure 6.11 we show a specific instance of the generic workflow shown in Figure 6.3. As previously shown, this example has seven network views and fourteen Network Elements. This results in fourteen device configurations being generated. As all devices in this example are routers, we can use the same router device compiler for each Network Element.

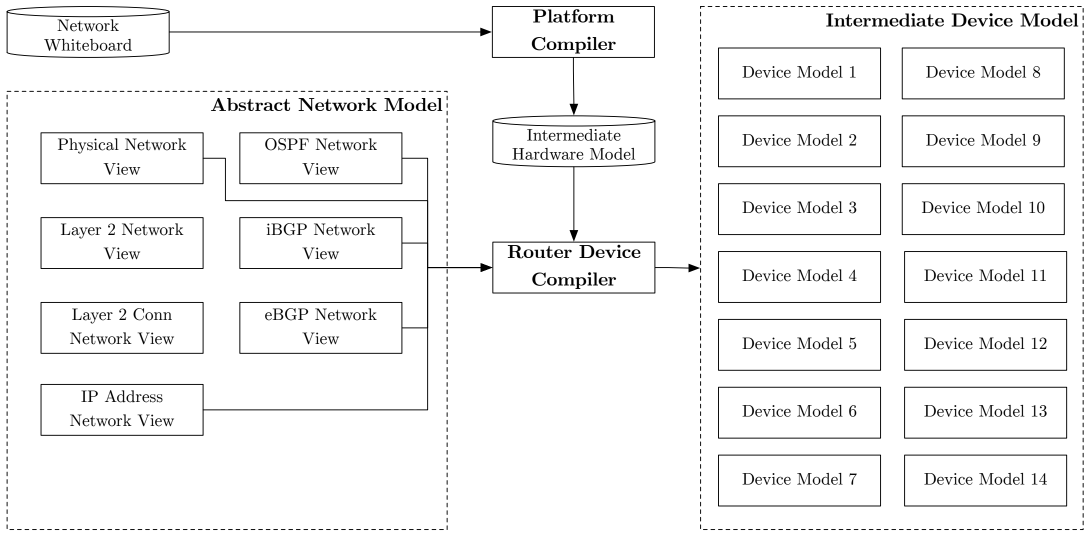
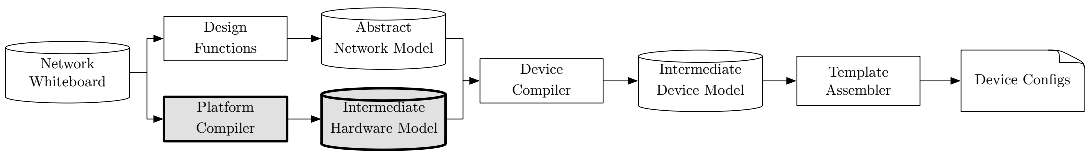


Figure 6.11: Workflow for compilation process from the Network Whiteboard and Abstract Network Model to the Intermediate Device Model using the Platform and Device Compilers.

6.4.1 *Platform Compiler*



In this section we describe how the Platform Compiler takes the interface information from the Network Whiteboard into an Intermediate Hardware Model. We will explain how the management interfaces are allocated and named. An example of this workflow is shown in Figure 6.12. We will walk through each component of the Intermediate Hardware Model shown previously in Figure 6.6.

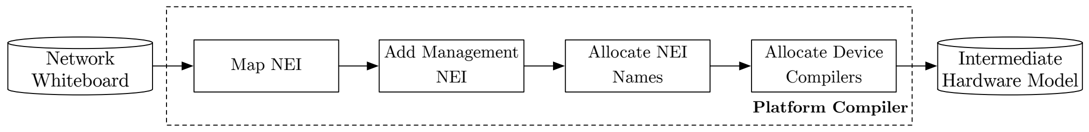


Figure 6.12: Workflow for Intermediate Hardware Model.

Map Network Element Interfaces

The first step is to collate the interfaces in the Intermediate Hardware Model. This is done by iterating over the physical Network Element Interfaces in the Network Whiteboard. This is shown in Figure 6.13.

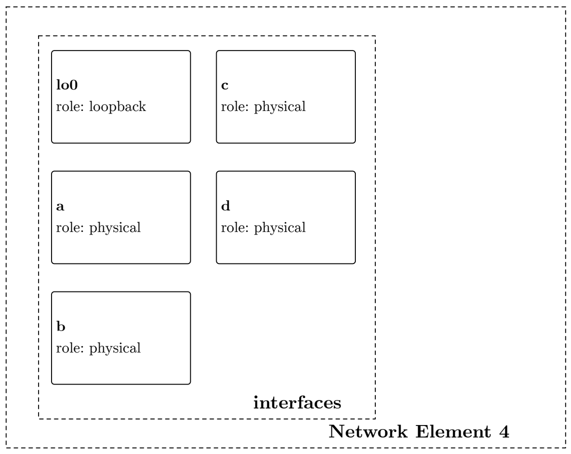


Figure 6.13: Example of Intermediate Hardware Model for Network Element 4 of Simplified Small Internet Example showing mapping of interfaces from the Network Whiteboard.

Add Management Network Element Interfaces

The next step is to allocate out-of-band management interfaces. This is particularly useful in simulation or emulation environments, where the environment requires the use of an interface for management purposes. An example of this allocation is shown in Figure 6.14.

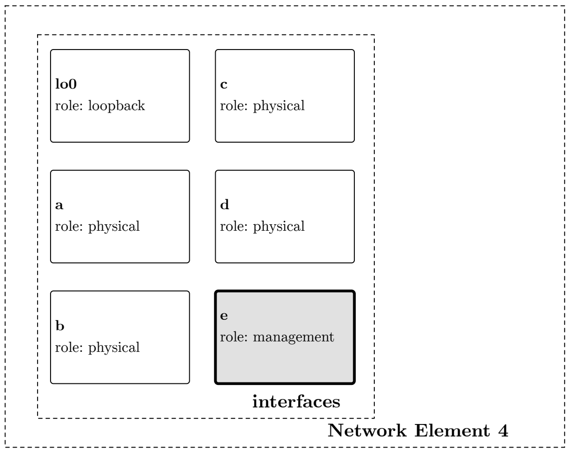


Figure 6.14: Example of Intermediate Hardware Model for Network Element 4 of Simplified Small Internet Example showing addition of management interfaces.

Network Element Interface Naming

The next step is to apply naming to the interface types. For example, the data and management interfaces are allocated as *eth0*, *eth1*, *eth2*, *etc*, and the loopback is allocated as *lo:1*. This is shown in Figure 6.15.

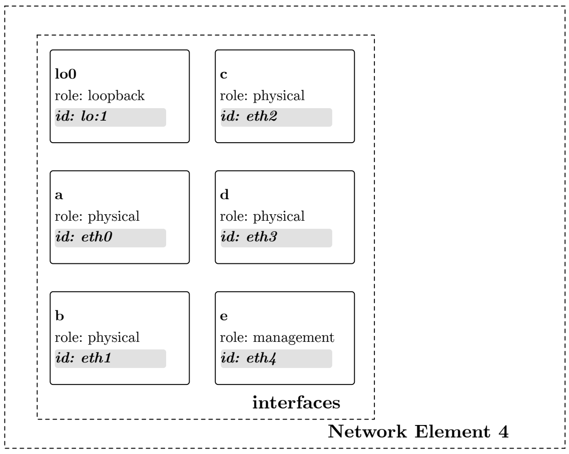


Figure 6.15: Example of Intermediate Hardware Model for Network Element 4 of Simplified Small Internet Example showing interface name allocation.

Specification of Device Compiler

The final step is to specify the appropriate Device Compiler for the target device. This can take into account both the platform, and the relevant Network Element Labels such as *device_type* or *syntax*. An example is shown in Figure 6.16, where the *example* compiler has been specified.

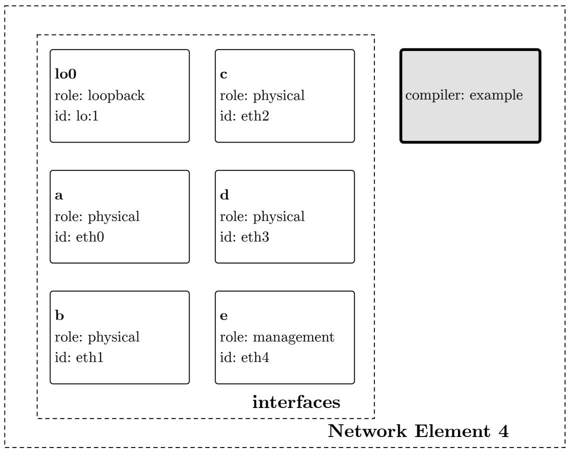
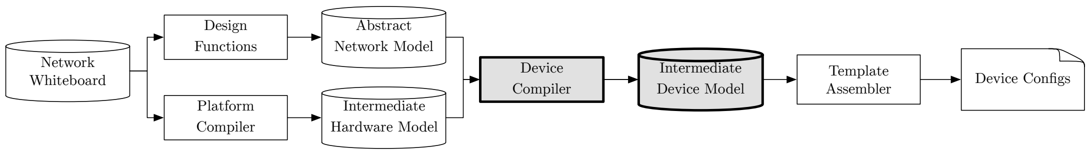


Figure 6.16: Example of Intermediate Hardware Model for Network Element 4 of Simplified Small Internet Example showing Device Compiler specification

Conclusion

In this section we have shown the key steps involved in the Platform Compiler process, including the Mapping of Network Element Interfaces, addition of management interfaces, naming of interfaces, and specification of the device compiler. This process can be used to produce the example Intermediate Device Model shown in Figure 6.6. In the next section we will show the device compilation process.

6.4.2 *Device Compiler*



We now show the steps involved in the Router Device Compiler, which takes the Network Views and the Intermediate Hardware Model, and generates the Intermediate Device Model. For this example we show this for a specific Network Element, *r4*. An overview of this process is shown in Figure 6.17. This Router Device Compiler will construct the Intermediate Device Model for an idealised target device, which is based on a simplified version of the Quagga configuration syntax. We use this simplified Device model to explain the concepts, and provide a more complete example of the device compilation process, in Chapter 6 and Chapter 7.

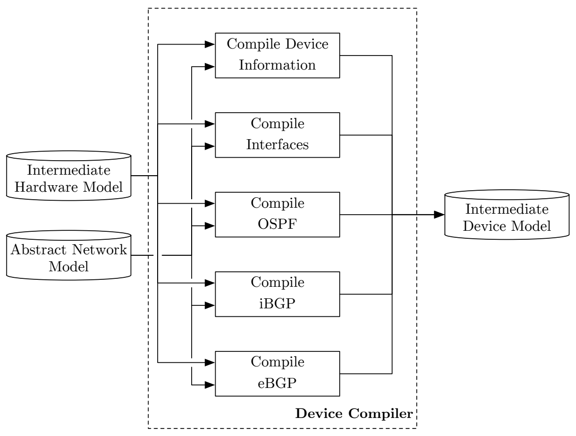


Figure 6.17: Workflow for Intermediate Device Model construction

We first describe the general device information. We then discuss the how interface information is generated for the Intermediate Device Model. For each routing protocol we then describe the Intermediate Device Model is generated for each protocol.

Device Model Example

An example Intermediate Device Model for Network Element 4 of the Simplified Small Internet Example is shown in Figure 6.19. This Intermediate Device Model is highlighted in Figure 6.18.

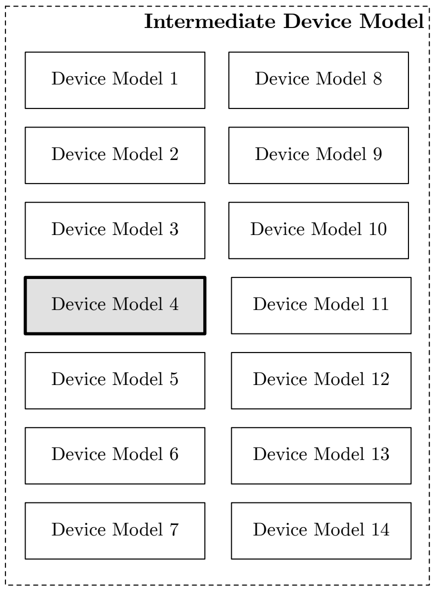


Figure 6.18: Set of all Intermediate Device Models with Device Model for Network Element *4* highlighted.

Device Information

The device information captures information relating to the device in general at the device-global level, rather than for specific protocols. An example is shown in Listing 6.11, based on the information in the Device Model which has been highlighted in Figure 6.20.

```
{
  "asn": 20,
  "hostname": "r4"
}
```

Listing 6.11: Example device global information

We also describe an example utility function for interface description shown in Algorithm 6.13. This is used in the other components of the Device Compiler.

Algorithm 6.13 Pseudo-Code for Interface Description utility function

```
function INTERFACE_DESCRIPTION($\theta_i, \pi_i, \tau_i$)
    $X \leftarrow$ NEIGH_NE($\theta_{phy}, \pi_i, \tau_i$)            ▷ Neighbor NEs of NEI
    names $\leftarrow$ NAME_Π($\theta_i, \pi_i$)$\forall \pi_i \in X$  ▷ Name of each neighbor NE
    return STR_CONCAT(names, ',')        ▷ Concatenate with comma
end function
```

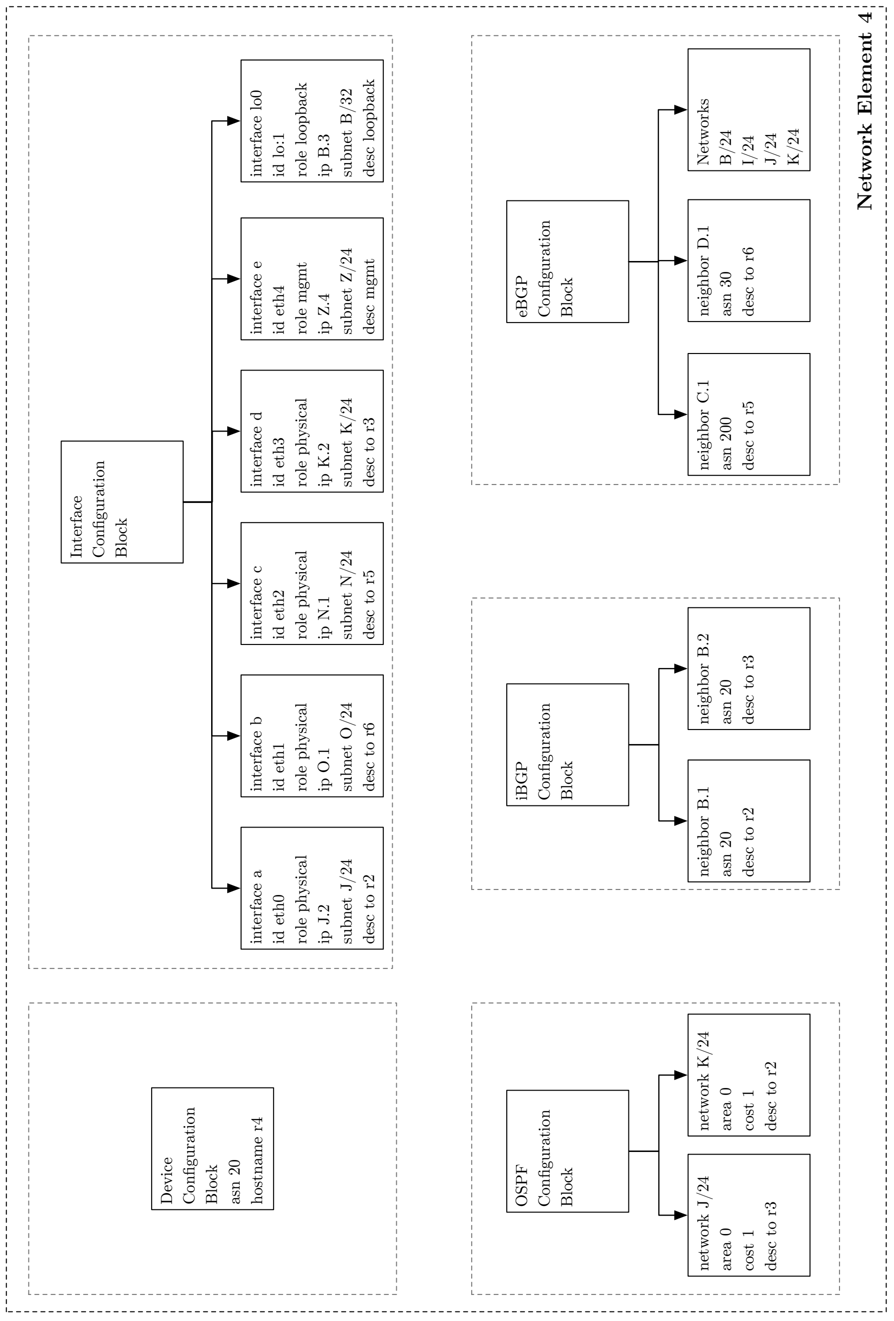

Network Element 4
Device Configuration Block asn 20 hostname r4
Interface Configuration Block
interface a id eth0 role physical ip J.2 subnet J/24 desc to r2
interface b id eth1 role physical ip O.1 subnet O/24 desc to r6
interface c id eth2 role physical ip N.1 subnet N/24 desc to r5
interface d id eth3 role physical ip K.2 subnet K/24 desc to r3
interface e id eth4 role mgmt ip Z.4 subnet Z/24 desc mgmt
interface lo0 id lo:1 role loopback ip B.3 subnet B/32 desc loopback
OSPF Configuration Block
network J/24 area 0 cost 1 desc to r3
network K/24 area 0 cost 1 desc to r2
iBGP Configuration Block
neighbor B.1 asn 20 desc to r2
neighbor B.2 asn 20 desc to r3
eBGP Configuration Block
neighbor C.1 asn 200 desc to r5
neighbor D.1 asn 30 desc to r6
Networks B/24 I/24 J/24 K/24

Figure 6.19: Example Intermediate Device Model for Network Element *4* of Simplified Small Internet Example

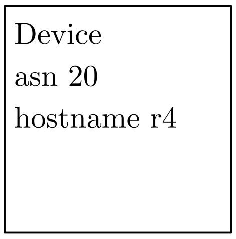
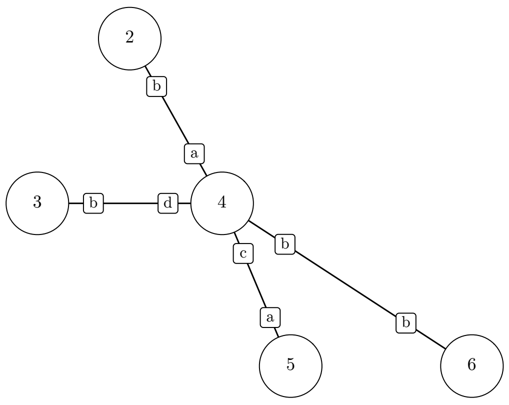
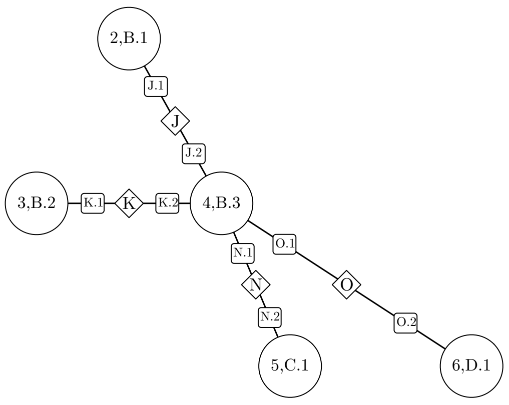


Figure 6.20: Subsection of Device Model of Network Element 4, showing Device Information.

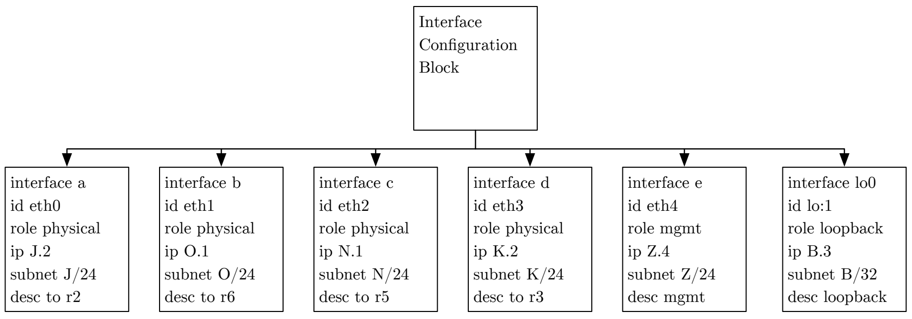
Interfaces



(a) Subsection of Physical Network View for Network Element 4



(b) Subsection of IP Address Network View for Network Element 4



(c) Interfaces subsection of Device Model of Network Element 4

Figure 6.21: Subsection of Physical Network View and Interfaces section of Device Model for Network Element 4

The Interfaces Device Model section is constructed by iterating over the neighbours of the Network Element in the Physical Network View, and the *Loopback* Network Element Interface, and is shown in Figure 6.21. The Interface identifier in the Device Model is mapped from the Network Element Interface Identifier in the Physical Network View. The Interface Role is mapped from the Network Element

Interface Role, and can be used in the template logic in the Assembly step of the compilation process. The IP label is mapped from the IP Address Label on the Network Element Interface, Subnet from the Subnet Label of the neighbour Pseudo Network Element in the IP Address Network View, and the description label from the neighbour Network Element name in the Physical Network View. This process is shown in the pseudo-code example in Algorithm 6.14. An example section of the Intermediate Device Model for the interface information is shown in Listing 6.16.

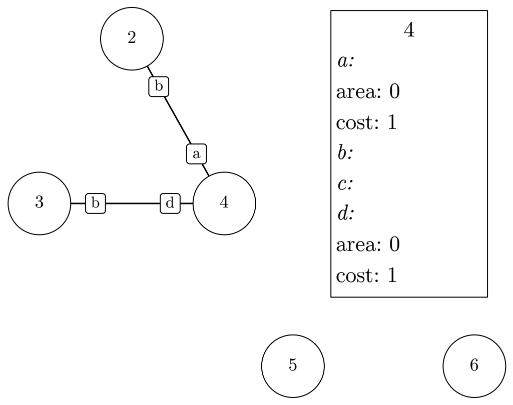
```
[ {
  "desc": "to r2",
  "id": "a",
  "ip": "J.2",
  "role": "physical",
  "subnet": "J"
 }, {
  "desc": "to r6",
  "id": "b",
  "ip": "O.1",
  "role": "physical",
  "subnet": "O"
 }, {
  "desc": "to r5",
  "id": "c",
  "ip": "N.1",
  "role": "physical",
  "subnet": "N"
 }, {
  "desc": "to r3",
  "id": "d",
  "ip": "K.2",
  "role": "physical",
  "subnet": "K"
 }, {
  "desc": "loopback",
  "id": "lo0",
  "ip": "B.3",
  "role": "loopback",
  "subnet": "B"
 } ]
```

Listing 6.12: Example JSON format for Interfaces Component of Intermediate Device Model for Network Element 4

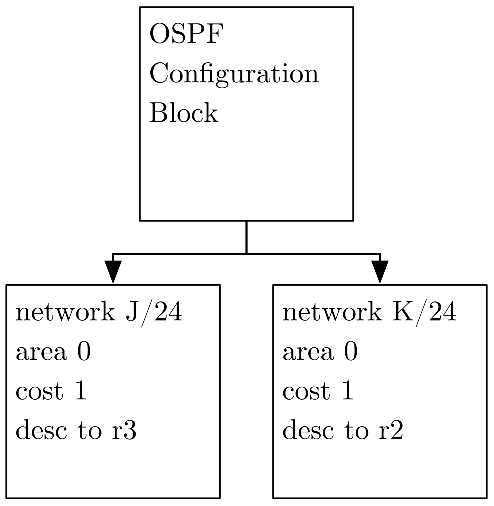
Algorithm 6.14 Pseudo-Code for interfaces component of device compiler

```
function INTERFACES($\Theta$, IHM, $\pi_i$)
    $X \leftarrow \varnothing$
    $\theta_{phy} = NV(\Theta, physical)$                ▷ Physical Network View
    $\theta_{ip} = NV(\Theta, ip)$                        ▷ IP Network View
    $t \leftarrow NEI(\theta_{phy}, \pi_i)$               ▷ NEI for NE $\pi_i$
    for $\tau_i \in t$ do
        $x_{id} \leftarrow \lambda_{IHM}(IHM, \pi_i, \tau_i, id)$     ▷ id from Intermediate Hardware Model
        $x_{role} \leftarrow \rho_T(\theta_{phy}, \pi_i, \tau_i)$
        $x_{ip} \leftarrow \lambda_T(\theta_{ip}, \pi_i, \tau_i, ip)$         ▷ IP from IP NV
        $x_{subnet} \leftarrow \lambda_T(\theta_{ip}, \pi_i, \tau_i, subnet)$   ▷ Subnet from IP NV
        $x_{descr} \leftarrow$ INTERFACE_DESCRIPTION$(\theta_{phy}, \pi_i, \tau_i)$
        $X \cap \{(id, x_{id}), (role, x_{role}), (ip, x_{ip}), (subnet, x_{subnet}), (desc, x_{desc})\}$
    end for
    return X
end function
```

OSPF



(a) Subsection of OSPF Network View for Network Element 4. The Labels for the Network Element Interfaces of Network Element 4 are shown in the rectangle.



(b) OSPF subsection of Device Model of Network Element 4

Figure 6.22: Subsection of OSPF Network View and Device Model for Network Element 4

The OSPF Device Model section is constructed by iterating over the neighbours of the Network Element in the OSPF Network View, and is shown in Figure 6.22.

The Network label in the Device Model is mapped from the Subnet label of the Pseudo Network Element in the IP Address Network View. It can be seen that the Network Label is the same value as used in the Interfaces section of the Device Model. This demonstrates the repetition which occurs in construction the Device Model. The Area and Cost Labels in the Device Model are mapped from the Area and Cost labels of the Network Element interfaces in the OSPF Network View. A pseudo-code example of this process is shown in Algorithm 6.15, with an example Intermediate Device Model extract for OSPF shown in Listing F.15.

```
[
 {
  "area": 0,
  "cost": 1,
  "desc": "to r5",
  "network": "J/24"
 }, {
  "area": 0,
  "cost": 1,
  "desc": "to r6",
  "network": "K/24"
 }
]
```

Listing 6.13: Example JSON format for OSPF Component of Intermediate Device Model for Network Element 4

Algorithm 6.15 Pseudo-Code for OSPF component of device compiler

```
function OSPF($\Theta, \pi_i$)
    $X \leftarrow \varnothing$
    $\theta_{ospf} = NV(\Theta, ospf)$                    ▷ Physical Network View
    $\theta_{ip} = NV(\Theta, ip)$                        ▷ IP Network View
    $t \leftarrow NEI(\theta_{ospf}, \pi_i)$              ▷ NEI for NE $\pi_i$
    for $\tau_i \in t$ do
        $x_{network} \leftarrow \lambda_T(\theta_{ip}, \pi_i, \tau_i, ip)$        ▷ Network from IP NV
        $x_{area} \leftarrow \lambda_T(\theta_{ospf}, \pi_i, \tau_i, area)$       ▷ Area from OSPF NV
        $x_{cost} \leftarrow \lambda_T(\theta_{ospf}, \pi_i, \tau_i, cost)$       ▷ Cost from OSPF NV
        $x_{descr} \leftarrow$ INTERFACE_DESCRIPTION$(\theta_{ospf}, \pi_i, \tau_i)$
        $X \cap \{(network, x_{network}), (area, x_{area}), (cost, x_{cost}), (desc, x_{desc})\}$
    end for
     return X
end function
```

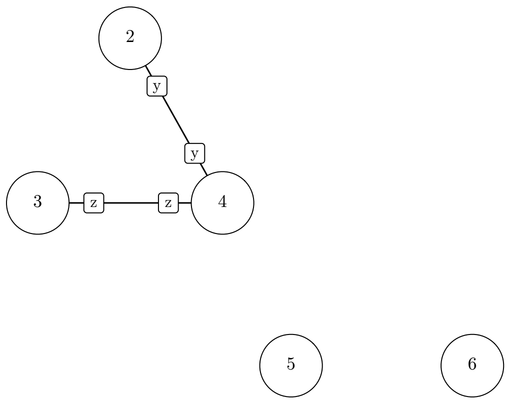
BGP

In this section we describe the configuration of the intermediate device model for the BGP routing protocol. As we have shown in previous chapters, the design approach used for creating the topologies for the interior (iBGP) and exterior (eBGP) forms of the BGP routing protocol. As such, we treated these separately, by using two different Network Views. However, when configuring a router, these are often configured together. The router itself will determine whether a peering session is iBGP, or eBGP, by looking at the autonomous system number in the configuration of that session. In terms of configuration, we therefore combine the two iBGP and eBGP Network Views into the same part of the intermediate device model.

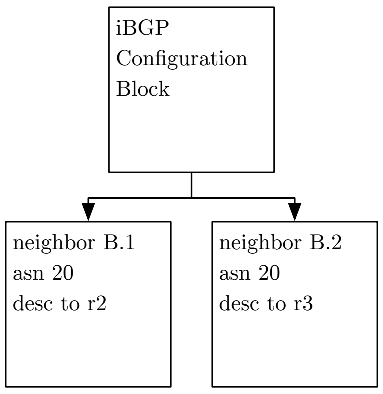
The way in which each device configures the sessions, in particular how the autonomous system number can be represented, can differ between devices. However the general approach, of mapping the sessions from the Network View into a list of sessions applies to the different target devices.

This is in contrast to a protocol such as OSPF, which can vary in the configuration data location between different target devices. For OSPF we need to use a different function depending on the target device. However to avoid repetition, we create two functions for BGP, which map the session information from the Network View into a list of sessions. This list of sessions can then be handled as appropriate by the target device compiler.

iBGP



(a) Subsection of iBGP Network View for Network Element *4*



(b) iBGP subsection of Device Model of Network Element *4*

Figure 6.23: Subsection of iBGP Network View and Device Model for Network Element *4*

The iBGP Device Model section is constructed by iterating over the neighbours of the Network Element in the iBGP Network View, and is shown in Figure 6.23. The IP Address Network View is used to map the loopback IP address of the neighbour, and the ASN label to map the ASN of the neighbour. In this case, the ASN of Network Element 4 could be used, since by definition, iBGP connects Network Elements which have the same ASN.

We note here that our use of Network Views is declarative, in the sense that allows a Design Function to express the desired topology, of a network design. This is decoupled from the actual low-level implementation of how to realise such a network design. For instance, the compiler simply creates a session for each Network Element connection in the iBGP Network View, regardless of how this Network Element Connection was created. The connection is created the same, whether that topology was constructed as either a full mesh, as a hierarchy with the roles explicitly set on the Network Whiteboard, as a hierarchy with the roles automatically assigned using an algorithm, or as a multiple level hierarchy.

In each of these cases, we create a peering session in the Intermediate Device Model according to the structure of the Network View. This allows us freedom in how we create the Design Functions to create the Network Views, with the compiler decoupling the low-level implementation from the design policy.

This process is shown in the pseudo-code example in Algorithm 6.16, and an extract of the Intermediate Device Model shown in Listing 6.14.

```
[
 {
  "asn": 20,
  "desc": "to r2",
  "neighbor": "B.1"
 }, {
  "asn": 20,
  "desc": "to r3",
  "neighbor": "B.2"
 }
]
```

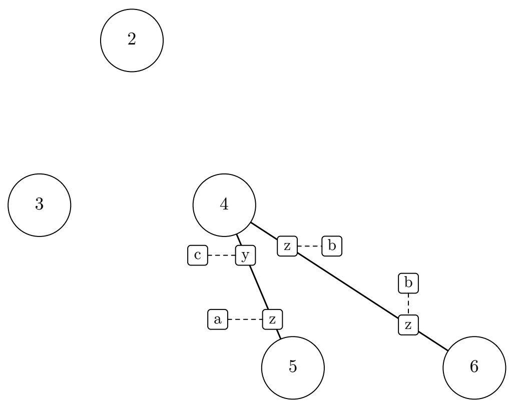
Listing 6.14: Example JSON format for iBGP Component of Intermediate Device Model for Network Element 4

Algorithm 6.16 Pseudo-Code for iBGP component of device compiler

```
function IBGP($\Theta, \pi_i$)
    $X \leftarrow \varnothing$
    $\theta_{ibgp} = NV(\Theta, ibgp)$                                  ▷ iBGP Network View
    $\theta_{ip} = NV(\Theta, ip)$                                      ▷ IP Network View
    $E \leftarrow E(\theta_{ibgp}, \pi_i)$                              ▷ NEC for NE $\pi_i$
    for $\epsilon_i \in E$ do
        $\pi_{remote} = \text{NEC\_OTHER\_NE}(\theta_i, \epsilon_j, x)$       ▷ Far-end NE
        $x_{neighbor} \leftarrow \lambda_{\Pi}(\theta_{ip}, \pi_{remote}, ip)$    ▷ NE IP from IP NV
        $x_{asn} \leftarrow \lambda_{ASN}(\Theta, \pi_{remote})$
        $x_{descr} \leftarrow \text{INTERFACE\_DESCRIPTION}(\theta_{ibgp}, \pi_i, \tau_i)$
        $X \cap \{(neighbor, x_{neighbor}), (asn, x_{asn}), (desc, x_{desc})\}$
    end for
    return $X$
end function
```

eBGP

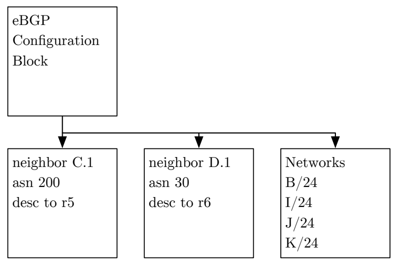
The eBGP Device Model section is constructed by iterating over the neighbours of the Network Element in the eBGP Network View, and is shown in Figure 6.24. The IP Address Network View is used to map the loopback IP address of the neighbour, and the ASN label to map the ASN of the neighbour. This process is shown in the pseudo-code example in Algorithm 6.17, with an extract of the eBGP section of the Intermediate Device Model shown in Listing F.21. The Pseudo-Code to generate the eBGP section of the Intermediate Device Model is shown in Algorithm 6.17.



(a) Subsection of eBGP Network View for Network Element 4

Loopback	Infrastructure
1: A	1:
20: B	20: I, J, K
30: D	30:
40: E	40:
100: F	100: R, S, T
200: C	200:
300: G	300: U, V, W

(b) IP Address Blocks allocated to each ASN



(c) eBGP subsection of Device Model of Network Element of the Simplified Small Internet Example

Figure 6.24: Subsection of eBGP Network View and Device Model for Network Element 4

```
{
 "neighbors": [
  {
   "asn": 200,
   "desc": "to r5",
   "neighbor": "C.1"
  }, {
   "asn": 30,
   "desc": "to r6",
   "neighbor": "D.1"
  }
 ],
 "networks": [
   "B/24",
   "J/24",
   "K/24",
   "L/24"
 ]
}
```

Listing 6.15: Example JSON format for eBGP Component of Intermediate Device Model for Network Element 4

Algorithm 6.17 Pseudo-Code for eBGP component of device compiler

```
function EBGP($\Theta, \pi_i$)
    $X \leftarrow \varnothing$
    $\theta_{ebgp} = NV(\Theta, ebgp)$                         ▷ ebgp Network View
    $\theta_{ip} = NV(\Theta, ip)$                             ▷ IP Network View
    $E \leftarrow NEC(\theta_{ebgp}, \pi_i)$                    ▷ NEC for NE $\pi_i$
    for $\epsilon_i \in E$ do
        $\pi_{remote} = \text{NEC\_OTHER\_NE}(\theta_i, \epsilon_j, x)$          ▷ Far-end NE
        $x_{neighbor} \leftarrow \lambda_{\Pi}(\theta_{ip}, \pi_{remote}, ip)$       ▷ NE IP from IP NV
        $x_{asn} \leftarrow \lambda_{ASN}(\Theta, \pi_{remote})$
        $x_{descr} \leftarrow \text{INTERFACE\_DESCRIPTION}(\theta_{ebgp}, \pi_i, \tau_i)$
        $X \cap \{(neighbor, x_{neighbor}), (asn, x_{asn}), (desc, x_{desc})\}$
    end for
    $Y \leftarrow \varnothing$
    $ASN = \lambda_{ASN}(\Theta, \pi_i)$
    loopbacks $\leftarrow \lambda_{\Theta}(\theta_{ip}, loopback)$        ▷ Loopback IP block label from IP NV
    infra $\leftarrow \lambda_{\Theta}(\theta_{ip}, loopback)$  ▷ Infrastructure IP block label from IP NV
    loopbacks_asn $\leftarrow$ GET_KEY_VAL$(loopbacks, ASN)$    ▷ Loopback IPs for this ASN
    infra_asn $\leftarrow$ GET_KEY_VAL$(loopbacks, ASN)$  ▷ Infra IPs for this ASN
    $Y \leftarrow Y\cap$ loopbacks_asn                        ▷ Add Loopback pool
    $Y \leftarrow Y\cap$ infra_asn                       ▷ Add Infrastructure IP pool
     return X
end function
```

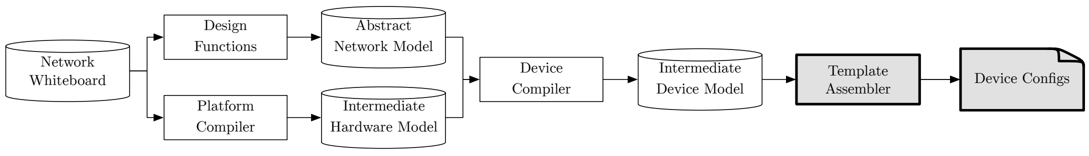
Combining

As they are independent, each of these sub-compilers can be combined to produce a single device compiler for the target device.

6.4.3 *Conclusion*

In this section we have provided an example of how information from Network Views and the Network Whiteboard has led to the generation of the Intermediate Device Model artefact. In the next section we will explain Phase Two of the Low-Level Device Configuration state, describing rendering using templates.

6.5 configuration assembly using templates



Assembling the configurations through template rendering is straightforward in our toolchain, due to the Platform Compiler and Device Compiler approach to handling the compilation logic. It follows closely the guidelines for simple templates explained in earlier chapters.

Listing 6.16 provides the interfaces part of the Intermediate Device Model for the device *r4* of the Simplified Small Internet Example. Listing Listing 6.17 shows a simplified example template for a Cisco IOS-like device syntax. The rendered template output is shown in Listing 6.18 using the provided Intermediate Data Model.

The remaining template processing steps are shown in the appendix as per Table 6.2. We combine these to form the combined Intermediate Device Model shown in Listing 6.19, a combined template in Listing 6.20 , and the resulting combined output configuration in Listing 6.21.

```
[ {
  "desc": "to r2",
  "id": "a",
  "ip": "J.2",
  "role": "physical",
  "subnet": "J"
 }, {
  "desc": "to r6",
  "id": "b",
  "ip": "O.1",
  "role": "physical",
  "subnet": "O"
 }, {
  "desc": "to r5",
  "id": "c",
  "ip": "N.1",
  "role": "physical",
  "subnet": "N"
 }, {
  "desc": "to r3",
  "id": "d",
  "ip": "K.2",
  "role": "physical",
  "subnet": "K"
 }, {
  "desc": "loopback",
  "id": "lo0",
  "ip": "B.3",
  "role": "loopback",
  "subnet": "B"
 } ]
```

Listing 6.16: Interfaces JSON

```
{% for e in data %}
interface {{e.id}}
  description {{e.desc}}
  ip address {{e.ip}}
{% endfor %}
```

Listing 6.17: Interfaces Template

```
interface a
    description to r2
    ip address J.2
interface b
    description to r6
    ip address O.1
interface c
    description to r5
    ip address N.1
interface d
    description to r3
    ip address K.2
interface lo0
    description loopback
    ip address B.3
```

Listing 6.18: Interfaces Output

Table 6.2: IDM, Template, and output for OSPF, iBGP and eBGP

Protocol	IDM	Template	Rendered Output
OSPF	Listing F.15	Listing F.16	Listing F.17
iBGP	Listing F.18	Listing F.19	Listing F.20
eBGP	Listing F.21	Listing F.22	Listing F.23

```
{
  "asn": 20,
  "ebgp": {
    "neighbors": [{
      "asn": 200,
      "desc": "to r5",
      "neighbor": "C.1"
    }, {
      "asn": 30,
      "desc": "to r6",
      "neighbor": "D.1"
    }],
    "networks": [
      "B/24",
      "J/24",
      "K/24",
      "L/24"
    ]
  },
  "hostname": "r4",
  "ibgp": {
    "neighbors": [{
      "asn": 20,
      "desc": "to r2",
      "neighbor": "B.1"
    }, {
      "asn": 20,
      "desc": "to r3",
      "neighbor": "B.2"
    }]
  },
  "interfaces:": [{
    "desc": "to r2",
    "id": "a",
    "ip": "J.2",
    "role": "physical",
    "subnet": "J"
  }, {
    "desc": "to r6",
    "id": "b",
    "ip": "O.1",
    "role": "physical",
    "subnet": "O"
  }, {
    "desc": "to r5",
    "id": "c",
    "ip": "N.1",
    "role": "physical",
    "subnet": "N"
  }, {
    "desc": "to r3",
    "id": "d",
    "ip": "K.2",
    "role": "physical",
    "subnet": "K"
  }, {
    "desc": "loopback",
    "id": "lo0",
    "ip": "B.3",
    "role": "loopback",
    "subnet": "B"
  }]
}
```

Listing 6.19: Combined Intermediate Device Model

```
!
hostname {{data.hostname}}
!
{% for e in data.interfaces %}
interface {{e.id}}
  description {{e.desc}}
  ip address {{e.ip}}
{% endfor %}
!
router ospf
  {% for e in data %}
  network {{e.network}} area {{e.
       area}}
  {% endfor %}
!
router bgp {{data.asn}}
  ! ibgp
  {% for e in data.ibgp.neighbors %}
  neighbor {{e.neighbor}} remote-as
       {{e.asn}}
  {% endfor %}
  ! ebgp
  {% for e in data.ebgp.neighbors %}
    neighbor {{e.neighbor}} remote-
       as {{e.asn}}
  {% endfor %}
  ! networks
  {% for e in data.ebgp.networks %}
  network {{e}}
  {% endfor %}
```

Listing 6.20: Combined Jinja2 Template

```
!
hostname r4
!
interface a
  description to r2
  ip address J.2
interface b
```

```
  description to r6
  ip address O.1
interface c
  description to r5
  ip address N.1
interface d
  description to r3
  ip address K.2
interface lo0
  description loopback
  ip address B.3
!
router ospf
  network J/24 area 0
  network K/24 area 0
!
router bgp 20
  ! ibgp
  neighbor B.1 remote-as 20
  neighbor B.2 remote-as 20
  ! ebgp
  neighbor C.1 remote-as 200
  neighbor D.1 remote-as 30
  ! networks
  network B/24
  network J/24
  network K/24
  network L/24
```

Listing 6.21: Combined example device configuration

6.6 TRANSFER FUNCTION APPROACH

In this section we provide an overview of how the Platform Compiler and Device Compiler approach described in this Chapter to produce the Intermediate Device Model can be viewed as a transfer function applied to the Abstract Network Model and Network Whiteboard. This follows a similar discussion we presented in Section 5.8.

We denote the Intermediate Hardware Model is represented as M with each entry μ_i for device i. We then denote the Intermediate Device Model as Ψ, and each individual device entry as ψ_i for Network Element π_i. For convenience, we denote each component entry in ψ_i by σ, e.g σ_{phy} σ_{ospf}, etc. Therefore, for Network Element π_i, the Intermediate Device Model entry ψ_i is defined as:

$$\begin{aligned} g_{device}(\theta_{phy}) &= \sigma_{device} \\ g_{interfaces}(\theta_{phy}, \theta_{ip}, \mu_i) &= \sigma_{interfaces} \\ g_{ospf}(\theta_{ospfl}, \theta_{ip}) &= \sigma_{ospf} \\ g_{ibgp}(\theta_{ibgp}, \theta_{ip}) &= \sigma_{ibgp} \\ g_{ebgp}(\theta_{phy}, \theta_{phy}) &= \sigma_{ebgp} \end{aligned} \tag{6.1}$$

For each function g we define a modified version g' which takes the complete Abstract Network Model Θ rather than an individual

Network View θ_i. This can be performed by a selection function to obtain θ_i from Θ,

$$\begin{aligned} g'_{device}(\Theta) &= \sigma_{device} \\ g'_{interfaces}(\Theta, \mu_i) &= \sigma_{interfaces} \\ g'_{ospf}(\Theta) &= \sigma_{ospf} \\ g'_{ibgp}(\Theta) &= \sigma_{ibgp} \\ g'_{ebgp}(\Theta) &= \sigma_{ebgp} \end{aligned} \tag{6.2}$$

Now μ_i is defined as $h(\omega_i)$ for Network Whiteboard ω_i, and h the Platform Compiler function. The Intermediate Device Model for π_i is:

$$\begin{aligned} g'_{device}(\Theta) &= \sigma_{device} \\ g'_{interfaces}(\Theta, h(\omega_i)) &= \sigma_{interfaces} \\ g'_{ospf}(\Theta) &= \sigma_{ospf} \\ g'_{ibgp}(\Theta) &= \sigma_{ibgp} \\ g'_{ebgp}(\Theta) &= \sigma_{ebgp} \end{aligned} \tag{6.3}$$

Now recall that $\Theta = F(\omega_i)$, therefore

$$\begin{aligned} \psi_i &= (\sigma_{device}, \sigma_{interfaces}, \sigma_{ospf}, \sigma_{ibgp}, \sigma_{ebgp}) \\ &= (g'_{device}(\Theta), g'_{interfaces}(\Theta, \mu_i), \\ &\quad g'_{ospf}(\Theta), g'_{ibgp}(\Theta), g'_{ebgp}(\Theta)) \\ &= G(F(\omega_i), h(\omega_i)) = G'(\omega_i) \end{aligned} \tag{6.4}$$

We can apply this for each Network Element π_i to form the complete Intermediate Device Model $\Psi = (\psi_i, \psi_j, \psi_k, \ldots)$. We can see that our functional approach allows the Intermediate Device Model Ψ to be constructed from the Network Whiteboard ω_i.

This allows us to attain the property $F(model) = configuration$ described in industry presentations such as by Shields *et al.* [39].

6.7 CONCLUSION

In this chapter we have shown how a combination of a device-level intermediate representation and a template-based code generator can be used to transform the network-level Network Views into low-level device configurations. This approach is flexible to accommodate new Network Views and additional compilation target devices.

In this chapter we have shown how the Platform Compiler and Device Compiler approach addresses Research Question 4: *How can the configurations for a diversity of network devices be generated systematically from a graph-based intermediate representation?*

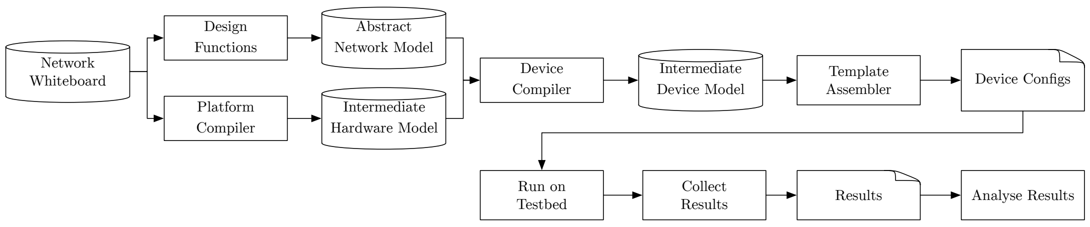
Part III

IMPLEMENTATION AND CASE STUDIES

7

SYSTEM IMPLEMENTATION AND EXPERIMENTATION SETUP

7.1 INTRODUCTION



In Chapter 4, Chapter 5, and Chapter 6 we explained the theory supporting the generation of network device configurations from high-level policy descriptions. In this chapter we present AutoNetkit, an implementation of a tool that supports the toolchain and theoretical framework previously discussed. This includes a description of our experimental infrastructure for verifying the correctness of the tool. The key contribution of this chapter is to describe the system and tool implementation that demonstrates the practicality of the theoretical approach. AutoNetkit is written in Python and available on GitHub[1], where it has over 80 "stargazers"[2] (favourites). It has been used in research demonstrations and by industry as part of the VIRL [79] network simulation platform from Cisco.

We now present an overview of AutoNetkit and demonstrate its use to implement our approach and toolchain. We will validate the configurations generated on the Netkit simulation platform. This sets up the experimentation framework to allow us to address Research Question 5: *What are the scalability and extensibility characteristics for the compilation of High-Level Network Configuration Policy to device con-*

1 https://github.com/sk2/autonetkit
2 https://github.com/sk2/autonetkit/stargazers

figurations in terms of network size and diversity of network protocols and target devices?, which we will address in the case studies of Chapter 8.

7.2 SYSTEM IMPLEMENTATION

7.2.1 *Introduction*

The architecture of the AutoNetkit implementation is shown in Figure 7.1. AutoNetkit is written in Python. There are two use-case scenarios for the system. The first is as a black-box, where a Network Whiteboard is provided to a tool which then produces low-level device configurations, and a series of visualisations of the protocol topologies. The device configurations can then be fed directly to an emulated network environment. A collection framework can be used to gather operational data from the emulated data to enable to be compared to the requirements provided in the original network description.

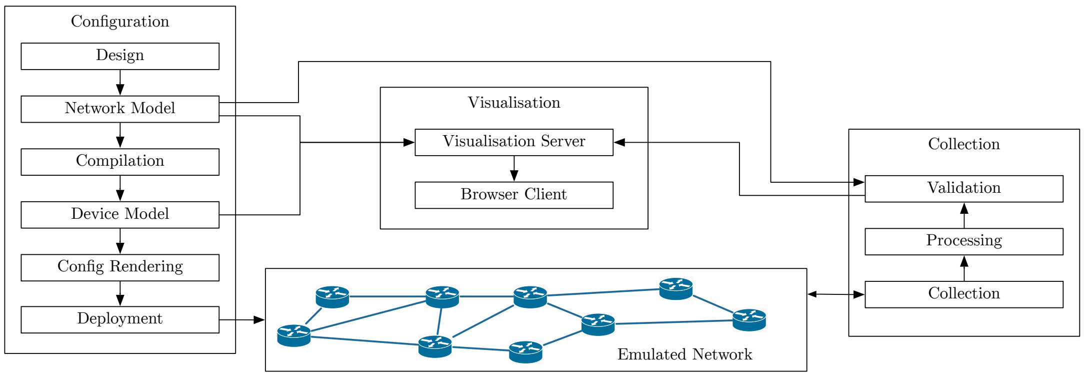


Figure 7.1: System Implementation and Experiment Setup

The second use-case allows for API use of the library. This is the focus of this chapter, as this walks through the steps involved. An understanding of this use-case makes the black-box use-case straightforward.

We now provide an overview of the architecture. A more detailed description of the Python implementation was provided at the PyCon Australia 2013 Conference [52]. A recording is available on YouTube[3].

3 https://www.youtube.com/watch?v=EGK5jjyUBCQ

7.2.2 *Configuration*

The Network Whiteboard can be provided in a number of formats. It can be programmatically described, or imported using a graph description format such as GraphML. We provide an example of this in the example in this chapter, and an example of using GraphML in the case studies in Chapter 8.

The Network Model is based on the NetworkX [93] Python graph library. We provide an API on top of NetworkX to simplify the accessing of nodes, edges and interfaces, and the setting and getting of attributes. This Abstract Network Model can be seen in the AutoNetkit code on GitHub[4]. The Design Functions can be written using this Network Model, which we provide examples of in this chapter. The Platform and Device Compilers described in Chapter 6 are written using an object-oriented approach allowing for extensibility. The templates are written in the Jinja2 template format, which are then rendered using the output of the Platform and Device Compilers.

Some of the terminology used in the implementation can differ from that described in the theoretical chapters. A mapping of the major terms is shown in Table 7.1. For simplicity of access, we include the Network Whiteboard as the *input* overlay in the Abstract Network Model. This simplifies accessing elements and labels from the Network Whiteboard in the Design Functions to construct the other Network Views.

Table 7.1: Mapping of theory to implementation terms

Theory	Implementation
Network Whiteboard	*input* Overlay
Network View	Overlay
Network Element	Node
Network Element Connection	Edge
Network Element Interface	Interface
Label	Attribute
Intermediate Device Model (IDM)	Network Information Database (NIDB)

4 https://github.com/sk2/autonetkit/tree/thesis16/autonetkit/anm

Black-Box Console Script

A console script is also provided which allows the use of the use of the system as a black-box. An example of such usage is provided in Listing 7.1. This black-box approach can be constructed using the workflow components, such as the Design Functions and Compilers which we discuss in the Simplified Example later in this chapter.

```
$ autonetkit -f house.graphml
INFO AutoNetkit 0.11.0
INFO Automatically assigning input interfaces
INFO Allocating from IPv4 infrastructure block: 10.0.0.0/8
INFO Allocating from IPv4 loopback block: 192.168.0.0/22
INFO Compiling Netkit for localhost
INFO Configuration engine completed

$ tree --charset unicode -L 1 rendered/localhost/netkit
rendered/localhost/netkit
|-- 1
|-- 1.startup
|-- 2
|-- 2.startup
|-- 3
|-- 3.startup
|-- 4
|-- 4.startup
|-- 5
|-- 5.startup
‘-- lab.conf
```

Listing 7.1: Running AutoNetkit as a console script, with output directory structure shown

7.2.3 *Visualisation*

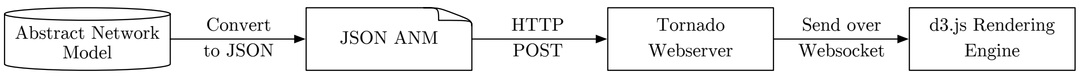


Figure 7.2: Flowchart of visualisation process

The visualisation process is shown in Figure 7.2. The Abstract Network Model is converted to a JSON representation of the Abstract Network Model, which is then sent to a Tornado webserver. A browser-based Javascript library interprets the JSON from the webserver and renders it using the d3.js [11] framework. The system also allows highlighting of specific nodes, edges and paths. This can be used to display data collected from running network simulations such as traceroutes.

A screenshot is shown in Figure 7.3. The first dropdown menu selects the overlay such as *physical* or *ospf*; the second selects the node attribute to display, the third selects the link attribute, and the fourth the interface attribute. The node icon is selected based on the *device_type* node attribute. A grouping is displayed behind the nodes: the *physical* overlay the nodes are grouped by their *asn* attribute, and on the *ospf* overlay they are grouped by both their *asn* and *area* attributes. The first checkbox allows the toggle of the interface display, and the second animates transitions. The topology diagrams used in Section 7.4 and Chapter 8 were generated using this framework. As the visualisation framework generates SVG graphics, plots can exported to PDF using a web browser.

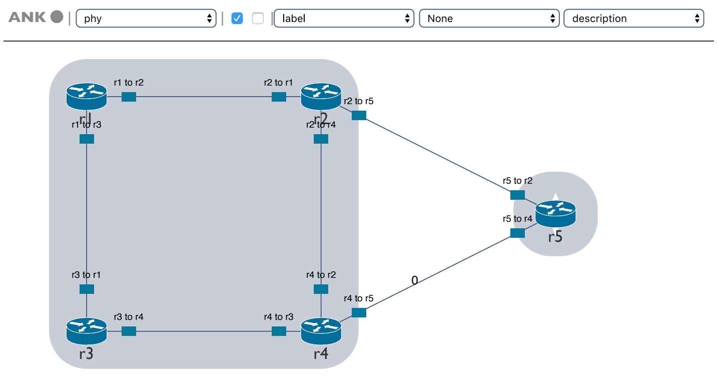
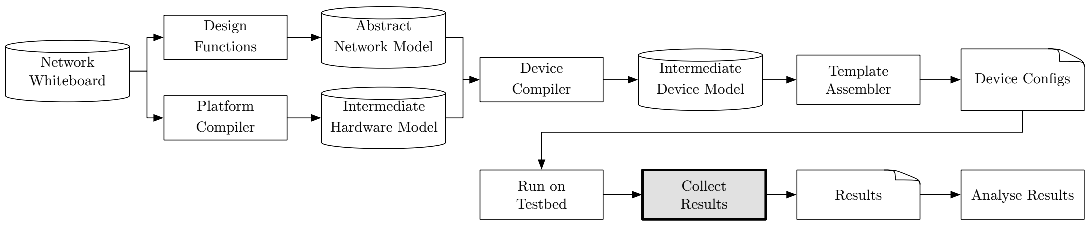


Figure 7.3: Screenshot of visualisation system.

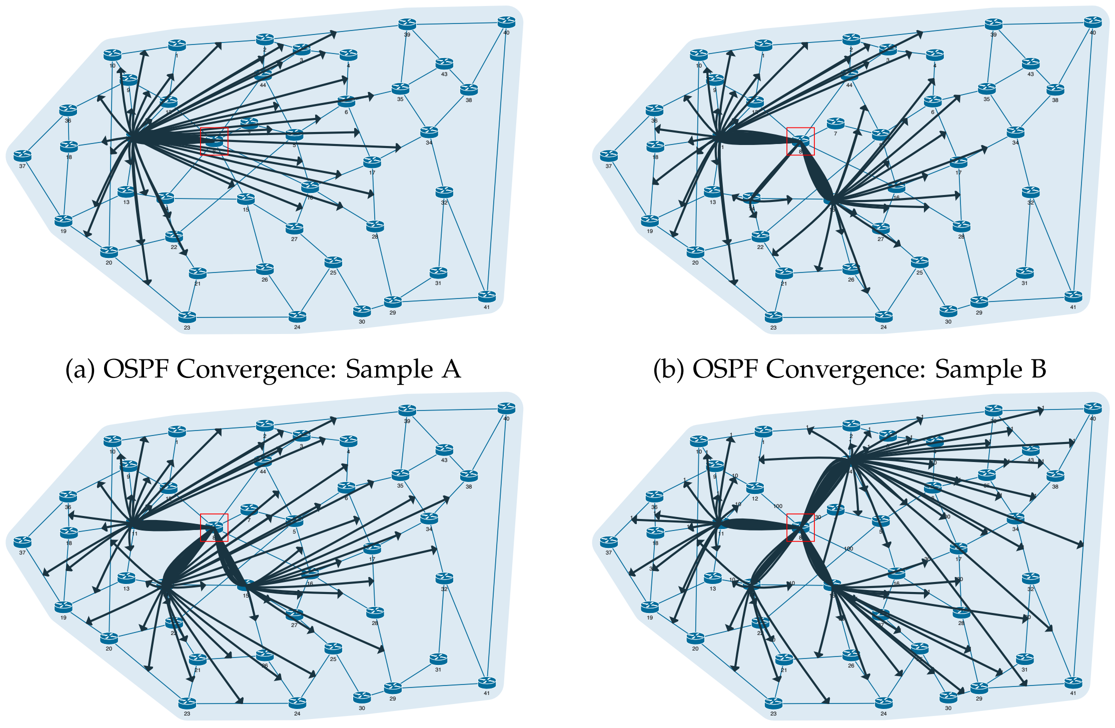
The visualisation system can also be used independently. Edelman [27] demonstrated how the system can be integrated with the *Schprokits* network automation framework.

7.2.4 *Collection*



The collection components provides a wrapper to log into the individual network device, and run diagnostic commands such as routing tables or traceroute commands. This then can be processed using a library such as TextFSM [40], which can then be compared to the original configuration models in the Abstract Network Model, or visualised using the visualisation framework.

An example of such a visualisation was shown at a demonstration at SIGCOMM 2014 [54] for data collection, and in the frames from the time-series visualisation animation shown in Figure 7.4. The collection approach was briefly described in Knight *et al.* [57]. This approach has been expanded further as part of the Live Visualisation feature of the Virtual Internet Routing Lab [79] platform from Cisco, which uses the AutoNetkit Abstract Network Models and automated visualisation framework.



(a) OSPF Convergence: Sample A

(b) OSPF Convergence: Sample B

(c) OSPF Convergence: Sample C

(d) OSPF Convergence: Sample D

Figure 7.4: Visualisation of collected OSPF data over a time-series as the network converges. Captures are taken from the time-series animation at `https://www.youtube.com/watch?v=SOZog3n8d5s`.

7.2.5 *Emulated Network*

The emulated network is a series of virtual machines that emulate the network operating system of a device such as router. These run the configurations generated by the Device Compilers in the system. The emulated network also requires a description of the virtual machine to instantiate and the wiring between these machines. These can be generated by the Platform Compiler.

7.2.6 *Availability and Installation*

The system is available on GitHub[5], and can be installed from the Python Package Index[6]. The visualisation framework is also available on GitHub[7].

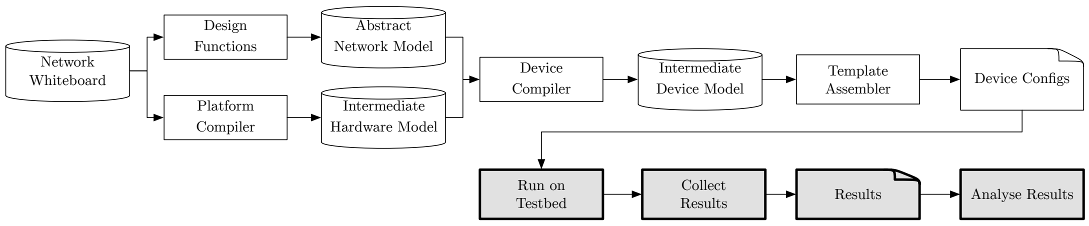
The version of AutoNetkit used in the following examples and case studies is available on GitHub in the thesis16[8] branch. This can be installed using the command shown in Listing 7.2.

```
pip install https://github.com/sk2/autonetkit/archive/thesis16.zip
```

Listing 7.2: Installation of *thesis16* branch using the PIP package manager

5 https://github.com/sk2/autonetkit
6 https://pypi.python.org/pypi/autonetkit
7 https://github.com/sk2/autonetkit_vis
8 https://github.com/sk2/autonetkit/tree/thesis16

7.3 EXPERIMENTATION SETUP



In Section 7.4 we describe generating configurations for the Netkit [84] platform. In the Case Studies shown in Chapter 8 we generate Netkit configurations, and also demonstrate generating configurations for the Cisco VIRL [79], and the C-BGP [88] simulation platforms. This allows us to demonstrate the use of the system across a variety of different target platforms and devices, showing the flexibility of the low-level compilation approach discussed in Chapter 6. For verification we will show the output of diagnostic commands such as TRACEROUTE or the output of routing tables.

We now look at an example of an emulation setup for the House topology example using Netkit. The topology is shown in Figure 7.5, and consists of five routers and six links. All of the links are point-to-point between the router pairs: we do not have any hubs or switches in this topology. The interface names are specified as interface labels and range from *eth0* to *eth3*. We wish to reproduce this topology in Netkit using the Quagga router software package. This requires launching the appropriate virtual machines to represent the network devices, running Quagga on them so they act as routers, and establishing the virtual links between these devices. We also provide an out-of-band management network to be able to log into the devices and run diagnostic commands for our verification process.

The Netkit emulation to run this topology is shown in Figure 7.6. This is based on Figure 3 from [84], and has been adapted for the House topology shown in Figure 7.5. It shows the virtual machines representing the routers running Quagga routing software, which provides the *zebrad*, *ospfd*, and *bgpd* processes. These lightweight virtual machines run in User-Mode Linux. Virtual Network Interfaces, listed as *VNI* represent the interfaces between the virtual machines, and are connected by *virtual hubs*. The Quagga processes exchange routing information over these Virtual Network Interfaces and virtual hubs, with the resulting routes added to the Linux route table running in the Linux Kernel. Finally, we can use the “Tap” virtual hub to interface to the host system, allowing out-of-band access to the virtual

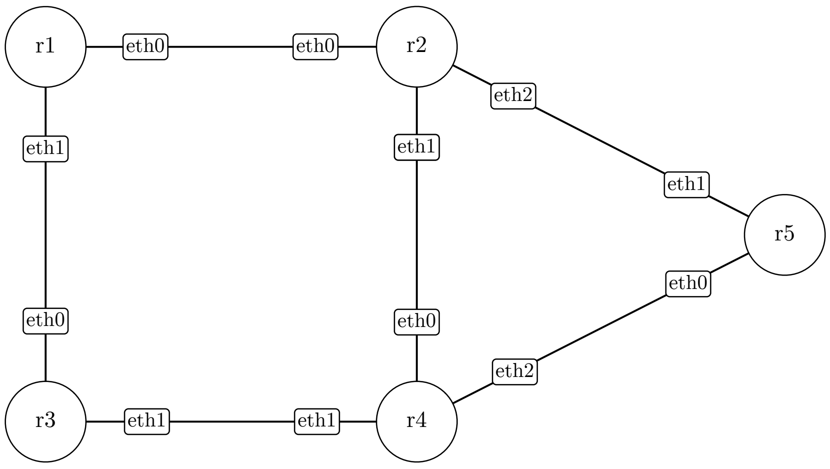


Figure 7.5: House example Physical Topology.

machines. The virtual machines, Virtual Network Interfaces, and virtual hubs are specified by the Platform Compiler to generate the Netkit *lab.conf* topology specification file, and the Device Compilers generate configuration files for the Quagga processes.

In the next section we will walk through the process of generating and verifying configuration files for the Netkit network emulation platform. In Chapter 8 we will also generate configurations for the VIRL network emulation platform. This follows the same general concept of virtual machines running network operating systems, with virtual links established between them.

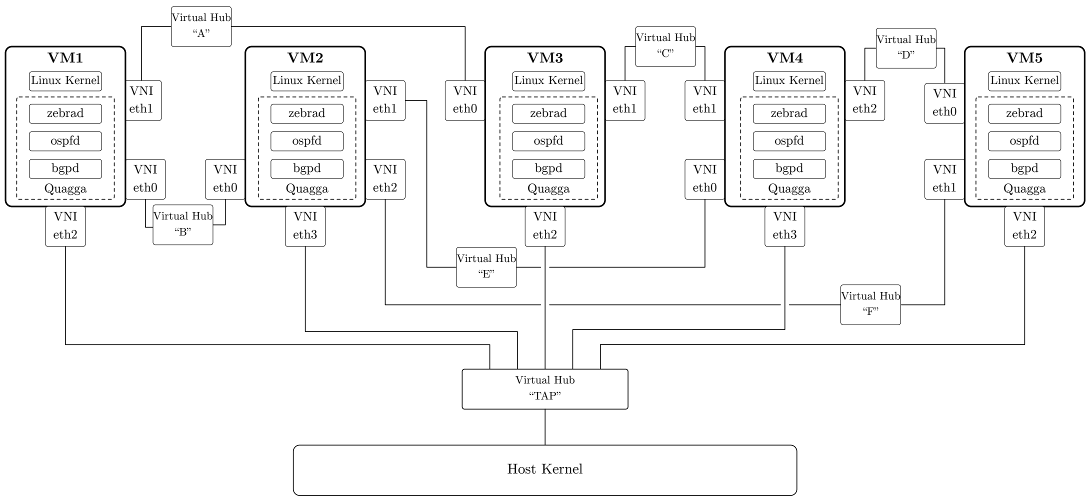


Figure 7.6: Overview of using Netkit emulation system as a testbed to validate the House example topology. Netkit emulation diagram based on Figure 3 from Pizzonia *et al.* [84]

7.4 SIMPLIFIED EXAMPLE: HOUSE TOPOLOGY

In this example, we present a simplified walk-through of the key components of the configuration process. This allows us to introduce the components, so that we can focus on their extension in the case studies of Chapter 8.

This uses the House topology we introduced in Section 4.3 with two autonomous systems of routers, and the links between them. We show the OSPF routing protocol within an autonomous system, and the eBGP protocol between autonomous systems. We also configure the iBGP protocol within the four-router autonomous system to share the routes learnt by eBGP.

7.4.1 *Methodology*

The methodology used to implement and verify this example is shown in Figure 7.7. This introduces the details of the toolchain and the methodology which we use in Chapter 8. The configuration process is shown on the left. The Network Whiteboard is transformed into the Network Views of the Abstract Network Model using Design Functions. These Design Functions follow the theoretical approach outlined in Section 5.6.The ordering is for the Physical Design Function to transform the network whiteboard into the Physical Network View, which is then used by the Layer 2 Design Function to create the Layer 2 Network View. This is then used by the IP Address Design Function, and the Layer 2 Connectivity Design Function which creates the Layer 2 Connectivity Network View. The layer Two Connectivity Network View is then used by each of the routing protocol Design Functions for OSPF, iBGP, and eBGP. This Abstract Network Model can also be sent to the Visualisation Server to be viewed in a browser-based Visualisation Client. We use the Visualisation Client to automatically generate the figures presented in this example.

The Platform Compiler compiles the Network Whiteboard into the Intermediate Platform Model, and the Intermediate Hardware Model. The Intermediate Platform Model is used to specify the simulation details required by the Netkit *lab.conf* specification. The intermediate hardware model assigns interface names and the Device Compiler to use for each device. In this example the compiler used is the Quagga Device Compiler. This device compiler combines the physical inventory in the intermediate hardware model with the logical inventory described in the abstract network model, to produce the Intermediate Device Model. This Intermediate Device Model is then assembled by the Template Assembler with Quagga Templates to produce the Quagga device configurations. The Intermediate Platform Model is assembled with a Netkit Template to produce the Netkit Platform Configurations.

These configurations can then be launched on a Netkit Simulation, providing us with a running network using the high-level topology specified on the Network Whiteboard. We then use a series of collection commands to allow us to gather data about the functioning of the simulated network. Here we use the TRACEROUTE, SHOW IP BGP, and SHOW IP ROUTE commands to view the path taken through a network, and the diagnostic information about the functioning of the routing protocols. We discuss the output of these collection commands to confirm that the network has been correctly configured as intended on the Network Whiteboard description.

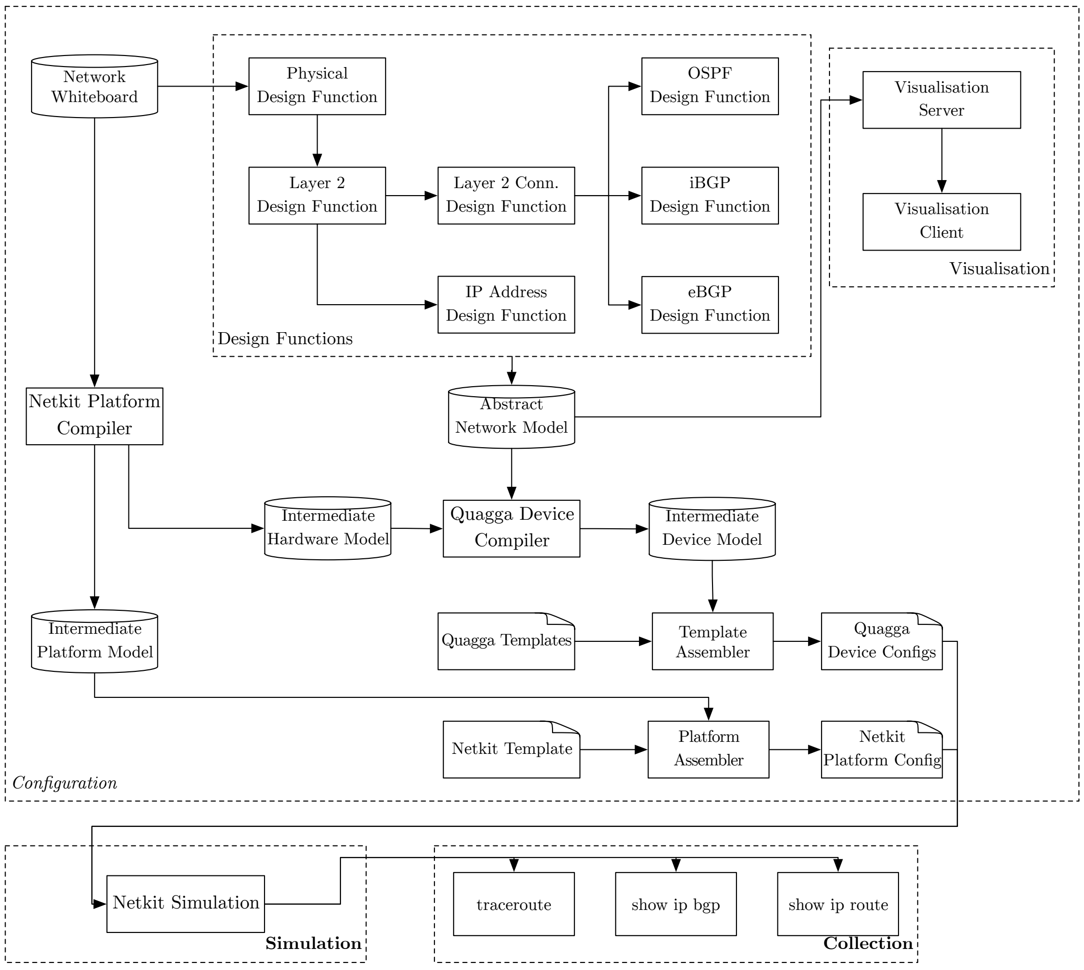
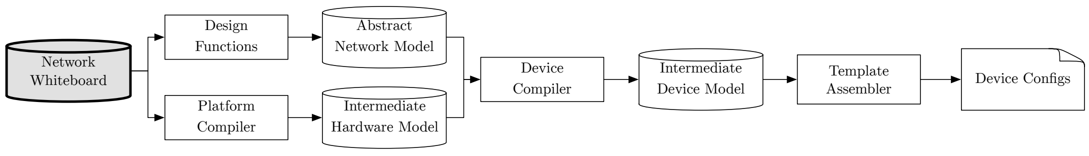


Figure 7.7: Configuration Generation and Verification methodology for House Example

7.4.2 *Network Whiteboard*



As the Network Whiteboard is represented by the *input* overlay in the Abstract Network Model, the first step is to create the Abstract Network Model object. This is shown in Listing 7.3 and Listing 7.4.

```
# Create Abstract Network Model
import autonetkit
anm = autonetkit.NetworkModel()
```

Listing 7.3: Initialisation

```
# Create Network Whiteboard
g_in = anm.add_overlay("input")

# Specify list of nodes to add
nodes = ["r1", "r2", "r3", "r4", "r5"]
# Add nodes to Network Whiteboard
g_in.add_nodes_from(nodes)
# Set device_type and asn for all nodes
g_in.update(device_type="router", asn=1)
# Set asn for specific device r5
g_in.update("r5", asn=2)

# Update node positions from dictionary
positions = {
    "r1": (0, 0),
    "r2": (250, 0),
    "r3": (0, 250),
    "r4": (250, 250),
    "r5": (500, 125)}

for n in g_in:
    n.x, n.y = positions[n]

# Update visualisation
autonetkit.update_http(anm)
```

Listing 7.4: Create Network Whiteboard

Once we have the Abstract Network Model, we can add the *input* overlay. We then define a list of five nodes, *r1*, *r2*, *r3*, *r4*, and *r5*, and add these to the *input* overlay, using the ADD_NODES_FROM function. Next, we use the UPDATE function to set attributes to the nodes in the *input* overlay. We set the *device_type* to *router*, and the *asn* to 1. We then specify the *asn* of the node *r5* to be 2, placing it in a separate Autonomous System for our design functions. Finally, we define a dictionary of x and y co-ordinates for the nodes, and then iterate over the nodes in the *input* overlay, setting the appropriate attribute for that node. The code for these steps is shown in Listing 7.4, and a visualisation of the resulting overlay is shown in Figure 7.8.

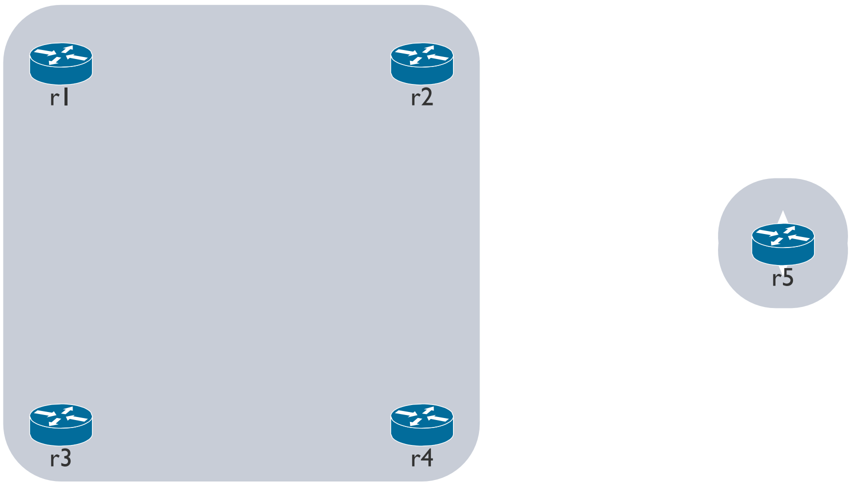


Figure 7.8: Visualisation of adding nodes step for Network Whiteboard

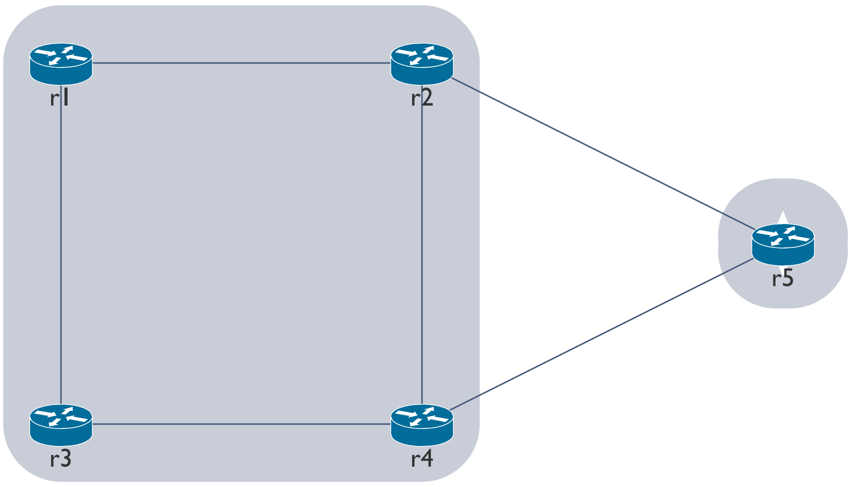


Figure 7.9: Visualisation of Network Whiteboard with links added

Now that the nodes have been added, the next step is to add the edges, representing the links. We first create a list of the *(node1, node2)* pairs for the edges, and then use the ADD_EDGES_FROM function. We then use the ALLOCATE_INPUT_INTERFACES function, to automatically allocate interfaces to the endpoint of each of the edges in the *input* overlay. These steps are shown in Listing 7.6. This completes the steps to create the Network Whiteboard. This has demonstrated programmatic creation of the Network Whiteboard. In Chapter 8 we explore methods that use specification file formats such as GraphML to specify the Network Whiteboard. We then update the visualisation as shown in Listing 7.5, to give the visualisation of the completed *input* overlay shown in Figure 7.9.

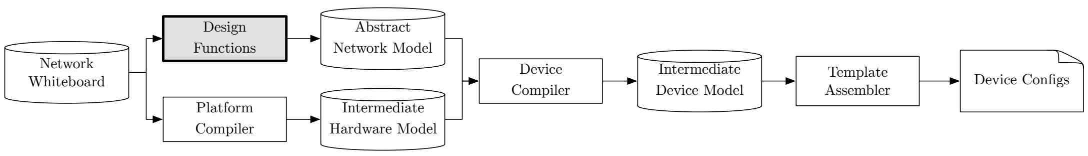
```
autonetkit.update_http(anm)
```

Listing 7.5: Update Visualisation

```
# Specify list of edges
edges = [("r1", "r2"), ("r2", "r4"), ("r1", "r3"),
         ("r3", "r4"), ("r2", "r5"), ("r4", "r5")]
# Add all edges from list to g_in
g_in.add_edges_from(edges)
# Automatically allocate interfaces to added edges
g_in.allocate_input_interfaces()
```

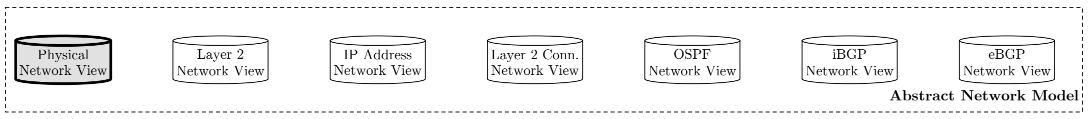
Listing 7.6: Set Edges in Network Whiteboard

7.4.3 *Design Functions*



In this section we show how the Design Functions described in Section 5.6 are implemented in AutoNetkit. This illustrates how the theoretical framework can be used in our reference implementation.

Physical



The *physical* Design Function is responsible for creating the *physical* overlay in the Abstract Network Model, corresponding to the *physical* topology. This is used by the subsequent Design Functions discussed in this section.

Here we make the simplifying assumption that all links in the Network Whiteboard are physical. We could generalise this, by explicitly setting a default parameter of physical links in the Network Whiteboard, and then selecting through a query the links from the whiteboard which have the type of physical.

As the *physical* overlay is fundamental to the network design process, it is created automatically as part of the Abstract Network Model. The first step in the *physical* Design Function is to add the appropriate nodes from the *input* overlay, using the ADD_NODES_FROM function. We use the *retain* parameter to copy the key attributes from the *input* overlay, for the *asn*, *device_type*, and the *x* and *y* co-ordinates. These are stored on the *physical* overlay, and used as the cross-view labels discussed in Section D.1.4. A number of default attributes are then set for nodes in the *physical* overlay, to simplify the other Design Functions and the Compilers. These are to enable IPv4 by setting the *use_ipv4* boolean value to *true*, and then specifying the *platform* as *netkit* and *syntax* as *quagga* for the *Platform Compiler*, and the *host* for the grouping of nodes to the relevant simulation host.

The edges from the *input* overlay are added to the *physical* overlay using the ADD_EDGES_FROM function. If required, these could be filtered in this step, as discussed above for the case of different edge types being specified in the Network Whiteboard step. The code for the *physical* Design Function is shown in Listing 7.7. Finally, the output is visualised, as shown in Figure 7.10.

```
# Access automatically created physical Network View
g_phy = anm['phy']
# Copy nodes from g_in and retain key attributes
g_phy.add_nodes_from(g_in, retain=["asn", "device_type", "x", "y"])
# Set default attributes
g_phy.update(use_ipv4=True, host="localhost",
             platform="netkit", syntax="quagga")

# Add edges from g_in
g_phy.add_edges_from(g_in.edges())
autonetkit.update_http(anm)
```

Listing 7.7: Physical Design Function

Layer 2

The first step is to create the *layer2* overlay in the Abstract Network Model. We then add the nodes and edges from the *physical* overlay.

The next step is to split the point-to-point edges to form the collision domains. This is performed by using the Python *list comprehension* on the edges in the *layer2* overlay, and filtering to only list the edges that connect layer 3 devices (routers in this case). This is performed using the IS_L3DEVICE function, which looks at the *device_type* attribute we set on the *physical* overlay. This demonstrates the use of the pass-through attributes: the *device_type* attribute is passed through to return the value set on the *physical* overlay. In this case, as all edges connect routers, the list of edges to be split is the list

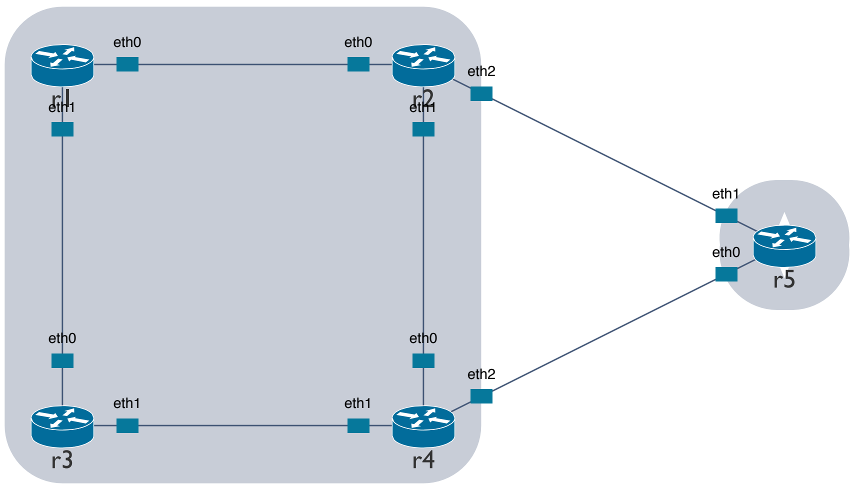
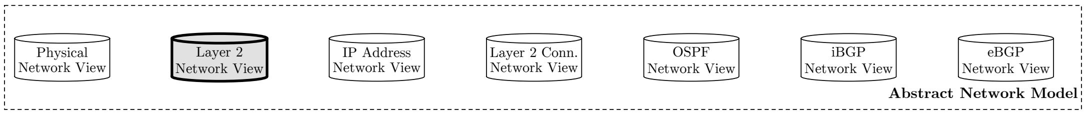


Figure 7.10: Visualisation of Physical Network View. The interface labels shown have been allocated in the later Platform Compiler stage.



of all edges in the topology. For more advanced cases, as discussed in Section 5.7.1, this filtering step becomes more important.

We then split these edges using the SPLIT function, which implements the *split* High-Level Primitive discussed in Figure 5.4.3. We prepend the *bd_* prefix to each of the pseudo-nodes created using the SPLIT function. The suffix of the pseudo-node is formed using the labels of the nodes at each end of the split. For example the label *bd_r3_r4* is generated for the pseudo-node created on the edge between *r3* and *r4*.

We also calculate the mean x and y value based on the nodes at each end of the split, and allocate this to the newly created pseudo-node. For IP allocation purposes, we allocate the most common *asn* value to the pseudo-node, based on the nodes which were split. This is used when allocating IP addresses for the inter-asn edges. Finally, we mark the *device_type* as *broadcast_domain* and the *broadcast_domain* attribute to be *true*, to assist in querying and filtering in subsequent design functions.

The *Layer 2* Network View is constructed using the Design Function described in Listing 7.8, with the output of this Design Function is shown in Figure 7.11.

```
from autonetkit.ank import split
from collections import Counter
```

```
# Create Layer 2 Network View
g_l2 = anm.add_overlay("layer2")
# Add nodes and edges from g_phy
g_l2.add_nodes_from(g_phy)
g_l2.add_edges_from(g_phy.edges())

# Split the point-to-point edges to add a collision domain
edges_to_split = [edge for edge in g_l2.edges()
                  if edge.src.is_l3device() and edge.dst.is_l3device()]

# Mark split edges with attribute
for edge in edges_to_split:
    edge.split = True  # mark as split for use in building nidb

# Split the edges with broadcast domain pseudo-node
split_created_nodes = split(g_l2, edges_to_split, id_prepend='bd_')

for node in split_created_nodes:
    # Set the x,y of split nodes based on neighbor average
    neighs = node.neighbors()
    x = sum(neigh["phy"].get("x") for neigh in neighs) / len(neighs)
    y = sum(neigh["phy"].get("y") for neigh in neighs) / len(neighs)
    node.set("x", x)
    node.set("y", y)

    # Set asn attribute of broadcast domain to be most frequent
    c = Counter(n.get("asn") for n in neighs)
    most_common_asn, _ = c.most_common(1)[0]
    node.set("asn", most_common_asn)

    # Set boolean and device_type attributes
    node.set("broadcast_domain", True)
    node.set("device_type", "broadcast_domain")

# Update visualisation
autonetkit.update_http(anm)
```

Listing 7.8: Layer 2 Design Function

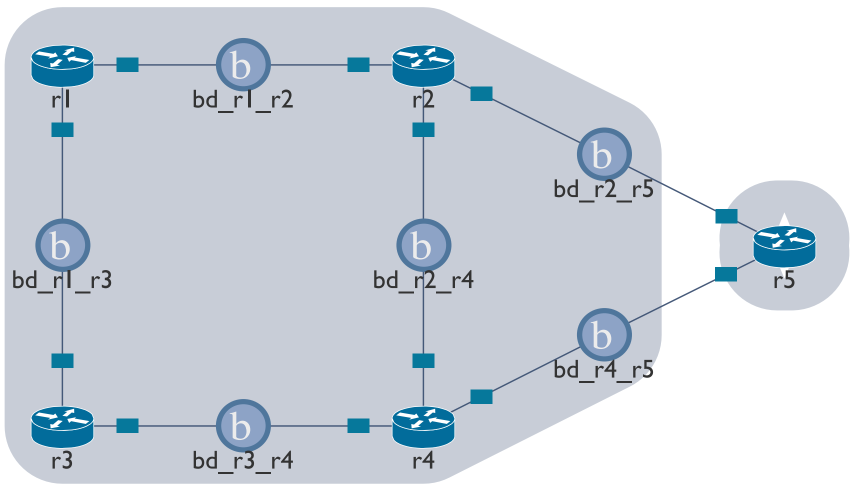
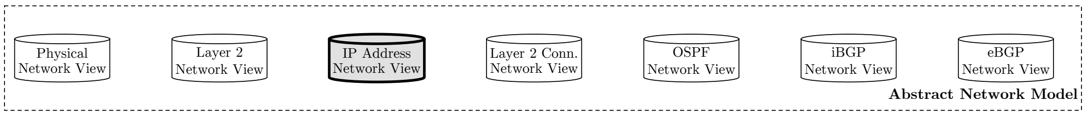


Figure 7.11: Visualisation of Layer 2 Network View

IP Addressing



The *IP Addressing* Design Function uses the simplified algorithm outlined in Section 5.6.3. As discussed in Section 9.3.1 IP Addressing is a complex topic, with many possible algorithms. We use a simplified algorithm in this example. The code for this Design Function is shown in Listing G.3.

The first step in the *IP Addressing* Design Function is to create the overlay. The nodes and edges are added from the *Layer 2* overlay. We also explicitly copy the *broadcast_domain*, *device_type* and *asn* attributes from the *Layer 2* overlay, for the *broadcast_domain* pseudo-nodes. This is because these pseudo-nodes aren't present in the *physical* overlay, and so the pass-through attribute approach does not apply. Future work can generalise the pass-through attribute approach for pseudo-nodes in the implementation. The *x* and *y* co-ordinates are cached as part of the visualisation process, so they do not need to be copied across. The next step is to create the IP Address blocks to be allocated. We use the *netaddr* [71] Python module for this. We first allocate the *loopback* addresses to the routers. We use the *10.0.0.0/16* block for loopback addresses, and allocate a */24* subnet to each AS. We group the nodes based on their *asn* attribute allocated on the *input* overlay, using the GROUP_BY function, which is based on the GROUPING High-Level Primitive. This function gives a list of *(asn, nodes)* tuples, where *nodes* is the list of nodes for the *asn*.

The next step is to iterate over each of these tuples, and get the next *asn_block* from the subnet allocation. We record the address block used for the *asn* into the *loopback_allocations* dictionary, for use in the prefix announcement of the routing protocol Design Functions. We then iterate over each *node* in the list of *nodes* for the *asn*, and allocate the next free IP Address onto the node, and onto the *loopback_zero* interface of the node. The result of this step is shown in Figure 7.12.

We then allocate the *infrastructure* IP addresses to the physical interfaces of the nodes. For this, we use the *192.168.0.0/16* address block. Like for the loopback addresses, we break this into a */24* block for each AS, which we refer to as the *asn_block*. We then filter the nodes in the *IP Addressing* overlay, to obtain the list of *broadcast_domain* nodes. We use the GROUP_BY function to group only these *broadcast_domain* nodes by their *asn* attribute. For each *asn* we get the next IP address block, and record this into the *infra_allocations* dictionary, for advertisement in the routing protocol Design Functions.

As this example contains only point-to-point subnets, we can simply divide the *asn_block* into /30 subnets. This simplifying assumption can be relaxed for more complex topologies involving switches. We then iterate over the grouped *(asn, nodes)* tuple of *broadcast_domain* nodes. We obtain the next /30 prefix from the *asn_block*, and record this onto the *broadcast_domain* node. We use the NEIGHBOR_INTERFACES function for the *broadcast_domain* node to obtain the neighbor interfaces. This function is based on the NEI_BY_NE Low-Level Primitive described in Figure D.2.4, and returns the physical interfaces of the router nodes, connected to this *broadcast_domain* node. We then iterate over the IP addresses of the /30 prefix for this *broadcast_domain*, and allocate them to these physical interfaces. This gives the resulting overlay shown in Figure 7.12.

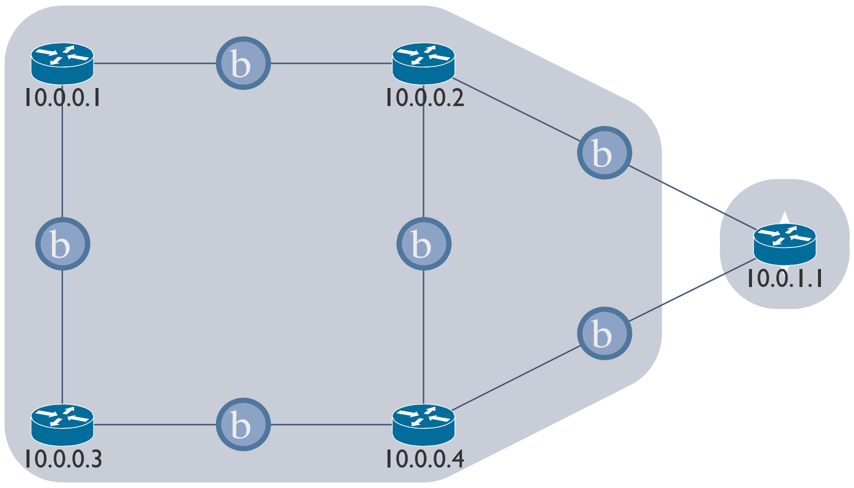
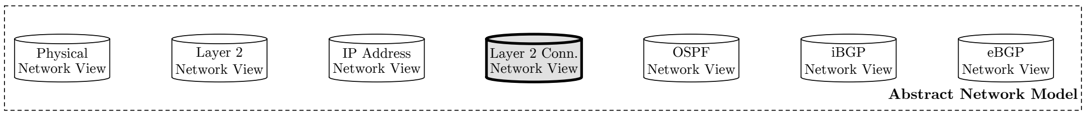


Figure 7.12: Visualisation of IP Network View. Node labels show loopback IP allocations.

Layer 2 Connectivity



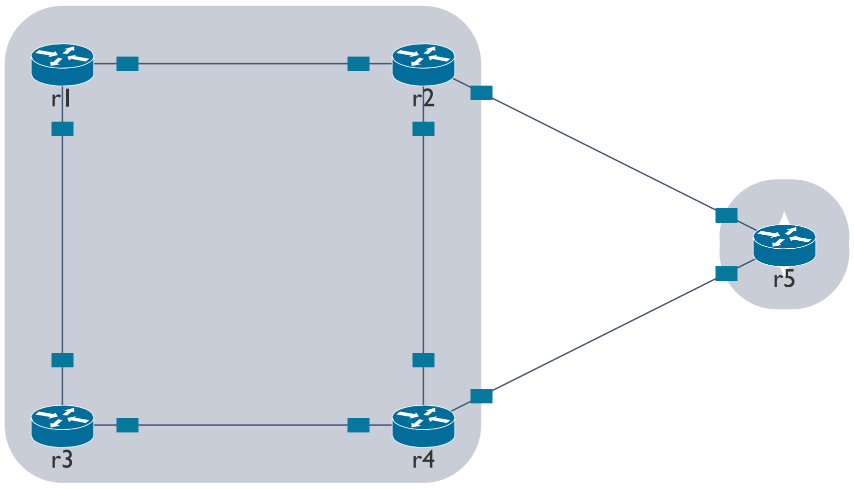


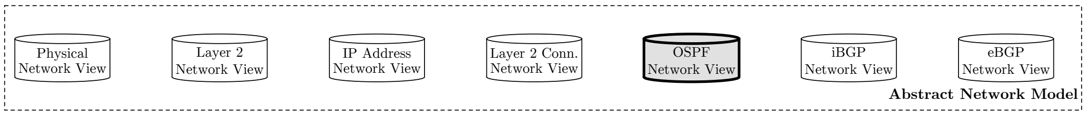
Figure 7.13: Visualisation of Layer 2 Connectivity Network View

The *Layer 2 Connectivity* Network View is used for the routing Design Functions. It is an *intermediate* Network View. The code for the Design Function is shown in Listing G.2. This code is shown in the appendix. This Design Function first creates the *Layer 2 Connectivity* overlay in the Abstract Network Model, and adds the nodes and edges from the *Layer 2* overlay. It then uses the node query function to select the nodes which have the *broadcast_domain* attribute set to *true*. This generalisation allows nodes such as switches to be also allocated with the *broadcast_domain* attribute, to handle more complex topologies without needing to modify the *Layer 2 Connectivity* Design Function.

We then use the EXPLODE function, based on the [] EXPLODE High-Level Primitive. This explodes out the links connected to the *broadcast domain* nodes. In this simplified example, where we only have point-to-point links between routers, this results in a *Layer 2 Connectivity* overlay which has the same connectivity as the *physical* overlay. For more complex topologies as discussed in Section 5.7.1 the *Layer 2 Connectivity* overlay would be different to the *physical* overlay. This generalisation and use of an *intermediate* network view also decouples the routing Design Functions from the physical topology. We demonstrate the power of this approach in the case studies in Chapter 8.

The result of the *Layer 2 Connectivity* Design Function is shown in Figure 7.13.

OSPF



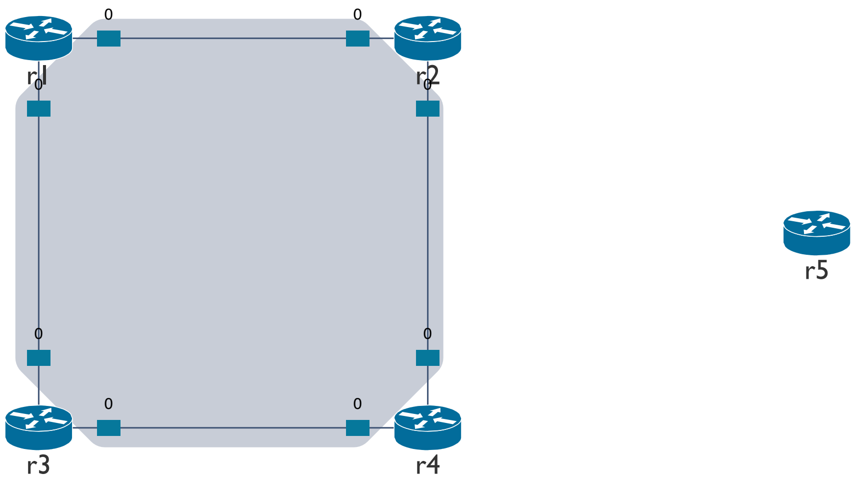


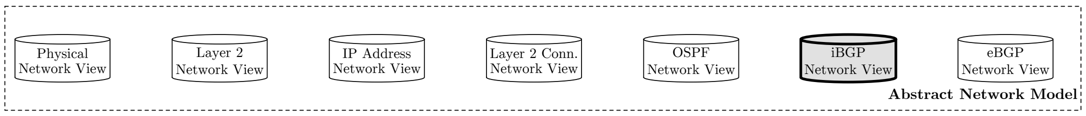
Figure 7.14: Visualisation of OSPF Network View. Interface labels show the value of the *area* attribute.

The *OSPF* Network View is shown in Figure 7.14. As per the Design Functions shown in Listing G.4, the *OSPF* Network View is constructed from the routers and interfaces from the *physical* Network View, and the edges from the Layer 2 Connectivity Network View, where the edge connects two nodes in the same AS. These edges represent OSPF adjacencies.

A default OSPF *area* attribute of *0* is set on each interface that is connected in the OSPF Network View. A default OSPF *area* attribute of *0* is also set on the Loopback Zero interface of each node in the *OSPF* Network View.

Listing G.5 shows the process to map the prefixes to be advertised by the OSPF routing process. For each node, these are the prefixes allocated to each of the interfaces of the node, from the *IP Addressing* Network View. These include the *infrastructure* and *loopback zero* IP address prefixes.

iBGP



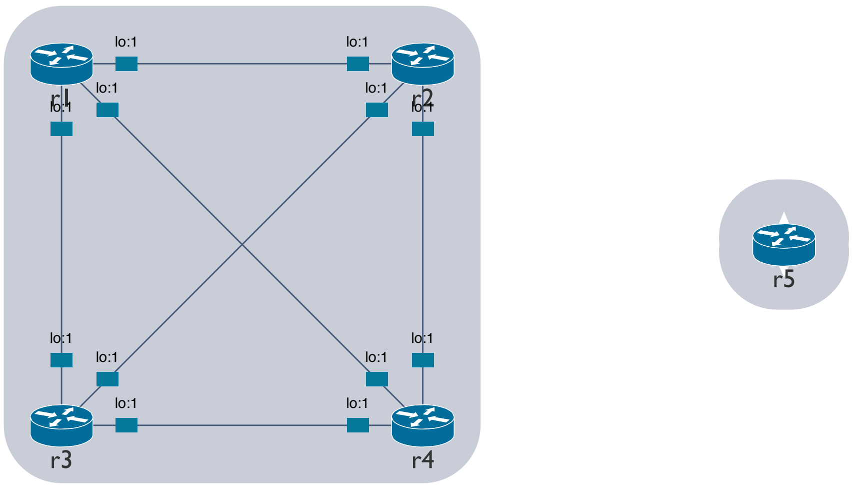
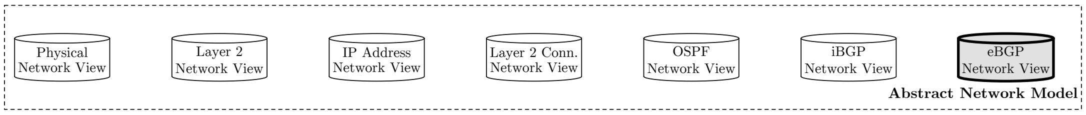


Figure 7.15: Visualisation of iBGP Network View. iBGP Session endpoints have been annotated with the bound_to interface labels

The *iBGP* Network View is shown in Figure 7.15. The Design Functions used to construct is it shown in Listing G.6. The Design Function first adds the routers and interfaces from the Layer 2 Network View. It then groups the nodes by their *asn* attribute. Within each AS group of nodes, a full-mesh is constructed by connecting each node to each other node in the AS group with an edge. These edges represent iBGP sessions.

A logical interface is created to represent the BGP session endpoint, and is connected by the edge. To establish the iBGP session between loopback interfaces, the logical interface is associated to the Loopback Zero interface. This association is shown on the interface labels in Figure 7.15.

eBGP



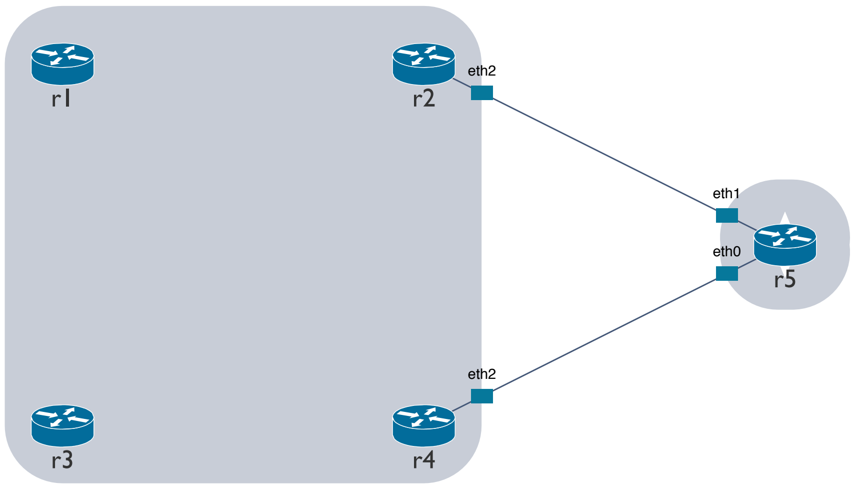


Figure 7.16: Visualisation of eBGP Network View. eBGP Session endpoints have been annotated with the bound_to interface attributes

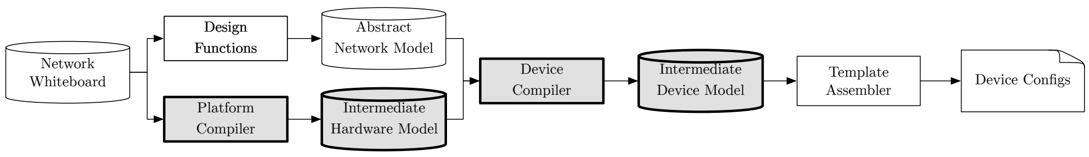
The *eBGP* Network View is shown in Figure 7.16, with the associated Design Function shown in Listing G.7. The eBGP Design Function first adds the routers and interfaces from the Layer 2 Network View. It then adds the edges from the *Layer 2 Connectivity* Network View, where the edge connects routers that do not have the same *asn* value. These edges represent eBGP sessions.

To construct these edges we first add a logical interface to the nodes at each end of the edge. This logical interface is then associated to the corresponding physical interface from the Layer 2 Connectivity Network View. This establishes the eBGP session on the physical interface.

The edge is then created with these logical interfaces as the eBGP session termination points. This is shown in Figure 7.16, where the *interface* attributes show the associated physical interface for the eBGP session. The edge from r2 to r5 is from the *Layer 2 Connectivity* edge between *eth2* of *r2* and *eth0* of *r5*, and the edge from *r4* to *r5* is from the *Layer 2 Connectivity* edge between *eth2* of *r4* and *eth0* of *r5*.

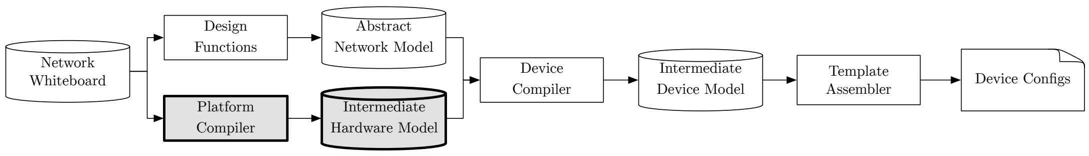
Finally, the code shown in Listing G.8 is used to aggregate the ASN prefixes to be advertised over eBGP.

7.4.4 *Compiling*



We now provide an overview of implementing the Platform Compiler and Device Compiler process in AutoNetkit.

Netkit Platform Compiler



The Platform Compiler performs the roles discussed previously in Section 6.3.2. The code for the Netkit Platform Compiler is shown in Listing G.9, and we provide an overview of the code here.

The Netkit Platform Compiler is created as a Python Class, which allows it to be extended in the future using inheritance, as discussed in Chapter 6. The main compiler logic is performed in the COMPILE function. This function returns an *nidb* data structure, which is a Python dictionary containing the information required for the templates. This is the Intermediate Hardware Model for each node, a list of the subnets, and a list of the *tap* management interfaces.

As we did not allocate interface ids in the Network Whiteboard step, we first iterate over each of the nodes in the topology, and run the SETUP_INTERFACES command. This allocates the interface ids, such as *eth0*, *eth1*, *eth2*, etc.

We then create the wiring information for the simulation to specify which interfaces on the simulated devices get connected together. Netkit connectivity is specified in the form of a 3-tuple of (*node*, *interface*, *collision_domain_alias*). All (*node*, *interface*) tuples sharing the same *collision_domain_alias* label will be connected to the same collision domain. For simplicity in this example, we use the IP subnet as the *collision_domain_alias*.

We then allocate the *tap* interfaces for the out-of-band management access. This is performed by adding an additional interface to each node, with the appropriate next free interface id.

The result of all of these operations are stored in the *nidb* dictionary as a *lab*. This allows expansion to generate multiple labs from a single Network Whiteboard and Abstract Network Model. These multiple labs could be launched on different hosts or on different simulation environments, and interconnected.

An example of the lab output for this topology is shown in Listing 7.9.

```
{
  "machines": ["r1", "r2", "r3", "r4", "r5"],
  "subnets": [
    {"host": "r1", "interface_id": "0", "subnet": "192.168.0.0.30"},
    {"host": "r1", "interface_id": "1", "subnet": "192.168.0.4.30"},
    {"host": "r2", "interface_id": "0", "subnet": "192.168.0.0.30"},
    {"host": "r2", "interface_id": "1", "subnet": "192.168.0.8.30"},
    {"host": "r2", "interface_id": "2", "subnet": "192.168.0.12.30"},
    {"host": "r3", "interface_id": "0", "subnet": "192.168.0.4.30"},
    {"host": "r3", "interface_id": "1", "subnet": "192.168.0.16.30"},
    {"host": "r4", "interface_id": "0", "subnet": "192.168.0.8.30"},
    {"host": "r4", "interface_id": "1", "subnet": "192.168.0.16.30"},
    {"host": "r4", "interface_id": "2", "subnet": "192.168.0.20.30"},
    {"host": "r5", "interface_id": "0", "subnet": "192.168.0.20.30"},
    {"host": "r5", "interface_id": "1", "subnet": "192.168.0.12.30"}
  ],
  "taps": {
    "linux_host": "172.16.0.1",
    "tap_hosts": [
      {"interface_id": 2, "node": "r1", "tap_ip": "172.16.0.5"},
      {"interface_id": 3, "node": "r2", "tap_ip": "172.16.0.6"},
      {"interface_id": 2, "node": "r3", "tap_ip": "172.16.0.7"},
      {"interface_id": 3, "node": "r4", "tap_ip": "172.16.0.3"},
      {"interface_id": 2, "node": "r5", "tap_ip": "172.16.0.4"}
    ],
    "tap_vm_ip": "172.16.0.2"
  }
}
```

Listing 7.9: Lab output showing *nidb* information

The final step is to run the Device Compiler for each node, and to store the result into the *nidb*. This is discussed in the next step.

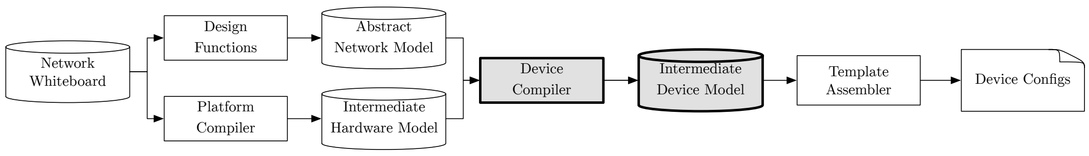
The lab configuration for this Intermediate Platform Model is shown in Listing 7.10. This configuration is rendered using the lab template discussed later in this section.

```
machines = "r4, r5, r1, r2, r3"

r4[0]=192.168.0.8.30
r4[1]=192.168.0.16.30
r4[2]=192.168.0.20.30
r5[0]=192.168.0.20.30
r5[1]=192.168.0.12.30
r1[0]=192.168.0.0.30
r1[1]=192.168.0.4.30
r2[0]=192.168.0.0.30
r2[1]=192.168.0.8.30
r2[2]=192.168.0.12.30
r3[0]=192.168.0.4.30
r3[1]=192.168.0.16.30
```

```
r4[3]=tap,172.16.0.1,172.16.0.3
r5[2]=tap,172.16.0.1,172.16.0.4
r1[2]=tap,172.16.0.1,172.16.0.5
r2[3]=tap,172.16.0.1,172.16.0.6
r3[2]=tap,172.16.0.1,172.16.0.7
```

Listing 7.10: Example Netkit lab configuration for House topology

Quagga Device Compiler



We now discuss the Device Compiler, which transforms the Abstract Network Model into the Intermediate Device Model format suitable to render the device configuration using templates. The code for the Quagga Device Compiler is shown in Listing G.12, which we provide a summary of here.

Our target is an individual configuration file for each of the Quagga daemons (OSPF, BGP, etc). We note that a large part of the same compiler logic could be used for a single-configuration file *vtysh* approach.

The compile function takes a *node* and returns a dictionary containing each of the entries required for the templates. These entries include the *hostname* and *asn* attributes of the node. Further, there are data structures for each of *zebra*, *ssh*, *interfaces*, *ospf* and *bgp*. These data structures are provided by further functions called by the compile function. This dictionary returned is the Intermediate Hardware Model for the specific device provided to the compile function.

The zebra function is used to specify the password for the Zebra routing daemon process. The ssh function specifies whether SSH key access is allowed to automate logins. For this example it is disabled.

The interfaces function iterates over each of the interfaces of the *nidb*. This includes the interfaces from the Abstract Network Model, including the loopback interface, and also those interfaces added by the Platform Compiler, such as the *tap* out-of-band management interfaces. This function is used for the *ifconfig* commands to configure the interface in Linux within Netkit.

The ospf function iterates over each of the interfaces of the given node, and appends the *area* and *network* subnet to the list of OSPF networks for that node.

Finally, the BGP function handles both iBGP and eBGP. For the *ibgp* and *ebgp* overlays, the edges (corresponding to sessions) are iterated over. The interface binding is looked up, and a dictionary is created for each session. This dictionary contains the IP address of the neighbour, the *asn* attribute of the neighbour, a short description for the session, and the IP address to use for the *update source* BGP configuration directive. For this example, the *update source* is set to the interface of which the session is bound. For *ibgp* this is the loopback interface, and for *ebgp* this is the physical interface. The ASN prefixes to be advertised are included, and the Loopback IP address is used for the *router_id* to use to uniquely identify the BGP speaker in the network

To summarise, the return value of the BGP function is a dictionary containing a list of the *ibgp_neighbors* dictionaries, a list of the *ebgp_neighbors* dictionaries, the *networks* to announce, and the *router_id*.

An example of the Intermediate Device Model for node *r4* is shown in Listing 7.11.

```
{
  "asn": 1,
  "bgp": {
    "ebgp_neighbors": [
      {"asn": 2, "desc": "Router r5",
        "neigh_ip": "192.168.0.22", "update_source": "192.168.0.21"}
    ],
    "ibgp_neighbors": [
      {"asn": 1, "desc": "Router r1",
        "neigh_ip": "10.0.0.1", "update_source": "10.0.0.4"},
      {"asn": 1, "desc": "Router r2",
        "neigh_ip": "10.0.0.2", "update_source": "10.0.0.4"},
      {"asn": 1, "desc": "Router r3",
        "neigh_ip": "10.0.0.3", "update_source": "10.0.0.4"}
    ],
    "networks": ["10.0.0.0/24", "192.168.0.0/24"],
    "router_id": "10.0.0.4"
  },
  "hostname": "r4",
  "interfaces": [
    {"broadcast": "192.168.0.11", "id": "eth0",
    "ip": "192.168.0.10", "netmask": "255.255.255.252"},
    {"broadcast": "192.168.0.19", "id": "eth1",
    "ip": "192.168.0.18", "netmask": "255.255.255.252"},
    {"broadcast": "192.168.0.23", "id": "eth2",
    "ip": "192.168.0.21", "netmask": "255.255.255.252"},
    {"broadcast": "172.16.255.255", "id": "eth3",
    "ip": "172.16.0.3", "netmask": "255.255.0.0"},
    {"broadcast": null, "id": "lo:1", "ip": "10.0.0.4",
    "netmask": "255.255.255.255"}
  ],
  "ospf": {
    "networks": [
      {"area": 0, "network": "10.0.0.4/32"},
      {"area": 0, "network": "192.168.0.8/30"},
      {"area": 0, "network": "192.168.0.16/30"}
    ]
  },
  "ssh": {"use_key": false},
  "zebra": {"password": "zebra"}
}
```

Listing 7.11: Example Intermediate Device Model for node *r4* of House topology

The configurations for OSPF and BGP are shown in Listing 7.12 and Listing 7.13 respectively. These are rendered using the templates discussed later in this section

```
!
hostname r4
password zebra
enable password zebra
!
!
router ospf
    network 192.168.0.4/30 area 0
    network 192.168.0.16/30 area 0
!
log file /var/log/zebra/ospfd.log
!
```

Listing 7.12: Example OSPF configuration for *r4* of House topology

```
!
hostname r4
password zebra
enable password zebra
!
router bgp 1
network 172.16.0.0/24
network 192.168.0.0/24
! 172.16.0.2
neighbor 172.16.0.2 remote-as 1
neighbor 172.16.0.2 description Router r1
! 172.16.0.3
neighbor 172.16.0.3 remote-as 1
neighbor 172.16.0.3 description Router r2
! 172.16.0.4
neighbor 172.16.0.4 remote-as 1
neighbor 172.16.0.4 description Router r3
!
!
neighbor 192.168.0.2 remote-as 2
neighbor 192.168.0.2 description Router r5
!
!
log file /var/log/zebra/bgpd.log
!
debug bgp
debug bgp events
debug bgp filters
debug bgp fsm
debug bgp keepalives
debug bgp updates
!
```

Listing 7.13: Example BGP configuration for *r4* of House topology

Compilation Result

An example of running the created Platform Compiler is shown in Listing 7.14.

```
# Create Platform Compiler and Compile
sim_plat = simple_platform_compiler(anm, nidb)
sim_plat.compile()
```

Listing 7.14: Compile Platform

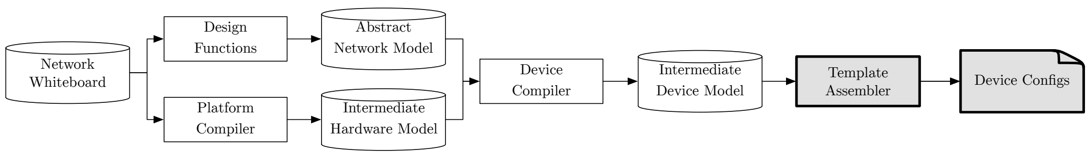
7.4.5 *Templates*

The Netkit Lab.conf Template Listing G.22 is for the lab, specifying the nodes to start, the links between them, and the TAP management interfaces.

There are four templates for configuring the Quagga routing processes. The Zebra Template Listing G.17 is for the base Zebra setup for Quagga, such as setting login to the Zebra shell, and static routes. The OSPF template Listing G.16 contains the OSPF configuration information OSPF Template, and the BGP template Listing G.14 contains the BGP (both iBGP and eBGP) configuration information. Finally, the Quagga Daemons Template Listing G.15 specifies the Zebra routing processes to run, in this case Zebra, OSPF and BGP.

Each virtual machine has a Netkit Device Startup Template Listing G.19, which configures the Linux ethernet interfaces and IP addresses, and starts the SSH and Zebra processes. It also sets the SSH password to allow SSH login. The Hostname Template Listing G.18 sets the hostname of the device, and the Shadow Template Listing G.20 and SSH Config template Listing G.21 are used to setup the SSH login process.

7.4.6 *Template Assembly*



Once we have the templates, we can render the data into the templates, as shown in Listing G.23 and Listing G.24. For convenience, a number of shortcuts into the *NIDB* dictionary are defined as shown in Listing G.10. This step implements the configuration assembly step discussed in Chapter 6.

The rendered output is written into a gzip archive, which can be copied to the Netkit server, extracted, and the lab launched. Writing to a *.tar.gz* archive in memory, and then writing the final archive to disk can significantly optimise performance by reducing disk I/O operations. This is especially important for the large-scale case study discussed in Section 8.6.

7.4.7 *Example Output*

An example of output of the process is shown below. The directory file structure is shown in Listing 7.15. We also show example router configs for the router *r4*. Listing 7.16 shows the ospfd.conf output, and Listing 7.17 shows the bgpd.conf output.

```
|-- lab.conf
|-- r1
|   -- etc
|       |-- hostname
|       |-- shadow
|       |-- ssh
|       |   -- ssh_config
|       -- zebra
|           |-- bgpd.conf
|           |-- daemons
|           |-- ospfd.conf
|           -- zebra.conf
|-- r1.startup
|-- r2
|   -- etc
|       |-- hostname
|       |-- shadow
|       |-- ssh
|       |   -- ssh_config
|       -- zebra
|           |-- bgpd.conf
|           |-- daemons
|           |-- ospfd.conf
|           -- zebra.conf
|-- r2.startup
|-- r3
|   -- etc
|       |-- hostname
|       |-- shadow
|       |-- ssh
|       |   -- ssh_config
|       -- zebra
|           |-- bgpd.conf
|           |-- daemons
|           |-- ospfd.conf
|           -- zebra.conf
|-- r3.startup
|-- r4
|   -- etc
|       |-- hostname
|       |-- shadow
|       |-- ssh
|       |   -- ssh_config
|       -- zebra
|           |-- bgpd.conf
|           |-- daemons
|           |-- ospfd.conf
|           -- zebra.conf
|-- r4.startup
|-- r5
|   -- etc
|       |-- hostname
|       |-- shadow
|       |-- ssh
|       |   -- ssh_config
|       -- zebra
|           |-- bgpd.conf
|           |-- daemons
|           |-- ospfd.conf
|           -- zebra.conf
-- r5.startup

20 directories, 41 files
```

Listing 7.15: Tree Output for Lab

```
!
hostname r4
password zebra
enable password zebra
!
```

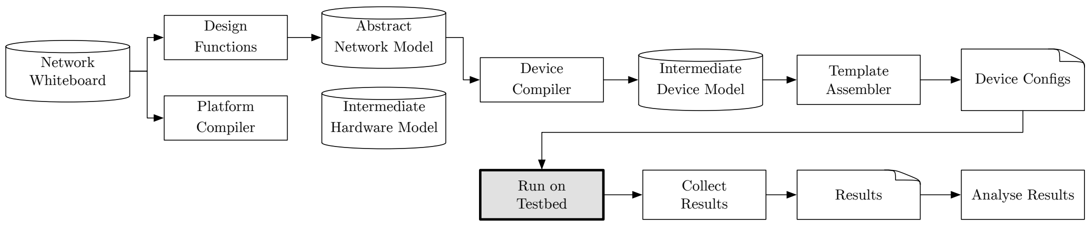
```
!
router ospf
    network 192.168.0.4/30 area 0
    network 192.168.0.16/30 area 0
!
log file /var/log/zebra/ospfd.log
!
```

Listing 7.16: Example ospfd.conf output for r4

```
!
hostname r4
password zebra
enable password zebra
!
router bgp 1
network 172.16.0.0/24
network 192.168.0.0/24
! 172.16.0.2
neighbor 172.16.0.2 remote-as 1
neighbor 172.16.0.2 description Router r1
! 172.16.0.3
neighbor 172.16.0.3 remote-as 1
neighbor 172.16.0.3 description Router r2
! 172.16.0.4
neighbor 172.16.0.4 remote-as 1
neighbor 172.16.0.4 description Router r3
!
!
neighbor 192.168.0.2 remote-as 2
neighbor 192.168.0.2 description Router r5
!
!
log file /var/log/zebra/bgpd.log
!
debug bgp
debug bgp events
debug bgp filters
debug bgp fsm
debug bgp keepalives
debug bgp updates
!
```

Listing 7.17: Example bgpd.conf output for r4

7.4.8 *Launching*

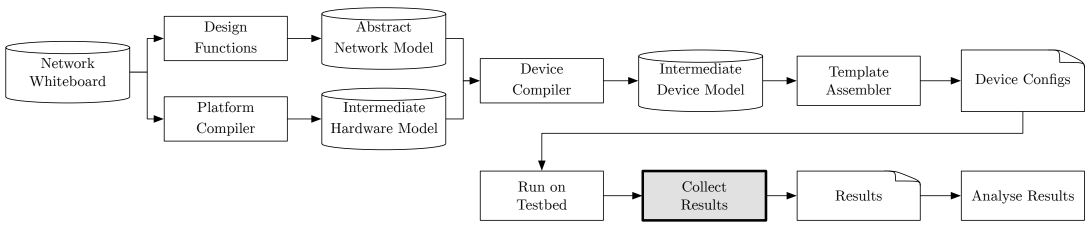


The topology can be launched by transferring the *.tar.gz* lab generated by AutoNetkit across to the destination server. It can then be extracted and launched using the Netkit *lstart* command, as shown in Listing 7.18.

```
ubuntu@ip-172-31-14-185:~/lab$ lstart -o --con0=none -p5

======================== Starting lab ===========================
Lab directory: /home/ubuntu/lab
Version:       <unknown>
Author:        <unknown>
Email:         <unknown>
Web:           <unknown>
Description:
<unknown>
=================================================================
You chose to use parallel startup.
Starting "r1"...
Starting "r2"...
Starting "r3"...
Starting "r5"...
Starting "r4"...

The lab has been started.
=================================================================
```

Listing 7.18: Launching Lab

7.4.9 *Measuring*



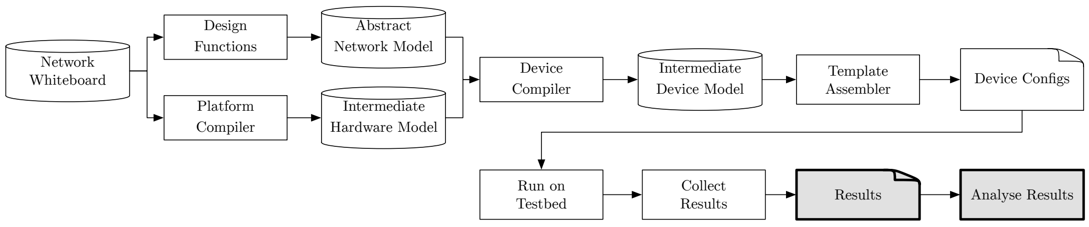
The running simulation can be measured by logging into one of the virtual routers using a command such as telnet or ssh, and then running a diagnostic command. Listing 7.19 shows the output of displaying the routing table, using *show ip route* for the node *r4*. We can see the login process through telnet, and then the running of the command. This can be automated using expect scripting, in the collection framework discussed in Section 7.2.4.

```
ubuntu@ip-172-31-14-185:~/lab$ telnet 172.16.0.3 zebra
Trying 172.16.0.3...
Connected to 172.16.0.3.
Escape character is '^]'.
MOTD file not found

User Access Verification

Password:
r4> en
```

```
Password:
r4# sh ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

C>* 127.0.0.0/8 is directly connected, lo
C>* 172.16.0.0/16 is directly connected, eth3
B>* 172.16.1.0/24 [20/0] via 192.168.0.2, eth2, 00:00:52
C>* 192.168.0.0/30 is directly connected, eth2
O   192.168.0.4/30 [110/10] is directly connected, eth0, 00:00:58
C>* 192.168.0.4/30 is directly connected, eth0
O>* 192.168.0.12/30 [110/20] via 192.168.0.18, eth1, 00:00:03
O   192.168.0.16/30 [110/10] is directly connected, eth1, 00:00:58
C>* 192.168.0.16/30 is directly connected, eth1
O>* 192.168.0.20/30 [110/20] via 192.168.0.6, eth0, 00:00:06
```

Listing 7.19: Logging in and running SHOW IP ROUTE from *r4*

7.4.10 *Simulation Results*



The results of the measurement process can then be analysed. The routing table of *r3* is shown in Listing G.25 and post-processed, where the IP addresses are reverse-mapped to the respective interface elements, is shown in Listing 7.20. Additional diagnostic outputs include the OSPF neighbours, shown in Listing G.26, or the BGP neighbours, shown in Listing 7.22, and Listing G.27. Another form of results collection is to view the path taken across a network using the *traceroute* command, as shown in Listing 7.21.

```
r3# sh ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

B>* 10.0.0.0/24 [200/0] via lo:1.r4 (recursive via eth1.r4, eth1),
    00:04:56
O>* lo:1.r1/32 [110/20] via eth1.r1, eth0, 00:06:12
O>* lo:1.r2/32 [110/30] via eth1.r1, eth0, 00:06:05
  *                     via eth1.r4, eth1, 00:06:05
O   lo:1.r3/32 [110/10] is directly connected, lo, 00:06:57
C>* lo:1.r3/32 is directly connected, lo
O>* lo:1.r4/32 [110/20] via eth1.r4, eth1, 00:06:05
B>* 10.0.1.0/24 [200/0] via lo:1.r4 (recursive via eth1.r4, eth1),
    00:04:56
C>* 127.0.0.0/8 is directly connected, lo
C>* 172.16.0.0/16 is directly connected, eth2
```

```
B>* 192.168.0.0/24 [200/0] via lo:1.r4 (recursive via eth1.r4, eth1),
    00:04:56
O>* 192.168.0.0/30 [110/20] via eth1.r1, eth0, 00:06:12
O   192.168.0.4/30 [110/10] is directly connected, eth0, 00:06:57
C>* 192.168.0.4/30 is directly connected, eth0
O>* 192.168.0.8/30 [110/20] via eth1.r4, eth1, 00:06:05
O   192.168.0.16/30 [110/10] is directly connected, eth1, 00:06:05
C>* 192.168.0.16/30 is directly connected, eth1
```

Listing 7.20: Post-processed SHOW IP ROUTE from *r3*

```
hostname r3:~# traceroute 10.0.1.1
traceroute to 10.0.1.1 (10.0.1.1), 64 hops max, 40 byte packets
 1  192.168.0.18 (192.168.0.18)  1 ms  0 ms  0 ms
 2  10.0.1.1 (10.0.1.1)  1 ms  1 ms  1 ms
```

Listing 7.21: Output of TRACEROUTE from *r3* to *r5*

```
r4# sh ip bgp summary
BGP router identifier 10.0.0.4, local AS number 1
RIB entries 5, using 320 bytes of memory
Peers 4, using 10064 bytes of memory

Neighbor        V   AS MsgRcvd MsgSent   TblVer  InQ OutQ Up/Down  State/PfxRcd
10.0.0.1        4    1      27      30        0    0    0 00:25:20        0
10.0.0.2        4    1      30      30        0    0    0 00:25:15        3
10.0.0.3        4    1      27      30        0    0    0 00:25:20        0
192.168.0.22    4    2      31      31        0    0    0 00:27:15        1

Total number of neighbors 4
```

Listing 7.22: SHOW IP BGP SUMMARY from *r4*

7.4.11 *Conclusion*

In this section we have presented an implementation which we allowed us to demonstrate our approach is feasible and able to generated correct configurations.

7.5 PRIOR PUBLICATIONS

The work of this chapter has been previously published. In an early joint paper, Nguyen *et al.* [77], a framework for creating testbed for networks was proposed. This used an object-oriented approach. This paper was primarily focussed on BGP configuration.

A second paper was presented as demonstration [55]. This was more closely aligned to this thesis in that it introduced a graph-based intermediate format, visual capture of network topology and policy, high-level network design, automated topology visualisation, extensible configuration, and template-based low-level compilation.

A third paper [54] refined these concepts, enhanced the automated visualisation system, and presented a workflow for data collection and processing from the network simulation.

Finally, the work presented in Knight *et al.* [57] presented the concepts described in this thesis with an emphasis on an experimentation framework. This introduced the attribute graph concept which was expanded to Network Views in Chapter 4.

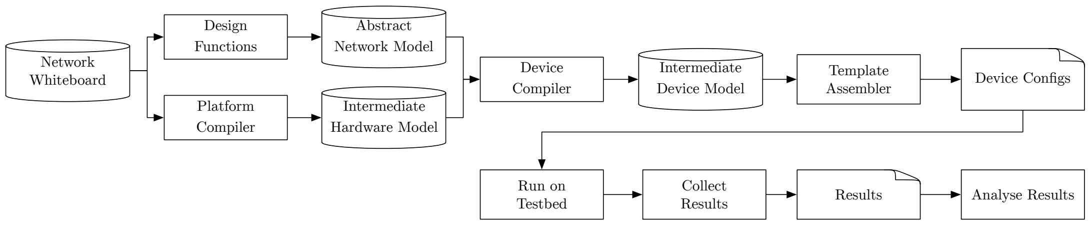
7.6 CONCLUSION

In this chapter we have described how the theoretical methodology described in Chapter 4, Chapter 5, and Chapter 6 has been implemented and validated using software infrastructure. This implementation and validation enables us to address Research Question 5: *What are the scalability and extensibility characteristics for the compilation of High-Level Network Configuration Policy to device configurations in terms of network size and diversity of network protocols and target devices?*, in the four case studies in Chapter 8 which illustrate the correctness, scalability and flexibility of the tool.

8

CASE STUDIES

8.1 INTRODUCTION



In this chapter we use the experimentation setup described in Chapter 7 to evaluate the approach described in this thesis, with four case studies. This allows us to demonstrate the flexibility of the approach to accommodate different topologies, protocol designs, and target devices and platforms.

This will allow us to demonstrate how our approach and toolchain addresses Research Question 5: *What are the scalability and extensibility characteristics for the compilation of High-Level Network Configuration Policy to device configurations in terms of network size and diversity of network protocols and target devices?*

8.2 CHAPTER OVERVIEW

This chapter looks at four case studies.

The first case study revisits the Simplified Small Internet Example topology from Chapters 4, 5, and 6, showing an implementation from a Network Whiteboard description through to a running simulation on Quagga routers in Netkit. This builds upon the *House Topology* example described in Section 7.4.

The second case study relaxes the simplifying assumptions made in Chapter 4, to implement the full Simplified Small Internet Example topology from Di Battista *et al.* [25]. This demonstrates the use of the system to reproduce a more realistic example, with specific IP addresses, multi-homed eBGP connections, and BGP routing policy.

The third case study focusses on a managed Layer 2 switching network, through the use of VLANs. This presents the use of the system in an Enterprise/Campus environment, in addition to the Wide-Area Network examples of the first two case studies. This case study revisits the theoretical VLAN topology and approach outlined in Section 5.7.2. In order to validate this case study we use the Cisco IOSv router and IOSvL2 managed switch virtual machines, running inside the Cisco VIRL simulation platform. This allows us to also demonstrate the use of our framework to target alternative device operating systems, and simulation platforms. We validate the configurations through the output of switch and router *show* commands, and a traceroute across the network topology.

Finally, the fourth case study looks at the performance of the system in a large-scale example of 1,154 routers divided into 40 Autonomous Systems. We show that the system is able to allocate IP addresses, designate route reflectors, build the OSPF, iBGP and eBGP topologies, and generate configuration files for the routers in under ten seconds on a modern MacBook-Pro. This demonstrates how our approach is able to handle large-scale networks.

8.3 SIMPLIFIED NETKIT SMALL-INTERNET LAB

8.3.1 *Introduction*

In this section we revisit the Small Internet lab example from Netkit, used in Chapters 4, 5, and 6, to show how it can be reproduced using AutoNetkit, the Python reference implementation of our approach.

We begin by describing the Network Whiteboard used to capture the high-level requirements. We then discuss the use of the Design Functions originally discussed in Section 7.4.3, We then compile for the Netkit target platform, using the Netkit platform compiler, and Quagga device compiler developed in Section 7.4.4. We conclude by presenting experimental results gathered from a running Netkit simulation. We show the output of the TRACEROUTE command and plot this against the topology diagram to validate the routing protocol operation. We also discuss the output of the routing table and BGP peering diagnostic commands.

A mapping of the names used in the theoretical examples in Chapter 4, Chapter 5, and Chapter 6 to those used in this and the next case study is shown in Table 8.1.

Table 8.1: Translation of names from theoretical example

Theoretical Name	Case Study Name	Theoretical Name	Case Study Name
1	as1r1	8	as10r2
2	as20r3	9	as10r1
3	as20r2	10	as10r3
4	as20r3	11	as300r1
5	as200r	12	as300r3
6	as30r1	13	as300r2
7	as40r1	14	as300r4

8.3.2 *Methodology*

The methodology used to implement and verify this case study is shown in Figure 8.1. In this case study we only vary the Network Whiteboard, with the rest of the toolchain as shown in Figure 7.7. This allows us to demonstrate the versatility of decoupling the network design and configuration generation from the high-level specification.

For this example the Network Whiteboard is represented in GraphML format rather than specified programmatically in Python. We discuss the benefits of this approach. The remainder of the Design Functions and Compilation steps are as per the House example in Chapter 7 and follows the same methodology. We use the same series of collection commands, TRACEROUTE, SHOW IP BGP, and SHOW IP ROUTE to gather results about the running network and to allow us to verify the configurations generated accurately represent the high-level Network Whiteboard specification.

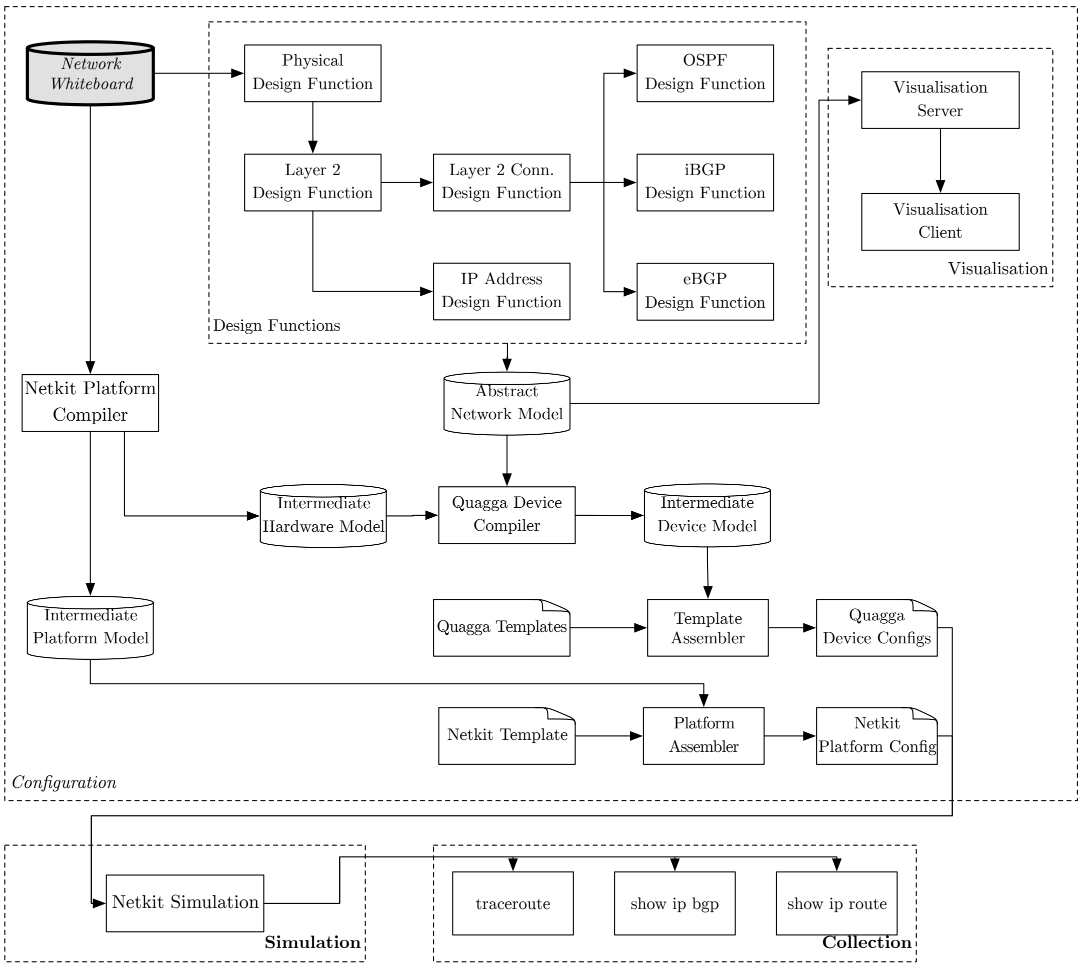


Figure 8.1: Configuration Generation and Verification methodology for Simplified Small Internet Case Study

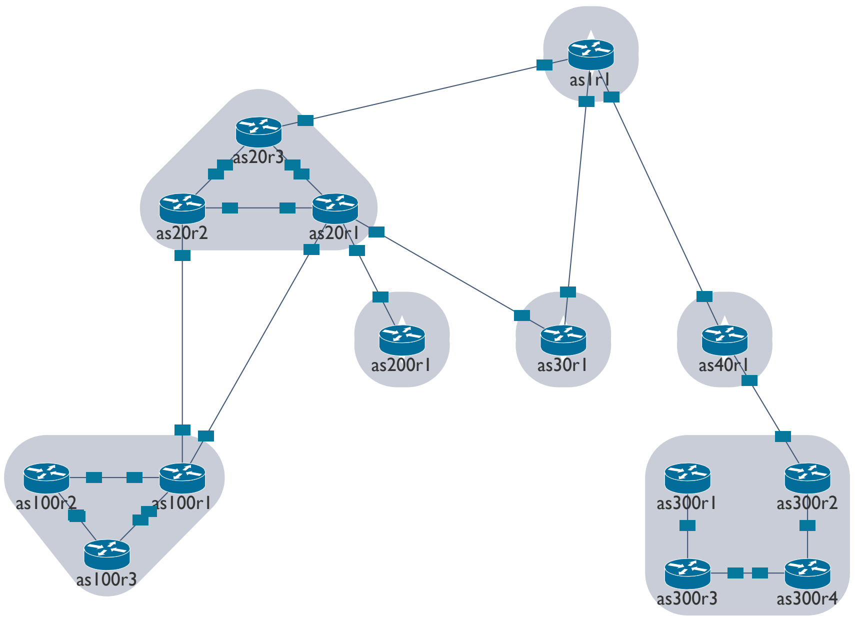


Figure 8.2: Visualisation of Network Whiteboard

8.3.3 *Network Whiteboard*

The Network Whiteboard for the Small Internet example is created as a GraphML [12] file. This captures the key components of the topology from Di Battista *et al.* [25] shown previously in Figure 4.31.

The GraphML file was initially created with the freely available yEd[1] GUI graph editor software tool, as shown in Figure 8.3. We then load the GraphML file using the built-in support for GraphML in NetworkX. This is loaded using the AutoNetkit wrapped around the NetworkX GraphML loader, as shown in Listing H.1.

For clarity, in this example we then iterate over the nodes and edges of the GraphML file to produce two Python dictionaries. These are reproduced in Listing H.2 for the nodes, and Listing H.3 for the edges. This shows the simplicity of the data required for the Network Whiteboard. We can see the node information contains the *asn* value, which is used to create the OSPF, iBGP and eBGP Network Views. This then creates the *input* overlay for the Abstract Network Model which corresponds to the Network Whiteboard.

Finally, we update the visualisation with the updated Abstract Network Model, as shown in Listing H.4. This single command shows the simplicity of visualising the Abstract Network Model using the automated visualisation component of our implementation. The result of the visualisation of the Abstract Network Model, showing the

1 http://www.yworks.com/products/yed

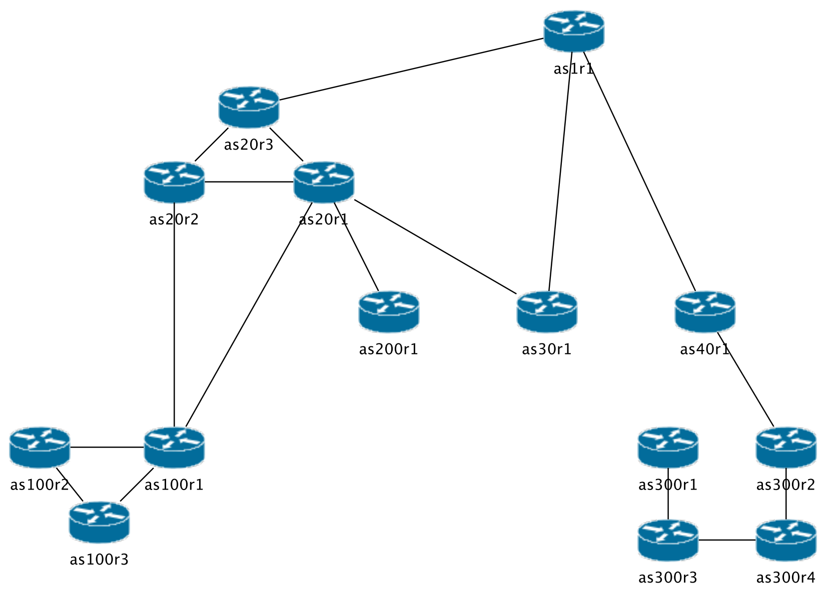


Figure 8.3: Drawing of Network Whiteboard using yEd editor.

input overlay (corresponding to the Network Whiteboard) is shown in Figure 8.2.

8.3.4 *Design Functions*

In this section we show how the Design Functions which have been discussed in previous chapters can be implemented using Python code within our framework. We build upon the Design Functions presented in Section 7.4.3 to show their use in implementing the Design Functions discussed from a theoretical perspective in Section 5.6.

We note that many of these Design Functions share the same logic as those shown in the House Example in Section 7.4.3. This illustrates how the same Design Functions can be used to create the appropriate Network Views for different input topologies, using the Network Whiteboard abstraction.

For each Design Function, we show a visualisation of the resulting Network View using our automated visualisation system, which we discussed in Section 7.2.3.

Physical Network View

The *physical* Design Function selects the nodes, edges, and interfaces from the *input* overlay. Here we assume that all links are physical. If other link types were used, then a filtering step could be used to filter only the physical links from the *input* overlay.

The Design Function for the *physical* overlay is shown in Listing H.5, with the visualisation shown in Figure 8.4. As the topology only contains physical devices and links, the *physical* overlay visualisation shown in Figure 8.4 looks the same as the *input* overlay visualisation shown in Figure 8.2.

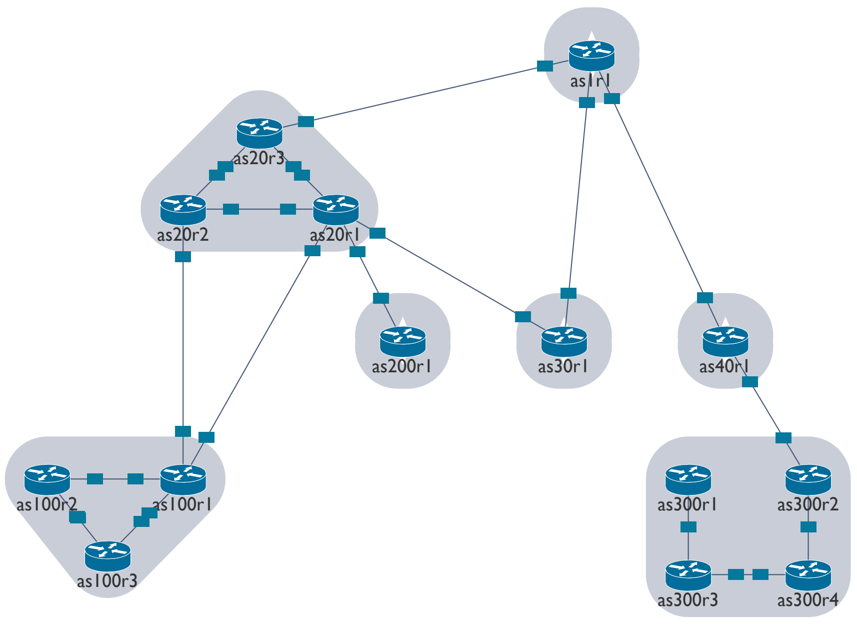


Figure 8.4: Visualisation of Physical Network View

Layer 2 Network View

The *Layer 2* Network View is shown in Figure 8.5. The Design Function used to construct it is shown in Listing H.6. These are the same *Layer 2* Design Functions as shown in Section 7.4.3 and used in the House Example in Chapter 7.

IP Addressing Network View

Like the previous Design Functions, the *IP Addressing* Design Function uses the same logic as in the House Example shown in Figure 7.4.3. The infrastructure addresses are shown in Figure 8.6, and loopbacks in Figure 8.7. The code for this Design Functions is shown in Listing H.8

Layer 2 Connectivity Network View

The Layer 2 Connectivity Network View is shown in Figure 8.8. The Design Function used to construct it is shown in Listing H.7, and also is identical to that used in the House Example shown in Section 7.4.3. This illustrates the power of decoupling Design Functions from the

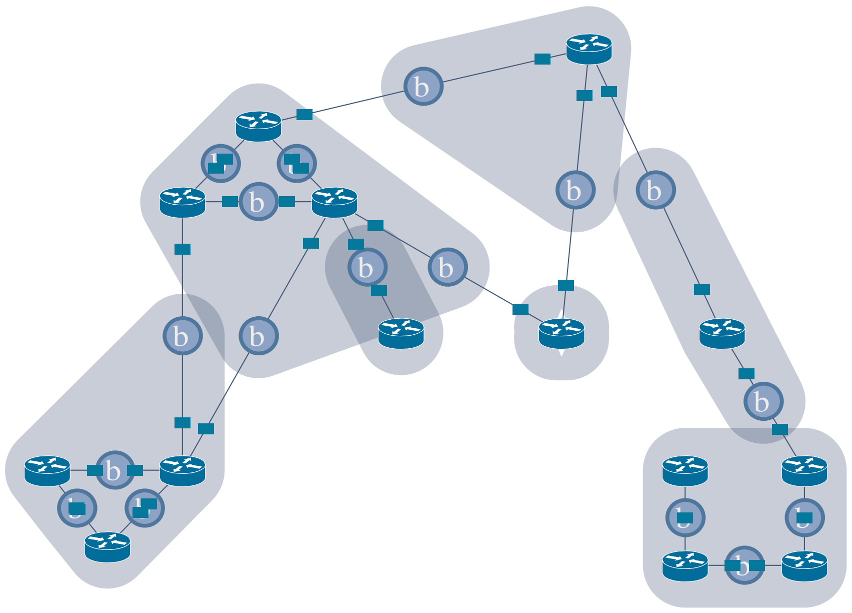


Figure 8.5: Visualisation of Layer 2 Network View

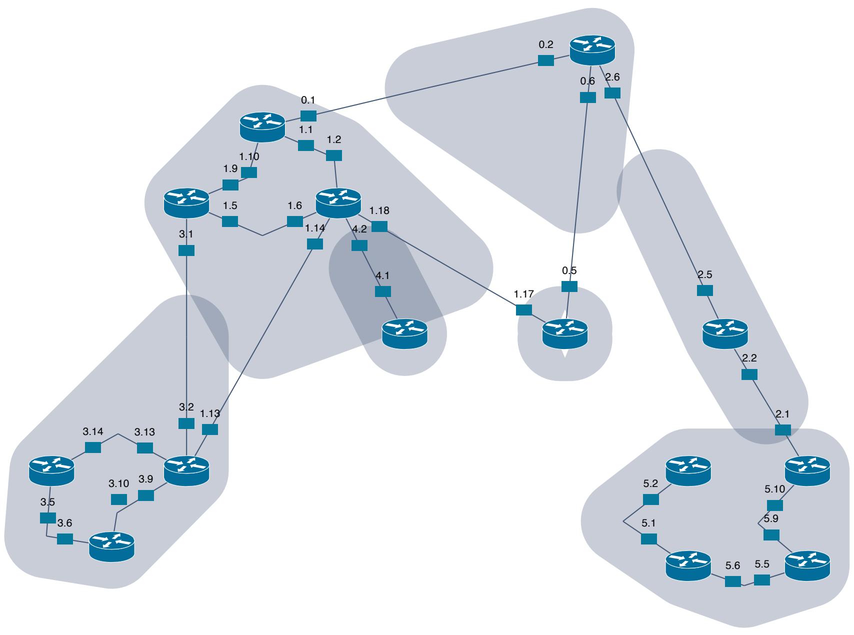


Figure 8.6: Visualisation of IP Network View. Interface labels show IP addresses. For visual clarity, broadcast domains have been removed, links manually re-aligned, and only the 3rd and 4th octets of 192.168.x.y are shown.

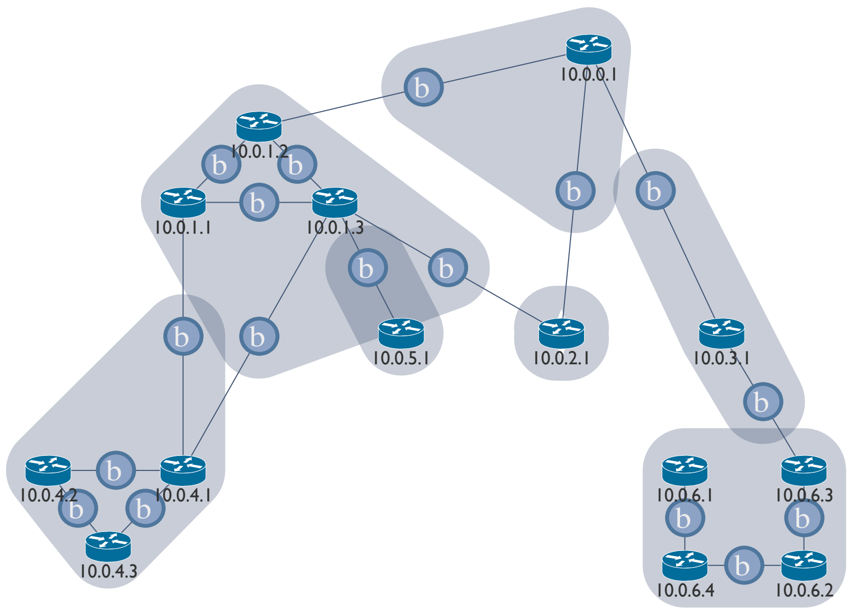


Figure 8.7: Visualisation of IP Network View. Node labels show loopback IP Addresses.

topology specification, in that the same Design Functions can be used for different *input* topologies, express using the Network Whiteboard abstraction.

OSPF Network View

The *OSPF* Design Function is shown in Listing H.9, and for the advertisement of network prefixes in Listing H.10. This again uses the same logic as Section 7.4.3. The Visualisation of this Design Function is shown in Figure 8.9.

iBGP Network View

The *iBGP* Network View is constructed using the code shown in Listing H.11. This Design Function follows the same approach of that shown in Section 7.4.3. The result of applying this Design Function to the Small Internet topology is shown in Figure 8.10.

eBGP Network View

The final Design Function is for the *eBGP* Network View. The code to create this Network View is shown in Listing H.12, and the code to advertise the relevant network prefixes over eBGP is shown in Listing H.13. Like the other Design Functions, this follows the same logic as Section 7.4.3. The result of this Design Function is shown in Figure 8.11.

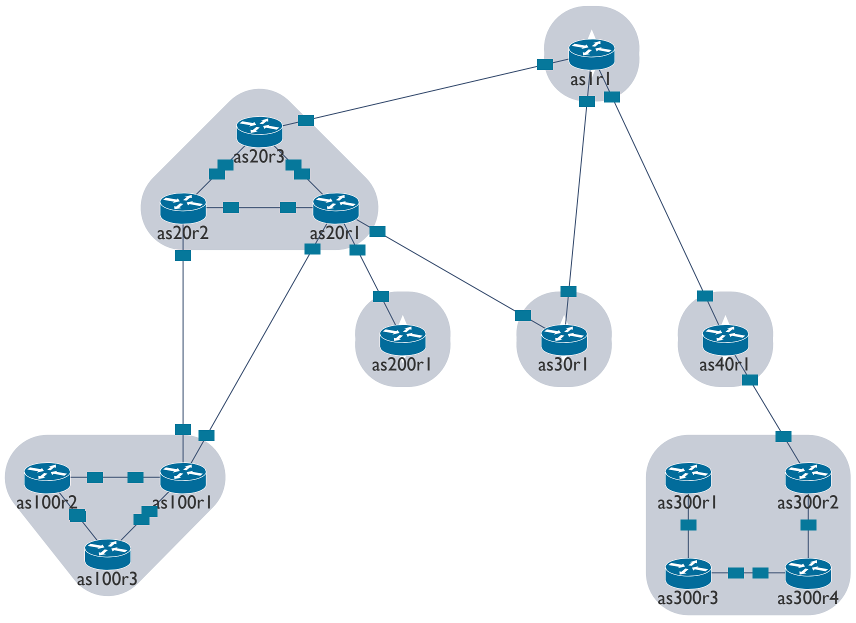


Figure 8.8: Visualisation of Layer 2 Connectivity Network View

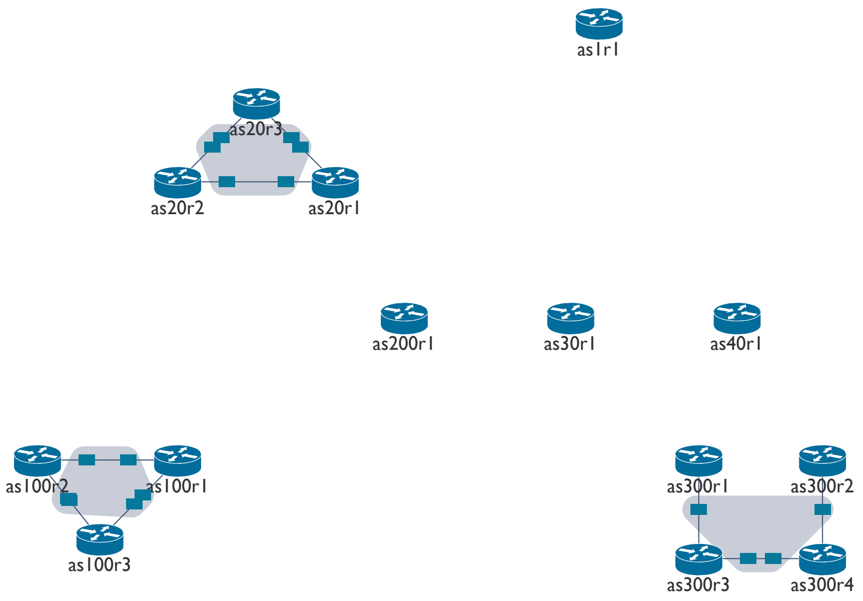


Figure 8.9: Visualisation of OSPF Network View. OSPF grouping shows interface areas, within an ASN.

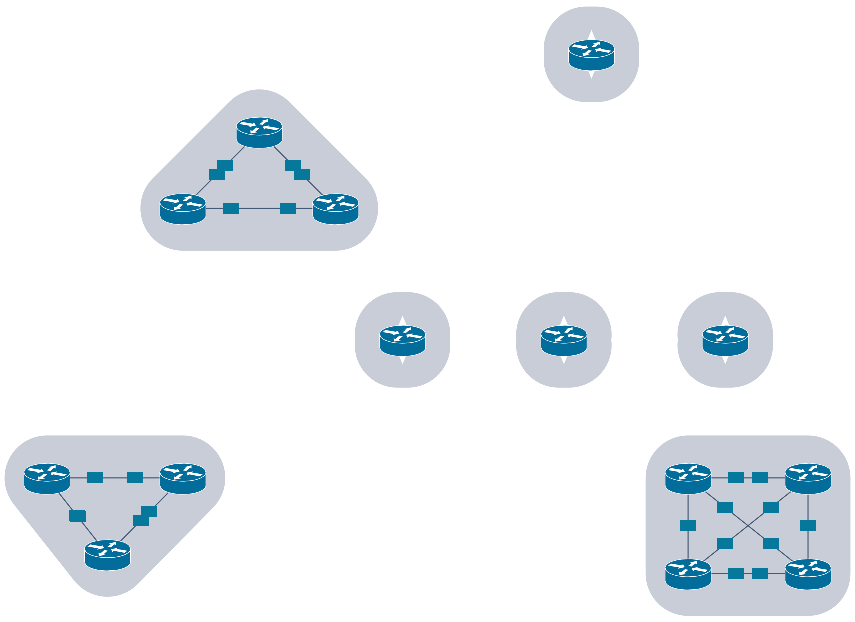


Figure 8.10: Visualisation of iBGP Network View

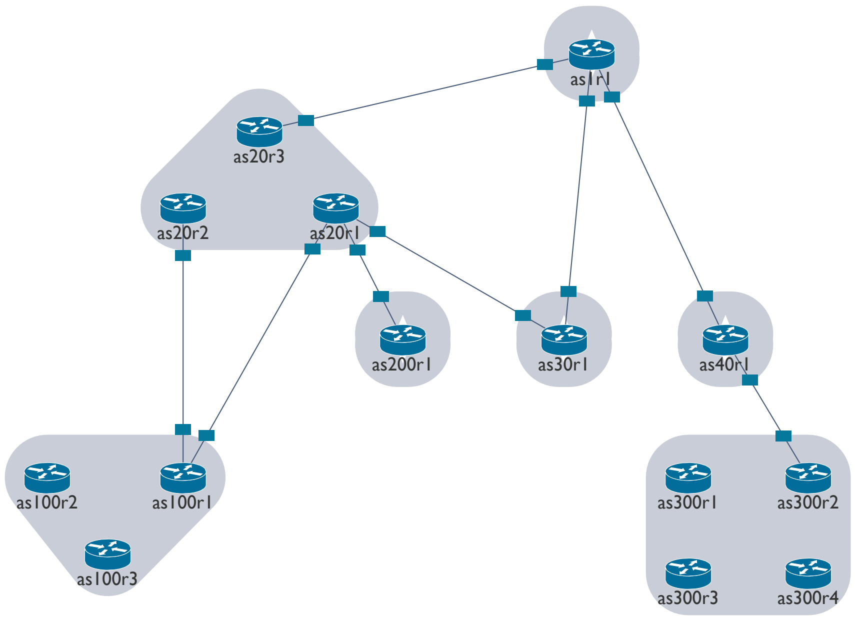


Figure 8.11: Visualisation of eBGP Network View

Conclusion

In this section we have shown how the theoretical Design Functions described in Section 5.6 can be used to construct the Network Views shown in Section 4.5.

Further, we have shown how the Network Whiteboard, Network View, and Design Function abstractions have allowed us to re-use many of the Design Functions described in the House Example in Section 7.4 to construct the appropriate Network Views for a different *input* topology, expressed using a different Network Whiteboard.

In the Section 8.4, we will show how these Design Functions can be extended to realise the full requirements of the Netkit Small Internet Lab, by relaxing the simplifying assumptions made in Section 4.5.1.

We now show how the Abstract Network Model constructed using the Design Functions in this section can be used to compile Quagga configurations for the Netkit platform, and provide a series of results that validate the correctness of the generated configurations.

8.3.5 *Compilation*

Netkit Platform Compiler

The Netkit Platform Compiler is shown in Listing H.14, and follows the same logic as the Netkit Platform Compiler discussed in the House Example in Section 7.4.4. This shows how the platform compilers can be re-used for different input topologies.

Quagga Device Compiler

The Quagga Device Compiler shown in Listing H.15 also follows the same logic as the Quagga Device Compiler discussed in the House Example in Section 7.4.4.

8.3.6 *Simulation Results*

We now discuss the simulation results collected from running the lab in Netkit.

The routing table of *as1r1* shown in Listing 8.1, and *as20r1* shown in Listing H.17 show their IGP and BGP routes.

```
as1r1# sh ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

C>* 127.0.0.0/8 is directly connected, lo
C>* 172.16.0.0/16 is directly connected, eth3
O   172.16.0.1/32 [110/10] is directly connected, lo, 00:03:13
```

```
C>* 172.16.0.1/32 is directly connected, lo
B>* 172.16.1.0/24 [20/0] via 192.168.0.1, eth0, 00:02:49
B>* 172.16.2.0/24 [20/0] via 192.168.0.5, eth1, 00:02:21
B>* 172.16.3.0/24 [20/0] via 192.168.2.5, eth2, 00:02:21
B>* 172.16.4.0/24 [20/0] via 192.168.0.1, eth0, 00:02:19
B>* 172.16.5.0/24 [20/0] via 192.168.0.1, eth0, 00:02:19
B>* 172.16.6.0/24 [20/0] via 192.168.2.5, eth2, 00:01:51
C>* 192.168.0.0/30 is directly connected, eth0
C>* 192.168.0.4/30 is directly connected, eth1
B>* 192.168.1.0/24 [20/0] via 192.168.0.1, eth0, 00:02:49
B>* 192.168.2.0/24 [20/0] via 192.168.2.5, eth2, 00:02:21
C>* 192.168.2.4/30 is directly connected, eth2
B>* 192.168.3.0/24 [20/0] via 192.168.0.1, eth0, 00:02:19
B>* 192.168.4.0/24 [20/0] via 192.168.0.1, eth0, 00:02:19
B>* 192.168.5.0/24 [20/0] via 192.168.2.5, eth2, 00:01:51
```

Listing 8.1: *show ip route* output for *as1r1*

The BGP peers of *as1r1* shown in Listing H.18 shows eBGP and iBGP peers. The traceroute from *as300r1* to *as100r2* is shown in Listing 8.2, and shows the path through multiple ASes over eBGP and OSPF to the destination Listing H.20. A processed result is shown in Listing H.20, with the plotted path shown in Figure 8.12.

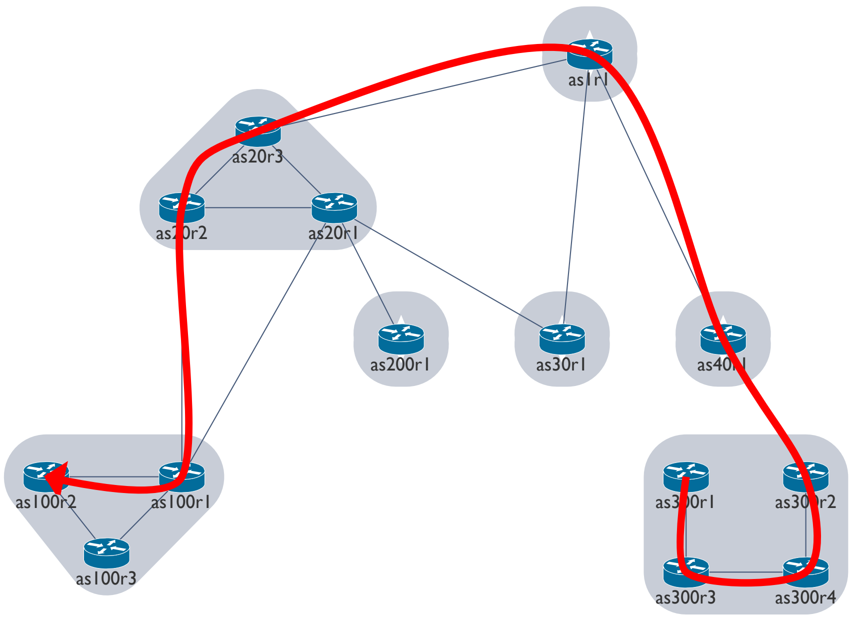


Figure 8.12: TRACEROUTE from *as300r1* to *as100r2*.

```
as300r1:~# traceroute 10.0.4.2
traceroute to 10.0.4.2 (10.0.4.2), 64 hops max, 40 byte packets
 1  192.168.5.1  (192.168.5.1)   0 ms  0 ms  0 ms
 2  192.168.5.5  (192.168.5.5)   1 ms  0 ms  0 ms
 3  192.168.5.10 (192.168.5.10)  0 ms  1 ms  0 ms
 4  192.168.2.2  (192.168.2.2)   1 ms  1 ms  1 ms
 5  192.168.2.6  (192.168.2.6)   1 ms  1 ms  1 ms
 6  192.168.0.1  (192.168.0.1)   1 ms  1 ms  1 ms
 7  192.168.1.9  (192.168.1.9)   1 ms  1 ms  1 ms
 8  192.168.1.13 (192.168.1.13)  1 ms  1 ms  1 ms
 9  10.0.4.2     (10.0.4.2)      1 ms  1 ms  1 ms
```

Listing 8.2: Output of TRACEROUTE from *as300r1* to *as100r2*

8.3.7 *Conclusion*

In this section we have shown how we can use our system to reproduce the example that was described in the theoretical Chapter 4, Chapter 5, and Chapter 6. We have shown how the system can be used to launch and measure valid configurations for a larger scale topology, with multiple autonomous systems interconnected. The measurement results confirm that the network is configured correctly. We note that this approach is scalable, and that adding new Network Elements, new Network Element Connections, or changing IP address pools can be readily handled, with the resulting configurations generated automatically.

In the next case study we relax the simplifying assumptions made in Chapter 4, in order to accurately reproduce the Small Internet topology described in Di Battista *et al.* [25].

8.4 complete netkit small-internet lab

8.4.1 *Introduction*

In this case study we build upon the simplified small Internet lab described in the previous case study, by relaxing our simplifying assumptions made in Section 4.5.1. This allows us to reproduce the topology described in Di Battista *et al.* [25]. We use this to demonstrate the flexibility of our approach in a more realistic scenario involving specific IP addresses, route redistribution, and BGP policies.

We begin by describing the simplifying assumptions that we relax. We then explain the Network Whiteboard that is used to describe the network topology and policy to be built, and provide an overview of each of the Design Functions. These Design Functions build on the previous case study, so we focus on the additions that enhance functionality in order to realise the complete small Internet example. We then explain the enhancements required in the compilation and template process. Finally, we launch the resulting configurations, and show the outputs of both the traceroute command, and the routing protocols to demonstrate a correctly configured network. The physical topology and IP addressing is shown in Figure B.2, and the BGP policy used is shown in Figure B.1.

8.4.2 *Relaxation of Simplifications*

In this Case Study we relax the assumptions we made in Section 4.5.1. We first list the assumptions made, and then discuss how we will remove each simplifying assumption. The assumptions we made are as follows:

1. There are no end-host subnets.
2. Autonomous systems are not multi-homed.
3. IP addresses are automatically allocated.
4. OSPF is used at the IGP within each autonomous system.
5. All iBGP peerings are full-mesh, and routes are not redistributed into the IGP.
6. There is no eBGP policy.
7. All links are point-to-point.

We relax these simplifications as follows:

1. We introduce the use of a *server* device type to represent end-host subnets. As shown in Figure 8.14, each of the autonomous systems *20*, *30*, *40*, *100*, *200*, and *300* have one or more server nodes representing end-host subnets. These have the relevant prefix, such as *20.1.1.0/24* or *200.2.0.0/16*, specified on their physical interface.
2. We add the eBGP peering links from *as100r1* to *as20r1*, and from *as300r1* to *as20r1*.
3. We manually specify the IP addresses on the Network Whiteboard, as shown in Listing 8.4.
4. We use RIP as the routing protocol in the autonomous systems, and create the appropriate Design Function in Figure 8.4.5.
5. We specify the appropriate iBGP or route-redistribution approach on the Network Whiteboard, as shown in Listing 8.5. These are then handled in the Design Functions in 8.4.5, Figure 8.4.5, and Figure 8.4.5.
6. eBGP policies are specified on the Network Whiteboard and Listing 8.5. These can be seen in Figure 8.16. The application of these policies is detailed in the eBGP Design Functions of Figure 8.4.5.
7. We introduce unmanaged layer 2 *switch* device types into autonomous systems *20* and *300*, as shown in Figure 8.14.

With the removal of these simplifications, we will now demonstrate how our approach can be used to produce a replication of the Netkit BGP lab Small Internet described in Di Battista *et al.* [25], through the Network Whiteboard high-level specification.

8.4.3 *Methodology*

The methodology used to implement and verify this case study is shown in Figure 8.13. We follow the same general approach as the methodology for the House example and Simplified Small Internet case study, but make a number of variations.

The Network Whiteboard is specified in GraphML format, and contains additional information to represent the switches and end-hosts in the network, as well as IP address information and BGP primary and backup links. We then programmatically augment this with information on BGP policy relationships. The Layer 2 Design Function is extended to take into consideration the switches in AS20 and AS300, and the end-host devices in numerous autonomous systems. The IP Address Design Function is modified to use the addresses specified on the Network Whiteboard rather than an automated allocation algorithm. The OSPF Design Function is replaced with the RIP Design Function, and takes into consideration route redistribution. The iBGP Design Function is extended to consider the redistribution policies specified on the Network Whiteboard. Finally, the eBGP Design Function is extended to allow for BGP policy to be expressed.

We extend the Quagga Device Compiler and Quagga Templates to allow for the RIP routing protocol, and BGP policy. The remainder of the process such as the Netkit Platform Compiler and the collection commands run are as per previous methodologies. We use these collection commands to verify that the configurations generated match the Netkit Small Internet Lab described by Battista *et al.* [25].

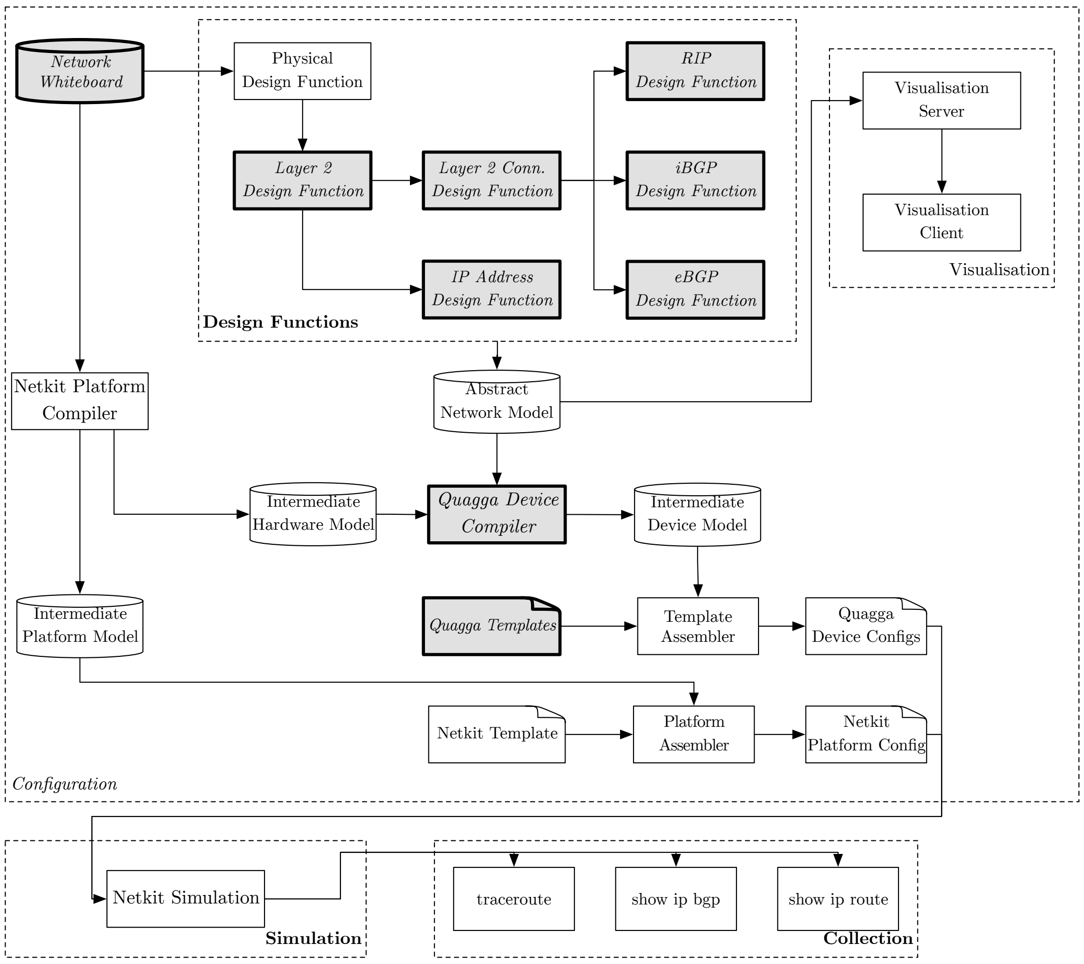


Figure 8.13: Configuration Generation and Verification methodology for Complete Small Internet Case Study

8.4.4 *Network Whiteboard*

The Network Whiteboard for this topology is shown in Figure 8.14. We first create an Abstract Network Model, as shown in Listing H.21.

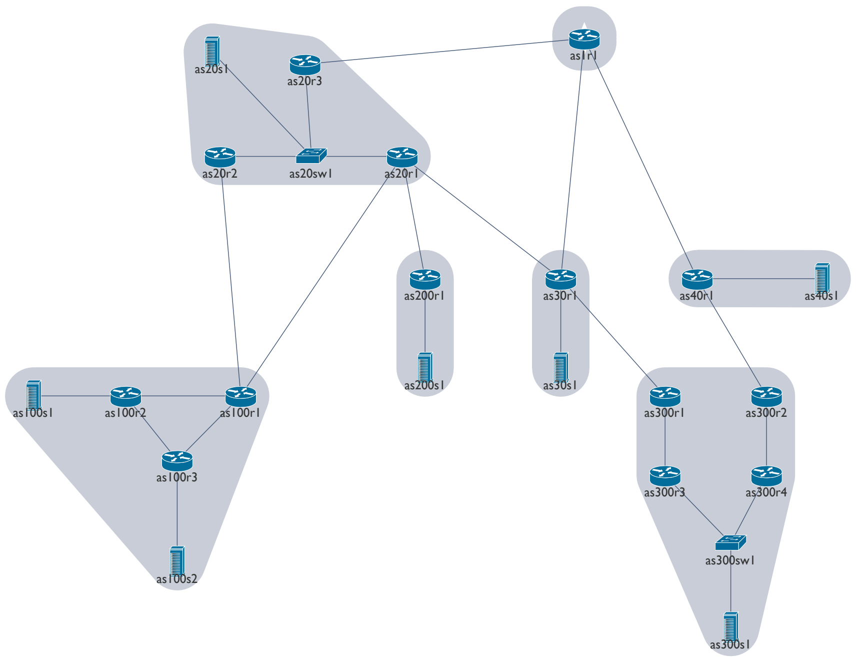


Figure 8.14: Visualisation of Network Whiteboard

Our Network Whiteboard is based on a GraphML file constructed using the yEd editor, as shown in Figure 8.15. In this Network Whiteboard, we have three types of network devices, denoted using the *device_type* attribute. These are *router*, *server*, and *switch*. In this case study, *switch* nodes represent basic unmanaged switches, and *server* nodes represent end hosts. The edges between the nodes are used to carry information regarding IP Addressing, and where applicable, BGP policy. We thus have used the Network Whiteboard abstraction to capture the high-level aspect of network configuration policy.

As shown in the code of Listing H.22, we first load this GraphML file. We then post-process the nodes by updating their attributes. We form a list of both the nodes and the edges, which we then add in a later step, covered below. The raw dictionary form of the post-processed nodes and their associated attribute data is shown in Listing 8.3, and the raw dictionary form of the edges and their associated attribute data is shown in Listing 8.4.

```
{
 "as1r1": {"asn": 1, "device_type": "router", "x": 519, "y": -209},
 "as100r1": {"asn": 100, "device_type": "router", "x": 38, "y": 291},
 "as100r2": {"asn": 100, "device_type": "router", "x": -123, "y": 291},
```

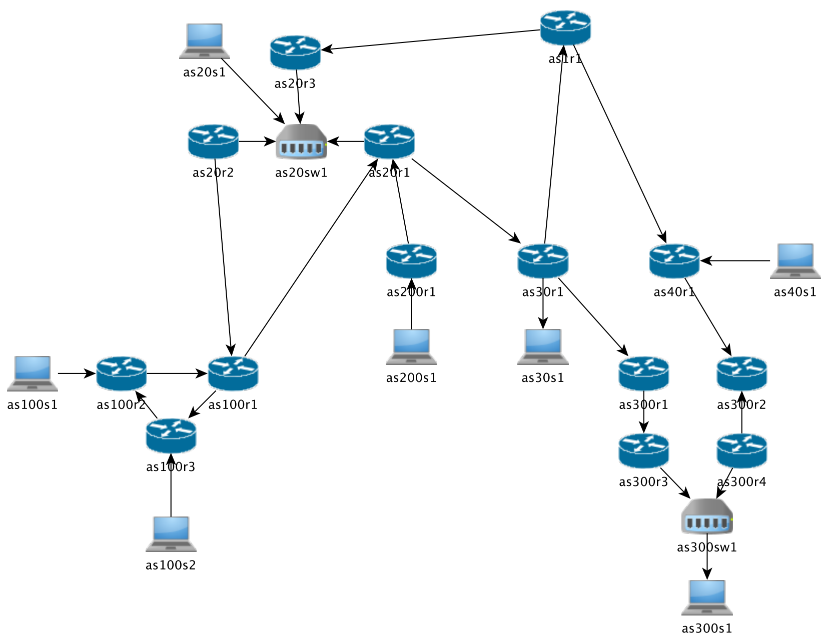


Figure 8.15: Drawing of Network Whiteboard using yEd editor.

```
"as100r3": {"asn": 100, "device_type": "router", "x": -51, "y": 381},
"as100s1": {"asn": 100, "device_type": "server", "x": -252, "y": 291},
"as100s2": {"asn": 100, "device_type": "server", "x": -51, "y": 523},
"as20r1": {"asn": 20, "device_type": "router", "x": 264, "y": -44},
"as20r2": {"asn": 20, "device_type": "router", "x": 9, "y": -44},
"as20r3": {"asn": 20, "device_type": "router", "x": 128, "y": -173},
"as20s1": {"asn": 20, "device_type": "server", "x": -2, "y": -190},
"as20sw1": {"asn": 20, "device_type": "switch", "x": 137, "y": -44},
"as200r1": {"asn": 200, "device_type": "router", "x": 296, "y": 127},
"as200s1": {"asn": 200, "device_type": "server", "x": 296, "y": 252},
"as30r1": {"asn": 30, "device_type": "router", "x": 486, "y": 127},
"as30s1": {"asn": 30, "device_type": "server", "x": 486, "y": 252},
"as300r1": {"asn": 300, "device_type": "router", "x": 632, "y": 291},
"as300r2": {"asn": 300, "device_type": "router", "x": 774, "y": 291},
"as300r3": {"asn": 300, "device_type": "router", "x": 632, "y": 402},
"as300r4": {"asn": 300, "device_type": "router", "x": 774, "y": 402},
"as300s1": {"asn": 300, "device_type": "server", "x": 724, "y": 615},
"as300sw1": {"asn": 300, "device_type": "switch", "x": 724, "y": 497},
"as40r1": {"asn": 40, "device_type": "router", "x": 677, "y": 127},
"as40s1": {"asn": 40, "device_type": "server", "x": 852, "y": 127}
}
```

Listing 8.3: List of nodes with their attribute data

```
[
  ("as1r1", "as20r3", "eth2", "eth0", "11.0.0.20/30", 22, 21, None),
  ("as1r1", "as40r1", "eth0", "eth2", "11.0.0.28/30", 30, 29, None),
  ("as100r1", "as100r3", "eth2", "eth0", "100.1.0.0/30", 1, 2, None),
  ("as100r1", "as20r1", "eth1", "eth1", "11.0.0.4/30", 5, 6, "backup"),
  ("as100r2", "as100r1", "eth0", "eth3", "100.1.0.4/30", 6, 5, None),
  ("as100r3", "as100r2", "eth1", "eth1", "100.1.0.8/30", 10, 9, None),
  ("as100s1", "as100r2", None, "eth2", "100.1.2.0/24", None, 1, None),
  ("as100s2", "as100r3", None, "eth2", "100.1.3.0/24", None, 1, None),
  ("as20r1", "as20sw1", "eth2", None, "20.1.1.0/24", 1, None, None),
  ("as20r1", "as30r1", "eth3", "eth3", "11.0.0.16/30", 17, 18, None),
  ("as20r2", "as100r1", "eth0", "eth0", "11.0.0.0/30", 2, 1, "primary"),
  ("as20r2", "as20sw1", "eth1", None, "20.1.1.0/24", 2, None, None),
  ("as20r3", "as20sw1", "eth1", None, "20.1.1.0/24", 3, None, None),
  ("as20s1", "as20sw1", None, None, None, None, None, None),
  ("as200r1", "as20r1", "eth0", "eth0", "11.0.0.32/30", 33, 34, None),
  ("as200s1", "as200r1", None, "eth1", "200.2.0.0/16", None, 1, None),
  ("as30r1", "as1r1", "eth2", "eth1", "11.0.0.24/30", 25, 26, None),
  ("as30r1", "as300r1", "eth0", "eth0", "11.0.0.8/30", 10, 9, None),
  ("as30r1", "as30s1", "eth1", None, "30.3.3.0/24", 1, None, None),
  ("as300r1", "as300r3", "eth1", "eth1", "200.1.0.0/18", 1, 2, None),
  ("as300r3", "as300sw1", "eth0", None, "200.1.128.0/17", 1, None, None),
  ("as300r4", "as300r2", "eth0", "eth1", "200.1.64/18", 2, 1, None),
  ("as300r4", "as300sw1", "eth1", None, "200.1.128.0/17", 2, None, None),
  ("as300sw1", "as300s1", None, None, None, None, None, None),
  ("as40r1", "as300r2", "eth0", "eth0", "11.0.0.12/30", 14, 13, None),
  ("as40s1", "as40r1", None, "eth1", "40.4.4.0/24", None, 1, None),
]
```

Listing 8.4: List of edges with their attribute data, of form *(src_id, dst_id, src_int_ip, dst_int_ip, subnet, src_ip, dst_ip, bgp_policy)*

We also provide supplementary data describing the commercial relationships between each Autonomous System, to capture the BGP policy relationships shown in Figure B.1. This provides a practical demonstration of the theoretical approach to capturing BGP policy described in Chapter 5. We also capture the redistribution policy used within the Autonomous Systems. These pieces of information are captured as dictionaries, where the dictionary key corresponds to the *asn* label of the node. This is shown in Listing 8.5. A visualisation of the *input* overlay with the nodes annotated with their BGP policy roles is shown in Figure 8.16.

```
bgp_roles = {
    1:   "b",
    20:  "p",
    30:  "p",
    40:  "p",
    100: "c",
    200: "c",
    300: "c"
}

bgp_redist_policy = {
    20:  "ibgp",
    100: "redistribute_igp",
    300: "redistribute_igp",
}

bgp_load_share_policy = {
    300: True
}
```

Listing 8.5: Specification of BGP Roles

We note that the policy role attributes could also be stored in the dictionary, and used directly in the BGP policy creation. This would also be a valid approach, and would avoid duplication of information. We apply the policy attribute to each node in the Autonomous System for simplicity of explaining the Design Functions.

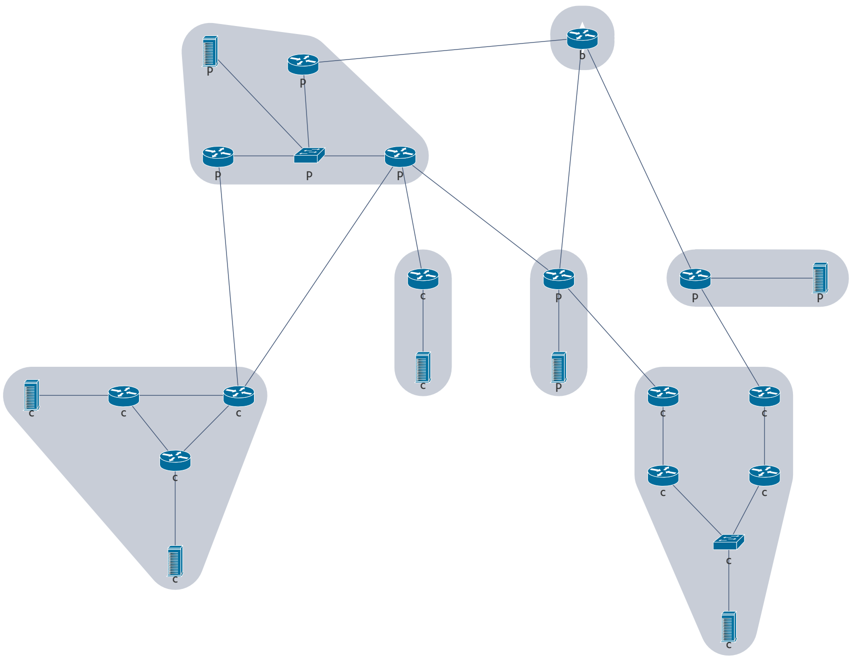


Figure 8.16: Visualisation of Network Whiteboard showing BGP Roles

We then add the nodes to the Network Whiteboard. This is shown in Listing H.23. In this step, we first add the nodes with the properties shown in the raw dictionary in Listing 8.3. We mark each router with a *simulate* attribute to be *true*, to allow us to filter out only the nodes which we wish to simulate on the Netkit platform. For this example we do not simulate the unmanaged switches (as these are represented by a shared virtual wire in Netkit), or the servers (which act as placeholders for end-host subnets). We also map the *bgp_role* previously discussed for the node, based on the *asn* attribute of the node. Finally, we map the redistribution policy previously described. If the redistribution policy for an *asn* is *ibgp* then we also mark the *ibgp* attribute of the node to bet *true*. This is used in the *ibgp* Design Function. If the redistribution policy is not *ibgp* then we set the *redistribute_bgp_to_igp* to *true*, to be used for routing protocol redistribution.

The next step is to add the edges. This is done in Listing H.24, where we iterate over each edge in the edges dictionary. In this dictionary, each edge is a tuple of the form *(src_id, dst_id, src_int_id, dst_int_id, subnet, src_ip, dst_ip, bgp_policy)*. We first map the *src_id* and *dst_id* into the source and destination node objects. We then respectively add the interface for the *src_int_id* and *dst_int_id*. We use directed edges in the GraphML file, so the *src* and *dst* correspond to the source and destination node and interface of the directed edge.

This allows us to use GraphML and associated tools such as yEd to easily create the Network Whiteboard for this case study.

The next step is to process the IP Address from the *subnet* which is of the form *a.b.c.d/x* where *x* is the prefix length. For simplicity, we express only the last octet of the IP Address in the *src_ip* and *dst_ip* attributes in the GraphML input. For instance a *src_ip* of *1* and a *dst_ip* of *2* for a subnet of *192.168.0.0./24* would correspond to *192.168.0.1* and *192.168.0.2*. We perform this transformation to form the relevant *ip* and *subnet* attributes on the interfaces. Finally, we create the edge connecting the interfaces. If the *bgp_policy* attribute is set, then this is stored on the edge, for use in the eBGP Design Functions. We also define a utility function, NEIGH_AVE_XY , to return the sum of neighbour *x* and *y* co-ordinates for a given node. This is shown in Listing H.26, and is used in later Design Functions. The result of the *input* overlay Design Function is shown in Figure 8.14.

8.4.5 *Design Functions*

Physical Network View

The *physical* Network View is shown in Figure 8.17. The Design Function to create this overlay is shown in Listing H.25, as is similar to previous *physical* design functions, where the nodes and edges are added from the *input* overlay, with the *platform* and *syntax* attributes set to *netkit* and *quagga* respectively.

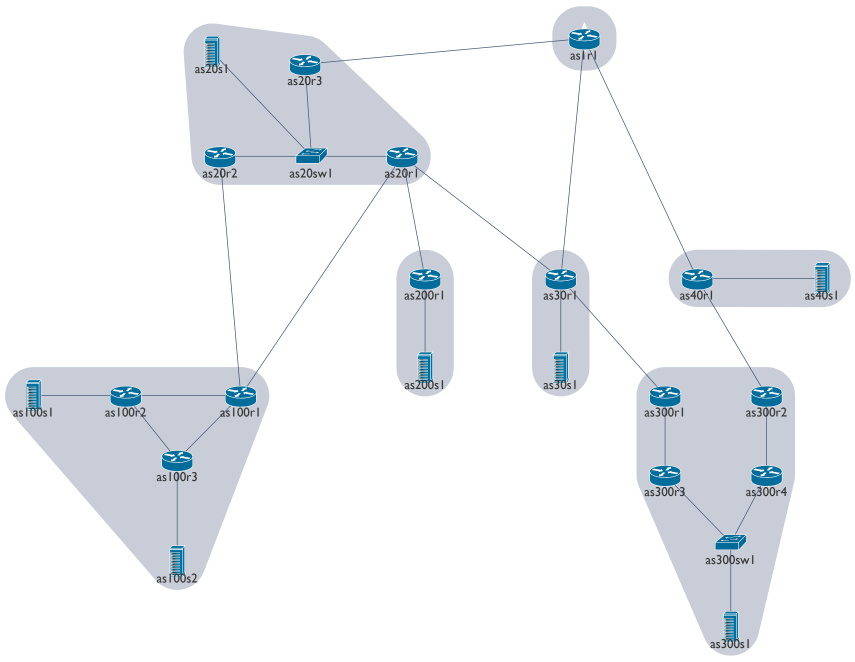


Figure 8.17: Visualisation of Physical Network View

Layer 2 Network View

The *layer 2* Network View is shown in Figure 8.18. We first create the *layer 2* Network View, by adding the nodes from the *physical* Network View. We then form the broadcast domains, which consists of two main steps: one to split point-to-point links between routers, and the other to form the broadcast domains that arise from the unmanaged switches.

The step to split the point-to-point links follows the same logic as in previous case studies. We use the neigh_ave_xy utility function to simplify the logic of averaging out the location of the broadcast domain pseudo-node. This step is performed on the edges which connect a *layer 3* device to another *layer 3* device, by using the edge filtering function. This ensures that we only split point-to-point links

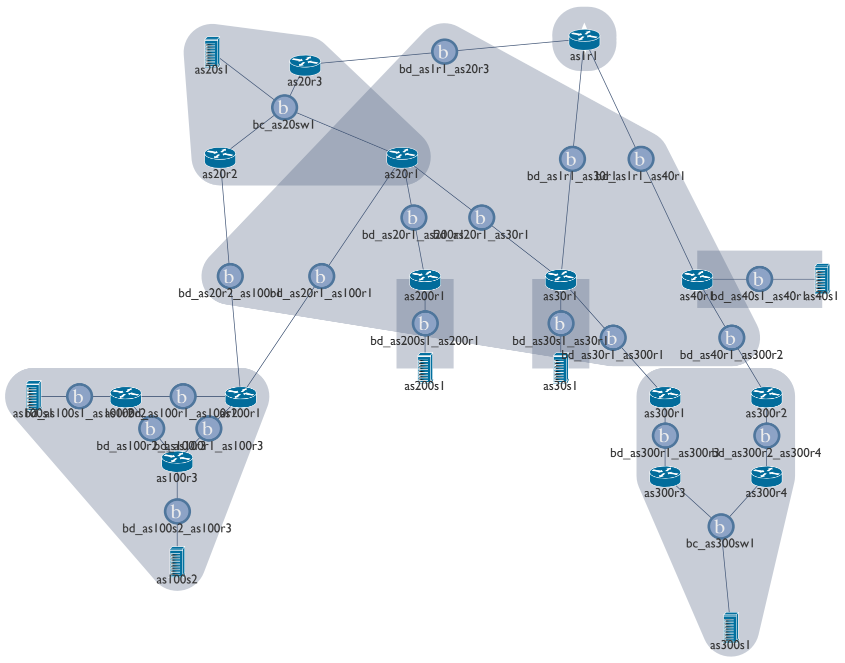


Figure 8.18: Visualisation of Layer 2 Network View

between *layer 3* devices, and therefore do not need to modify this logic to handle the unmanaged *layer 2* switches.

The second step differs to previous case studies, as we need to handle unmanaged switches. To do this, we create a *broadcast domain* pseudo-node representing the switch, create the appropriate interface on it, and connect it to the nodes connected to the switch. This step could be done in alternative ways, such as mapping the switch directly to a *broadcast domain* pseudo-node.

The code for this step is shown in Listing H.27, with the resulting *layer 2* overlay visualised in Figure 8.18.

IP Addressing Network View

The *IP Addressing* Network View differs from previous examples. Previously we have used automated address allocation algorithms to allocate loopback and infrastructure IP Addresses. In this case study, we wish to use the IP addresses specified on the topology as shown in Figure B.2. These addresses should then be advertised by the appropriate routing protocols. Where possible, we wish to present an IP Addressing Network View that is similar to as if it were constructed using automated allocation algorithms. This includes the interface IP address attributes, the subnet being stored on the broadcast domain pseudo-nodes, and the relevant address-block allocations being stored as dictionaries on the overlay itself. This allows us to re-use previous Design Functions for the routing protocols, and to decouple automated from manual allocation from the specification of the routing Design Functions.

The first step is to create the *ip* overlay in the Abstract Network Model. We add the nodes and the edges from the *layer 2* overlay, and then copy across the *interface* attribute for the *ip* and *subnet* from the *input* overlay. We then create two dictionaries, one to store the infrastructure IP address allocations, and one to store the end host IP allocations. These are *infra_allocations* and *host_allocations*. These allocations are stored separately as they are advertised differently in the routing protocols. We then use the GROUP_BY function to group the nodes by their *asn* attribute. We provide the list of *broadcast domain* pseudo-nodes as the second parameter to this function. This results in a series of *(asn, pseudo-nodes)* tuples to iterate over.

We then iterate over each *broadcast domain* pseudo-node within the *asn* grouping. Similar to in automated IP addressing, we then look at the neighbours of the broadcast domain. We iterate over each neighbour interface of the broadcast domain. For convenience, we map the *ip* attribute and *subnet* attribute to be a Netaddr [71] *IPAddress* and *IPNetwork* class, respectively. This simplifies IP addressing tests and aggregation, by allowing us to leverage the inbuilt functions of the Netaddr Python library.

For each neighbour interface, we store the *subnet* attribute in a temporary variable. We then set this attribute on the *broadcast domain* pseudo-node itself as the *subnet* variable. In this example we simply choose one of the *subnet* values. We could make this more rigorous by ensuring that each *subnet* attribute is identical for each neighbour interface of a broadcast domain, by following the verification functions we described in Section 9.3.3. Finally, if the *asn* attribute is not *None*, then we record the *subnet* value into the *infra_allocations* dictionary for the *asn* value to which the *broadcast domain* belongs. If the *broadcast domain* is connected to any servers, then we also record it into the *host_allocations* dictionary. The result of these dictionaries is shown in Listing 8.6 and Listing 8.7.

```
{
  20: ["20.1.1.0/24"],
  30: ["30.3.3.0/24"],
  40: ["40.4.4.0/24"],
  100: ["100.1.3.0/24", "100.1.0.8/30", "100.1.0.0/30",
       "100.1.0.4/30", "100.1.2.0/24"],
  200: ["200.2.0.0/16"],
  300: ["200.1.0.0/18", "200.1.64.0/18", "200.1.128.0/17"]
}
```

Listing 8.6: Small Internet Infrastructure Address Allocations

```
{
  20: ["20.1.1.0/24"],
  30: ["30.3.3.0/24"],
  40: ["40.4.4.0/24"],
  100: ["100.1.3.0/24", "100.1.2.0/24"],
  200: ["200.2.0.0/16"], 300: ["200.1.128.0/17"]
}
```

Listing 8.7: Small Internet Loopback Address Allocations

The result of this step is shown in Figure 8.19, where the *broadcast domain* pseudo-node labels show the *subnet* attribute, and the interface labels show the *ip* attribute. The code for this Design Function is shown in Listing H.29.

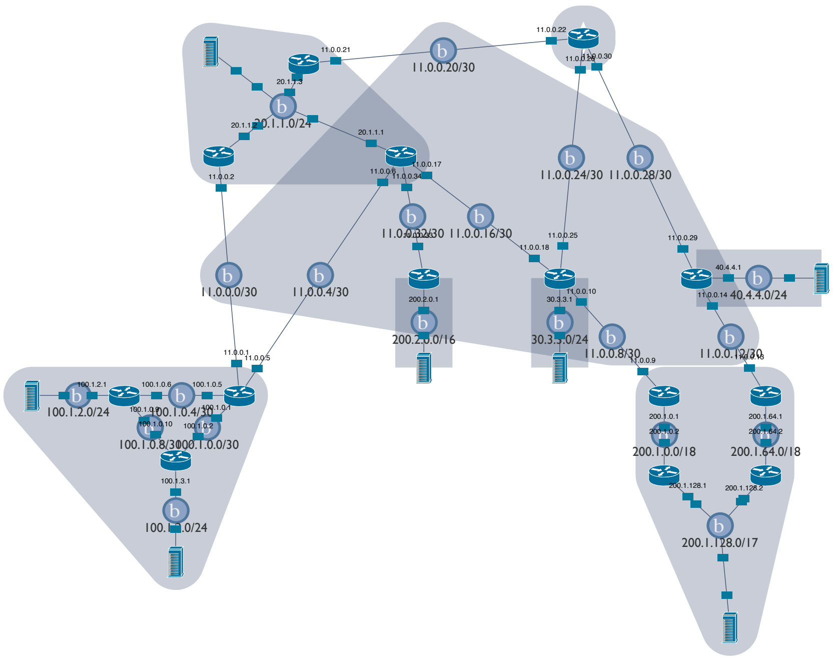


Figure 8.19: Visualisation of IP Addressing Network View. Node labels show the *subnet* attribute, and interface labels show the *ip* attribute

Layer 2 Connectivity Network View

The *Layer 2 Connectivity* Network View is constructed like in previous examples, by exploding the *broadcast domain* nodes. This is shown in Figure 8.20, with the Design Function code shown in Listing H.28.

RIP Network View

We now can construct the routing protocol Network Views. The Netkit Small Internet Lab shown in Di Battista *et al.* [25] uses the *RIP* (Routing Information Protocol) as the IGP within the autonomous

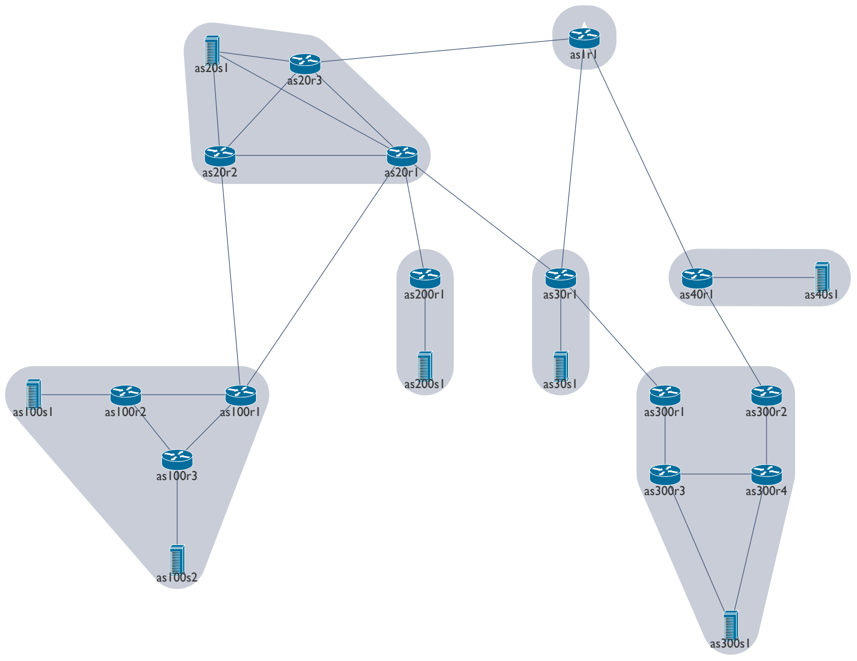


Figure 8.20: Visualisation of Layer 2 Connectivity Network View

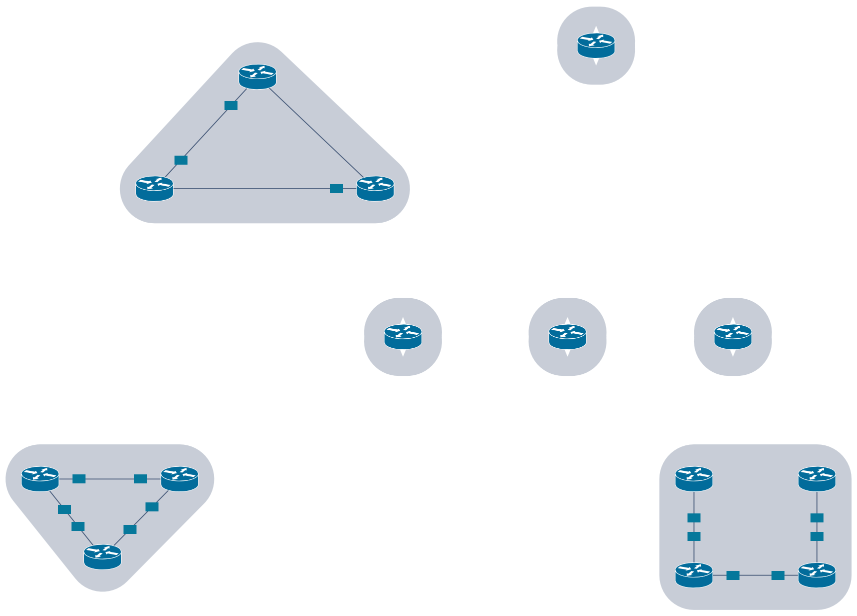


Figure 8.21: Visualisation of RIP Network View

systems. This differs to our previous examples which used the *OSPF* routing protocol. As we will see in this section, the Design Function for *RIP* is similar to *OSPF*, it is only in the compiler that we need to generate the different syntax. The abstraction of Network Views, where edges represent connectivity, allows the common patterns across protocols to be expressed.

The *rip* Network View is constructed by adding the *routers* from the *input* overlay (the *physical* overlay could also be used here), and the edges from the *layer 2 connectivity* overlay. This creates the basic connectivity for the overlay, as shown in Figure 8.21. The code for this step is shown in Listing H.30.

To reproduce this case study, we also need to consider the redistribution of routes, and the advertisement of relevant IP Address prefixes. We now discuss the Design Function steps to perform these tasks.

Through inspection of the router configurations of the lab in Di Battista *et al.* [25], we can infer that the policy is to use the *redistribute connected* command on a *router*, if that *router* connects to a *server*. This means that we will redistribute connected routes on routers which connect to end-host prefixes, as on our Network Whiteboard we denote end-host prefixes through the use of the *server* node.

The set of nodes for which the *redistribute connected* attribute is set to *true* is shown in Listing 8.8.

```
[as20r1, as20r2, as20r3, as100r2, as100r3, as300r3, as300r4]
```

Listing 8.8: List of nodes for which redistribute connected are set for RIP Network View

Finally, we set the attributes to advertise the relevant IP address prefixes for each router through the *RIP* protocol. The code to perform this is shown in Listing H.32.

iBGP Network View

The *ibgp* Network View is shown in Figure 8.22. This differs to previous examples, where we previously connected all nodes in an ASN in a full-mesh configuration. In this case study, we follow the iBGP topology policy inferred from the routing configurations of Di Battista *et al.* [25]. For this, only the routers in *asn20* are connected using iBGP. The other ASes use redistribution to share the routes.

This case study also differs to previous examples in that the iBGP sessions are bound to the physical interfaces, rather than the loopback interface.

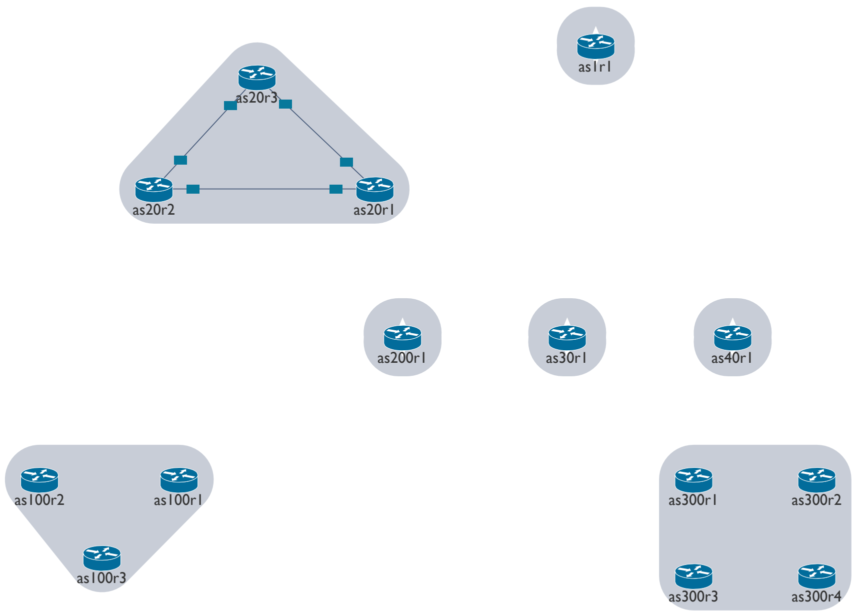


Figure 8.22: Visualisation of iBGP Network View

The code to show how these cases are handled is shown in Listing H.33. Most notably, we first filter only the nodes which have the *ibgp* attribute in the *input* overlay set to *true*, and we bind the iBGP session endpoint to the physical connection interface, rather than the loopback interface, as this example does not use loopback addressing, as shown in Figure B.2.

eBGP Network View

Note: we hard-code policies, such as which to apply for backbone, provider, customer, for prefix advertisement and prefix list/route maps, here as a first step. These could be expanded later to be defined externally, with the appropriate policy function then applied. This would generalise the policy mechanics further.

CREATE EBGP NETWORK VIEW The final Network View to create is the *ebgp* Network View. The Design Function for this Network View is more complicated than previous examples, as we need to incorporate BGP policy, and form the appropriate access lists and prefix lists to implement this policy.

The first step is to create the *ebgp* Network View, as shown in Listing H.34, and visualised in Figure 8.23. This logic follows that of previous case studies. It is in the mapping of policies that we expand on previous logic.

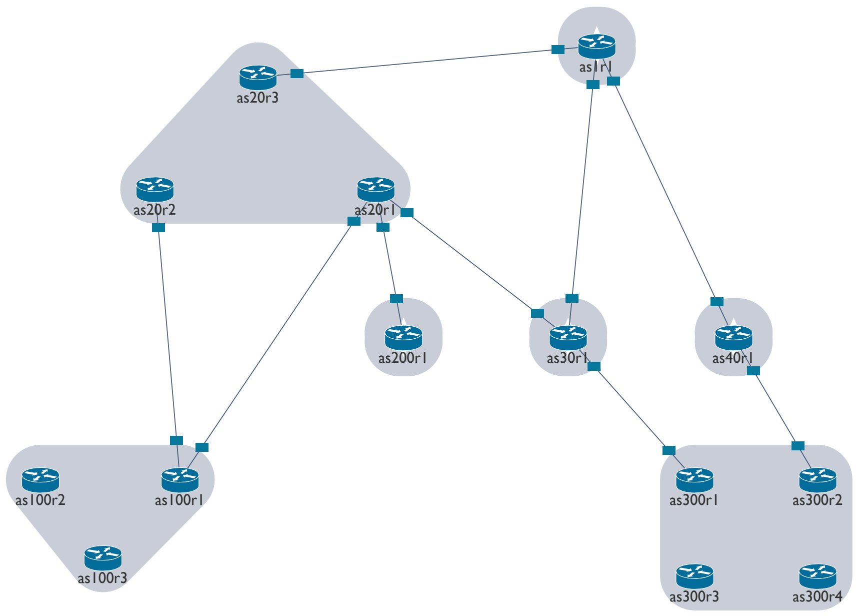


Figure 8.23: Visualisation of eBGP Network View

map policies The first step is to map the eBGP Policy Roles, as shown in Listing H.35. This function copies the *bgp_role* attribute for each node from the Network Whiteboard to the eBGP Network View. We also define a bgp_rank function, that orders the roles of *b*, *p* and *c* (*backbone*, *provider*, and *customer*) to 1, 2, and 3), to allow for comparisons in later functions.

ip address prefix advertisements The next step is to mark the prefixes to be advertised by eBGP. This code is shown in Listing H.36 and is more complex than in previous case studies. This applies the policy designated on the Network Whiteboard, and is based on the logic implied from the configurations and slides of Di Battista *et al.* [25]. We now examine each component of this step.

We first iterate over each node in the eBGP Network View, and get the infrastructure and loopback address block for the *asn* to which the node belongs. If the node has the *bgp_role* of *b* (is in a backbone network), then we also advertise the default route of *0.0.0.0/0*.

We then iterate over the eBGP sessions previously created in Listing H.34, and look at the *bgp_role* of each side of the session. We use the bgp_rank function to easily compare the roles. If the session is from a node that is lower in rank than the destination (such as backbone to provider, or provider to customer) then we advertise the prefixes of the node across the session. For instance we will advertise backbone routes to a provider, and provider routes to a customer. It can also occur that the rank is the same, such as for provider to provider routes (for example from *as20r1* to *as30r1*). In this case, we advertise the subnet of the interface of the node with the lower *asn* (in this example *as20r1* advertises the prefix of *11.0.0.16/30* to *as30r1*. This is also inferred from the policy design rules of the example in Di Battista *et al.* [25]. Additional logic could be handled through designations on the Network Whiteboard.

We then advertise the prefixes of the ASN for both loopback, and infrastructure, as per previous case studies. If the *bgp_role* of the node is a customer, then we aggregate the prefixes into a */16* address block, to be consistent with the example of Di Battista *et al.* [25]. Finally, we set the list of prefixes to be advertised onto the node, to be used in the compiler.

This case study also uses a form of load sharing policy for AS300. This is specified using the *bgp_load_share_policy* data structure in Listing 8.5. It could also be marked on the nodes in the Network Whiteboard, like the *bgp_role* attribute. If an AS is marked as load sharing, then we gather the eBGP nodes in that ASN, and find the common networks that the nodes advertise, based on the advertisement attribute described in the previous paragraph. For simplicity we assume that the prefixes are */16*, as aggregated in the previous step for the customer role autonomous systems. We then split this */16* into two */17* prefixes, and allocate each of these */17* prefixes to the eBGP routers, in addition to the */16* prefix. As the */17* prefixes are more specific than the */16* prefix, they will be preferred in the route selection process run on the routers.

A summary of the prefixes announced by each eBGP router is shown below in Listing 8.9.

```
as1r1 ["0.0.0.0/0", "11.0.0.20/30", "11.0.0.24/30", "11.0.0.28/30"]
as20r1 ["11.0.0.4/30", "11.0.0.16/30", "11.0.0.32/30", "20.1.1.0/24"]
as20r2 ["11.0.0.0/30", "20.1.1.0/24"]
as20r3 ["20.1.1.0/24"]
as30r1 ["11.0.0.8/30", "30.3.3.0/24"]
as40r1 ["11.0.0.12/30", "40.4.4.0/24"]
as100r1 ["100.1.0.0/16"]
as200r1 ["200.2.0.0/16"]
as300r1 ["200.1.0.0/16", "200.1.0.0/17"]
as300r2 ["200.1.0.0/16", "200.1.128.0/17"]
```

Listing 8.9: Prefixes announced by each eBGP router

The final step in this component is to map the prefixes advertised by the *customer* eBGP routers onto their peers, to be used in creating the prefix-lists on the *provider* eBGP router. For instance, on the *provider* router *as40r1*, there is a prefix lists to only accept the *customer* routes over the session from *as300r2*. To create this prefix-list on *as40r1* we need to map the customer prefixes onto the node, to then use in the compiler. A summary of these prefixes is shown below in Listing 8.10.

```
as20r1 {200:["200.2.0.0/16"], 100:["100.1.0.0/16"]}
as20r2 {100:["100.1.0.0/16"]}
as30r1 {300:["200.1.0.0/17", "200.1.0.0/16"]}
as40r1 {300:["200.1.128.0/17", "200.1.0.0/16"]}
```

Listing 8.10: Customer prefixes for each eBGP provider router

applying policies The next step is to apply the eBGP policies. This is shown in Listing H.37. For simplicity we first form sets of nodes for each of the backbone, provider, and customer nodes. This simplifies comparison logic. We then create a series of utility functions to append a policy, or route-map in both inbound and outbound directions, as well as to allow appending a route-map to default-originate.

We then iterate over each backbone node, and look at the eBGP peering sessions. If the session is to a provider, we set the *default_originate* attribute to *True*, and append the *acceptAny* inbound prefix-list, and the *defaultOut* outbound prefix-list. Next we iterate over the provider nodes. If the session is to another provider, we add the default originate route-map of *dontUseMe*, the outbound route-map of *dontUseMe*, and the outbound prefix-list of *defaultOut*. If the session is to a customer, then we set the *default_originate* attribute to *True*, and add the outbound prefix-list of *defaultOut*. We also specify the customer-specific prefix-list based on the ASN. For instance the session from *as40r1* to *as300r2* would have the *as300In* prefix-list. This prefix-list is created in a later step.

Finally, we iterate over the customer nodes. If the session is to a backbone node, we do nothing. If it is to a provider, then we look at the *bgp_policy* attribute specified on the edge. This is from the Network Whiteboard. In this example, if the *bgp_policy* is *backup*, then we append the outbound route-map of *metricOut*, and the inbound route-map of *localPrefIn*. This is applied on the link from the original specification in the Network Whiteboard shown in Listing 8.4, where the final value in the edge tuple indicates the *bgp_policy* attribute. The link from *as100r1* to *as20r1* is designated as the backup. The effect of the inbound and outbound routing policy is to make this route less appealing to the routing decision process, so that the alternative link via the primary path of *as100r1* to *as20r2* is preferred. This is confirmed in the trace routes shown in Section 8.4.8. A visualisation of the eBGP Network View, showing the default originate label marked on eBGP sessions, is shown in Figure 8.24.

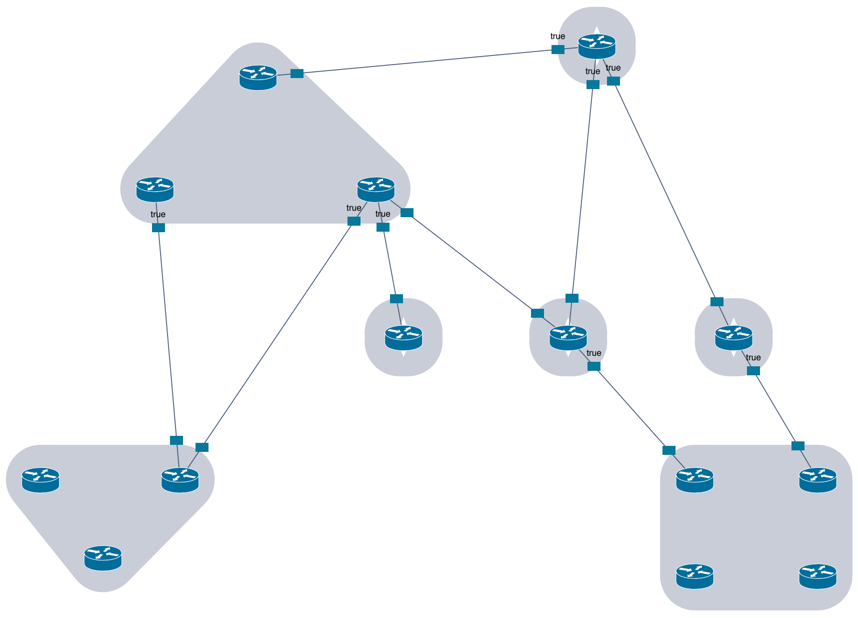


Figure 8.24: Visualisation of eBGP Network View, showing Default Originate label

CREATING PREFIX LISTS AND ROUTE MAPS The final step in the eBGP Design Function is to create the prefix lists and route-maps. The code to create the prefix lists is shown in Listing H.38. We first specify a library of the inbuilt prefix-lists, which are *defaultOut*, *defaultOk*, *defaultIn*, *defaultOut*, and *acceptAny*. We then iterate over each node in the eBGP Network View. We first check if the node has any *customer_asn_prefixes* defined, and store this result for later processing.

We then iterate over each eBGP session from the node, and look at the prefix-lists defined on the session. For each prefix-list that exists in the library discussed above, we apply the value from the library. If the prefix-list is *mineOutOnly*, we form the prefix-list values from the networks the node advertises over eBGP, from the *networks* attribute on the node. Finally, we examine the *customer_asn_prefixes* value. If there are any prefixes set, we create the appropriate prefix-list name, and set the list of customer prefixes as the value. A visualisation of the eBGP Network View showing the prefix lists on the eBGP sessions is shown in Figure 8.25.

We then create the route-maps. The code to create the route-maps is shown in Listing H.39. We again iterate over each eBGP node, and then iterate over the eBGP sessions from the node. We create a running list of the route-maps defined on each session, in order to define these route-maps globally on the node.

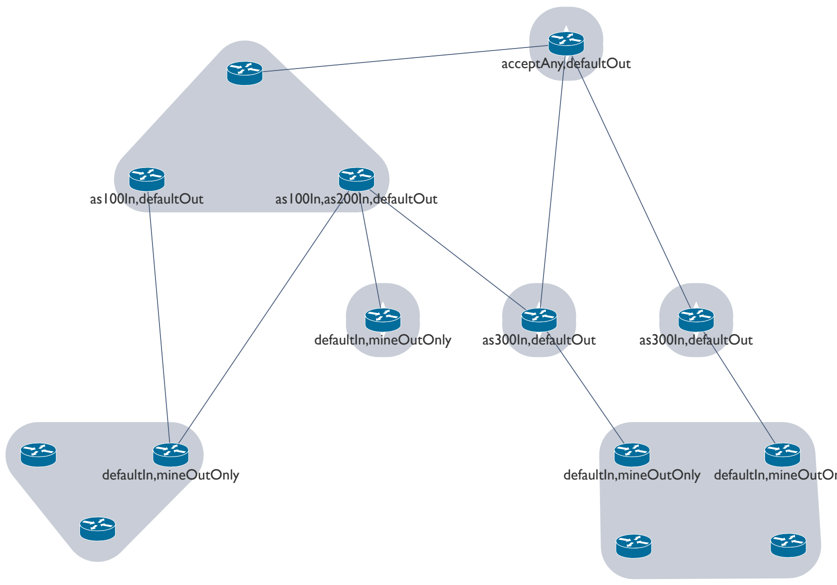


Figure 8.25: Visualisation of eBGP Network View, showing node Prefix Lists

We then iterate over the route-maps collected on the node. If the route-map is *dontUseMe*, then we prepend the node's *asn* attribute three times. This is as-per the policies defined in Di Battista *et al.* [25], and once again, an alternative definition could be incorporated and activated through the use of policy attributes from the Network Whiteboard stage. If the route-map is *localPrefIn*, we set the *local-preference* to *90*. Finally, if the route-map is *metricOut*, we set the *match-ip-adddress* to be the *myAggregate* access-list, and the *metric* to be *10*. We create the *myAggregate* access-list as the list of each of the infrastructure prefixes allocated to the *asn* attribute of the node. This could also be expanded further to accommodate other policy requirements. Finally, we set the *route_maps* and *access_lists* attributes on the node. This concludes the Design Function steps for the eBGP Network View.

Routing Protocol Redistribution

The final step of the Design Function process is to set the redistribution of BGP into RIP. This is shown in Listing H.40, where we look at the *redistribute_bgp_to_igp* set on the Network Whiteboard. If this attribute is set to *True*, and the node has at least one eBGP peer, then we set the *redistribute_bgp* attribute in the RIP Network View to True. This is then used by the RIP component of the Quagga device compiler to set the appropriate redistribution command. This is a further example of high-level policy expressed on the Network Whiteboard being used to set low-level device configuration directives.

8.4.6 *Compilation*

Netkit Platform Compiler

The Netkit Platform compiler is similar to that used in previous examples. It is shown in Listing H.43. The only significant difference to previous examples is that we do not need to allocate interface identifiers to the data interfaces, as these have been designated on the Network Whiteboard. We do however allocate a tap interface, and automatically assign this an interface identifier.

Quagga Device Compiler

The Quagga device compiler for this case study needs to be extended to handle both the RIP routing protocol, and the BGP routing policy. The code for this extended compiler is shown in Listing H.44.

The RIP component of the compiler maps two key pieces of information: the networks to advertise, and the redistribution information. The networks are obtained from the node in the *rip* overlay.. Similarly, the *redistribute_bgp* and *redistribute_connected* values are obtained from the node in the *rip* overlay, and were also specified using the RIP Design Function.

The BGP component of the Quagga device compiler consists of the iBGP and eBGP session handling code. The iBGP code uses the same logic as previous examples, as only the iBGP overlay has differed according to the Design Functions discussed in Section 8.4.5.

The eBGP component of the compiler follows similar logic to previous examples, with extensions to handle the routing policy. The inbound and outbound *prefix lists* and *route maps*, as well as the default originate *route map* are mapped from the session in the *ebgp* overlay. Similarly, the *default_orginate* boolean value is also mapped from the session in the *ebgp* overlay. This demonstrates how the design of the policy has been specified in the Design Functions discussed in Section 8.4.5, with the compiler simply mapping the designated policies onto the appropriate session in the Intermediate Device Model. The networks to be advertised over BGP are mapped as per previous examples. Finally, the *prefix lists*, *access lists*, and *route maps* for the node are mapped from the node in the *ebgp* overlay.
A utility function, PROCESS_ROUTE_MAPS in the Quagga device compiler has been created to assist with the creation of the route maps. This is responsible for mapping the generic format expressed on the node in the *ebgp* overlay into the format appropriate for the specific target device (Quagga in this case). This allows the generic format to be mapped into the device-specific format, by performing a transla-

tion of the route-map syntax. This could be expanded in future by extending the language for the generic expression of route-maps, as discussed in Section 9.4.2.

Templates

The templates used for generating the device-specific syntax are largely the same as in previous examples. The notable difference is the addition of the RIP template, shown in Listing H.41, to generate the *ripd.conf* output, and the extension of the BGP template to handle the addition of routing policy. The extended BGP template is specified in Listing H.42. This extended BGP template shows the iteration over the route maps, prefix lists, and access lists as specified in the Intermediate Device Model by the Quagga device compiler. This applies for each session, as well as the specification of the route maps, prefix lists, and access lists at the global level of the *bgpd.conf* configuration file.

8.4.7 *Example Intermediate Device Model and Device Configurations*

An example of the Intermediate Device Model for the node *as20r1* is shown below in Listing 8.11. This shows the *global*, *interfaces*, *rip*, and *bgp* components. The *global* (hostname, ssh, etc) and *interfaces* components are similar to previous examples.

The *rip* component shows the networks to be advertised over the RIP routing protocol, and that connected prefixes are to be redistributed into RIP. BGP learnt prefixes are not to be redistributed into RIP.

The *bgp* component shows the *ibgp* neighbours as per previous examples. The *networks* shows the prefixes to advertise over BGP, also as per previous examples. We can see the *ebgp* neighbours shows the new attributes for the *default_originate* boolean, and the lists for the prefix list and route map names. We can also see the prefix lists and route maps defined at the *bgp* level.

```
{
  "asn": 20,
  "bgp": {
    "access_lists": {},
    "ebgp_neighbors": [
      {
        "asn": 200, "default_originate": true,
        "desc": "Router as200r1", "neigh_ip": "11.0.0.33",
        "plIn": ["as200In"],
        "plOut": ["defaultOut"],
        "rmDo": [], "rmIn": [], "rmOut": []
      }, {
        "asn": 30, "default_originate": false,
        "desc": "Router as30r1", "neigh_ip": "11.0.0.18",
        "plIn": [], "plOut": ["defaultOut"],
        "rmDo": ["dontUseMe"], "rmIn": [],
        "rmOut": ["dontUseMe"]
      }, {
        "asn": 100, "default_originate": true,
        "desc": "Router as100r1", "neigh_ip": "11.0.0.5",
        "plIn": ["as100In"], "plOut": ["defaultOut"],
        "rmDo": [], "rmIn": [], "rmOut": []
      }
    ],
    "ibgp_neighbors": [
      {"asn": 20, "desc": "Router as20r2",
        "neigh_ip": "20.1.1.2"},
      {"asn": 20, "desc": "Router as20r3",
        "neigh_ip": "20.1.1.3"}
    ],
    "networks": ["11.0.0.4/30", "11.0.0.16/30", "11.0.0.32/30", "20.1.1.0
        /24"],
    "prefix_lists": {
      "as100In": [
        ["permit", "100.1.3.0/24"],
        ["permit", "100.1.2.0/24"]
      ],
      "as200In": [
        ["permit", "200.2.0.0/16"]
      ],
      "defaultOut": [
        ["permit", "0.0.0.0/0"]
      ]
    },
    "route_maps": { "dontUseMe": [
        ["set as-path prepend", "20 20 20"]
      ]
    }
  },
  "hostname": "as20r1",
  "interfaces": [
    {"broadcast": "11.0.0.7", "id": "eth0",
      "ip": "11.0.0.6", "netmask": "255.255.255.252"},
    {"broadcast": "20.1.1.255", "id": "eth1",
      "ip": "20.1.1.1", "netmask": "255.255.255.0"},
    {"broadcast": "11.0.0.19", "id": "eth2",
      "ip": "11.0.0.17", "netmask": "255.255.255.252"},
    {"broadcast": "11.0.0.35", "id": "eth3",
      "ip": "11.0.0.34", "netmask": "255.255.255.252"},
    {"broadcast": "172.16.255.255", "id": "eth4",
      "ip": "172.16.0.8", "netmask": "255.255.0.0"}
  ],
  "rip": {
    "networks": ["20.1.1.0/24"],
    "redistribute_bgp": false,
    "redistribute_connected": true
  },
  "ssh": {"use_key": false},
  "zebra": {"password": "zebra"}
}
```

Listing 8.11: Intermediate Device Model for *as20r1*

The generated configuration files are shown below for RIP as *ripd.conf* in Listing 8.12 and BGP as *bgp.conf* in Listing 8.13. The RIP configuration shows the network prefixes advertised, and the redistribute commands. The BGP configuration shows the BGP sessions with the route maps, default originate, and prefix lists set, as per the Design Functions discussed in Section 8.4.5. It also shows the specification of the prefix lists and route maps.

```
!
!
hostname ripd
password zebra
!
router rip
redistribute connected
network 20.1.1.0/24
!
log file /var/log/zebra/ripd.log
```

Listing 8.12: Generated *ripd.conf* configuration file for *as20r1*

```
!
hostname bgpd
password zebra
enable password zebra
!
router bgp 20
network 11.0.0.4/30
network 11.0.0.16/30
network 11.0.0.32/30
network 20.1.1.0/24
!
neighbor 20.1.1.2 remote-as 20
neighbor 20.1.1.2 description Router as20r2 (iBGP)
!
neighbor 20.1.1.3 remote-as 20
neighbor 20.1.1.3 description Router as20r3 (iBGP)
!
!
neighbor 11.0.0.33 remote-as 200
neighbor 11.0.0.33 description Router as200r1 (eBGP)
neighbor 11.0.0.33 default-originate
neighbor 11.0.0.33 prefix-list defaultOut out
neighbor 11.0.0.33 prefix-list as200In in
!
neighbor 11.0.0.18 remote-as 30
neighbor 11.0.0.18 description Router as30r1 (eBGP)
neighbor 11.0.0.18 prefix-list defaultOut out
neighbor 11.0.0.18 default-originate route-map dontUseMe
neighbor 11.0.0.18 route-map dontUseMe out
!
neighbor 11.0.0.5 remote-as 100
neighbor 11.0.0.5 description Router as100r1 (eBGP)
neighbor 11.0.0.5 default-originate
neighbor 11.0.0.5 prefix-list defaultOut out
neighbor 11.0.0.5 prefix-list as100In in
!
!
ip prefix-list as100In permit 100.1.3.0/24
ip prefix-list as100In permit 100.1.2.0/24
!
ip prefix-list as200In permit 200.2.0.0/16
!
ip prefix-list defaultOut permit 0.0.0.0/0
!
!
route-map dontUseMe permit 10
```

```
set as-path prepend 20 20 20
!
!
!
log file /var/log/zebra/bgpd.log
!
debug bgp
debug bgp events
debug bgp filters
debug bgp fsm
debug bgp keepalives
debug bgp updates
!
```

Listing 8.13: Generated *bgpd.conf* configuration file for *as20r1*

Finally, Listing H.45 shows a further example of a rendered *bgpd.conf* configuration file for *as30r1*.

8.4.8 *Simulation Results*

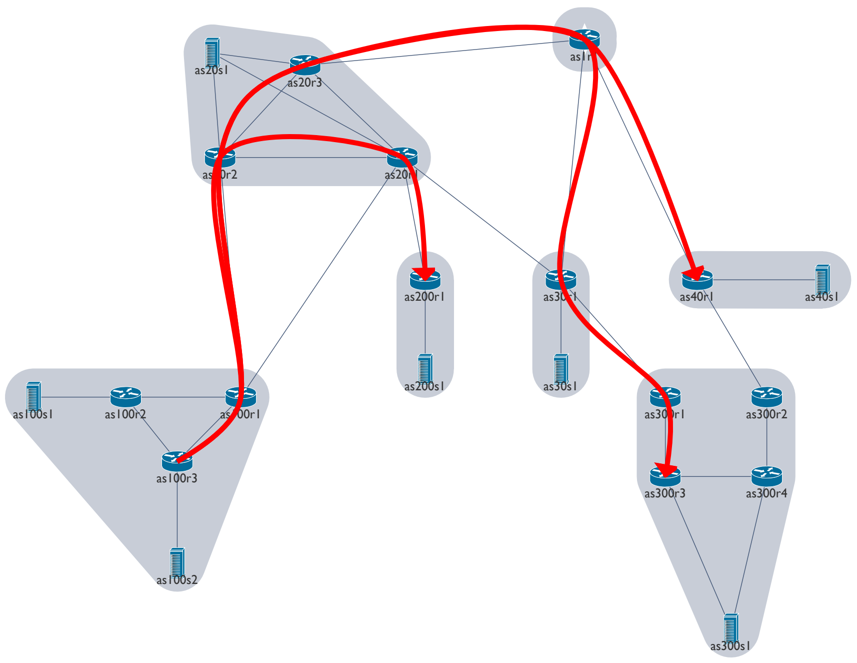


Figure 8.26: Visualisation of traceroutes from *as100r1* to *eth1.as200r1*, *eth1.as40r1* and *eth1.as300r3*

A trace from *as100r3* to *as40r1* is shown in Listing 8.4.8, and shows the correct advertisement of BGP prefixes. We note that the path taken flows up through the provider of AS20, and then through the backbone of AS1, to reach AS40. We can also see the correct use of the gateway router of *as100r1* for AS100 on the first hop.

```
as100r3:~# traceroute eth1.as40r1
traceroute to eth1.as40r1 (40.4.4.1), 64 hops max, 40 byte packets
```

```
 1  eth2.as100r1 (100.1.0.1)    0 ms  0 ms  0 ms
 2  eth0.as20r2  (11.0.0.2)     0 ms  0 ms  0 ms
 3  eth1.as20r3  (20.1.1.3)     0 ms  0 ms  0 ms
 4  eth2.as1r1   (11.0.0.22)    1 ms  1 ms  0 ms
 5  eth1.as40r1  (40.4.4.1)     1 ms  1 ms  1 ms
```

The trace from *as100r3* to *as200r1*, shown in Listing 8.14, shows the use of the primary path via *as20r2* even though it would be shorter via *as20r1*. This confirms the correct application of the BGP routing policy to prefer the *primary* link over the *backup* link.

```
as100r3:~# traceroute eth1.as200r1
traceroute to eth1.as200r1 (200.2.0.1), 64 hops max, 40 byte packets
 1  eth2.as100r1 (100.1.0.1)    0 ms  0 ms  0 ms
 2  eth0.as20r2  (11.0.0.2)     0 ms  0 ms  0 ms
 3  eth2.as20r1  (20.1.1.1)     1 ms  0 ms  0 ms
 4  eth1.as200r1 (200.2.0.1)    1 ms  0 ms  0 ms
```

Listing 8.14: Traceroute from *as100r3* to *eth0.as200r1*

The trace from *as100r3* to *eth1.as300r3* again shows the preference of the *primary* link from *as100r1* to *as20r1* over the path from *as100r1* to *as20r2*. It also shows the preference of the provider-backbone-provider path of AS20 to AS1 to AS30, over the provider-provider path of *as20r1* to *as30r1*. This is also due to the application of the routing policy. Finally, we can see the route going through *as300r1*, demonstrating the load sharing advertising the /*17* prefix of *200.1.0.0/17*.

```
as100r3:~# traceroute eth1.as300r3
traceroute to eth1.as300r3 (200.1.0.2), 64 hops max, 40 byte packets
 1  eth2.as100r1 (100.1.0.1)    13 ms  0 ms  0 ms
 2  eth0.as20r2  (11.0.0.2)     0 ms   0 ms  0 ms
 3  eth1.as20r3  (20.1.1.3)     0 ms   0 ms  0 ms
 4  eth2.as1r1   (11.0.0.22)    1 ms   0 ms  0 ms
 5  eth2.as30r1  (11.0.0.25)    1 ms   1 ms  1 ms
 6  eth0.as300r1 (11.0.0.9)     1 ms   1 ms  1 ms
 7  eth1.as300r3 (200.1.0.2)    1 ms   1 ms  1 ms
```

Listing 8.15: Traceroute from *as100r3* to *eth1.as300r3*

These three traceroutes are visualised in Figure 8.26.

The traceroute from *as100r3* to *eth1.as300r3* at *200.1.0.2*, and to *eth0.as300r3* at *200.1.128.1* shows the load sharing advertisement working correctly for the *200.1.0.0/17* and *200.1.128.0/17* prefixes. In the traceroute shown in Listing 8.16, we can see the path to *eth0.as300r3* is via AS40, in contrast to the path to *eth1.as300r3* which was via AS30. This confirms the load sharing policy is applied correctly. This traceroute is visualised in Figure 8.27.

```
as100r3:~# traceroute eth0.as300r3
traceroute to eth0.as300r3 (200.1.128.1), 64 hops max, 40 byte packets
 1  eth2.as100r1 (100.1.0.1)    0 ms  0 ms  0 ms
 2  eth0.as20r2  (11.0.0.2)     0 ms  0 ms  0 ms
 3  eth1.as20r3  (20.1.1.3)     0 ms  0 ms  0 ms
 4  eth2.as1r1   (11.0.0.22)    0 ms  1 ms  0 ms
```

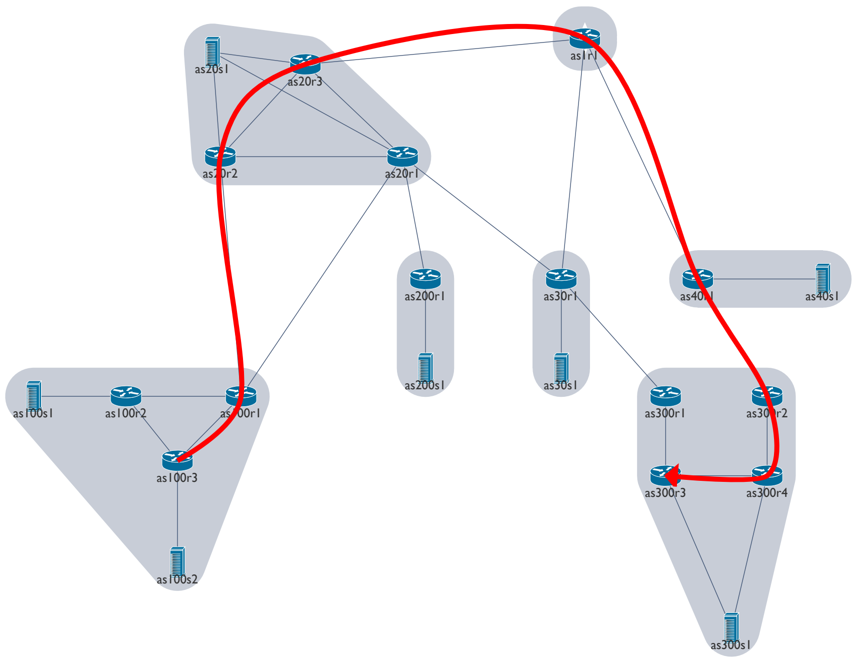


Figure 8.27: Visualisation of traceroute from *as100r1* to *eth0.as300r3*

```
5  eth2.as40r1  (11.0.0.29)    1 ms  1 ms  1 ms
6  eth0.as300r2 (11.0.0.13)    1 ms  1 ms  1 ms
7  eth0.as300r4 (200.1.64.2)   1 ms  1 ms  1 ms
8  eth0.as300r3 (200.1.128.1)  1 ms  1 ms  1 ms
```

Listing 8.16: Traceroute from *as100r3* to *eth0.as300r3*

We now discuss more detailed diagnostic output from the *show ip route* and *show ip bgp* commands from the *zebra* and *bgpd* processes.

The route table for *as1r1*, shown in Listing H.3.5 shows the prefixes learnt over *ebgp* and the peer which announces them.

The route table for *as20r1* shown in Listing H.3.5 shows the default route to *0.0.0.0/0* via *11.0.0.22* (*as1r1*), which is via *20.1.1.3* (*as20r3*). Thus the default route is to *as1r1*, announced from *as20r3*. The point-to-point inter-AS routes of *11.0.0.0/30* and *11.0.0..20/30* are advertised over RIP. There are a number of directly connected point-to-point inter-AS routes of *11.0.0.4/30* and *11.0.0.32/30* and *11.0.0.16/30*. The network for *AS200* of *200.2.0./16* is announced via *11.0.0.33* (*as200r1*). Finally, the network of *100.1.0.0/16* is reached via *11.0.0.1* (*eth0* of *as100r1*) via *20.1.1.2* (*as20r2*). This confirms the routing policy to prefer reaching *AS100* through *as20r2* vs the more direct route of *as20r1*. This is due to the routing policies applied.

In Listing 8.17 we can see the BGP table for *as1r1*. This shows the next hop to each network advertised, and the path. We can see some networks, such as *200.2.0.0/16* is reached via *11.0.0.21* (*as20r3*), and goes through AS *20* then AS *200*. We can also see multiple paths, such as to *200.1.0.0/16* which is advertised by both *as300r1* and *as300r2*. This prefix can be reached via *11.0.0.29* (*as40r1*) over the AS path of *40* then *300*, or via *11.0.0.25* (*as30r1*) over the AS path of *30* then *300*.

```
bgpd# sh ip bgp
BGP table version is 0, local router ID is 11.0.0.22
Status codes: s suppressed, d damped, h history, * valid, > best, i -
    internal,
              r RIB-failure, S Stale, R Removed
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*> 0.0.0.0          0.0.0.0                  0         32768 i
*> 11.0.0.0/30      11.0.0.21                              0 20 i
*> 11.0.0.4/30      11.0.0.21                              0 20 i
*> 11.0.0.8/30      11.0.0.25                0             0 30 i
*> 11.0.0.12/30     11.0.0.29                0             0 40 i
*> 11.0.0.16/30     11.0.0.21                              0 20 i
*> 11.0.0.20/30     0.0.0.0                  0         32768 i
*> 11.0.0.24/30     0.0.0.0                  0         32768 i
*> 11.0.0.28/30     0.0.0.0                  0         32768 i
*> 11.0.0.32/30     11.0.0.21                              0 20 i
*> 20.1.1.0/24      11.0.0.21                0             0 20 i
*> 30.3.3.0/24      11.0.0.25                0             0 30 i
*> 40.4.4.0/24      11.0.0.29                0             0 40 i
*> 100.1.0.0/16     11.0.0.21                              0 20 100 i
*  200.1.0.0/16     11.0.0.29                              0 40 300 i
*>                  11.0.0.25                              0 30 300 i
*> 200.1.0.0/17     11.0.0.25                              0 30 300 i
*> 200.1.128.0/17   11.0.0.29                              0 40 300 i
*> 200.2.0.0/16     11.0.0.21                              0 20 200 i

Total number of prefixes 18
```

Listing 8.17: show ip bgp output for *as1r1*

Listing H.3.5 shows the BGP table for *as30r1*.

The routing table for *as300r1* is shown in Listing H.3.5. It shows the directly connected and internal routes announced via RIP. The default route to *0.0.0.0/0* is reached through *11.0.0.10*, which is *as30r1*.

The BGP table for *as100r1* is shown in Listing 8.18, and shows the local-preference policy influencing the path to *11.0.0.6* (*as20r1*). This is setting the Local Preference value (shown as LocPrf) to *90*.

```
bgpd# show ip bgp
BGP table version is 0, local router ID is 100.1.0.5
Status codes: s suppressed, d damped, h history, * valid, > best, i -
    internal,
              r RIB-failure, S Stale, R Removed
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*  0.0.0.0          11.0.0.6                       90      0 20 i
*>                  11.0.0.2                               0 20 i
*> 100.1.0.0/16     0.0.0.0                  0         32768 i
```

Listing 8.18: show ip bgp output for *as100r1*

In the BGP table for *as20r1*, shown in Listing H.3.5, we can see the pre-pended AS path with the extra *30* values in the Path column.

Finally, we can see the default route of *0.0.0.0/0* announced over RIP (not iBGP) for the route tables of *as300r3* shown in Listing 8.19 and *as300r4* shown in Listing H.3.5.

```
as300r3# sh ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

R>* 0.0.0.0/0 [120/2] via 200.1.0.1, eth1, 00:24:07
C>* 127.0.0.0/8 is directly connected, lo
C>* 200.1.0.0/18 is directly connected, eth1
R>* 200.1.64.0/18 [120/2] via 200.1.128.2, eth0, 00:24:13
C>* 200.1.128.0/17 is directly connected, eth0
```

Listing 8.19: SHOW IP ROUTE output for *as300r3*

These diagnostic commands demonstrate the configurations generated using our approach from the Network Whiteboard high-level description meet the configuration and policy objectives of Di Battista *et al.* [25].

8.4.9 *Conclusion*

In this case study we have shown how we can build upon the previous Simplified Small Internet lab, and accurately reproduce the Full Small Internet Lab described in Di Battista *et al.* [25]. This provides a more realistic case study, and demonstrates how the framework is able to handle these requirements.

8.5 VLAN TOPOLOGY

8.5.1 *Introduction*

In this case study, we focus on the configuration of a Layer 2 Managed Switch topology, with the use of VLAN tags. We reproduce the theoretical example outlined in Section 5.7.2 to demonstrate how our approach can be applied to generate switch configurations for a campus or enterprise environment. This complements the previous examples which are focused on a wide-area-network environment.

As the Netkit simulation environment does not support dedicated managed switches, we generate configurations that can be on the Cisco IOSv and IOSvL2 virtual machine images. These virtual machines run router and managed switch operating systems, respectively. We run these on the Cisco VIRL simulation platform, described in [79].

This case-study affords us with an opportunity to demonstrate how the system can be used to generate configurations for routers of a different network operating system (IOS instead of Quagga), and to generate configurations for managed switches. It also allows us to demonstrate how the underlying framework can be applied to a different simulation platform by using the Platform Compiler approach discussed in Section 6.3.2.

We note that our Design Functions, such as for the routing protocols, do not need to be adapted from those used to generate the Abstract Network Models used in the previous case studies. The Design Functions in the previous case studies were designed for the Quagga routing platform, but can be applied, without modification, for the Cisco IOSv platform. This demonstrates the vendor independence of the Abstract Network Model and Design Function approach, which describe the declared network functionality, independent of the low-level device-specific implementation details.

8.5.2 *Methodology*

The methodology used to implement and verify this case study is shown in Figure 8.28. We add a VLAN Design Function to cater for the managed switches and their VLAN attributes. The remainder of the Design Functions are unmodified from previous case studies. We do not configure iBGP or eBGP in this topology, so these Design Functions have been removed.

We simulate this topology on the VIRL platform using the IOSv router and IOSvL2 managed switch. We therefore replace the Netkit Platform Compiler with a VIRL Platform Compiler, and the Quagga Device Compiler with an IOSv Router Device Compiler and IOSvL2 Switch Device Compiler. The VIRL Platform Compiler is responsible for selecting the correct Device Compiler to use based on the device specification on the Network Whiteboard. To generate configurations for these devices we provide a IOSv Router Device Template and IOSvL2 Switch Device Template, which are assembled to produce the device configurations. The VIRL Platform uses a single XML file as its input, so the Platform Assembler has been extended to first use the Intermediate Platform Model and VIRL XML Template to generate the simulated topology specification, and then to insert the device configurations.

The final VIRL XML file is simulated on the VIRL Simulation platform. To verify this case study we use the TRACEROUTE as per previous case studies, and the SHOW VLAN command on the managed switches to verify the switching configuration. We provide a detailed analysis of the TRACEROUTE output to confirm the correct configuration of the VLANs on the managed switches, and the OSPF routing protocol on the routers.

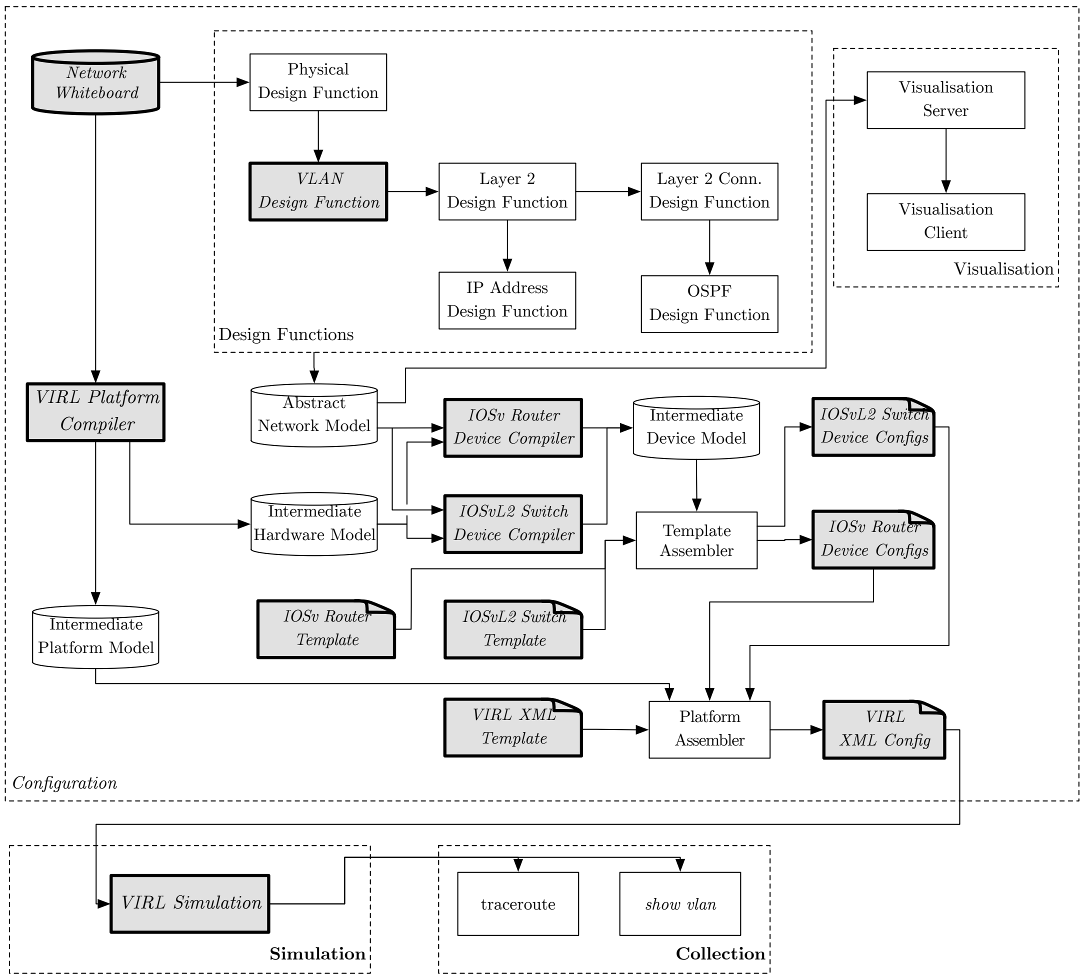


Figure 8.28: Configuration Generation and Verification methodology for VLAN Case Study

8.5.3 *Network Whiteboard*

The Network Whiteboard is created as a GraphML file, and then post-processed. The GraphML file uses directed edges to annotate the VLAN attributes on interfaces. The directed edges are used to mark the source and destination VLANs, which are then mapped onto the interfaces as they are created on the *input* overlay.

As this is the first case-study to use both *routers* and *switches*, we create a list of the nodes which are to have the role of *switch*. We then iterate across the nodes in the *input* overlay and mark them with the *device_type* of *switch* or *router* if they are in this list of switches, or not, respectively. These steps are displayed in Listing H.46.

We then explicitly create a dictionary of *nodes* and *edges*, as shown in Listing H.47 and Listing H.48, respectively. These explicitly denote the attributes to be set on the nodes, and the VLANs to set on the interfaces. We use the convention of *(src, dst, vlan)* for the edge list, and apply the *vlan* attribute to the interface of the *dst* node connected to the *edge*. This is shown in Figure 8.29, where the *vlan* attribute is displayed on the interface.

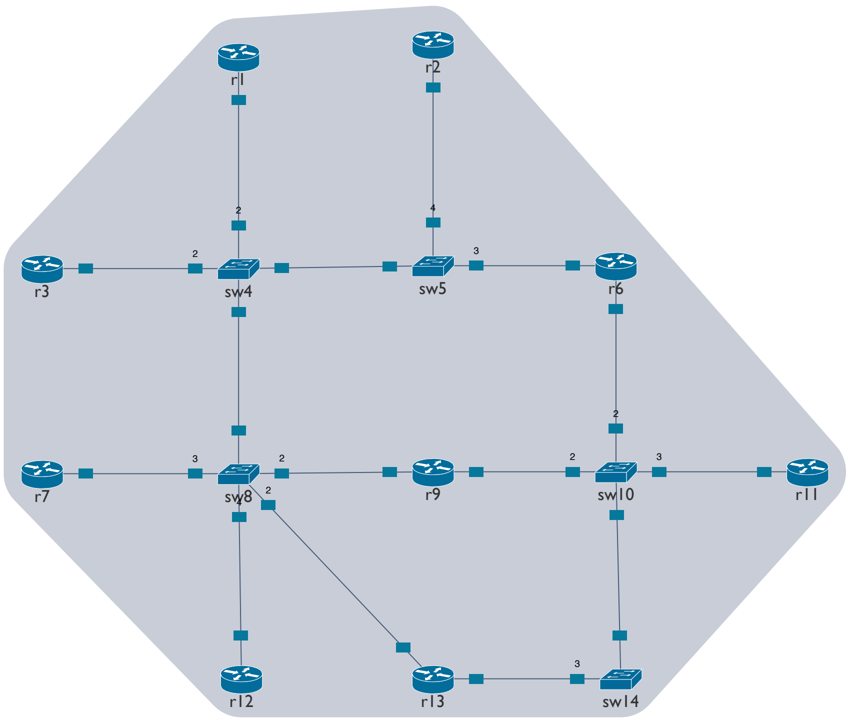


Figure 8.29: Network Whiteboard for VLAN topology showing VLANs on Interfaces.

8.5.4 *Design Functions*

Physical Network View

The next step is to create the *physical* network view. This step is performed like previous Design Functions, where the nodes and edges are added from the *input* overlay. Once again, as the edges only denote physical connectivity, we do not need to use any filtering functions. This again demonstrates the versatility of our generic approach to network design. The *physical* Design Function is shown in Listing H.49, and a visualisation of the result is shown in Figure 8.30.

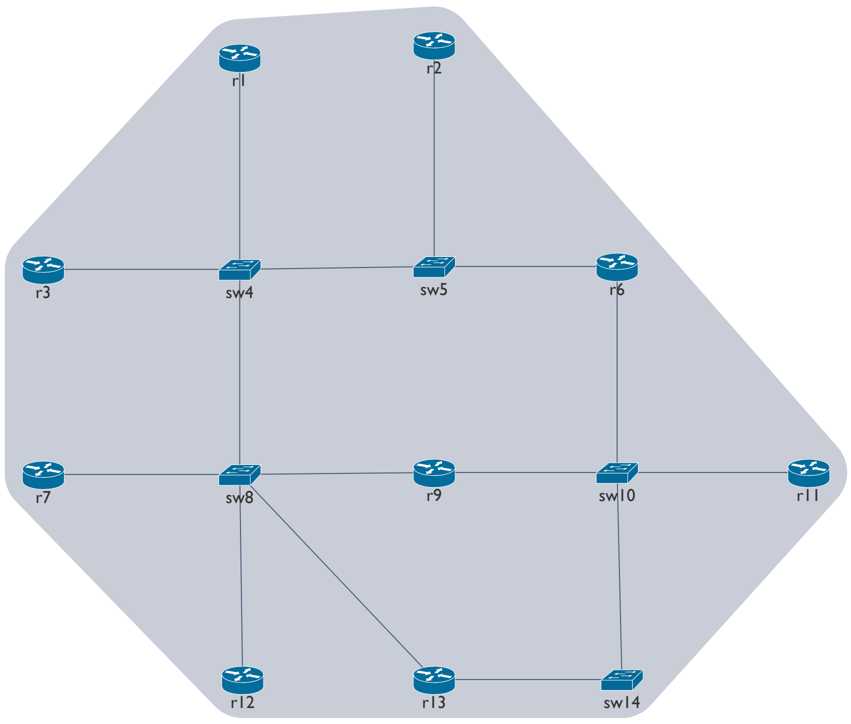


Figure 8.30: Physical Network View for VLAN topology.

VLAN Network View

The next step is to construct the *vlan* Network View. This differs to previous workflows for router-only topologies. We construct this Network View following the theoretical approach described in Section 5.7.2. This *vlan* Network View is then used to construct the *Layer 2* Network View. This demonstrates the ability of our approach to be adapted to different design approaches and requirements, without requiring modification of the latter Design Functions.

The first step in construction the *vlan* Network View is to add the nodes from the *physical* Network View, which are the *routers* and the *switches*, and then to add the edges representing physical connectivity. This is shown in Listing H.50.

AGGREGATE SWITCHES We then define the AGGREGATE_NODES function as an example of the MERGE High-Level Primitive described in Algorithm 5.4.3. This is shown in Listing H.51 .

We then copy the *vlan* interface attribute from the *input* overlay using the COPY_INT_ATTR function, and *merge* the switches using the AGGREGATE_NODES function. The code this step is shown in Listing H.52, with the result shown in Figure 8.31. Here we can see that two pseudo-nodes *pn0* and *pn1* have been created, from switches *sw4* and *sw8*, and *sw10* and *sw14* respectively.

We can also see that the *retain* parameter of *vlan* to the AGGREGATE_NODES function has retained the *vlan* attribute on the interfaces, as shown in the figure. A cross-comparison against the *vlan* attribute on the interfaces of the *input* overlay shown in Figure 8.29 confirms this.

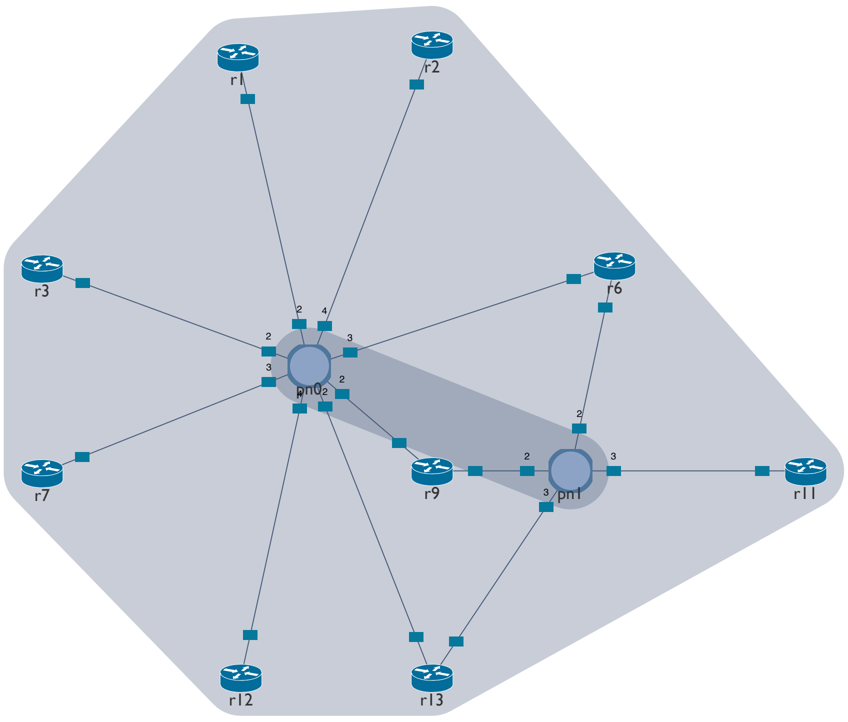


Figure 8.31: Step 2 of VLAN Network View for VLAN topology.

CREATE BROADCAST DOMAINS We then create the *broadcast_domain* pseudo_nodes, corresponding to the broadcast domains which arise from the *vlan* attributes set on the switches. The code for this step of the Design Function is shown in Listing H.50.

This is performed by iterating over each pseudo-node in the *vlan* Network View. We then create a dictionary of the edges of the pseudo-node corresponding to each *vlan* attribute. This is shown in Listing 8.20.

```
pn0 {"3": [(pn0, r7), (pn0, r6)], "2": [(pn0, r13), (pn0, r1), (pn0, r3),
     (pn0, r9)], "4": [(pn0, r12), (pn0, r2)]}
pn1 {"3": [(pn1, r11), (pn1, r13)], "2": [(pn1, r9), (pn1, r6)]}
```

Listing 8.20: VLANs grouped by pseudo-node

We then iterate over each vlan for each pseudo-node. We first create a new *broadcast_domain* pseudo-node. We then connect the destination interfaces of the original pseudo-node to the newly created *vlan* pseudo-node. We also set the *x* and *y* co-ordinate attributes for the *vlan* pseudo-node to be the average of the nodes to which it connects, for the visualisation.

Finally, we then remove the original pseudo-node from the Network View. The result of this step, defined in Listing H.53, is shown in Figure 8.32.

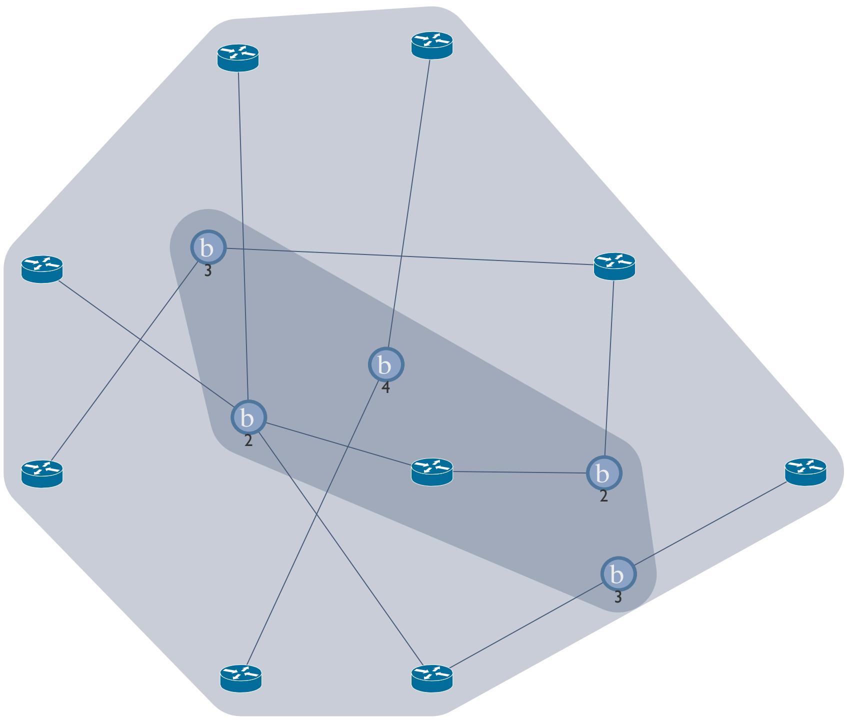


Figure 8.32: Step 3 of VLAN Network View for VLAN topology.

VLAN Switches Network View

The next step is to create the *switches* Network View, as shown in Listing H.54. This Network View is used to represent the trunking interfaces of the switches, and the Spanning Tree Protocol configuration.

This Network View is constructed by adding only the *switch* nodes from the *physical* Network View. We then add the edges from the *physical* Network View, which only adds the edges corresponding to the nodes which exist in the *switches* Network View, (the *switch* nodes).

We again use the COPY_INT_ATTR function to copy across the *vlan* attribute on the interfaces, from the *input* overlay. As the only edges in this Network View are between switches connected in the *physical* Network View, we can simply iterate across the edges in this Network View, and mark both the source and destination interfaces of the edge to be trunking interfaces, by setting the *trunking* attribute to *true*.

For the configuration of Spanning Tree Protocol, we also iterate over each connected subgraph, and record the set of all *vlan* attributes occurring within that subgraph. In this example, we have two connected subgraphs: one for *sw4* and *sw8*, and the other for *sw10* and *sw14*. For the first subgraph, the *vlan* attributes set are *2*, *3*, and *4*. For the second subgraph, the *vlan* attributes set are 2 and 3. These are recorded onto the switches in the subgraph.

The result of this step is shown in Figure 8.33, where the node labels show the VLANs used in the connected subgraph, and the interface labels show the trunking boolean.

Layer 2 Network View

The *Layer 2* Network View is created from the *vlan* Network View. This differs to the previous case studies, where the *Layer 2* Network View was constructed from the *physical* Network View.

Other than this difference, the remainder of the *Layer 2* Design Function is as per previous case studies. As shown in Listing H.55, *broadcast domain* nodes are retained as-is from the *Layer 2* Network View. Any point-to-point links between layer 3 devices are split with a broadcast domain. As there are no router-to-router connections in this topology, the *Layer 2* Network View, shown in Figure 8.34, looks the same as the final result of the *vlans* Network View previously shown in Figure 8.32. In the case that the topology contained both switches and point-to-point links, the *Layer 2* Design Function could accommodate for both.

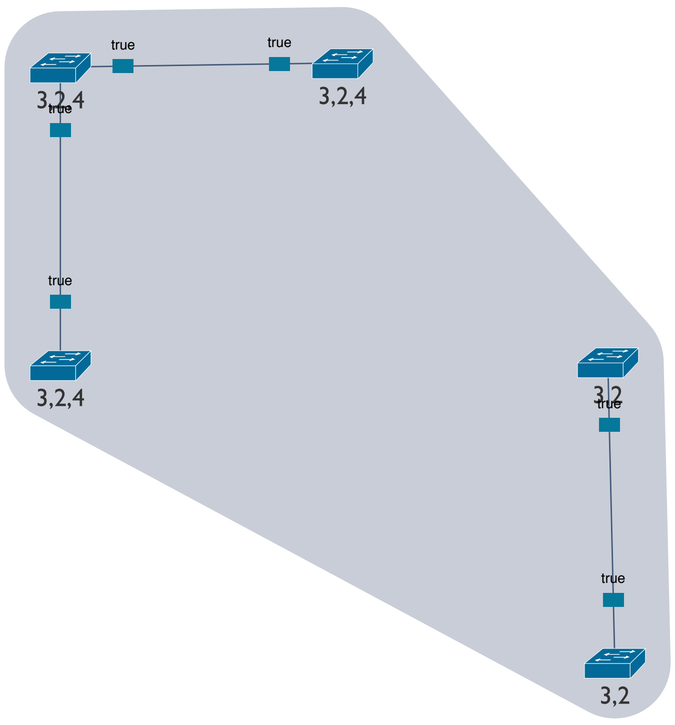


Figure 8.33: Switches Network View of VLAN topology. Node labels show VLANs for the Spanning Tree Protocol, and interface labels show trunking state.

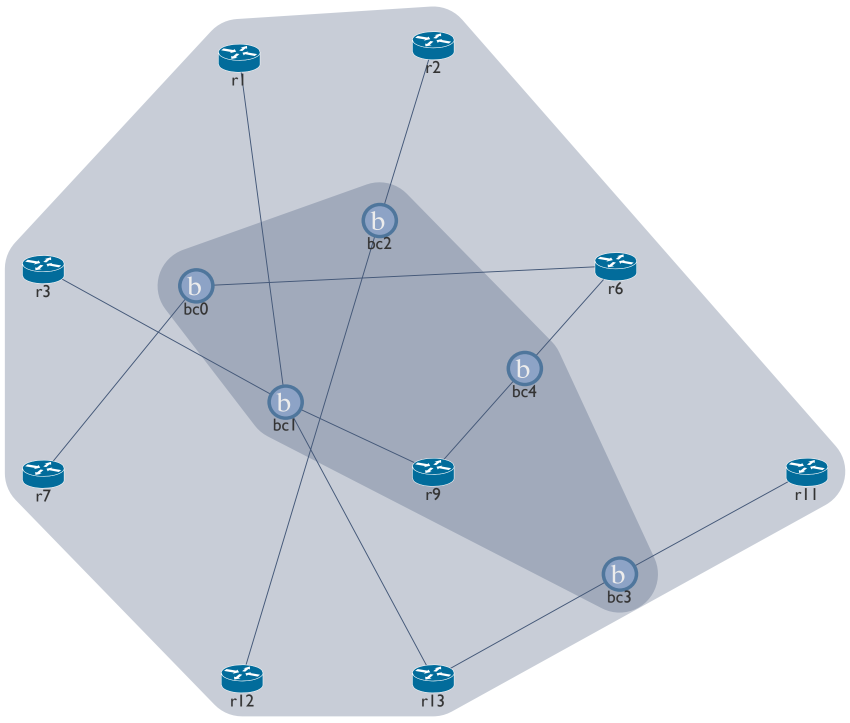


Figure 8.34: Layer 2 Network View for VLAN topology.

IP Address Network View

The *IP Address* Network View is constructed as shown in Listing H.57. This Design Function follows the same logic as for the Simplified Small Internet Example.

The result of the IP Address allocations are shown in Figure 8.35, where the *broadcast domain* pseudo-nodes show the subnet allocated, and the interfaces show the IP Address allocated.

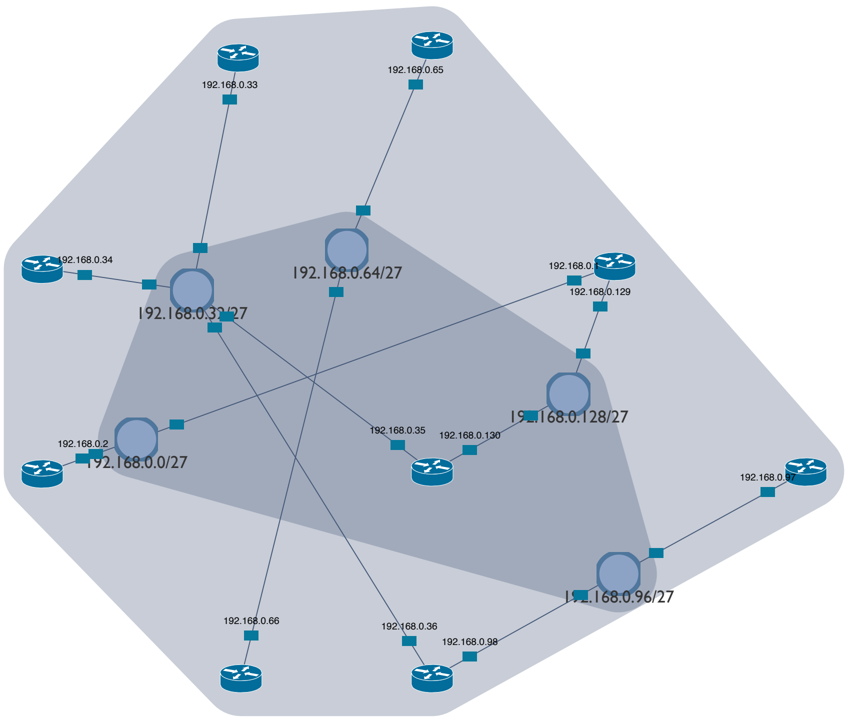


Figure 8.35: IP Address Network View for VLAN topology.

Layer 2 Connectivity Network View

The *Layer 2 Connectivity* Design Functions is shown in Listing H.56. This is the same as in previous case studies, and simply applies the EXPLODE function to the *broadcast domain* nodes added from the *Layer 2* Network View. The result of this Design Function is shown in Figure 8.36.

We note that even though the *Layer 2* Topology is created through the use of VLANs, we can apply the same *Layer 2 Connectivity* Design Function as in previous cases. This illustrates the power of the separation of concerns in our approach.

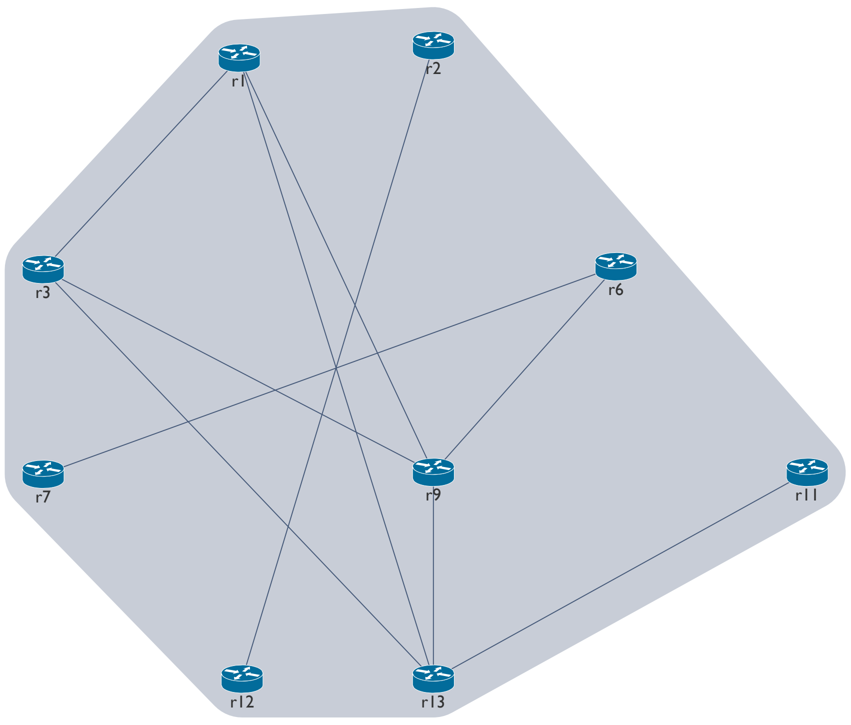


Figure 8.36: Layer 2 Connectivity Network View for VLAN topology.

OSPF Network View

The final Design Function in this case study is to create the OSPF Network View. The code for this example is shown in Listing H.58. This code is the same as that of the Small Internet case study, once again illustrating the modularity of our approach. A visualisation of the result of this Design Function is shown in Figure 8.37.

BGP Network View

We do not configure either iBGP or eBGP for this example. If required, these protocols could be configured by following the same approach discussed in previous case studies.

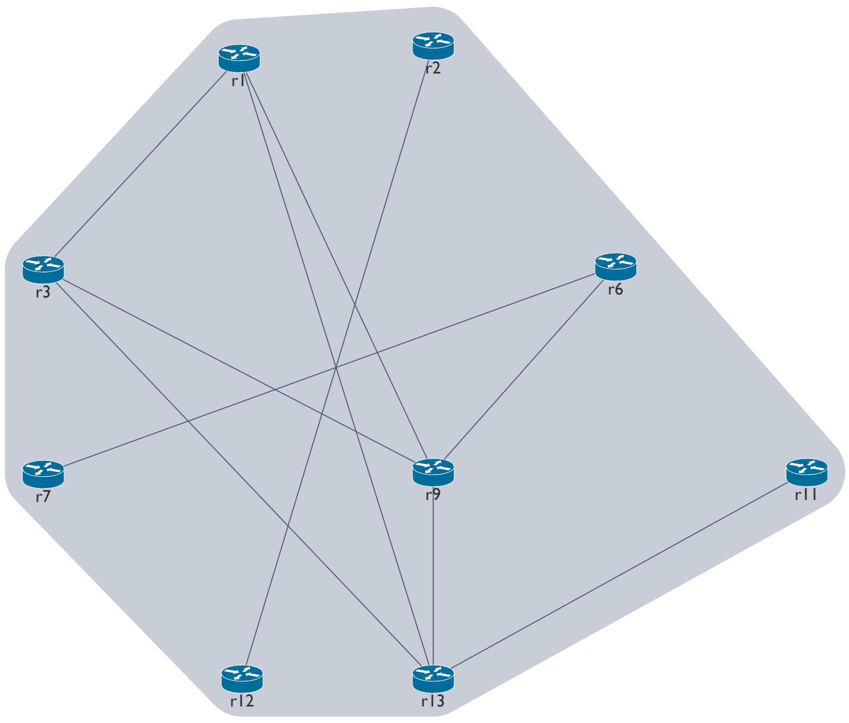


Figure 8.37: OSPF Network View for VLAN topology.

8.5.5 *Compilation*

Netkit doesn't easily provide for managed switches, so instead we use the Cisco VIRL platform [79] for this case study. This requires compiling for IOSv router, instead of the Quagga router, and creating a new device compiler for the IOSvL2 switch. This also allows us to demonstrate how our approach can be easily adapted to both a different simulation platform, and to a different set of target device syntaxes. We will also show the packaging of the device configurations into the VIRL XML topology specification file.

VIRL Platform Compiler

The structure of the .virl XML topology file is based on the example file from [103], and has been modified to this specific scenario. The .virl file used contains the nodes, interfaces, and connections. We then insert the configurations generated using the compilation process into the nodes.

The VIRL platform compiler is responsible for allocating the interface identifiers. This is done using one of two utility functions, ASSIGN_ROUTER_INTERFACE_IDS or ASSIGN_SWITCH_INTERFACE_IDS depending on whether the device is a router or a switch. These allocations are shown in Figure 8.38. The platform compiler also gen-

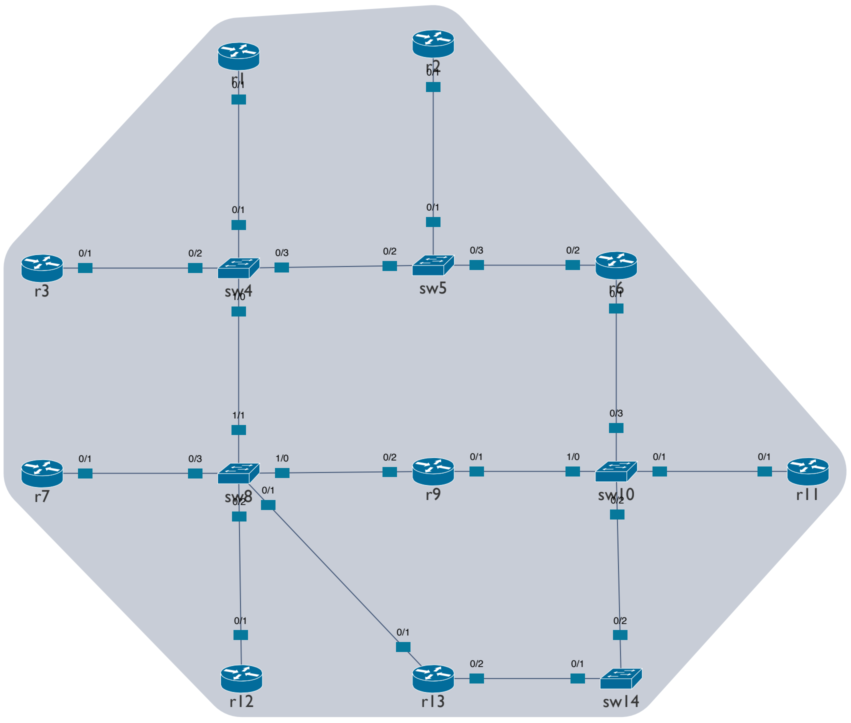


Figure 8.38: Physical Network View for VLAN topology with interface names allocated by the Platform Compiler. The *GigabitEthernet* prefix has been omitted from interface names.

erates the Intermediate Device Model using either the IOSv Router Compiler or the IOSvL2 Switch Compiler. These are discussed in Figure 8.5.5 and 8.5.5 respectively. Finally, the platform compiler assigns the appropriate render template to use depending on the device type. The templates are discussed in 8.5.5. The code for the VIRL Platform Compiler is shown in Listing H.59.

IOSv Router Compiler

The IOSv Router Compiler follows similar logic to the Quagga device compiler discussed in previous case studies. The code is shown in Listing H.60. For this case study we do not configure BGP, so this function is omitted from the IOSv Router Compiler. An example Intermediate Device Model, for router *r9*, is shown in Listing 8.21, including the interfaces and OSPF networks.

```
{
 "asn": 1,
 "hostname": "r9",
 "interfaces": [
  {"id": "GigabitEthernet0/1", "broadcast": "192.168.0.159",
   "ip": "192.168.0.130", "netmask": "255.255.255.224",
   "ospf_cost": 1, "physical": true},
  {"id": "GigabitEthernet0/2", "broadcast": "192.168.0.63",
   "ip": "192.168.0.35", "netmask": "255.255.255.224",
   "ospf_cost": 1, "physical": true},
  {"id": "Loopback0", "broadcast": null,
   "ip": "172.16.0.6", "netmask": "255.255.255.255", "physical": false}
 ],
 "ospf": {
  "networks": [
   {"area": 0, "hostmask": "0.0.0.31", "prefix": "192.168.0.128"},
   {"area": 0, "hostmask": "0.0.0.31", "prefix": "192.168.0.32"},
   {"area": 0, "hostmask": "0.0.0.0", "prefix": "172.16.0.6"}
  ],
  "process_id": 1
 }
}
```

Listing 8.21: Intermediate Device Model for *r9*

The generated configuration for router *r9* is shown in Listing 8.22.

```
!
hostname r9
boot-start-marker
boot-end-marker
!
no aaa new-model
!
ip cef
!
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
no service config
enable password cisco
ip classless
ip subnet-zero
no ip domain lookup
line vty 0 4
 transport input ssh telnet
 exec-timeout 720 0
 password cisco
 login
line con 0
 password cisco
!
interface GigabitEthernet0/0
  description OOB Management
  ! Configured on launch
  no ip address
  duplex auto
  speed auto
  no shutdown
!
interface GigabitEthernet0/1
  ip address 192.168.0.130 255.255.255.224
  duplex auto
  speed auto
  no shutdown
!
interface GigabitEthernet0/2
  ip address 192.168.0.35 255.255.255.224
  duplex auto
  speed auto
  no shutdown
!
interface Loopback0
  ip address 172.16.0.6 255.255.255.255
!
router ospf 1
  log-adjacency-changes
  passive-interface Loopback0
  network 192.168.0.128 0.0.0.31 area 0
  network 192.168.0.32 0.0.0.31 area 0
  network 172.16.0.6 0.0.0.0 area 0
!
```

```
end
```

Listing 8.22: IOSv router configuration for *r9*

IOSvL2 Switch Compiler

The IOSvL2 Switch Compiler is shown in Listing H.60. It consists of the INTERFACES and VLANS functions. The INTERFACES function is responsible for mapping the interface identifier from the *physical* Network View, the *trunking* attribute from the *switches* Network View, and the *vlan* attribute from the *vlan* Network View. This demonstrates how multiple Network Views can be combined in a device compiler. The VLANS function is responsible for mapping the list of all VLANs set on the node, from the *switches* Network View. This set of VLANs is specified from the VLAN Switches Design Function.

An example Intermediate Device Model, for switch *sw4*, is shown in Listing 8.23, showing the interfaces configuration, including the *trunking* and *vlan* attributes. The list of VLANs to configure on the device is shown as the global *vlans* attribute.

```
{
 "asn": 1,
 "hostname": "sw4",
 "interfaces": [
  {"id": "GigabitEthernet0/1", "trunking": false, "vlan": "2"},
  {"id": "GigabitEthernet0/2", "trunking": false, "vlan": "2"},
  {"id": "GigabitEthernet0/3", "trunking": true, "vlan": null},
  {"id": "GigabitEthernet1/0", "trunking": true, "vlan": null}
 ],
 "vlans": ["2", "3", "4"]
}
```

Listing 8.23: Intermediate Device Model for *sw4*

The generated configuration for switch *sw9* is shown in Listing 8.24.

```
!
version 15.2
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
service compress-config
no service config
enable password cisco
ip classless
ip subnet-zero
no ip domain lookup
!
line vty 0 4
transport input ssh telnet
exec-timeout 720 0
password cisco
login
!
line con 0
password cisco
!
hostname sw4
!
boot-start-marker
```

```
boot-end-marker
!
no aaa new-model
!
ip cef
no ipv6 cef
!
!
spanning-tree mode pvst
spanning-tree extend system-id
!
vlan internal allocation policy ascending
!
vtp domain test.lab
vtp mode transparent
vlan 3
vlan 2
vlan 4
!
interface Loopback0
  description Loopback
!
interface GigabitEthernet0/0
  description Mapped to Vlan1 for management
  ! Configured on launch
  switchport mode access
  no shutdown
!
interface GigabitEthernet0/1
  switchport mode access
  switchport access vlan 2
  no shutdown
!
interface GigabitEthernet0/2
  switchport mode access
  switchport access vlan 2
  no shutdown
!
interface GigabitEthernet0/3
  switchport trunk encapsulation dot1q
  switchport mode trunk
  no shutdown
!
interface GigabitEthernet1/0
  switchport trunk encapsulation dot1q
  switchport mode trunk
  no shutdown
!
interface Vlan1
  description OOB Management
  ! Configured on launch
  no ip address
!
ip forward-protocol nd
!
no ip http server
no ip http secure-server
!
control-plane
!
end
```

Listing 8.24: IOSvL2 Switch configuration for *sw4*

Templates

The template for the IOSv router is specified in Listing H.62, and for the IOSvL2 Switch is specified in Listing H.63. These follow the same general approach as previous templates, with the addition of the OOB (out-of-band) management interfaces hard-coded into the template for simplicity.

Packaging

The configuration files are generated using the same approach as previous case studies. However, instead of being inserted into a *.tar.gz* archive, the generate configurations are packaged into an XML file as required by the VIRL platform. The code to do this packaging step is shown in Listing H.64.

Once this step has been completed, the final VIRL XML file can be launched on the VIRL simulation platform. In the next section we show the results of this simulation step.

8.5.6 *Simulation Results*

The below results show that both the VLAN and routing configurations have been correctly generated. They also that Spanning Tree Protocol has been correctly setup to carry VLAN information between connected managed switches, as per the trunking topology shown in Figure 8.33.

Show VLAN from switches shown in Listing 8.25 and Listing H.65. Routing tables shown in Listing H.66 and Listing H.67. The latter mapped from IPs to interfaces is shown in Listing H.68.

```
sw4#show vlan

VLAN Name                             Status    Ports
---- -------------------------------- ---------
    -------------------------------
1    default                          active    Gi0/0
2    VLAN0002                         active    Gi0/1, Gi0/2
3    VLAN0003                         active
4    VLAN0004                         active
1002 fddi-default                     act/unsup
1003 token-ring-default               act/unsup
1004 fddinet-default                  act/unsup
1005 trnet-default                    act/unsup
```

Listing 8.25: Output of SHOW VLAN on *sw4*

Traceroute

```
r11#traceroute 172.16.0.5 probe 1
Type escape sequence to abort.
Tracing the route to 172.16.0.5
VRF info: (vrf in name/id, vrf out name/id)
  1 192.168.0.98 17 msec
  2 192.168.0.35 13 msec
  3 192.168.0.129 22 msec
  4 192.168.0.2 40 msec
```

Listing 8.26: Output of TRACEROUTE from *r11* to *r7*

A traceroute from *r11* to r7 is shown in Listing 8.26, and visualised in Figure 8.39.

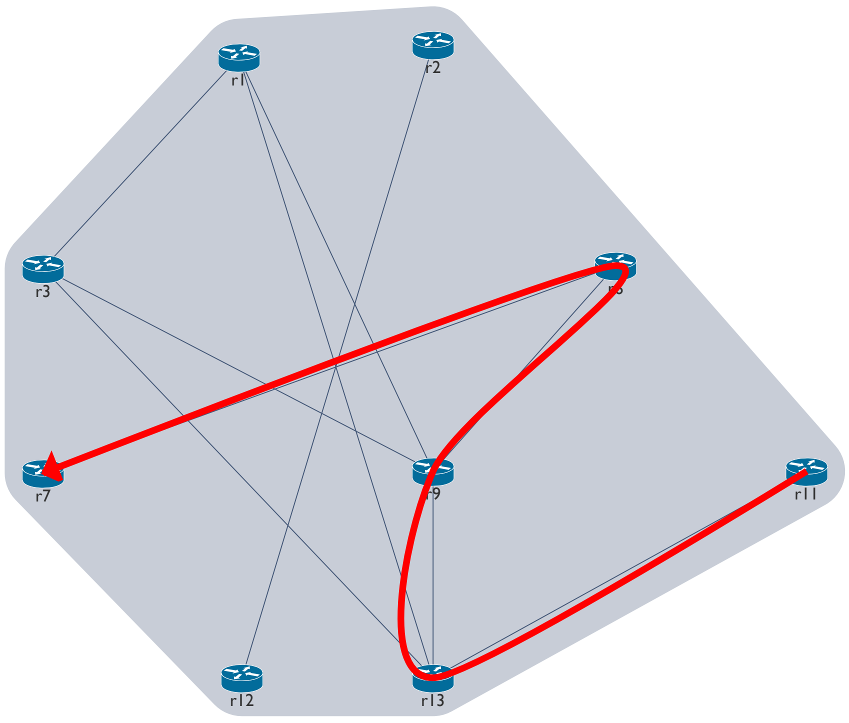


Figure 8.39: Visualisation of TRACEROUTE from r11 to r7.

If we step through the interfaces in the TRACEROUTE output, shown in Listing 8.26, and use Figure 8.38, we can see that the packet first leaves *GigE0/1.r11*, and enters *GigE0/1.sw1* on *VLAN3*.

```
GigE0/2.r13, GigE0/2.r9, GigE0/1.r6, GigE0/1.r7
```

The first hop is *GigE0/2.r13*, which is also on *VLAN3*. Therefore the packet leaves *GigE/2.sw10* and goes across the *trunk* link to *GigE0/2.sw14*. It then leaves *GigE/1.sw14* on *VLAN3*, and enters *GigE0/2* on *r13*. It then leaves *GigE0/1* on *r13* and enters *GigE0/1.sw8* on *VLAN2*.

The next hop is *GigE0/2.r9*. Therefore it leaves *GigE1/0* on *sw8* which is on *VLAN2*. It then enters *GigE0/2.r9*. The next hop is *GigE0/1* on *r6*. Therefore it leaves *GigE0/1* on *r9*, and enters *GigE1/0* on *sw10* which is on *VLAN2*. It leaves *GigE0/3* on *sw10* which is on *VLAN2*, and enters *GigE0/1* on *r6*.

The next hop is GigE0/1.r7. This is in *VLAN3* of *sw8*. Therefore the packet leaves *GigE0/2* of *r6* which is in *VLAN3*. It enters *GigE0/3* of *sw5*, and goes across the *trunk* link from *GigE0/2* of *sw5* to *GigE0/3* of *sw4*. It then goes across the *trunk* link from *GiE1/0* of *sw4* to *GigE1/1* of *sw8*. It then leaves *GigE0/3* of *sw8* on VLAN3, and enters *GigE0/1* of *r7*. The traceroute is to the *loopback0* IP address of *172.16.0.5*, which the router *r7* is announcing.

In list form this is *GigE/1.r11*, *GigE/2.sw10*, *GigE/1.sw14*, *GigE/1.r13*, *GigE1/0.sw8*, *GigE/1.r9*, *GigE/3.sw10*, *GigE/2.r6*, *GigE/2.sw5*, *GigE1/0.sw4*, *GigE/3.sw8*. A visualisation of this is shown in Figure 8.40.

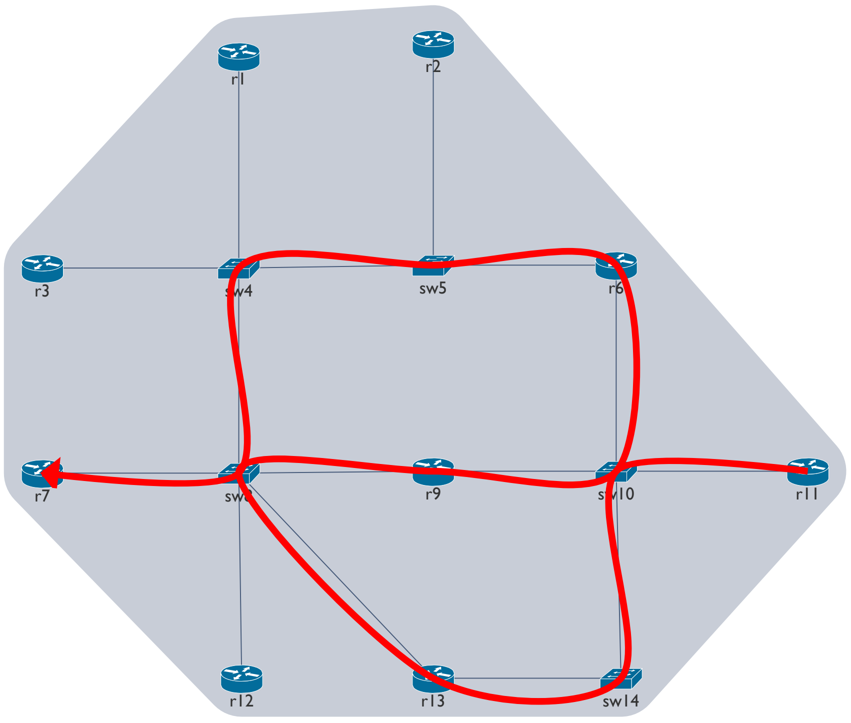


Figure 8.40: Visualisation of full path inferred from TRACEROUTE from r11 to r7, showing switches. The result is plotted on the Physical Network View.

8.5.7 *Conclusion*

In this case study we have shown how the system can be used to configure an enterprise network, using managed switches and VLANs. We have shown how the design function used in previous case studies can be adapted to this environment, with minimal changes required outside of the inherent changes required to support the VLAN configuration.

We have demonstrated how the system can be adapted to configure routers running different network operating systems, and how can be used to configure managed switches. We have also shown how the platform compiler approach can be readily adapted to support the management interface requirements, and topology description formats of a different simulation platform.

Finally, we have shown that the output of router and switch operating system commands, and the retreat example, that our system has been able to successfully configure the network that was expressed on the Network Whiteboard.

We note that this system can readily be used to automate the configuration of different topologies. For instance, devices can be placed in different VLANs by simply changing the label in the Network Whiteboard, and additional devices can be added to the Network Whiteboard, with no change required to the design functions or compilation process. This illustrates the declarative ability of our system to capture high-level network design policy.

8.6 LARGE-SCALE EXAMPLE

8.6.1 *Introduction*

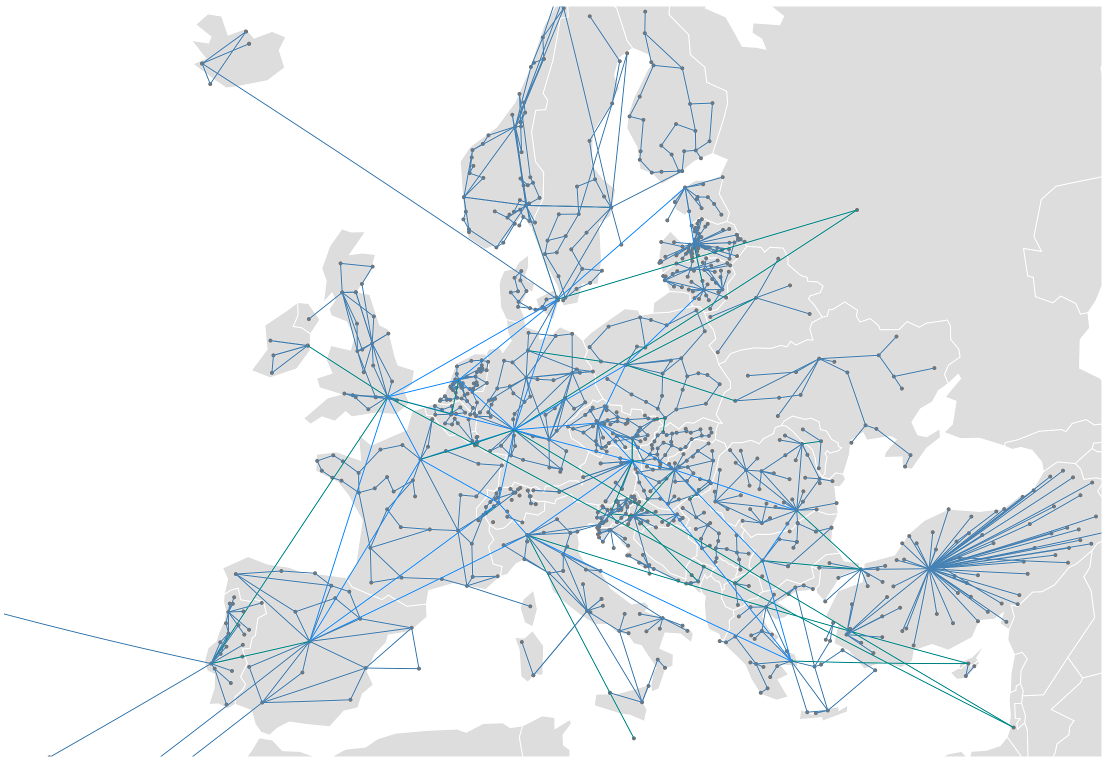


Figure 8.41: Geographic plot of NREN1400 network from Knight *et al.* [56]

Our final case study is a large-scale example. This demonstrates how the system can be used to configure large networks. Configuring networks of this size would be very time-consuming. However the use of an automated compiler, can reduce the time taken to configure such networks to seconds. Importantly, as we will discuss in future chapters, the ability to quickly generate configurations for a given input topology, lends itself well to variations for experimental what-if analysis.

For this example, we will use a large-scale topology based on the European academic research networks. This topology is constructed using topologies from the Internet Topology Zoo [53], an online repository of network topologies.

Details of the construction of this topology are outlined in the Technical Report of Knight *et al.* [56]. A summary is provided here. Individual networks in the Internet Topology Zoo have been obtained from network maps provided by network operators. These topologies have been traced, to be represented in a graph format. Typically these topologies represent city-level connectivity. For this example we make the assumption of one router per point-of-presence. This could be expanded through the use of a system such as the graph products [83] to expand out the connectivity based on a PoP template.

Individual networks in the Internet Topology Zoo from the European research networks are collated. Many topologies provide not only the internal structure, but also information on interconnectivity to other networks. As outlined in the technical report, this connectivity has been used, and combined with information collected through system such as traceroutes, to infer the Internet network connectivity of these research networks.

Table 8.2: Large-scale European NREN topology summary statistics

Network	ASN	Country	Size	Network	ASN	Country	Size
ACOnet	1853	Austria	18	LITNET	2847	Lithuania	42
AMRES	2107	Serbia	28	Malta	12046	Malta	1
ARNES	2107	Slovenia	34	MARNet	5379	Macedonia	17
BASNET	21274	Belarus	6	NIIF	1955	Hungary	35
BELnet	2611	Belgium	22	PSNC	9112	Poland	27
BREN	6802	Bulgaria	34	RedIris	766	Spain	19
CARNet	2108	Croatia	41	RENAM	9199	Moldova	3
CESNET	2852	Czech Republic	45	RENATER	2200	France	38
CYNET	3268	Cyprus	24	RESTENA	2602	Luxembourg	15
DFN	680	Germany	51	RHnet	15474	Iceland	14
EENet	3221	Estonia	12	RoEduNet	2614	Romania	40
FCCN	1930	Portugal	23	SANET	2607	Slovakia	35
FUNET	8624	Finland	24	SigmaNet	5538	Latvia	68
GARR	137	Italy	44	SUNET	2603	Sweden	31
GEANT	20965	Europe	18	SURFnet	1103	Netherlands	50
GRNET	5408	Greece	34	SWITCH	559	Switzerland	30
HEAnet	1213	Ireland	7	ULAKBIM	1967	Turkey	79
IUCC	378	Israel	10	Uni-C	1835	Denmark	22
JANET	786	United Kingdom	29	Uninett	224	Norway	64
JSCC	3058	Russia	1	URAN	12687	Ukraine	19

The topology used in this example is summarised in Table 8.2. There are a total of 1,154 devices, grouped into 40 Autonomous Systems. The autonomous system numbers of the networks have been allocated using publicly available lists of autonomous system allocations.

This results in a model containing the individual devices, the links within an AS, the links between the individual Autonomous Systems, and annotation on the devices of the Autonomous System to which they belong. This is stored in the GraphML format, which, as discussed in the previous chapter, can be imported directly to our implementation as a Network Whiteboard. This is shown in Figure 8.41.

Special Considerations

There are two main aspects of the system that we need to bear in mind with the larger topology. Due to the scale of the network being modelled, manually allocating IP addresses would be challenging. We therefore use a simple algorithm to automatically allocate IP addresses. For simplicity we allocate a class-full address block to each network.

One of the biggest performance impacts in configuring this internetwork model is in establishing the full-mesh iBGP connections within each Autonomous System. Due to the size of each of the networks, as shown in table Table 8.2 a full mesh of $O(N^2)/2$ can result in over 3,000 iBGP sessions for a network of 79 devices, such as ULAKBIM in Turkey. As we discussed in Chapter 5 this is a well-known problem in scalability in real-world networks. Approaches to solve this include the use of techniques such as confederations or route-reflectors. In this section will discuss how an automated algorithm can be used to choose the most central routers to use in a network for route reflection. An alternative approach would be to set label on the Network Whiteboard, for the network devices that we would like to use as a route reflectors.

8.6.2 *Methodology*

The methodology used to implement and verify this case study is shown in Figure 8.42. To cater for the large size of this network, we extend the IP address Design Function to allocate larger address blocks to each autonomous system. We also extend the iBGP Design Function to configure a route-reflector hierarchy instead of a full-mesh within each autonomous systems. We perform this by an algorithmic allocation of the route-reflector roles. We then use the same Netkit and Quagga compilers to measure the performance of our toolchain for a large-scale network. We do not simulate the network in Netkit, but instead compile for the C-BGP simulation platform. This uses a single configuration file which we produce by extending the C-BGP Platform Compiler to incorporate the Network Views of the Abstract Network Model. We then provide a C-BGP Template which is used by the Template Assembler to produce the C-BGP configuration file. This is loaded and run in the C-BGP Simulation. To verify the correctness of the generated configuration, we use the TRACEROUTE command, and then the C-BGP equivalents of the BGP and routing table commands: BGP SHOW RIB * and NODE SHOW RT *. We then discuss the results of this collected information.

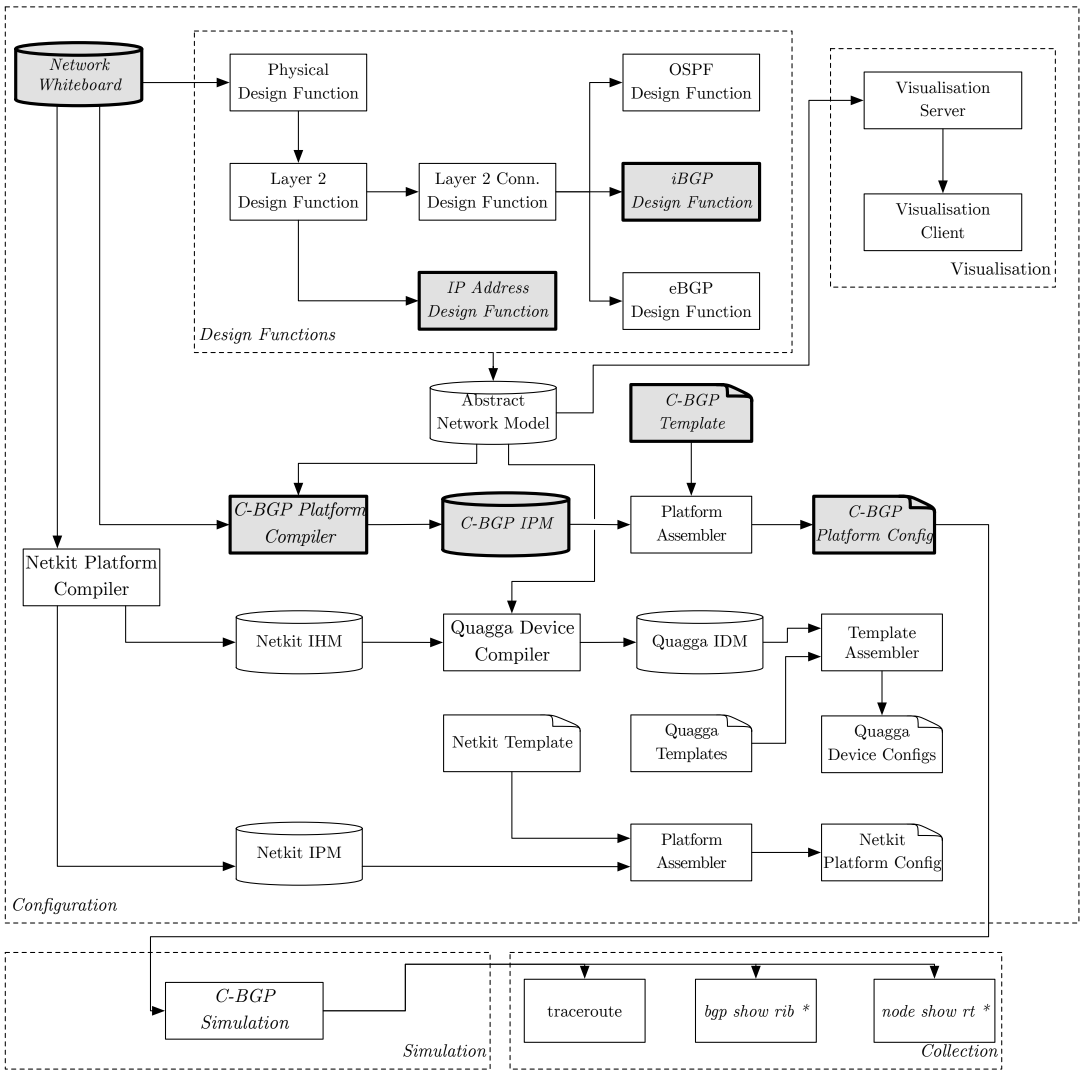


Figure 8.42: Configuration Generation and Verification methodology for Large Scale Case Study Case Study

8.6.3 *Network Whiteboard*

The procedure for the Network Whiteboard is as follows. We first load the source GraphML Listing H.69, add nodes to Network Whiteboard Listing H.70, and add edges to Network Whiteboard Listing H.71. This follows the same process as previous case studies. We then normalise the node locations Listing H.72 in order to visualise the nodes. This normalisation spaces the nodes based on a scaling factor, and allocates co-ordinates to nodes which may not have been geo-located. These co-ordinates are based on an average of the co-ordinates of the neighbours of the node based on the physical connectivity. Aside from the larger topology and the location normalisation, the process is the same as for previous case studies which had multiple autonomous systems, just that this topology has many more autonomous systems than previous case studies such as the Simplified Small Internet case study discussed in Section 8.3.

8.6.4 *Design Functions*

Physical Network View

The code to create the *physical* Network View is shown in Listing H.73. This is the same as in previous case studies. The visualisation is shown in Figure 8.43.

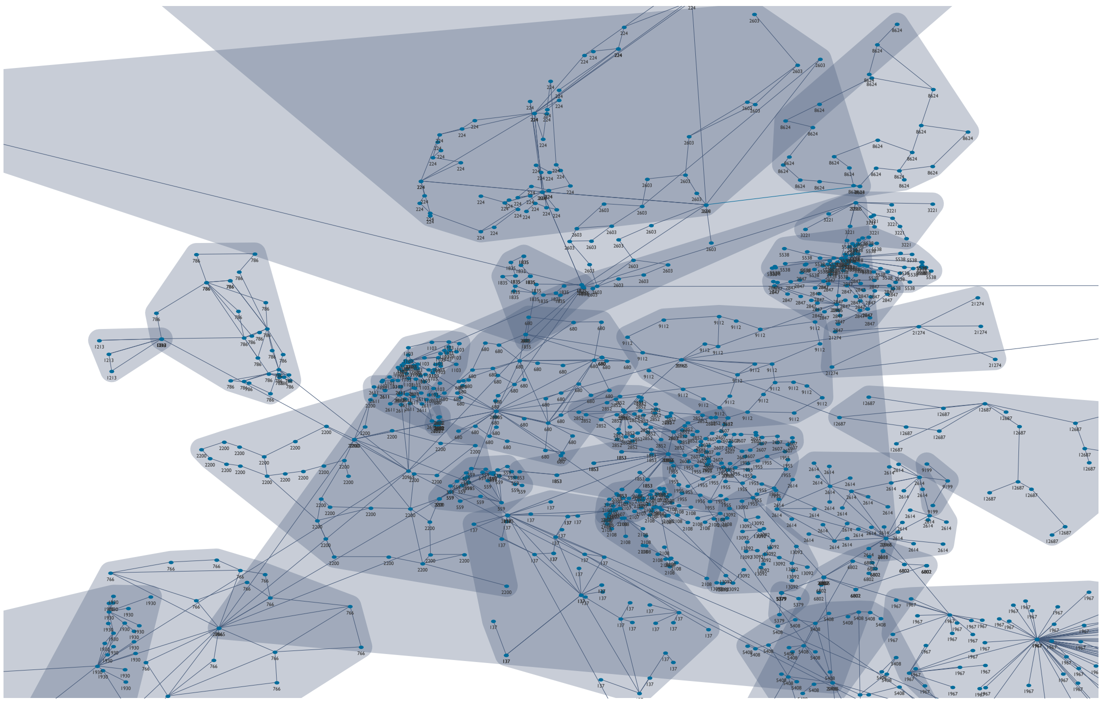


Figure 8.43: Visualisation of subset of Physical Network View, showing the *asn* attribute of each node

Layer 2 Network View

The *Layer 2* Network View described in Listing H.74, and *Layer 2 Connectivity* Network View described in Listing H.75 also follow the same Design Function logic as previous case studies.

IP Addressing Network View

Despite the much larger size of the topology, we can still use the same IP Address allocation logic used in previous case studies. As shown in Listing H.76, we iterate over each autonomous system, and allocate a */16* IP block for each ASN. We then iterate over each of the nodes (for loopbacks) and broadcast domains (for infrastructure) and assign the relevant IP address block. This shows the versatility of our approach, even for large-scale scenarios.

OSPF Network View

The OSPF Design Function, shown in Listing H.77, also follows the same approach as previous case studies, where connectivity is formed based on the physical connectivity within an autonomous system. As all links are point-to-point we use the Physical Network View rather than creating the Layer 2 Connectivity Network View, which would be identifical. An extract of the OSPF topology is shown in Figure 8.44.

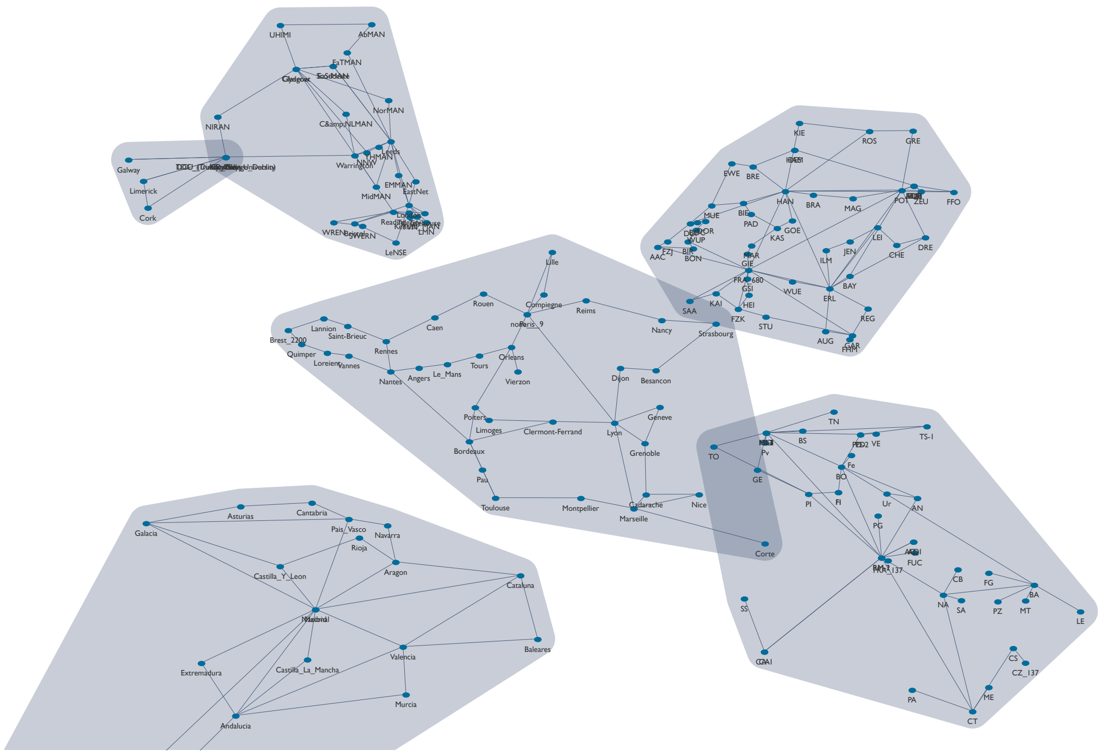


Figure 8.44: Visualisation of OSPF Network View. For visual clarity, a subset of autonomous systems is shown here.

iBGP Network View

As discussed in the introduction, we consider a route-reflector hierarchy for the iBGP topology. An extract of the iBGP topology is shown in Figure 8.45.

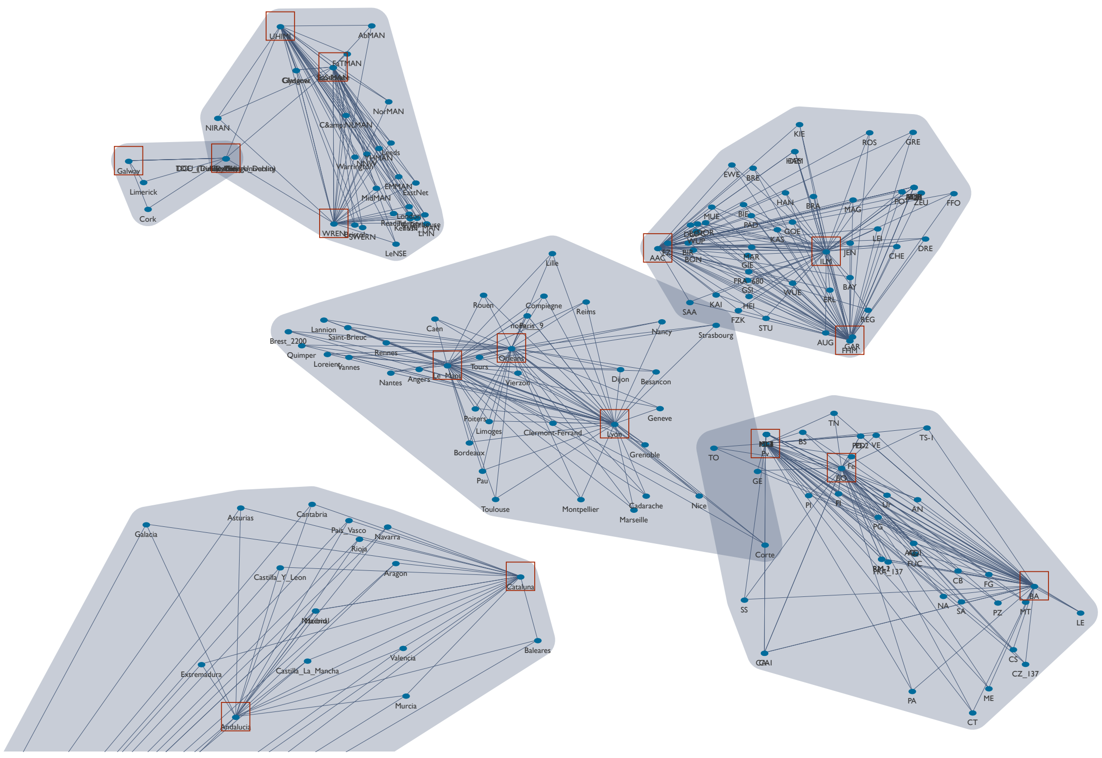


Figure 8.45: Visualisation of iBGP Network View. Nodes allocated the role of route-reflector have been highlighted in red. For visual clarity, a subset of autonomous systems is shown here.

ALLOCATING IBGP ROLES The first step is to create the iBGP Network View Listing H.78. We then use an automated algorithm to algorithmically allocate iBGP Roles Listing H.79, based on the centrality of the nodes within an autonomous system. This takes advantage of the underlying graph-based approach, and uses the BETWEENNESS_CENTRALITY algorithm from NetworkX. An alternative approach to allocate the roles could be substituted. This algorithm provides a list of nodes sorted by their centrality. We then choose the top third of most central nodes, and set their *ibgp_role* attribute to be *RR*, implying they are to perform the role of route-reflector. The other nodes are marked with the *ibgp_role* of *Client* as they will be clients of the route-reflectors.

CREATING IBGP SESSIONS We then use these designated roles to create the iBGP sessions. This follows a similar approach to creating iBGP sessions in previous examples, with the key difference being that we consider the *ibgp_role* of each node within the autonomous system. Within each autonomous system, we create two filtered sets, one for route-reflectors, and one for route-reflector clients. We then form a list of tuples for each pairing of route-reflector and route-reflector client nodes, and iterate over these pairs, adding an *up* session from the client to the route-reflector, and a *down* session from the route-reflector to the client. We also create *over* sessions between each pair of route-reflectors. These session directions are marked using the *direction* attribute on the session termination point, and is used in the template to set the "route-reflector client" configuration directive, as required. The code for the session creation is shown in Listing H.80, and the code used to highlight the route reflectors in the visualisation shown in Figure 8.45 is shown in Listing H.81.

eBGP Network View

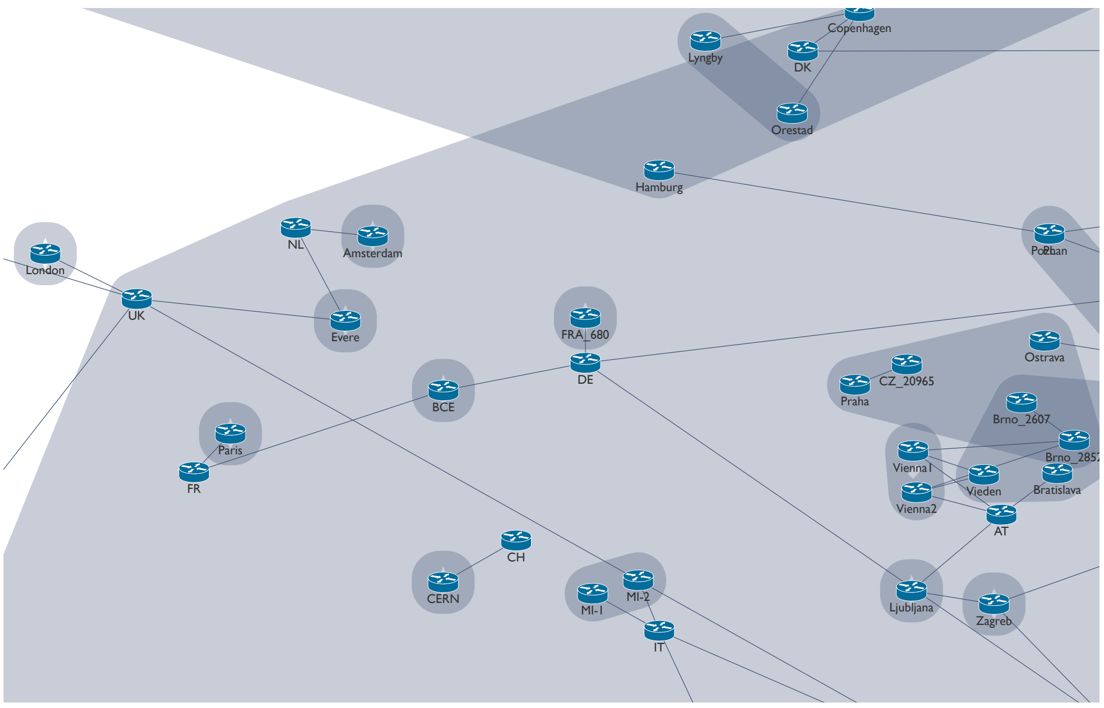


Figure 8.46: Subsection of eBGP Network View for NREN1400 topology. For clarity, non-eBGP nodes have been removed, and graphically coincident eBGP nodes have been manually moved.

An extract of the eBGP Network View is shown in Figure 8.46. The Design Function to create this Network View is shown in Listing H.82, and follows the same logic as previous case studies. Similarly, the code to specify the address prefixes to advertise over eBGP is shown in Listing H.83, and also follows the same logic as previous case studies.

Example Analysis

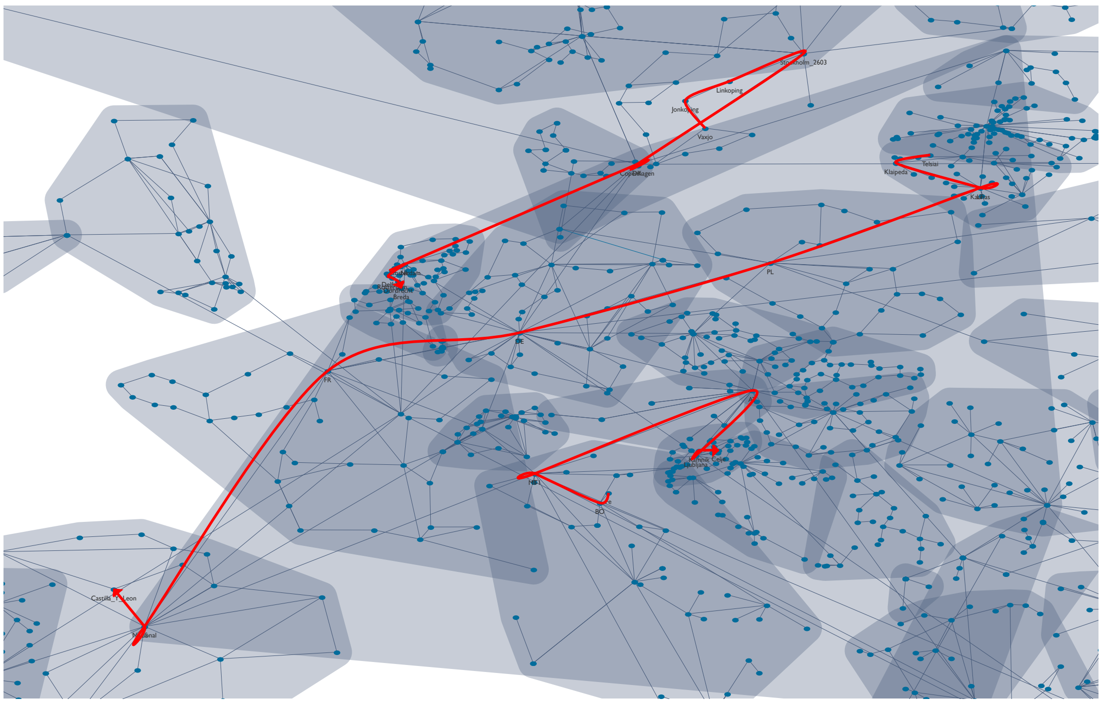


Figure 8.47: Example of paths highlighted on Physical Network View.

The visualisation in Figure 8.47 shows how the visualisation system can be used with the NetworkX shortest-path algorithm to highlight the shortest path between a given pair of nodes in the network. This can be used for offline analysis, which could later be compared to data collected from the running network. The code to produce these paths and an example output is shown in Listing H.84 and Listing H.85 respectively.

8.6.5 *Netkit*

The Netkit Platform Compiler is shown in Listing H.86 and follows the same logic as previous case studies. The Quagga device compiler is shown in Listing H.87 and also follows the same logic as in previous case studies. This demonstrates how the decoupling of the network design process from the low-level device configurations allows re-use of the compiler logic.

As shown in Listing 8.27, the system is able to generate the configurations for the Netkit platform and the Quagga devices in under ten seconds on a 2.2 GHz Intel Core i7 Retina MacBook Pro. This produces a zip archive, which when extracted, contains 3,460 directories and 8,072 files. This demonstrates the scalability of the system and approach for large topologies. Further refinements to the reference implementation described in Chapter 7 may be able to increase performance further.

```
$ sysctl -n machdep.cpu.brand_string
Intel(R) Core(TM) i7-4770HQ CPU @ 2.20GHz
$  time python nren1400.py
python nren1400.py  8.30s user 0.18s system 99% cpu 8.528 total
```

Listing 8.27: Netkit Compilation Performance

Due to the size of the topology, this topology is not launched in the Netkit environment. Instead we simulate the network using the C-BGP software package, which we discuss in the next section.

8.6.6 *C-BGP*

Introduction

Due to the size of the network, we use this opportunity to demonstrate how our framework can be used to generate configurations for the C-BGP [88] simulation software. C-BGP simulates the behaviour of routing protocols, rather than emulating the network operating systems like Netkit or VIRL. This allows for greater scalability, and can be used as a step in a workflow, where a topology is first simulated, and then emulated. This demonstrates the versatility of our approach: the same Network Whiteboard can be compiled to different targets, allowing a variety of options in a workflow.

The key difference to note in C-BGP is that nodes are identified by an IP address, and IGP and BGP sessions are established to this IP address, rather than to a physical interface such as *eth0* or *eth1*. For this example we will use the loopback interface of each device as the identifier. C-BGP also allows for a *full-mesh* to be established in a single command. We use this option, instead of the iBGP route-reflector hierarchy to simplify the configuration generation process. The approach shown could be extended to use route-reflectors in C-BGP if required. Additionally, the C-BGP configuration is expressed in a single file. We use the Platform compiler to produce this file. We do not require the use of a device compiler for this example.

C-BGP Platform Compiler

The compiler to produce this is shown in Listing H.88. It is based on the C-BGP tutorial[2], and the example configuration file[3].

The first component of the compiler is to create the physical links. This is grouped by each autonomous system, and describes the routers and the links between them. We then define the inter_domain links, which are the connections between different autonomous systems. These are based on the Physical Network View, and make use of the SUBGRAPH utility function to form the subgraph of each autonomous system. This grouping makes the configuration file easier to read.

The third step is to define the IGP protocol configuration. We base this on the OSPF Network View, again using the SUBGRAPH utility function to group by each autonomous system.

We then configure the BGP protocol. This defines three pieces of data: the routers which speak BGP, the peers of each BGP speaking router, and the networks which the BGP router advertises. As noted in the introduction, we use the *full-mesh* command to form a full-mesh in each autonomous system. Therefore, the BGP speaking routers to be configured are those that have peerings in the eBGP Network View. We then iterate over each of these routers, and list their peer routers. An example of the peers in the Intermediate Device Model is shown in Listing 8.28. The prefixes to advertise are obtained from the *networks* attribute of the node in the eBGP Network View.

```
"192.168.10.2": {
  "ebgp": [{
        "domain": 2852, "ip": "192.168.23.18",
        "next_hop_self": true, "up": true
      }, {
        "domain": 20965, "ip": "192.168.38.14",
        "next_hop_self": true, "up": true
      }, {
        "domain": 2607, "ip": "192.168.19.12",
        "next_hop_self": true, "up": true
      }
    ]
}
```

Listing 8.28: Extract of Intermediate Platform Model showing eBGP peers

2 http://c-bgp.sourceforge.net/tutorial.php
3 http://c-bgp.sourceforge.net/downloads/tutorial-simple.cli

This Intermediate Platform Model is then rendered using the template shown in Listing H.89. This template follows the example given in the C-BGP tutorial. We can see the *bgp domain full-mesh* command added to the BGP routers. We also add a static route for each eBGP peering to allow the prefixes to be reached. An extract showing the key parts of the output configuration generated is shown in Listing H.90. The total generated configuration is 7,658 lines in length, demonstrating the value of an automated approach to configuration generation.

Performance

As shown in Listing 8.29, the process of running the Design Functions, the C-BGP Platform Compilation, and rendering the configuration using the template is performed in under 6.5 seconds on a modern MacBook Pro laptop.

```
$ sysctl -n machdep.cpu.brand_string
Intel(R) Core(TM) i7-4770HQ CPU @ 2.20GHz
$  venv time python nren1400.py
python nren1400.py  6.35s user 0.16s system 98% cpu 6.606 total
```

Listing 8.29: C-BGP Compilation Performance

Simulation Results

Output from running the topology in the C-BGP simulation environment is captured, with the output of the routing table shown in Listing H.92, and the output of the BGP RIB shown in Listing H.91.

The result of a TRACEROUTE from London.JANET to Roznava.SANET is shown in Listing 8.30, with the post-processed output shown in Listing 8.31. The post-processed output shows the node and the network to which it belongs for each hop of the TRACEROUTE output.

```
cbgp> net node 192.168.6.17 traceroute 192.168.19.21
  1 192.168.38.18 (192.168.38.18)   icmp error (time-exceeded)
  2 192.168.38.9 (192.168.38.9)     icmp error (time-exceeded)
  3 192.168.38.14 (192.168.38.14)   icmp error (time-exceeded)
  4 192.168.19.24 (192.168.19.24)   icmp error (time-exceeded)
  5 192.168.19.23 (192.168.19.23)   icmp error (time-exceeded)
  6 192.168.19.29 (192.168.19.29)   icmp error (time-exceeded)
  7 192.168.19.6 (192.168.19.6)     icmp error (time-exceeded)
  8 192.168.19.17 (192.168.19.17)   icmp error (time-exceeded)
  9 192.168.19.11 (192.168.19.11)   icmp error (time-exceeded)
 10 192.168.19.3 (192.168.19.3)     icmp error (time-exceeded)
 11 192.168.19.21 (192.168.19.21)   reply
```

Listing 8.30: TRACEROUTE from London.JANET to Roznava.SANET

```
cbgp> net node 192.168.6.17 traceroute 192.168.19.21
  1 UK.GEANT (192.168.38.18)            icmp error (time-exceeded)
  2 DE.GEANT (192.168.38.9)             icmp error (time-exceeded)
```

```
 3 AT.GEANT (192.168.38.14)          icmp error (time-exceeded)
 4 Bratislava.SANET (192.168.19.24)  icmp error (time-exceeded)
 5 Sala.SANET (192.168.19.23)        icmp error (time-exceeded)
 6 Nove_Zamky.SANET (192.168.19.29)  icmp error (time-exceeded)
 7 Levice.SANET (192.168.19.6)       icmp error (time-exceeded)
 8 Zvolen.SANET (192.168.19.17)      icmp error (time-exceeded)
 9 Lucenec.SANET (192.168.19.11)     icmp error (time-exceeded)
10 R.Sobota.SANET (192.168.19.3)     icmp error (time-exceeded)
11 Roznava.SANET (192.168.19.21)     reply
```

Listing 8.31: Post-processed TRACEROUTE from London.JANET to Roznava.SANET

We also show the BGP RIB for the *DE* node of the GEANT network in Listing H.93. This shows the various prefixes learnt from different peers. A post-processed result with the prefixes mapped to the advertising network name, and host IPs mapped to their hostname, is shown in Listing 8.32. A subsection of the Physical Network View, showing the physical connectivity from the *DE.GEANT* node is shown in Figure 8.48. The directly connected physical peers such as *BCE.RESTENA* or *FRA_680.DFN* can be seen in the prefixes learnt. The other prefixes are learnt from other GEANT nodes via iBGP.

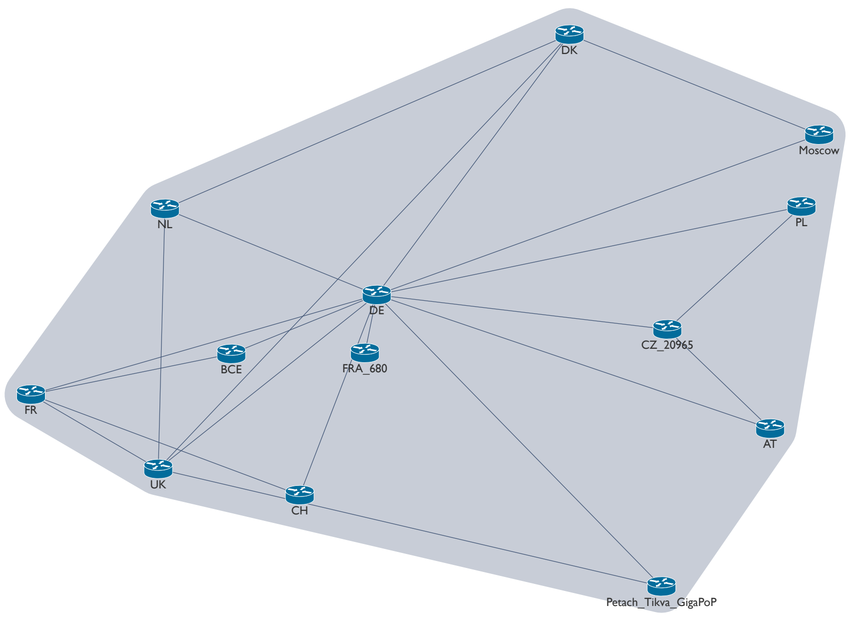


Figure 8.48: Subsection of Physical Network View showing connectivity from *DE.GEANT*

```
cbgp> bgp router DE.GEANT show rib *
*> GARR     IT.GEANT 0  4294967295  137 i
*> Uninett  DK.GEANT 0  4294967295  2603 224    i
*> IUCC     Petach_Tikva_GigaPoP.IUCC 0 4294967295  378 i
*> SWITCH   CH.GEANT 0  4294967295  559 i
*> DFN      FRA_680.DFN 0   4294967295  680 i
*> RedIris  ES.GEANT 0  4294967295  766 i
*> JANET    UK.GEANT 0  4294967295  786 i
*> SURFnet  NL.GEANT 0  4294967295  1103    i
*> HEAnet   UK.GEANT 0  4294967295  1213    i
*> Uni-C    DK.GEANT 0  4294967295  2603 1835   i
*> ACOnet   AT.GEANT 0  4294967295  1853    i
*> FCCN     UK.GEANT 0  4294967295  1930    i
*> NIIF     HU.GEANT 0  4294967295  1955    i
*> ULAKBIM  RO.GEANT 0  4294967295  1967    i
*> ARNES    AT.GEANT 0  4294967295  2107    i
*> CARNet   HU.GEANT 0  4294967295  2108    i
*> RENATER  FR.GEANT 0  4294967295  2200    i
*> RESTENA  BCE.RESTENA 0   4294967295  2602    i
*> SUNET    DK.GEANT 0  4294967295  2603    i
*> SANET    AT.GEANT 0  4294967295  2607    i
*> BELnet   NL.GEANT 0  4294967295  2611    i
*> RoEduNet RO.GEANT 0  4294967295  2614    i
*> LITNET   LT.GEANT 0  4294967295  2847    i
*> CESNET   CZ_20965.GEANT 0    4294967295  2852    i
*> JSCC     Moscow.JSCC 0   4294967295  3058    i
*> EENet    EE.GEANT 0  4294967295  3221    i
*> CYNET    GR.GEANT 0  4294967295  3268    i
*> MARNet   BG.GEANT 0  4294967295  5379    i
*> GRNET    GR.GEANT 0  4294967295  5408    i
*> SigmaNet LT.GEANT 0  4294967295  5538    i
*> BREN     BG.GEANT 0  4294967295  6802    i
*> FUNET    DK.GEANT 0  4294967295  2603 8624   i
*> PSNC     PL.GEANT 0  4294967295  9112    i
*> RENAM    RO.GEANT 0  4294967295  2614 9199   i
*> URAN     PL.GEANT 0  4294967295  12687   i
*> AMRES    HU.GEANT 0  4294967295  1955 13092  i
*> RHnet    DK.GEANT 0  4294967295  2603 15474  i
i> GEANT    DE.GEANT 0  0   null    i
*> BASNET   PL.GEANT 0  4294967295  21274   i
*> GARR     IT.GEANT 0  4294967295  137 i
*> Uninett  DK.GEANT 0  4294967295  2603 224    i
*> IUCC     Petach_Tikva_GigaPoP.IUCC 0 4294967295  378 i
*> SWITCH   CH.GEANT 0  4294967295  559 i
*> DFN      FRA_680.DFN 0   4294967295  680 i
*> RedIris  ES.GEANT 0  4294967295  766 i
*> JANET    UK.GEANT 0  4294967295  786 i
*> SURFnet  NL.GEANT 0  4294967295  1103    i
*> HEAnet   UK.GEANT 0  4294967295  1213    i
*> Uni-C    DK.GEANT 0  4294967295  2603 1835   i
*> ACOnet   AT.GEANT 0  4294967295  1853    i
*> FCCN     UK.GEANT 0  4294967295  1930    i
*> NIIF     HU.GEANT 0  4294967295  1955    i
*> ULAKBIM  RO.GEANT 0  4294967295  1967    i
*> ARNES    AT.GEANT 0  4294967295  2107    i
*> CARNet   HU.GEANT 0  4294967295  2108    i
*> RENATER  FR.GEANT 0  4294967295  2200    i
*> RESTENA  BCE.RESTENA 0   4294967295  2602    i
*> SUNET    DK.GEANT 0  4294967295  2603    i
*> SANET    AT.GEANT 0  4294967295  2607    i
*> BELnet   NL.GEANT 0  4294967295  2611    i
*> RoEduNet RO.GEANT 0  4294967295  2614    i
*> LITNET   LT.GEANT 0  4294967295  2847    i
*> CESNET   CZ_20965.GEANT 0    4294967295  2852    i
*> JSCC     Moscow.JSCC 0   4294967295  3058    i
*> EENet    EE.GEANT 0  4294967295  3221    i
*> CYNET    GR.GEANT 0  4294967295  3268    i
*> MARNet   BG.GEANT 0  4294967295  5379    i
*> GRNET    GR.GEANT 0  4294967295  5408    i
*> SigmaNet LT.GEANT 0  4294967295  5538    i
*> BREN     BG.GEANT 0  4294967295  6802    i
*> FUNET    DK.GEANT 0  4294967295  2603 8624   i
*> PSNC     PL.GEANT 0  4294967295  9112    i
*> RENAM    RO.GEANT 0  4294967295  2614 9199   i
*> Malta    IT.GEANT 0  4294967295  12046   i
*> URAN     PL.GEANT 0  4294967295  12687   i
*> AMRES    HU.GEANT 0  4294967295  1955 13092  i
*> RHnet    DK.GEANT 0  4294967295  2603 15474  i
i> GEANT    DE.GEANT 0  0   null    i
*> BASNET   PL.GEANT 0  4294967295  21274   i
```

Listing 8.32: C-BGP bgp show rib result for DE.GEANT node. The first column shows the network name for the prefix, and the second column shows the router the prefix was learnt from.

8.6.7 *Conclusion*

In this case study we have shown how our system is able to handle large-scale topologies. We have also demonstrated how our system can be adopted for the use of the C-BGP simulation platform. As we will discuss in Section 9.5.2, the ability to target multiple platforms from the same Network Whiteboard description can be used as part of an engineering design and testing workflow.

8.7 system adoption

8.7.1 *Introduction*

In this section we describe the use of the system in both research and industry. Adoption by industry is an important validation for research which focusses on an approach which is acceptable and usable by a substantial number of network researchers and practitioners.

8.7.2 *Use in Research*

AutoNetkit was used to automate the deployment of the virtual network setup for the Content Delivery Network emulation system presented as a demonstration at the SIGCOMM 2013 conference [85]. This involved extending the AutoNetkit platform to allow the configuration of Linux hosts representing HTTP servers (acting as traffic sources) and end hosts (acting as traffic sinks).

8.7.3 *Use in Industry*

Junosphere

An earlier version of AutoNetkit was used to automatically generate topologies for the Juniper Junosphere simulation environment [5].

VIRL

AutoNetkit has been used as a component of the VIRL platform, [79], a commercial product from Cisco with approximately 10,000 [78] users of the Personal Edition, with additional users in the Cisco Modeling Labs enterprise version. This allows us the opportunity to include feedback from real-world network practitioners regarding the abstractions and transformations that we have presented in this thesis.

extensions AutoNetkit has been extended in the VIRL platform in a number of aspects. These demonstrate the flexibility of the approach presented in this thesis. We provide an overview of these extensions below.

The *.virl* XML topology format allows the description of a labelled network topology, where nodes, interfaces, and the topology itself can be labelled. This acts as a Network Whiteboard input to the configuration process. It also supports writing of the generated configurations back to the .virl file.

The Design Functions have been extended to support the *OSPFv2*, *OSPFv3*, *RIP*, *IS-IS*, *EIGRP*, *iBGPv4*, *eBGPv4*, *iBGPv6*, *eBGPv6*, *MPLS LDP*, *VRFs*, and *VPN* protocols. The IP address allocation supports both *IPv4* and *IPv6* addressing.

The Device Compilers have been extended to support a number of different device operating systems, including the *IOSv*, *CSR1000v*, *IOS-XRv*, and *NX-OSv* routers; the *ASAv* firewall; and the *IOSvL2* managed switch. *Linux* operating systems are bootstrapped using configurations generated for the cloud-init initialisation system.

The collection and data analysis framework has been extended to collect a variety of diagnostic information from running network devices, and form the appropriate Network Views. This allows the processing and display of protocols such as *OSPF*, *IS-IS*, *BGP*, *Spanning-Tree* and *VLANs* from a variety of device types, together with automated visualisation of traceroute results.

This demonstrates how the underlying open-source framework can be extended to be reliably used in production for a variety of protocols and platforms, for both configuration generation and data collection and analysis.

USER FEEDBACK In this section we provide a number of quotes regarding real-world users of AutoNetkit as part of the VIRL platform. These offer insight into the value of a high-level approach to automating the process of generating low-level device configurations.

In a 2016 tweet, Ethan Banks [33] writes:

> Reading up on IS-IS today, as well as watching @CiscoVIRL tutorial videos. AutoNetKit is nice. Building basic configs by hand goes away. (Banks [33])

In a 2013 blog post Pepelnjak [24] reviews AutoNetkit in CML:

> Having automatically generated initial configurations with IP addresses and reasonably configured routing protocols is awesome. Whenever I wanted to do a quick test of a new IOS feature in the past, I spent more time creating the initial router configurations (or browsing through previously created topologies and configurations, trying to figure out which ones would closely match my current requirements) than doing the actual tests (I even wrote a Perl script to create the configurations). (Pepelnjak [24])

Pepelnjak continues:

> The ability to create a topology on the fly and get a running network in a few minutes is priceless. (Pepelnjak [24])

In a blog post in 2014, Burke [14] describes:

> It is called Autonetkit. It takes a set of predefined variables and injects them into the environment. I will give you a glimpse of how I can cut down a massive amount of time configuring a base environment. When learning a new feature within BGP for example and you're setting up GNS3 or another environment you can spend a lot of time setting up an environment. There have been many times where I may have had 90 minutes of study time. 25–30 minutes may have been thinking of a topology or addressing scheme and configuring that. Whilst not a hard task it is time that gets sucked away. Now with this feature you can have the core infrastructure up and running very quickly and focus on deploying BGP communities, MP-BGP or working with advanced functions. (Burke [14])

Burke continues:

> . . . it already has saved me 30 minutes per study sessions since I picked it up. That was 25 minutes more time at the CLI working on technologies opposed to simply inputting addressing (Burke [14])

In a forum post describing VIRL, Maker [65] writes:

> Autonetkit. This is probably the key thing (that and layer2 and ASA) that keeps me interested in VIRL. You can autoconfigure your routing protocols. This is a huge time saver. I can put, say, 9 iosv devices on a map and use the selector to grab 3 at a time and say run EIGRP for these three and provide some parameters, select another 3 and say I want OSPF for these three, etc. Autonetkit is really the star of the show with VIRL. (Maker [65])

In his 2016 book *The VIRL Book*, Wang [109] writes:

> One of the best features that Cisco VIRL offers is the ability to automatically configure basic IP addressing, L3 routing, OSPF, BGP and VRFs on all the simulated nodes when they are first launched. In other words, you have a "converged" network as soon as the network is up. This feature is called AutoNetkit. (Wang [109])
>
> This feature is particuarly useful for network engineers who want to test a configuration or new features in an IOS image. It normally takes hours just to get a baseline network working, running multi-protocolls, BGPs in a fairl complex environment. With AutoNetkit, you can have a fully routed network up in minutes. (Wang [109])

8.7.4 *Conclusion*

This section has validated the first research question presented in Section 3.2. We have made the proposed declarative high-level policy representation and associated compilation steps available to research and industry practitioners. As we have shown in this section, the system has been able to be extended for use in research, and industry feedback has demonstrated that network practitioners can readily understand and recognise the benefits of the high-level policy representation.

8.8 CONCLUSION

In this chapter we have presented four case studies which validate three key aspects of the approach presented in this thesis. Firstly, we have reinforced the claim that the approach can generate correct device configurations. Secondly, we have shown how the approach can be extended to different network designs, emulation platforms, and target devices. Finally, we have shown how the approach and tooling is suitable for use in generating configurations for large-scale networks.

In this chapter we have shown how our approach and toolchain addresses Research Question 5: *What are the scalability and extensibility characteristics for the compilation of High-Level Network Configuration Policy to device configurations in terms of network size and diversity of network protocols and target devices?*

Part IV

FUTURE WORK AND CONCLUSIONS

9

FUTURE WORK

9.1 INTRODUCTION

In this section we provide a brief overview on possible future directions which could be used to extend the approach presented in this thesis. We present these grouped by the themes discussed in Chapter 4, Chapter 5 and Chapter 6. We also discuss how the system as a whole could be expanded. These expand on each of the Research Questions we outlined in Chapter 3, covering declarative representations of High Level Network Policy, graph-based network configuration intermediate representations, the transformations of these, expanding the configuration generation from the intermediate representation, and increasing the scalability and demonstrating the expandability of the approach and toolchain presented in this thesis.

9.2 SPECIFICATION AND REPRESENTATION USING NETWORK VIEWS

9.2.1 *Network Element Paths*

The current Network Whiteboard and Network Views contain Network Elements, Network Element Interfaces, and Network Element Connections. This could be expanded by allowing the specification and representation of **Network Element Paths**, as shown in Figure 9.1. They can be defined based on Network Elements as shown in Figure 9.2 or with Network Element Interfaces as shown in Figure 9.3

Network Element Paths can be used to represent explicit paths through the network, such as for traffic engineering, where traffic must traverse a specific series of Network Elements and/or Network Element Interfaces. Network Element Paths consist of the following:

1. A **Network Element Path Source**, either a source Network Element, or an optional source egress Network Element Interface.

2. A **Network Element Path Destination**, either a destination Network Element, or an optional destination ingress Network Element Interface (optional)

3. Zero or more **Network Element Path Hops**, where each Network Path Hop consists of a Network Element. Optionally, a Network Path Hop may include a Network Path Hop *ingress* Network Element Interface and/or a Network Path Hop *egress* Network Element Interface.

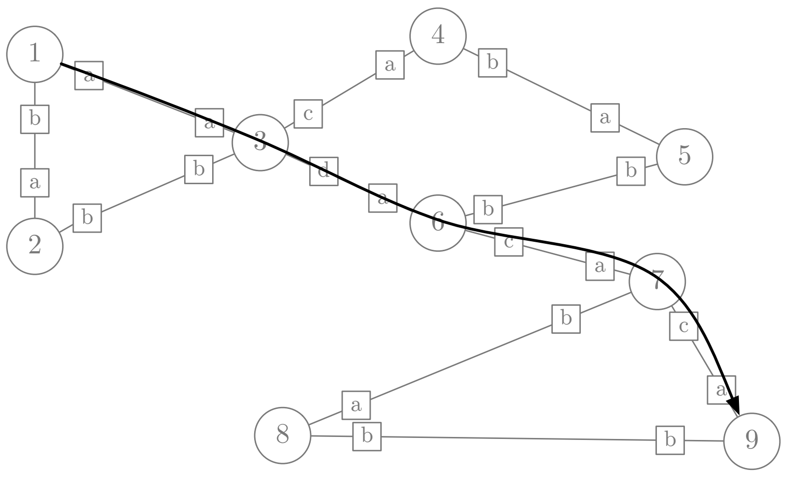


Figure 9.1: Example Network View showing a path through the Network View, from Network Element 1 to Network Element 9 via Network Elements 3, 6, and 7.

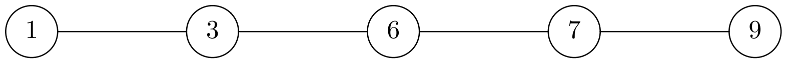


Figure 9.2: A Network Element Path specified using Network Elements.

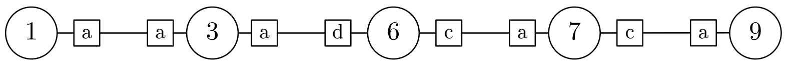


Figure 9.3: A Network Element Path specified using Network Elements and both Ingress and Egress Network Element Interfaces.

9.2.2 *Label Sets and Network View Schema*

We could formalise the approach of using labels. Currently labels are free-form in that no restrictions are placed on their setting by the

Design Functions. This could be refined by placing a schema on the label keys. For instance label keys could be restricted by the role of a Network Element, or its capabilities.

Network Element Capabilities are dictated by the target device to which the Network Element corresponds. They are constrained by the physical and logical capabilities of the target device. Some devices may support some features that others may not. For instance a modern router may allow for configuration of both IPv4 and IPv6, whilst an older router may only support IPv4. Another example is some routing protocols are only supported by certain vendors. For instance EIGRP was for a long time a proprietary routing protocol only available on Cisco devices. These constraints can also be captured in the capabilities, which are shown in Figure 9.4. In the examples provided in this thesis, we use a limited set of labels to indicate such capabilities. This could be extended to provide a strict alphabet of available labels depending on the Network Element.

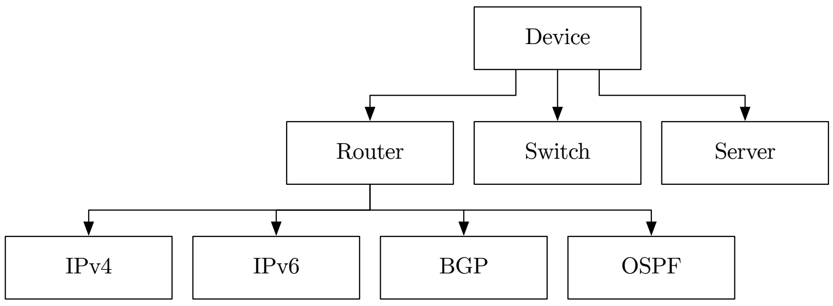


Figure 9.4: Network Element Capabilities

This could be extended further to introduce a schema-based approach like that discussed in the YANG work such as the "Yang Data Model for OSPF Protocol" by Yeung *et al.* [111].

9.2.3 *Higher Level Network Descriptions*

We could extend the labelled graph-based Network Whiteboard further by allowing a declarative grammar that is pre-processed with the Network Whiteboard to allow even higher level expression of design policy, or BGP policy. The Network Whiteboard then becomes an intermediate representation, with a higher-level language allowing description of business and network design policies which are then translated to the Network Whiteboard, and then through the system, as shown in Figure 9.5.

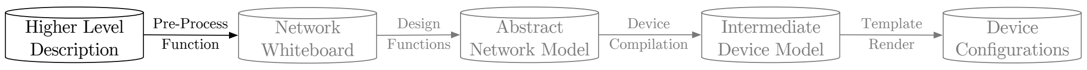
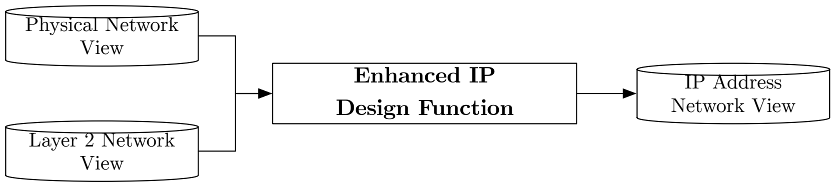


Figure 9.5: Approach with a higher-level grammar used to construct the Network Whiteboard.

9.3 NETWORK VIEW TRANSFORMATION

9.3.1 *Design Functions*

IP Addressing



So far we have used the simplified IP addressing algorithm discussed in Section 5.6.3, where we allocate a fixed sized block to each autonomous system for each of infrastruture and loopback addresses. This was continued in the case studies in Chapter 8, where we allocated a /24 IP block to each autonomous system. This was expanded to a /16 for the Large-Scale case study in Section 8.6. For each point-to-point broadcast domain Pseudo Network Element, we allocated /30, which was then iterated over to allocate individual IP addreses to the connected Network Element Interfaces. For the switches in the Small Internet Complete case study in Section 8.4 we used the static IP address allocation specific on the Network Whiteboard. Finally, for the VLAN case study in Section 8.5 we allocated as /27 to each broadcast domain Pseudo Network Element which arose from the VLAN Network View. This was known in advance to be of sufficient size given the topology.

However, these static allocations cannot be applied to larger topologies, or those with higher degree broadcast domains (a /27 allows for 30 host IP addresses). For this we require a more flexible IP allocation approach, which takes into account the degree of each broadcast domain Pseudo Network Element. From this we can build an IP addressing tree which determines the appropriate size block to allocate.

An example implementation of such an IPv4 allocation algorithm in AutoNetkit can be seen in the ipv4.py[1] module. This constructs a tree for each autonomous system based on the degree of each collision domain. An example for where loopbacks and infrastructure addresses are allocated from the same address block is shown in Figure 9.6, where a /27 block is allocated to the autonomous system, and split into two /28 blocks. One of these blocks is broken into two /29 blocks, with one allocated to a switch, and the other used for the /32 loopback of a host. The other /29 is broken into /30 blocks, and one is allocated to a point-to-point broadcast domain (arising from a collision domain hence the *cd_* prefix). For the other autonomous system, the autonomous system is allocated a /28 prefix, which is then allocated as /30 blocks as shown in Figure 9.7.

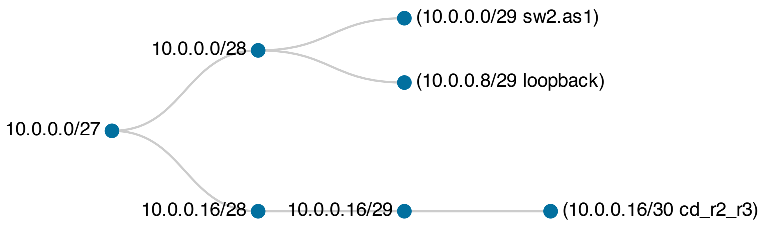


Figure 9.6: IP Addressing tree for an example AS1.

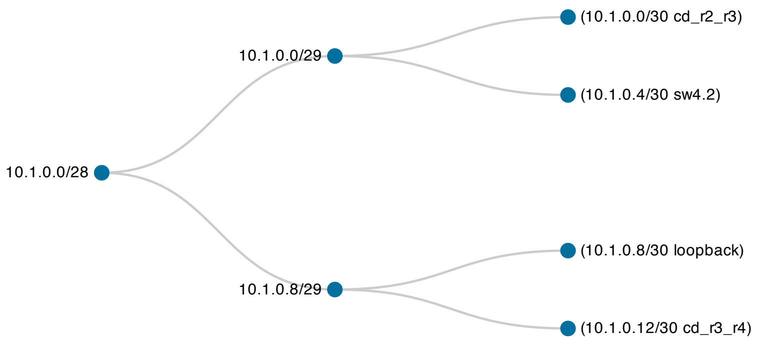
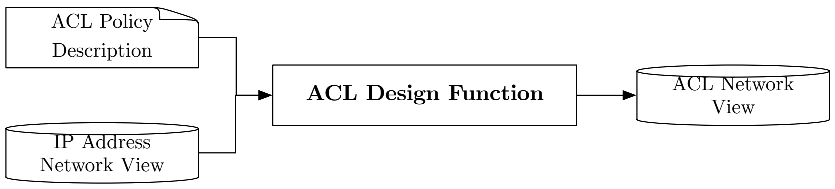


Figure 9.7: IP Addressing tree for an example AS2.

Our approach to IP addressing here can also be extended to IPv6 addressing. A simplified IPv6 allocation allocation algorithm can be seen in the ipv6[2] module.

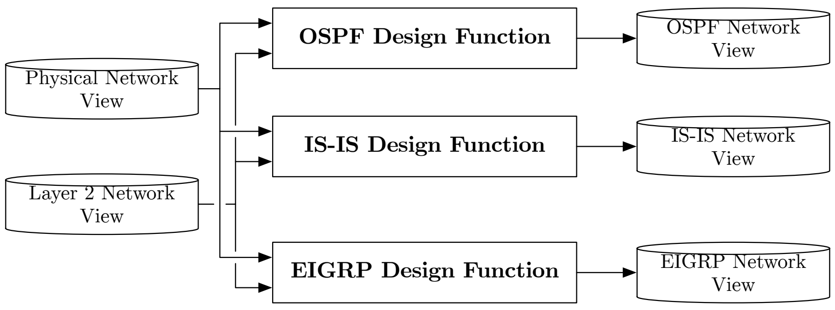
For more advanced IP addressing allocation approaches, we refer the reader to the work of Duerig *et al.* [26].



Access Control Lists

We provide a brief overview using the framework to configure access control lists, either on routers or on dedicated firewall devices. Access control lists filter data plane connectivity based on sets of IP address prefixes. Through the use of Design Functions, Network Elements, and the prefixes on them, can be collated using selection operations based on the labels on these elements. In the Network Whiteboard, these enterprises could either be labelled with a label, or the egress interface of the router or switch which connects to these devices, could be Labelled. This label could then be using a query operation, after the IP address allocation, to create a list of the prefixes which belong according to these criteria. From this, a list can be created on the Network Element which is to realise the access control list. We provided a simple example of this approach in the formation of prefix-lists in Section 8.4. We describe a more generalised approach based on abstract syntax trees in Section 9.4.2.

Multiple IGPs



We also note, that this approach can be used with a preprocessor step, to allocate a specific IGP to a specific autonomous system. This preprocessor step, is given a mapping of autonomous system number to the IGP label to set. It then maps the IGP value appropriate given this mapping. This approach is discussed further in Chapter 8 in the large-scale case study.

1 https://github.com/sk2/autonetkit/blob/thesis16/autonetkit/plugins/ipv4.py
2 https://github.com/sk2/autonetkit/blob/thesis16/autonetkit/plugins/ipv6.py

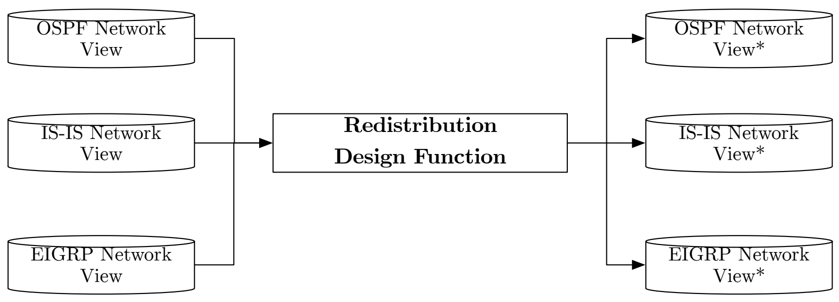


Figure 9.8: Workflow for Redistribution Design Function

In some scenarios, a network designer may wish to run more than one interior routing protocol in a network. This can be performed by generalising the routing protocol demonstration of a provided so far, of the OSPF routing protocol. So far we have discuss configuring all routers which belong to the same autonomous system, and have a direct physical connection, as having an adjacency in the OSPF Network View.

To generalise this, we can consider the examples described previously as having an implicit default label, which sets the IGP label to being OSPF. We now define this IGP label is being used to determine which interior Gateway protocol to run on that Network Element. If this label is not provided, then we will default to a label of OSPF.

The workflow for this process is shown in Figure 9.8. We note here that this Design Function modifies an set of existing Network Views, rather than creating a new Network View.

For this example we will consider the topology shown in Figure 9.9.

In order for multiple interior gateway protocols to run within a single autonomous system, the process of route redistribution needs to be considered. In particular, it is important to consider the running multiple independent router protocols within the network, and skewers topology information. Network designer must therefore consider the interactions between these protocols, and apply careful network design, to avoid routing loops from occurring. Such algorithms are beyond the scope of this basis, but we refer the reader to the discussion in Edgeworth *et al.* [28].

In order for router to exchange information between different writing protocols, the router must run multiple protocols. An example provided in Edgeworth *et al.* [28] shows an example of route redistribution. In this example, routers *R1*, *R2*, and *R3* run the EIGRP routing protocol, and routers *R3*, *R4*, and *R5* run the OSPF routing protocol. It can be seen that router *R3* runs both EIGRP and OSPF, and therefore the configuration of this router needs to consider the

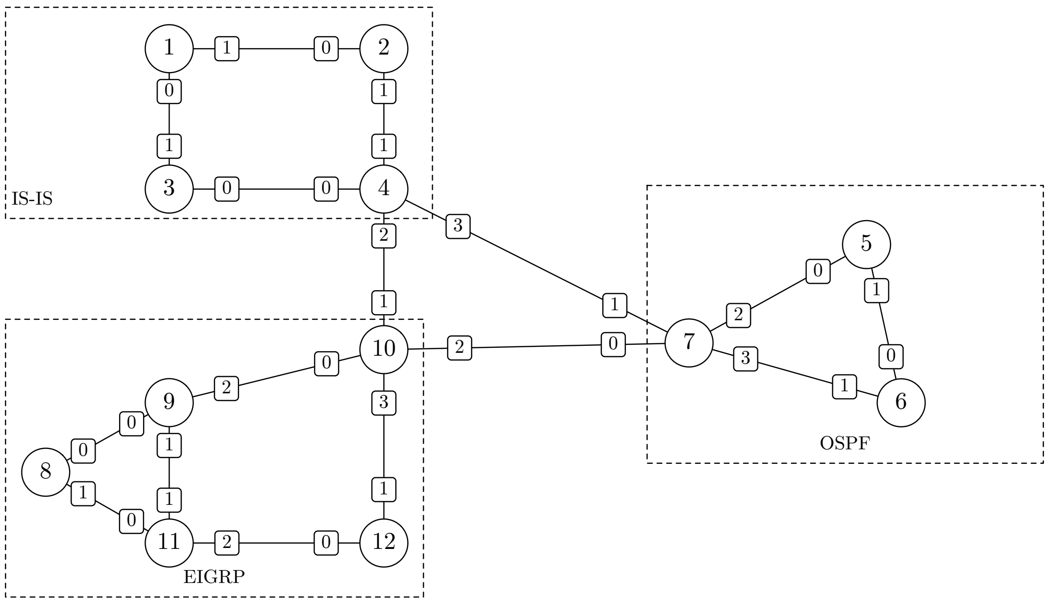


Figure 9.9: Topology showing multiple IGPs

sharing of information between the two protocols, for connectivity between *R1* and *R2*, and *R4* and *R5*.

The simplest form redistribution, would be to handle this and the compiler. If a Network Element participates in multiple interior gateway router protocol Network Views, that redistribution command can be added for each riding protocol it belongs to, to redistribute to the other protocols. However this shifts limitation into the compilers, and removes the ability to perform policy at the Network View design level.

An alternative approach, is to have a redistribution Design Function, which runs after the individual routing protocol Design Functions. This redistribution Design Function then sets the appropriate labels on the writing protocol Network Views, according to the redistribution policy of the network. The compilers can then look at these labels, and apply the appropriate policy of the network designer.

Multi-Level iBGP Hierarchy

The iBGP scalability approach described in Section 5.7.3 can be extended further to introduce a two-level hierarchy. For this example we will consider the topology shown in Figure 9.10. In this example we use a new label for route reflection to indicate the hierarchy. This label, *T*, denotes a router as being the top level route-Reflector within an autonomous system. We have also introduced a label to denote the grouping of route-reflectors at the first level of hierarchy. This label denotes a *cluster* and is shown in the figure by the dashed boxes around the sets of routers.

We then construct the Network Element Connections according to the following rules. We first divide the Network Elements into sets according to their autonomous system label, as before. We then divide these sets further, according to the cluster label. We then look at each individual cluster. We apply the same rules as before: Clients only connect to Route Reflectors, and Route Reflectors connect to other Route Reflectors.

In this example, the Network Elements in Cluster A are *1*, *2*, *3*, and *4*. Within this cluster, 1 and 2 are denoted Route Reflectors through the *R* label. *3* and *4* are Clients denoted by the C label. Thus we create Network Element Connections *(3, 1)*, *(3, 2)*, *(4, 1)* and *(4,2)*. We also create Network Element Connection *(1, 2)* between the Route Reflectors. The Network Elements in Cluster B are *7*, *8*, *9*, and *10*, where *7* and *8* are *Route Reflectors*, and *9* and *10* are *Clients*. We thus create Network Element Connections (9, 7), (9, 8) and *(*10, 7)*, (*10, 8)*.* We also create a Network Element Connection *(7, 8)* between the *Route Reflectors*.

We can now consider the hierarchical case. For this, we ignore the cluster label, as this relates to the two level hierarchy, in how we establish Network Element Connections between the R and the *C* routers. The top level, we only consider the autonomous system label, and the routers, labelled as T or R. We then follow a similar approach in creating Network Element Connections, as we did in the previous case. We create a client relation between *R* and *T* routers, and a peer relation between *T* routers. In this example, Network Elements *1*, *2*, *7*, and *8* are denoted *R*, and *5* and *6* are denoted *T*. We therefore create Network Element Connections *(1, 5)*, *(1, 6)*, *(2, 5)*, *(2, 6)*, *(7, 5)*, *(7, 6)* and *(8, 5)*, *(8, 6)*. We also create the peering session *(5, 6)*.

This example has shown how successive refinement of multiple labels can be used to introduce grouping. This use of labels for grouping and to denote roles, scales according to the required design policy. This can group Network Elements in multiple different manners, which differs to the more linear approach of an object-oriented Abstract Network Model. The use of labels allows more complicated policies to be expressed, and also allows for verification using verification Design Functions.

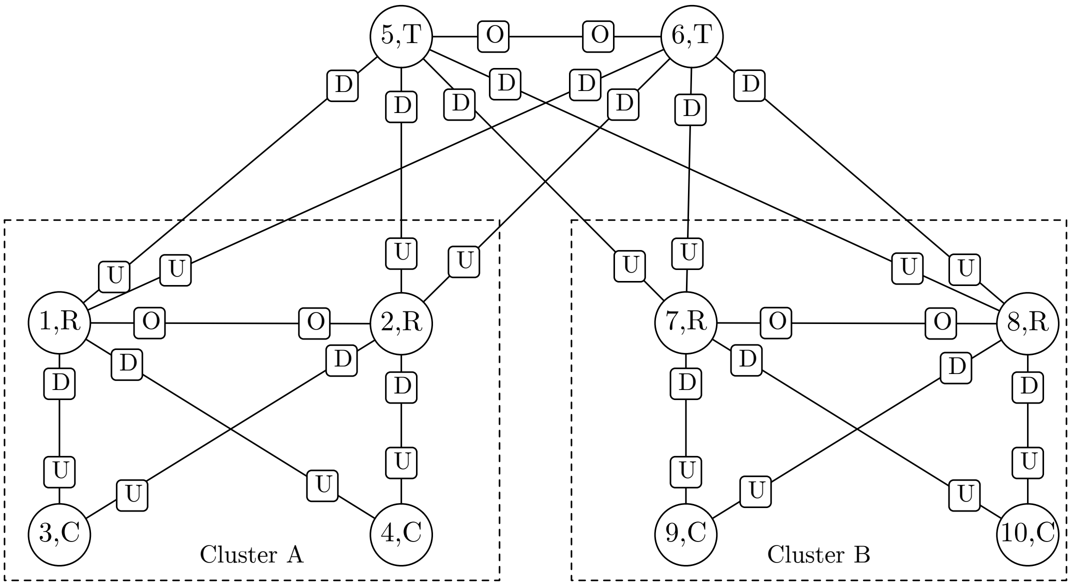


Figure 9.10: Topology showing two-level Route Reflector hierarchy

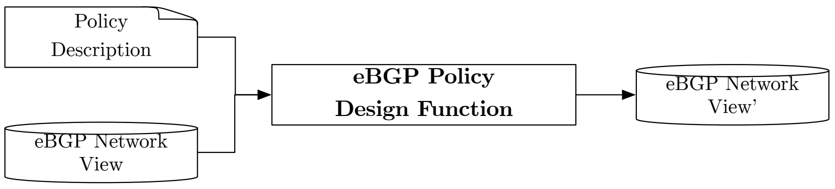


Figure 9.11: Workflow for BGP Policy Design Function

BGP Policy

In this section we provide a brief overview of using the framework to represent BGP policy. Routing Policy can be applied to the ingress or egress of a BGP session, or on the BGP process itself. This could be represented in our approach on the Network Views on the Network Element Interfaces. This would required abstracting the Network Element Interface away from representing the physical network interface, and extending the idea of a logical Loopback interface, which can terminate BGP sessions. Routing Policy typically applies to a group of devices, such as an autonomous system. To avoid repetition, the relationships could be described using a Policy Graph Model, where the nodes represent the device groups. This can then be transformed by the Design Function to map the policy onto the appropriate Network Elements.

Alternatively, this Policy Graph Model could be applied as a preprocessor step to the Network Whiteboard, to label the Network Elements as appropriate. Since the Network Views are constructed from the Network Whiteboard, the Design Functions on the BGP Network View can operate on the same information in either approach, whether it is applied to the Network Whiteboard, or applied at a later stage to the Network View. An example Policy Workflow is shown in Figure 9.12. An application of the policy relationships described in Figure B.1 is shown in Figure 9.13.

A more advanced version would use more complex data structures, and map to an Abstract Syntax Tree for traditional language transformation techniques to be used. We provide an overview of this approach in Section 9.4.2.

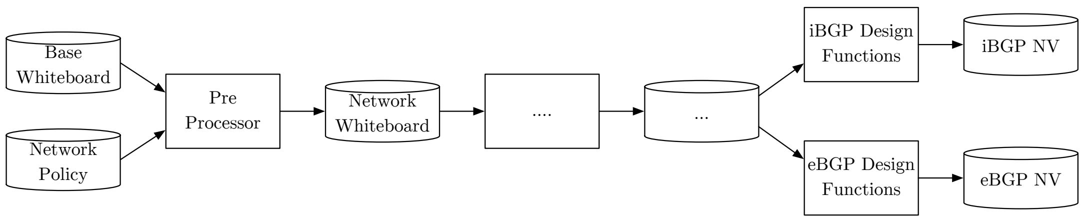


Figure 9.12: Routing Policy Workflow

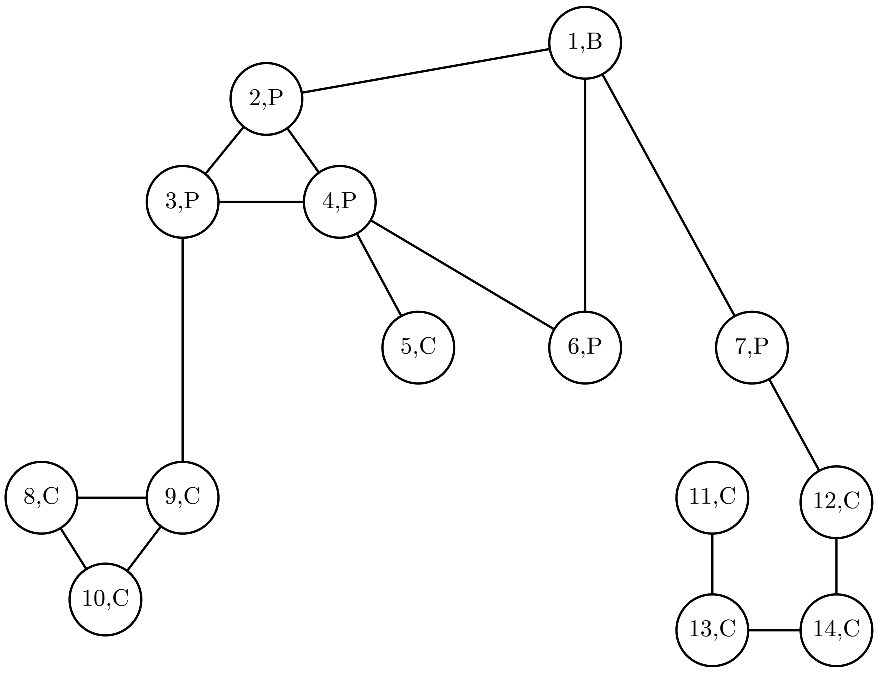
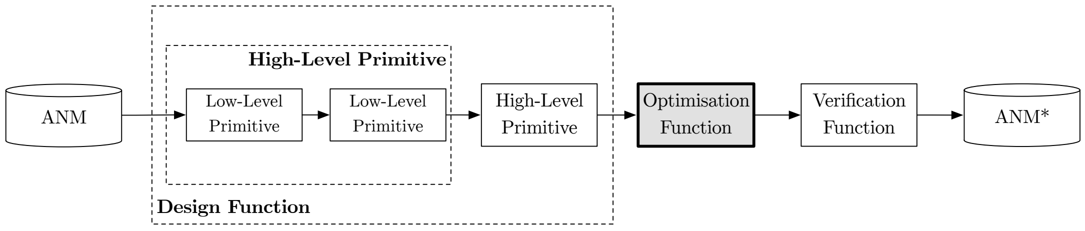


Figure 9.13: Routing Policy Labels applied to Network Elements. B represents Backbone, P represents Provider, and C represents Customer

9.3.2 *Optimisation Functions*



Optimisation functions can take two forms. The first is allocating labels according to an optimisation algorithm, such as setting OSPF costs to perform traffic engineering (such as preferring a certain primary link over a more expensive backup link), such as Fortz *et al.* [36]. An example of a workflow to perform this is provided in Figure 9.14.

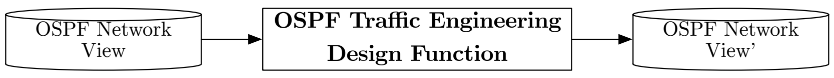
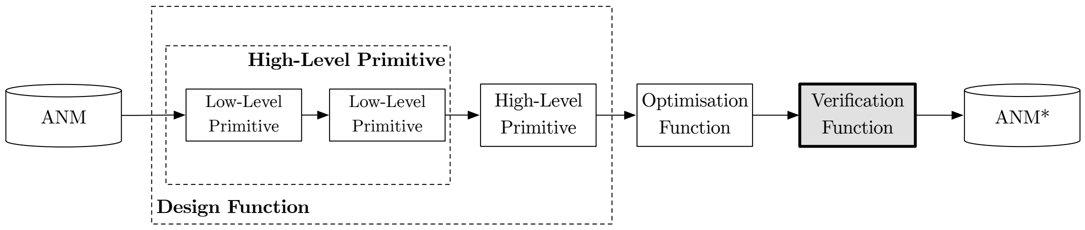


Figure 9.14: Workflow for OSPF Traffic Engineering Design Function

The second form is in modifying the structure of a Network View produced by a Design Function. Generally these optimisations would be better performed through the use of an enhanced Design Function, which incorporates both the initial Design Function to produce the Network View, and then applies the optimisation. This would ensure that the Network View produced by a Design Function is that which is used in subsequent Design Functions and in the compilers.

9.3.3 *Verification Functions*



Verification Functions are a extension of Design Functions. Rather than creating or modifying a Network View, they return a boolean value as to whether the Network View meets a set of verification criteria. This is analogous to *static checking* in Compiler Construction, where errors can be detected and reported during the View Generation process.

Such Verification Functions use the same primitives as Design Functions, but do not modify the Network View. The output of a Verification Function is a Boolean *true* or *false*, and can be expanded for optional messages such as Debug, Informational, Warning, or Exception.

Another example is to check the consistency of IP address allocations, such as all IP addresses in the same broadcast domain belonging to the same subnet. An example of verification functions using AutoNetkit is shown in Listing I.1.

An example of a simple Verification Function is checking that a Network View is fully-connected: that there are no partitions. This could be computed directly using a graph algorithm, such as the connected components algorithm discussed previously. If the number of connected components in a Network View is *one* then the graph is not partitioned. If the number of connected components is greater than one, then the graph is partitioned, which may indicate a problem, depending on the context of the Network View. Another form of verification is because a Network Element Interface can be bound to more than one Network Element Connection. This can arise in cases such as when layer 3 devices are connected to a Layer 2 device such as a switch. We discussed such an example in Section 5.7.1.

In this case the resulting topology for a routing protocol can result in multiple Network Element Connections being bound to the same Network Element Interface. In some situations, this can be valid but in other may be invalid. For instance, in the OSPF routing protocol a router cannot have more than one interface in the same broadcast domain. We can write a verification function to check this, by looking Layer 2 Connectivity Network View, and verifying that each broadcast domain connects to a most one Network Element Interface for each Network Element. This consistency check further demonstrates the power of the separation of concerns of the low-level configuration as such a verification becomes a simple task compared to trying to extract the topology from the device models.

A further example is an OSPF example, where one check may be that Area Border Routers are only connected by other Area Border Routers, or by Backbone routers. We refer to the topology shown in Figure 9.15. In order to configure the areas for the OSPF routing protocol, a user can express the desired area as an label, on the appropriate Network Element Interface, on the Network Whiteboard. Recall the following definitions, where the OSPF Role of a router is:

- *Area Border Router* (A) if it has an interface in area 0, and an interface not in area 0.
- *Backbone Router* (B) if all its interfaces are in area 0
- *Intermediate Router* (I) if no interfaces are in area 0

We can then define a Network Element label allocation function that maps the *ospf_role* based on these interface rules. We can then examine the topology to confirm that *Backbone Routers* only connect to *Backbone Routers* or *Area Border Routers*, and that *Intermediate Routers* only connect to *Intermediate Routers* or *Area Border Routers*.

The use of Verification Functions allows checks to be built into the network design workflow. Verification Functions can use the same Low-Level and High-Level Primitives used in Design Functions. They can also be applied before a Design Function to act as a Consistency Check Function of the Network Views which are used as inputs to the Design Function.

They can be written independent of the Design Functions, allowing for separation of concerns of the testing and design stages. This could be expanded further using the approaches discussed in Nc-Guard [104].

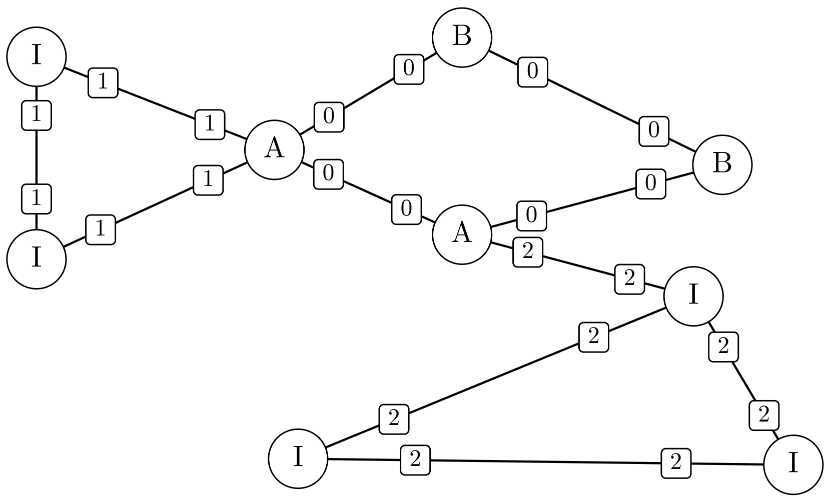


Figure 9.15: Reproduction of the OSPF View Example from Chapter 4, showing OSPF Area labels on Network Element Interfaces, from which the OSPF Role Label (A, B, or I) is derived.

9.3.4 *Composition of Design, Optimisation and Verification Functions*

The set of Design Functions, Verification Functions, and Optimisation Functions to build a specific Network Layer can be grouped into a module for that Network View.

We can compose these functions:

1. Expand a Network Whiteboard
2. Apply unmanaged switch transformations
3. Apply VLAN Design Function
4. Apply extended IP addressing Design Function
5. Apply ACLs
6. Set multiple IGPs
7. Set OSPF traffic engineering
8. Set iBGP scalability
9. Apply eBGP policy

Through separation of concerns, we can substitute more advanced Design Functions (IP addressing, iBGP scalability), introduce new intermediate Network Views (such as for switches and VLANws), apply Optimisation Functions (OSPF traffic engineering), or add new functionality in an existing Network View (eBGP policy). We can also apply Verification Functions to check the integrity of a Network View. This allows us to both re-use design logic, and to extend design workflows as necessary. An example flowchart of these extensions is shown in Figure 9.16.

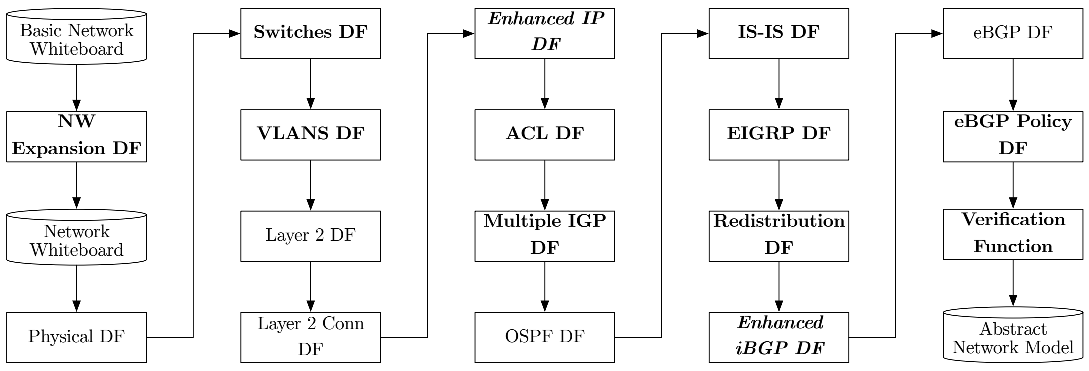


Figure 9.16: Workflow composed of the Design Function extensions described in this section. Bold indicates a new Design Function, Bold Italics indicates an enhanced Design Function, and unbolded indicate previously defined Design Functions unchanged.

9.4 GENERATION OF LOW-LEVEL CONFIGURATION STATE

9.4.1 *Intermediate Device Model Schema*

In Section 9.2.2 we describe how we could introduce a schema for Network Views. We could also introduce a schema for the Intermediate Device Model. This could leverage the work carried out to model device configurations in the YANG language, such as the collection of YANG models in the YangModels[3] GitHub page.

9.4.2 *Abstract Syntax Tree representation for BGP Policy*

An abstract-syntax tree approach used in compilers for programming languages would allow expansion of our current approach to represent BGP Policy and Access Control Lists. This would expand on the simple list structure we presented in Section 8.4.

It would allow for the if-then-else statements in a policy statement to be represented in a vendor-independent language in the Network Whiteboard and Network Views, and then transformed as appropri-

3 https://github.com/YangModels/yang

ate by the Device Compiler. This would systematically handle the different approaches used to represent routing policy in low-level configuration languages. This could then expand on work such as Boehm *et.al* [10] or RPSL [2].

9.4.3 *Device Compiler Inheritance*

As discussed in Section 6.2.7 there are similarities and differences in the logic used to compile the Intermediate Device Models from the Abstract Network Models for different target devices. The common code could be inherited using the programming concepts of inheritance and composition, as shown in Figure 9.17.

This would also allow for minor variations between releases of the same network operating system (such as versions of IOS) to be handled in a systematic manner, with the variations handled through the use of function inheritance, with the shared logic only required to be written once.

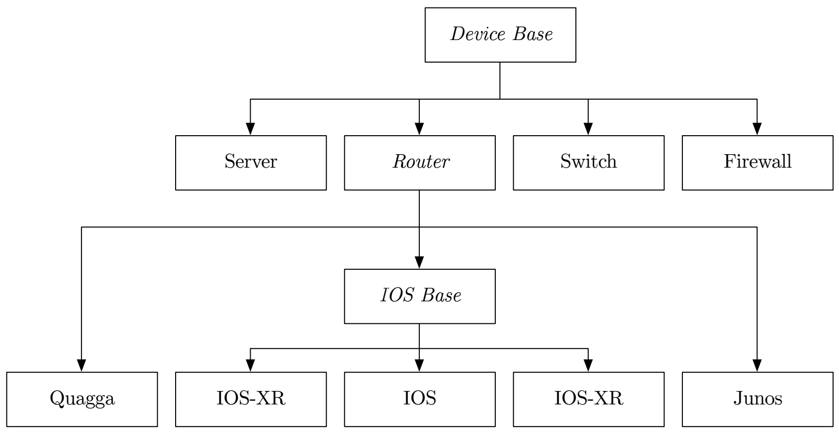


Figure 9.17: An example of device inheritance sharing common logic.

9.4.4 *Template Inheritance*

Similar to Device Compiler Inheritance, many templating engines allow composition of templates. This would allow large templates to be composed of a series of smaller templates. This would allow similar benefits to Device Compiler Inheritance, where shared template logic could be written once, and only variations between templates required to be written. For instance an IOS template could be composed of smaller templates for each of the interface, OSPF, and BGP blocks. If there was a change between releases of IOS, then the master template for that release could incorporate the relevant changes, and inherit the shared smaller templates for common configuration syntax.

9.4.5 *Compiling to Different Output Targets*

One of the main contributions of our approach has been to provide a systematic methodology to transform a high-level network description from the Network Whiteboard into the low-level Intermediate Device Model. This Device Model could be used as the input to network programmability work such as Ansible playbooks [61] or YANG/Netconf-based systems, or integration with existing template-based approaches which we discussed in Chapter 3. The key requirement for each of these systems is a description of desired network state. Our approach generates such network state from a high-level description.

As the Intermediate Device Models are represented in a simple Python dictionary or JSON structure, allowing the template engine to be substituted for a conversion into alternative formats used by current tools, such as YAML, XML (used by Netconf/YANG), or CSV. This would allow our systematic high-level approach to be integrated with existing workflows and tools.

9.4.6 *Compiling to Hardware*

The Platform Compiler approach presented in Section 6.4.1 is responsible for creating the Intermediate Hardware Model. In this thesis we used this to allocate interface names and add out-of-band management interface as required by the simulation environment.

This approach could be expanded further to allow the modelling of hardware requirements, such as interface names being determined by a *rack/chassis/line-card* hierarchy that is present on high-end real-world routers. Addition hardware-specific information could be added to the Intermediate Hardware Model by adding steps to the Platform Compiler approach. These could then be handled by the Device Compiler. This is a benefit of our approach of decoupling the device hardware specifics from the Device Compiler, and moving them into the Platform Compiler.

9.4.7 *Virtual Devices and Multi-Chassis Devices*

The system could be expanded to handle virtual devices (where a single device represents multiple Network Elements) or multi-chassis devices (where a single router is composed of multiple "devices"). The former could be handled through the use of labels on the Network Whiteboard and Network Views, which denote that multiple virtual routers belong to a single physical network device. This could then be aggregated in the Platform Compiler and Device Compiler steps. Multi-Chassis devices could also be handled in the Platform Compiler and Device Compiler steps, where the appropriate set configurations are generated for the single Network Element in the Network Whiteboard.

9.5 expansion and new applications of system

9.5.1 *Network Experimentation*

As we showed in Chapter 7, the system can be used in a black-box mode as a standalone console script, similar to a traditional programming language compiler. Combined with an automated collection system, this encourages network experimentation, where a user could modify parameters on the Network Whiteboard, automatically generate and deploy the configurations, and automatically collect results. This simplifies the steps required to carry out a scientific experimentation methodology. By storing the Network Whiteboards, this encourages reproducible research, realising similar benefits to reproducible research using the Mininet platform, as described by Handigol *et al.* [43].

9.5.2 *Engineering Workflows*

As we have shown, our approach can be used to compile to multiple target platforms and devices. This was demonstrated in the OSPF example in Section 6.2, where we compiled for multiple devices from the same Network Whiteboard. This was also demonstrated in the case studies of Chapter 8, in particular the Large-Scale Example in Section 8.6, where we compiled the same Network Whiteboard for Quagga on the Netkit emulation platform, and the C-BGP simulation platform. In this Large-Scale Example we also showed examples offline analysis using shortest-path algorithms.

This approach could be used to construct an engineering workflow, where the Network Whiteboard description is compiled to multiple targets. An example of such a workflow is shown in Figure 9.18. The *Design* step uses the Network Whiteboard and Design Functions as outlined in our previous examples. The *Algorithm* step could use offline analysis such as shortest path or other graph algorithms to check theoretical properties of network topologies (represented as graphs in Network Views). The *Simulation* step could use a platform such as C-BGP to model idealised protocol behaviour, and allow deep inspection of network behaviour. The *Emulation* step can use platforms such as Netkit, VIRL, or Junosphere to test device configurations against real operating systems, including real-world bugs and specification interpretations. Finally, as described in Section 9.4.6, we can compile to hardware devices for the *Testbed* step and to deploy to a *Production* network, like described in industry presentations such as Schmidt *et al.* [90].

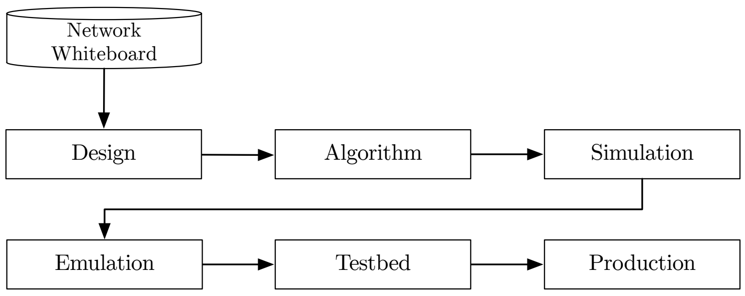


Figure 9.18: An example engineering workflow from algorithmic analysis to deployment in a production network.

9.5.3 *Integration with an Existing Network*

The work in this thesis focussed on the generation of configurations for a green-fields network. Integration with an existing network deployment could be performed using the techniques for "Bootstrapping an Existing Network" described by Caldwell *et al.* [16].

9.5.4 *Run-Time System*

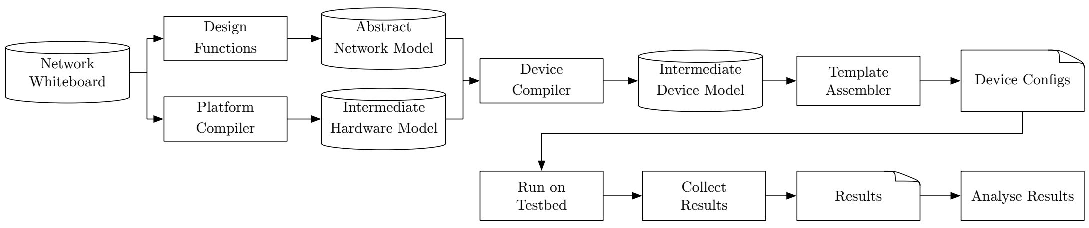
Finally, our work focussed on the static generation of device configurations. Our work could be integrated into a closed-loop system for real-time network management. This could use techniques such as those discussed by Chen *et al.* [22] to model the ordering of execution of commands internal to a router. Our framework could also be integrated with work such as that of Vanbever *et al.* [105] to generate and validate the intermediate Network Views from the current state to the desired state.

We note that much of the complexity in maintaining a run-time network is in modelling the sequence of changes to move from the current network state to the desired network state. The work on Netconf and YANG has made progress in this regard in allowing a device state specification to be uploaded to a device, with the device itself taking care of the internal sequencing to implement this desired state. This could be integrated with our approach through compilation of the Abstract Network Model to the YANG configuration format as described in Section 9.4.5. This would then allow us to focus on the network-level sequencing. This could be performed using the work of Vanbever *et al.* [105], which could be integrated with our graph-based approach to representing network topologies.

9.6 CONCLUSION

As this chapter demonstrates, there are numerous avenues for future research for the representation and implementation of network policy and associated compilation steps. These further demonstrate the ability of the approach and toolchain presented in this thesis to both answer the Research Questions presented in Chapter 3, and how they can be expanded further in future work to address an even larger set of network configuration tasks.

10 CONCLUSIONS



In this thesis we have addressed the major challenge of improving the reliability and cost of design and maintenance of modern large scale traditional computer networks by introducing a disciplined and well structured set of abstractions and transformations that automate the creation of router configurations from high-level network policy documents. In doing so we have emphasised the use of a representation language that is compatible with the skills and domain knowledge of current network engineers. We have adopted principles derived from computer language compilation where staged transformations are made to refine the design towards implementation. We have used a set of graph based intermediate representations and incorporated a flexible code generation arrangement that is compatible with all current network hardware.

The tool developed around these abstractions have been made available to the wider networking community and has been validated at scale in simulated network environments. We have shown through this process that it is feasible to transition to a new approach in static network design within the constraints current networks.

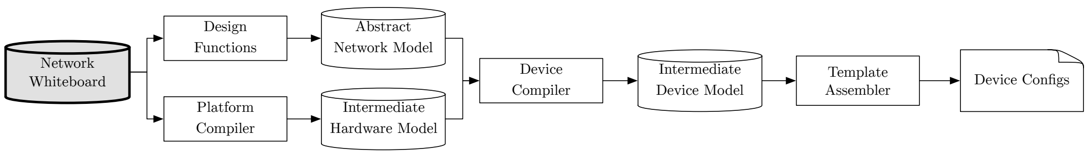
The adoption of new approaches in the networking industry, such as Software-Defined Networking has had a variable success rate. In reflecting on the work of this thesis it must be acknowledged that adoption depends of a combination of appropriate abstractions and languages but also on other factors. There is a strong "network effect" in adoption of new technology in networking whereby the operation and commitment of bigger players in the industry may be vital to adoption. It is not clear yet if there is sufficient commercial incentive for the large vendors to adopt an approach which may lead to the commodification of their hardware and software through standardisation. Our approach can be used with existing network hardware deployments.

10.1 RESEARCH QUESTIONS

The research questions we set out to address in this thesis are listed below, along with a summary of how they have been addressed in this thesis.

10.1.1 *Question 1*

Is there a declarative representation of High-Level Network Policy that is also likely to be widely adoptable by current network practitioners?

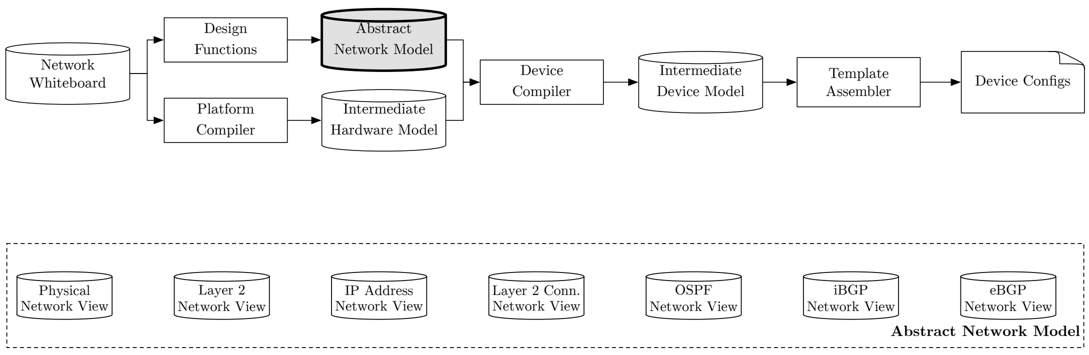


We have addressed this question through our use of the Network Whiteboard abstraction, which we presented in Chapter 4. As we discussed in Chapter 7 and Chapter 8, the Network Whiteboard can be constructed from a variety of methods, including programmatically or through a graph-based description format such as GraphML. This approach can be extended to support third-party formats such as the XML format used in a proprietary tool such as Cisco VIRL. The user feedback in Section 8.7.3 confirms that this approach is readily usable by current network practitioners.

10.1.2 *Question 2*

What is a better intermediate representation of network configuration that is based on graph theory, supports a well structured compilation process and provides clear separation of concerns?

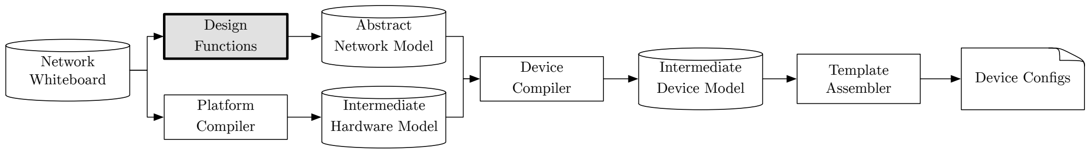
In Chapter 4 we introduced the Network View abstraction, which is an intermediate representation of network configurations. This is graph-based and allows the expression of network configurations using the Network Element, Network Element Interface, and Network Element Connections concepts. Through the use of labels we



have demonstrated how this abstraction can formally capture the key information in the network design process. The set of all Network Views forms the Abstract Network Model.

10.1.3 *Question 3*

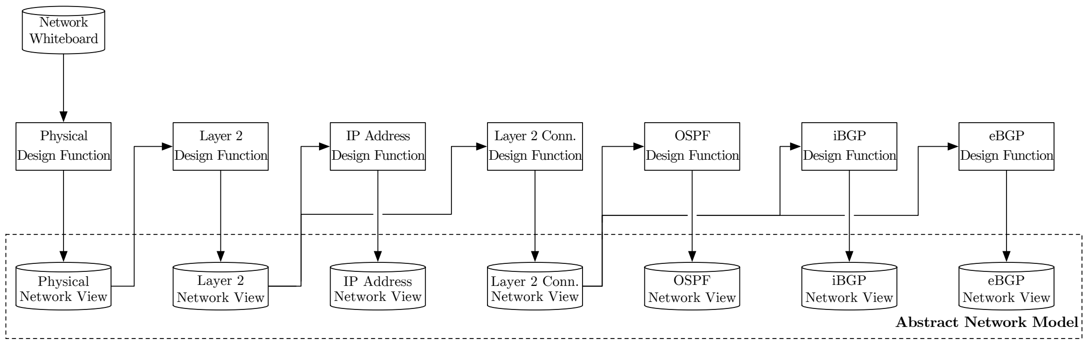
How can declarative High-Level Network Policy descriptions be transformed into a graph-based intermediate format using a compiler that is extensible in terms of new types of policies and new protocols?

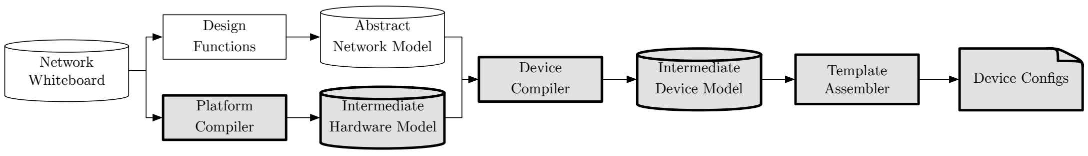


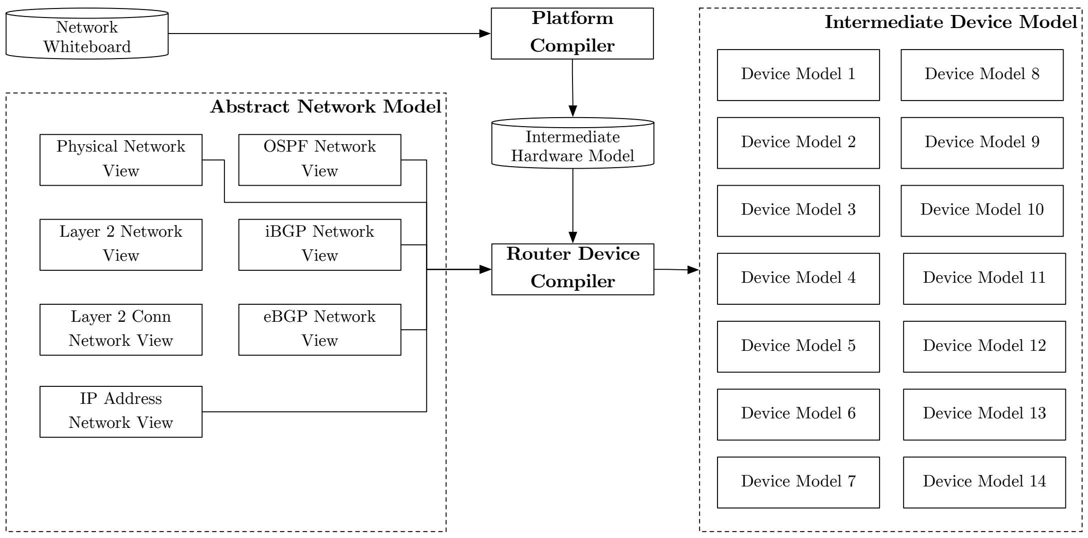
In Chapter 5 we introduced Design Functions which transform the High-Level Network Policy Description expressed using the Network Whiteboard, into the graph-based intermediate format of the Network Views. These Design Functions are built from the Low-Level Primitives and High-Level Primitives which in turn are based on set theory and graph theory. In Section 5.7 we discussed a number of ways in how Design Functions could be extended to support different devices such as switches, or different design policies such as for iBGP designs. In Chapter 8 we further demonstrated how our approach can be adopted to accommodate different network designs.

10.1.4 *Question 4*

How can the configurations for a diversity of network devices be generated systematically from a graph-based intermediate representation?


Network Whiteboard
Physical Design Function
Layer 2 Design Function
IP Address Design Function
Layer 2 Conn. Design Function
OSPF Design Function
iBGP Design Function
eBGP Design Function
Physical Network View
Layer 2 Network View
IP Address Network View
Layer 2 Conn. Network View
OSPF Network View
iBGP Network View
eBGP Network View
Abstract Network Model


Network Whiteboard
Design Functions
Abstract Network Model
Platform Compiler
Intermediate Hardware Model
Device Compiler
Intermediate Device Model
Template Assembler
Device Configs


Network Whiteboard
Platform Compiler
Intermediate Device Model
Abstract Network Model
Physical Network View
OSPF Network View
Layer 2 Network View
iBGP Network View
Layer 2 Conn Network View
eBGP Network View
IP Address Network View
Intermediate Hardware Model
Router Device Compiler
Device Model 1
Device Model 2
Device Model 3
Device Model 4
Device Model 5
Device Model 6
Device Model 7
Device Model 8
Device Model 9
Device Model 10
Device Model 11
Device Model 12
Device Model 13
Device Model 14

In Chapter 6 we presented the approach to generation of low-level device configurations. This is a two-step process, where we first construct an Intermediate Device Model, which is an intermediate representation of the information to be configured on that specific network device. This Intermediate Device Model is built using the Device Compiler, from the information in the Abstract Network Model. The second step assembles the individual device configurations from this Intermediate Device Model, using simple templates. In Chapter 6 we discussed how this is motivated from the diversity of low-level configurations for the configuration of interfaces and OSPF, and in Chapter 8 we showed how this approach can be used for the Quagga, IOSv, IOSvL2 devices, run on the Netkit and Cisco VIRL simulation testbeds, and the C-BGP simulation software. This demonstrated how our approach and toolchain can generate configurations for a diverse range of network devices and testbed platforms.

10.1.5 *Question 5*

What are the scalability and extensibility characteristics for the compilation of High-Level Network Configuration Policy to device configurations in terms of network size and diversity of network protocols and target devices?

In Chapter 5 and Chapter 6 we showed how our approach and toolchain can be extended to cater for both network protocols and target devices. We demonstrated this using AutoNetkit in Chapter 7 and Chapter 8, where we compiled our High-Level Network Configuration Policy to a variety of network protocols and target devices, showing extensibility. We also demonstrated scalability in quickly compiling a large-scale network. In Chapter 9 we discussed potential future directions that are enabled by these scalability and extensibility characteristics.

10.2 CONCLUSION

By addressing these research questions we have presented a theoretical approach and practical toolchain that can be used to express High-Level Network Configuration Policy together with intermediate representations and transformations, that can be used to generate low-level device configurations for a variety of network protocols and devices, in a manner that is both extensible and scalable, and able to be adopted by the research community and current network practitioners.

The approach presented in this thesis provides both a *Distributed State Abstraction* and a *Specification Abstraction*, which are two of the three abstractions which form Software-Defined Networking as outlined by Shenker [95]. We do not present a *Forwarding Abstraction*: changing the Forwarding Abstraction, such as to OpenFlow, would require replacing infrastructure and retraining support staff, which can be a significant expense.

Our approach to the Distributed State and the Specification Abstraction allows many of the benefits of a systematic and software defined approach to networking, whilst leveraging existing network deployments and network practitioner expertise.

Part V

APPENDICES

A

RFC 3139 REQUIREMENTS

The following is a reproduction of the full set of requirements identified in Sanchez *et al.* [89].

> 1. Provide means by which the behavior of the network can be specified at a level of abstraction (network-wide configuration) higher than a set of configuration information specific to individual devices,
>
> 2. Be capable of translating network-wide configurations into device-local configuration. The identification of the relevant subset of the network-wide policies to be down-loaded is according to the capabilities of each device,
>
> 3. Be able to interpret device-local configuration, status and monitoring information within the context of network-wide configurations,
>
> 4. Be capable of provisioning (e.g., adding, modifying, deleting, dumping, restoring) complete or partial configuration data to network devices simultaneously or in a synchronized fashion as necessary,
>
> 4a. Be able to provision multiple device-local configurations to support fast switch-overs without the need to down- load potentially large configuration changes to many devices,
>
> 5. Provide means by which network devices can send feedback information (configuration data confirmation, network status and monitoring information, specific events, etc.) to the management system,
>
> 6. Be capable of provisioning complete or partial configuration data to network devices dynamically as a result of network specific or network-wide events,
>
> 7. Provide efficient and reliable means compared to current versions of today's mechanisms (CLI, SNMP) to provision large amounts of configuration data,
>
> 8. Provide secure means to provision configuration data. The system must provide support for access control, authentication, integrity-checking, replay- protection and/or privacy security services. The minimum level of granularity for access control and authentication is host based. The system SHOULD support

user/role based access control and authentication for users in different roles with different access privileges,

9. Provide expiration time and effective time capabilities to configuration data. It is required that some configuration data items be set to expire, and other items be set to never expire,

10. Provide error detection (including data-specific errors) and failure recovery mechanisms (including prevention of inappropriately partial configurations when needed) for the provisioning of configuration data,

11. Eliminate the potential for mis-configuration occurring through concurrent shared write access to the device's configuration data,

12. Provide facilities (with host and user-based authentication granularity) to help in tracing back configuration changes,

13. Allow for the use of redundant components, both network elements and configuration application platforms, and for the configuration of redundant Network Elements.

14. Be flexible and extensible to accommodate future needs. Configuration management data models are not fixed for all time and are subject to evolution like any other management data model. It is therefore necessary to anticipate that changes will be needed, but it is not possible to anticipate what those changes might be. Such changes could be to the configuration data model, supporting message types, data types, etc., and to provide mechanisms that can deal with these changes effectively without causing inter-operability problems or having to replace/update large amounts of fielded networking devices,

15. Leverage knowledge of the existing SNMP management infrastructure. The system MUST leverage knowledge of and experience with MIBs and SMI. (Sanchez *et al.* [89])

B

NETKIT SMALL INTERNET LAB

The following two slides, and the example router configurations are reproduced from Di Battista *et al.* [25].

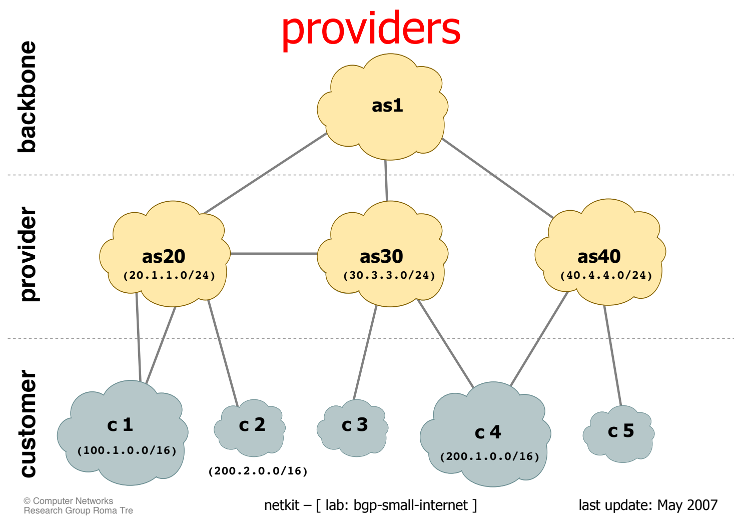


Figure B.1: Slide from Di Battista *et al.* [25] showing BGP Policy

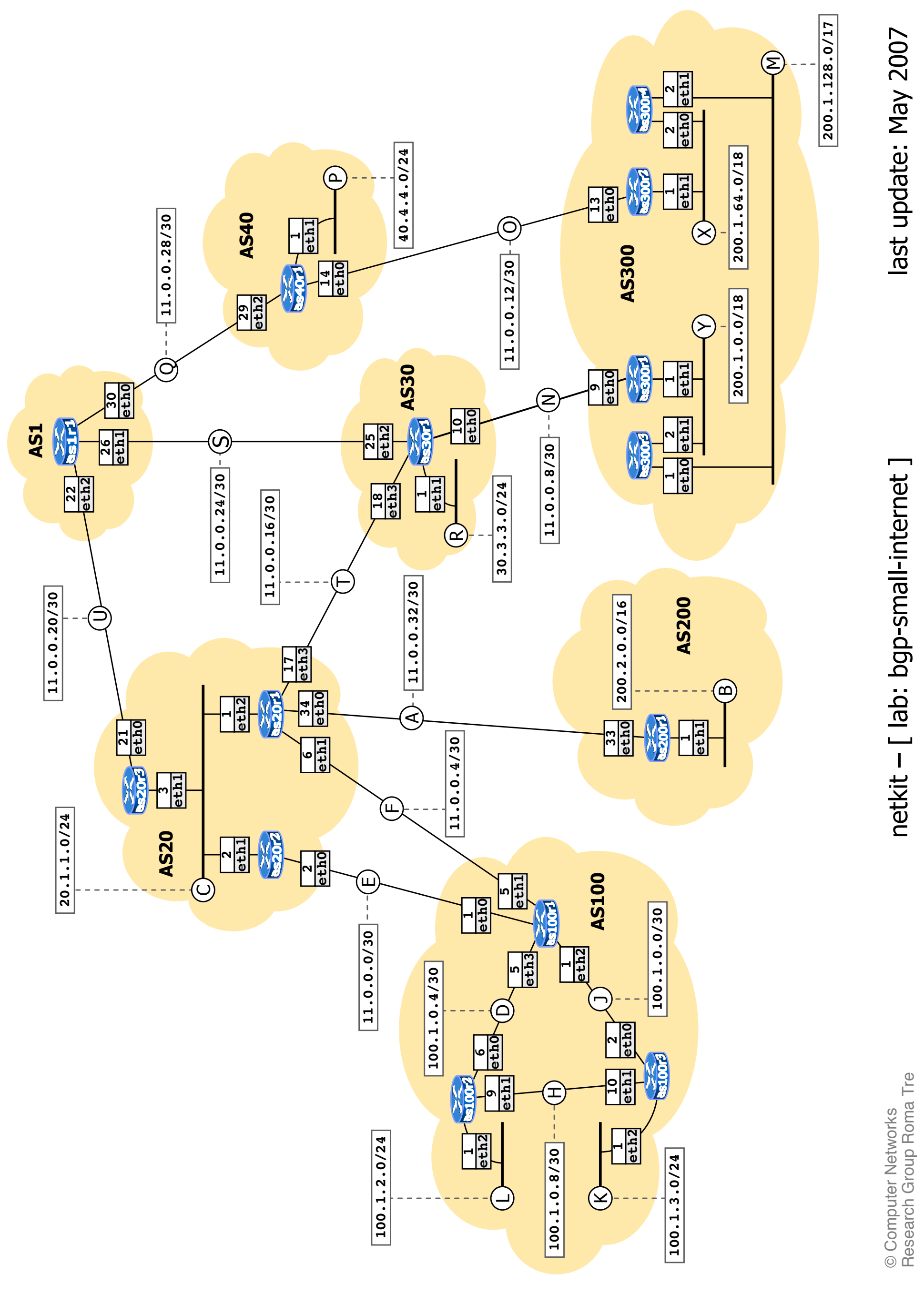


Figure B.2: Netkit Lab Small Internet Topology from Di Battista *et al.* [25]

B.1 EXAMPLE ROUTER CONFIGURATIONS

```
!
hostname bgpd
password zebra
enable password zebra
!
router bgp 20
network 20.1.1.0/24
network 11.0.0.4/30
network 11.0.0.16/30
network 11.0.0.32/30
!
neighbor 11.0.0.33 remote-as 200
neighbor 11.0.0.33 description Router as200r1
neighbor 11.0.0.33 default-originate
neighbor 11.0.0.33 prefix-list as200In in
neighbor 11.0.0.33 prefix-list defaultOut out
!
neighbor 11.0.0.5 remote-as 100
neighbor 11.0.0.5 description Router as100r1
neighbor 11.0.0.5 default-originate
neighbor 11.0.0.5 prefix-list as100In in
neighbor 11.0.0.5 prefix-list defaultOut out
!
neighbor 20.1.1.2 remote-as 20
neighbor 20.1.1.2 description Router as20r2 (iBGP)
!
neighbor 20.1.1.3 remote-as 20
neighbor 20.1.1.3 description Router as20r3 (iBGP)
!
neighbor 11.0.0.18 remote-as 30
neighbor 11.0.0.18 description Router as30r1 (eBGP)
! Use the route-map when announcing default route
neighbor 11.0.0.18 default-originate route-map dontUseMe
! Use the route-map in all the other cases
neighbor 11.0.0.18 route-map dontUseMe out
neighbor 11.0.0.18 prefix-list defaultOut out
!
ip prefix-list as200In permit 200.2.0.0/16
ip prefix-list as100In permit 100.1.0.0/16
ip prefix-list defaultOut permit 0.0.0.0/0
!
route-map dontUseMe permit 10
set as-path prepend  20 20 20
!
log file /var/log/zebra/bgpd.log
!
debug bgp
debug bgp events
debug bgp filters
debug bgp fsm
debug bgp keepalives
debug bgp updates
!
```

Listing B.1: BGPd configuration for *as20r1* from Di Battista *et al.* [25]

```
!
hostname ripd
password zebra
!
router rip
```

```
network 20.1.1.0/24
redistribute connected
!
log file /var/log/zebra/ripd.log
```

Listing B.2: RIPd configuration for *as20r1* from Di Battista *et al.* [25]

```
!
hostname bgpd
password zebra
enable password zebra
!
router bgp 30
network 30.3.3.0/24
network 11.0.0.8/30
!
neighbor 11.0.0.9 remote-as 300
neighbor 11.0.0.9 description Router as300r1
neighbor 11.0.0.9 default-originate
neighbor 11.0.0.9 prefix-list as300In in
!
neighbor 11.0.0.26 remote-as 1
neighbor 11.0.0.26 description Router as1r1 (eBGP)
!
neighbor 11.0.0.17 remote-as 20
neighbor 11.0.0.17 description Router as20r1 (eBGP)
! Use the route-map when announcing default route
neighbor 11.0.0.17 default-originate route-map dontUseMe
! Use the route-map in all the other cases
neighbor 11.0.0.17 route-map dontUseMe out
!
ip prefix-list as300In permit 200.1.0.0/16
ip prefix-list as300In permit 200.1.0.0/17
!
route-map dontUseMe permit 10
set as-path prepend  30 30 30
!
log file /var/log/zebra/bgpd.log
!
debug bgp
debug bgp events
debug bgp filters
debug bgp fsm
debug bgp keepalives
debug bgp updates
!
```

Listing B.3: BGPd configuration for *as30r1* from Di Battista *et al.* [25]

C

NOTATION FOR SMALL INTERNET EXAMPLE

Table C.1: Network Element Connections for Simplified Small Internet Example

Network Element Connection	Network Elements
ϵ_1	(π_1, π_2)
ϵ_2	(π_1, π_6)
ϵ_3	(π_1, π_7)
ϵ_4	(π_2, π_3)
ϵ_5	(π_2, π_4)
ϵ_6	(π_3, π_4)
ϵ_7	(π_3, π_9)
ϵ_8	(π_4, π_5)
ϵ_9	(π_4, π_6)
ϵ_{10}	(π_7, π_{12})
ϵ_{11}	(π_8, π_9)
ϵ_{12}	(π_8, π_{10})
ϵ_{13}	(π_9, π_{10})
ϵ_{14}	(π_{11}, π_{13})
ϵ_{15}	(π_{12}, π_{14})
ϵ_{16}	(π_{13}, π_{14})

Table C.2: Network Element Interfaces for Simplified Small Internet Example

Network Element	Network Element Interfaces
π_1	τ_a, τ_b, τ_c
π_2	τ_a, τ_b, τ_c
π_3	τ_a, τ_b, τ_c
π_4	$\tau_a, \tau_b, \tau_c\ \tau_d$
π_5	τ_a
π_6	τ_a, τ_b
π_7	τ_a, τ_b
π_8	τ_a, τ_b
π_9	τ_a, τ_b, τ_c
π_{10}	τ_a, τ_b
π_{11}	τ_a
π_{12}	τ_a, τ_b
π_{13}	τ_a, τ_b
π_{14}	τ_a, τ_b

Table C.3: Network Element Interface bindings for Simplified Small Internet Example

Network Element Connection	Binding 1	Binding 2
$\epsilon 1$	(π_1, τ_a)	(π_2, τ_a)
$\epsilon 2$	(π_1, τ_c)	(π_6, τ_a)
$\epsilon 3$	(π_1, τ_b)	(π_7, τ_a)
$\epsilon 4$	(π_2, τ_c)	(π_3, τ_a)
$\epsilon 5$	(π_2, τ_b)	(π_4, τ_a)
$\epsilon 6$	(π_3, τ_b)	(π_4, τ_d)
$\epsilon 7$	(π_3, τ_c)	(π_9, τ_c)
$\epsilon 8$	(π_4, τ_c)	(π_5, τ_a)
$\epsilon 9$	(π_4, τ_b)	(π_6, τ_b)
$\epsilon 10$	(π_7, τ_b)	(π_{12}, τ_a)
$\epsilon 11$	(π_8, τ_a)	(π_9, τ_c)
$\epsilon 12$	(π_8, τ_b)	(π_{10}, τ_b)
$\epsilon 13$	(π_9, τ_b)	(π_{10}, τ_a)
$\epsilon 14$	(π_{11}, τ_a)	(π_{13}, τ_a)
$\epsilon 15$	(π_{12}, τ_b)	(π_{14}, τ_a)
$\epsilon 16$	(π_{13}, τ_b)	(π_{14}, τ_b)

Table C.4: Loopback IP Address allocations for Simplified Small Internet Example

Network Element	τ_1	τ_2	τ_3	τ_4
π_1	H.2	M.1	L.1	-
π_2	H.1	J.1	I.1	-
π_3	I.2	K.1	P.1	-
π_4	J.2	O.1	N.1	K.2
π_5	N.2	-	-	-
π_6	L.2	O.2	-	-
π_7	M.2	Q.1	-	-
π_8	R.1	S.1	-	-
π_9	P.2	T.1	R.2	-
π_{10}	T.2	S.2	-	-
π_{11}	U.1	-	-	-
π_{12}	Q.2	V.1	-	-
π_{13}	U.2	W.1	-	-
π_{14}	V.2	W.2	-	-

$$E = \{\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4, \epsilon_5, \epsilon_6, \epsilon_7, \epsilon_8, \epsilon_9, \epsilon_{10}, \epsilon_{11}, \epsilon_{12}, \epsilon_{13}, \epsilon_{14}, \epsilon_{15}, \epsilon_{16}\} = \{ \\ (\pi_1, \pi_2), (\pi_1, \pi_6), (\pi_1, \pi_7), (\pi_2, \pi_3), (\pi_2, \pi_4), (\pi_3, \pi_4), \\ (\pi_3, \pi_9), (\pi_4, \pi_5), (\pi_4, \pi_6), (\pi_7, \pi_{12}), (\pi_8, \pi_9), (\pi_8, \pi_{10}), \\ (\pi_9, \pi_{10}), (\pi_{11}, \pi_{13}), (\pi_{12}, \pi_{14}), \\ (\pi_{13}, \pi_{14})\}$$

Figure C.1: Example of $E_{whiteboard}$ for Simplified Small Internet Example

$$T = \{ \\ (\pi_1, \{\tau_a, \tau_b, \tau_c\}), \quad (\pi_2, \{\tau_a, \tau_b, \tau_c\}), \quad (\pi_3, \{\tau_a, \tau_b, \tau_c\}), \\ (\pi_4, \{\tau_a, \tau_b, \tau_c\, \tau_d\}), \quad (\pi_5, \{\tau_a\}), \quad (\pi_6, \{\tau_a, \tau_b\}), \\ (\pi_7, \{\tau_a, \tau_b\}), \quad (\pi_8, \{\tau_a, \tau_b\}), \quad (\pi_9, \{\tau_a, \tau_b, \tau_c\}), \\ (\pi_{10}, \{\tau_a, \tau_b\}), \quad (\pi_{11}, \{\tau_a\}), \quad (\pi_{12}, \{\tau_a, \tau_b\}), \\ (\pi_{13}, \{\tau_a, \tau_b\}), \quad (\pi_{14}, \{\tau_a, \tau_b\}) \\ \}$$

Figure C.2: Example of T for Simplified Small Internet Example

$$
\begin{aligned}
&B = \{\\
&(\epsilon1,\{(\pi_1,\tau_a),(\pi_2,\tau_a)\}), && (\epsilon2,\{(\pi_1,\tau_c),(\pi_6,\tau_a)\}),\\
&(\epsilon3,\{(\pi_1,\tau_b),(\pi_7,\tau_a)\}), && (\epsilon4,\{(\pi_2,\tau_c),(\pi_3,\tau_a)\}),\\
&(\epsilon5,\{(\pi_2,\tau_b),(\pi_4,\tau_a)\}), && (\epsilon6,\{(\pi_3,\tau_b),(\pi_4,\tau_d)\}),\\
&(\epsilon7,\{(\pi_3,\tau_c),(\pi_9,\tau_c)\}), && (\epsilon8,\{(\pi_4,\tau_c),(\pi_5,\tau_a)\}),\\
&(\epsilon9,\{(\pi_4,\tau_b),(\pi_6,\tau_b)\}), && (\epsilon10,\{(\pi_7,\tau_b),(\pi_{12},\tau_a)\}),\\
&(\epsilon11,\{(\pi_8,\tau_a),(\pi_9,\tau_c)\}), && (\epsilon12,\{(\pi_8,\tau_b),(\pi_{10},\tau_b)\}),\\
&(\epsilon13,\{(\pi_9,\tau_b),(\pi_{10},\tau_a)\}), && (\epsilon14,\{(\pi_{11},\tau_a),(\pi_{13},\tau_a)\}),\\
&(\epsilon15,\{(\pi_{12},\tau_b),(\pi_{14},\tau_a)\}), && (\epsilon16,\{(\pi_{13},\tau_b),(\pi_{14},\tau_b)\}),\\
&\}
\end{aligned}
$$

Figure C.3: Example of B for Simplified Small Internet Example

$$
\begin{aligned}
&L_\Pi = \{\\
&(\pi_1,\{(asn,1),(role,router)\}), && (\pi_2,\{(asn,20),(role,router)\}),\\
&(\pi_3,\{(asn,20),(role,router)\}), && (\pi_4,\{(asn,20),(role,router)\}),\\
&(\pi_5,\{(asn,200),(role,router)\}), && (\pi_6,\{(asn,30),(role,router)\}),\\
&(\pi_7,\{(asn,40),(role,router)\}), && (\pi_8,\{(asn,100),(role,router)\}),\\
&(\pi_9,\{(asn,100),(role,router)\}), && (\pi_{10},\{(asn,100),(role,router)\}),\\
&(\pi_{11},\{(asn,300),(role,router)\}), && (\pi_{12},\{(asn,300),(role,router)\}),\\
&(\pi_{13},\{(asn,300),(role,router)\}), && (\pi_{14},\{(asn,300),(role,router)\})\\
&\}
\end{aligned}
$$

Figure C.4: Example of L_Π for Simplified Small Internet Example

$$
\begin{aligned}
&L_B = \{\\
&\{\pi_1, \{(\tau_a, \{(ip, H.2)\}), (\tau_b, \{(ip, M.1)\}), (\tau_c, \{(ip, L.1)\})\}\},\\
&\{\pi_2, \{(\tau_a, \{(ip, H.1)\}), (\tau_b, \{(ip, J.1)\}), (\tau_c, \{(ip, I.1)\})\}\},\\
&\{\pi_3, \{(\tau_a, \{(ip, I.2)\}), (\tau_b, \{(ip, K.1)\}), (\tau_c, \{(ip, P.1)\})\}\},\\
&\{\pi_4, \{(\tau_a, \{(ip, J.2)\}), (\tau_b, \{(ip, O.1)\}), (\tau_c, \{(ip, N.1)\}), (\tau_d, \{(ip, K.2)\})\}\},\\
&\{\pi_5, \{(\tau_a, \{(ip, N.2)\})\}\},\\
&\{\pi_6, \{(\tau_a, \{(ip, L.2)\}), (\tau_b, \{(ip, O.2)\})\}\},\\
&\{\pi_7, \{(\tau_a, \{(ip, M.2)\}), (\tau_b, \{(ip, Q.1)\})\}\},\\
&\{\pi_8, \{(\tau_a, \{(ip, R.1)\}), (\tau_b, \{(ip, S.1)\})\}\},\\
&\{\pi_9, \{(\tau_a, \{(ip, P.2)\}), (\tau_b, \{(ip, T.1)\}), (\tau_c, \{(ip, R.2)\})\}\},\\
&\{\pi_{10}, (\tau_a, \{(ip, T.2)\}), (\tau_b, \{(ip, S.2)\})\}\},\\
&\{\pi_{11}, \{(\tau_a, \{(ip, U.1)\})\}\},\\
&\{\pi_{12}, \{(\tau_a, \{(ip, Q.2)\}), (\tau_b, \{(ip, V.1)\})\}\},\\
&\{\pi_{13}, \{(\tau_a, \{(ip, U.2)\}), (\tau_b, \{(ip, W.1)\})\}\},\\
&\{\pi_{14}, \{(\tau_a, \{(ip, V.2)\}), (\tau_b, \{(ip, W.2)\})\}\},\\
&\}
\end{aligned}
$$

Figure C.5: Example of L_B for Simplified Small Internet Example

D

DESIGN DETAILS AND DEFINITIONS OF PRIMITIVES

D.1 DESIGN DETAILS

D.1.1 *Introduction*

In this section we provide a more precise set of definitions for the concepts presented in Section 4.4, and how they can be realised using a graph theory-based approach.

This section is organised as follows. We first outline the notation used for the implementation details. We then outline some fundamental graph theory terminology. We then discuss how graph theory can be adapted to represent the key concepts outline in this chapter, including Network Views, Network Elements, Network Element Connections, and Network Element Interfaces, and the labelling of each of these elements.

D.1.2 *Notation*

We adopt the common standard approach for notation suggested by Voloshin [108]. We also use associative arrays to represent key value pairs for labels. An example is

$$m = \{k_1 : v_1, k_2 : v_1, k_3 : v_3, \dots\}$$

These can be represented using tuples of key value pairs such as $\{(k_1, v_1), (k_2, v_2), (k_3, v_3), \dots\}$. We define the function HAS_KEY(X, KEY) to confirm the existence of the key key in the associative array X, GET_KEY(X, KEY) to obtain the value of key in X, and SET_KEY(X, KEY, VAL to set the value of key to val in X.

D.1.3 *Network Views and their elements*

A Network Whiteboard can viewed as a special case of a Network View. The functions that can be used on a Network View can also be used on a Network Whiteboard.

Network Views

Network Views can be implemented using graphs. A separate graph is used to represent each Network View.

This motivates the need for cross-Network View access of Network Elements and Network Element Interfaces, in both the Design Functions discussed in Chapter 5, and in the Configuration Generation process discussed in Chapter 6. This cross-Network View access of elements is enabled through the use of unique identifiers. We discuss these identifiers in Table D.1.3.

We define a Network View as NV, which has the form of (NE, NEC, NEI, B, L) where NE is the set of Network Elements, NEC is the set of Network Element Connections, NEI is the set of Network Element Interfaces, B is the set of Network Element Interface bindings for Network Element Connections, and L is the set of Labels.

For notational simplicity, we represent the elements using the symbols as shown in Table D.1. Using this notation, the Network View is then defined as the form $\theta = (\Pi, E, T, B, L)$

Table D.1: Table of element notation symbols

Element Type	Set Symbol	Element Symbol
Network Whiteboard	–	ω_i
Network View	Θ	θ_i
Network Element	Π	π_i
Network Element Connection	E	ϵ_i
Network Element Interface	T	τ_i
Network Element Interface Binding	B	b
Network Element Connection Distinguisher	δ	-

Network Whiteboard

Special case of Network View. Not defined in ANM, (covered later), but otherwise all functions within scope of a NV apply.

Network Elements

Network Elements are implemented using nodes. The set of Network Elements in a Network View corresponds to the set of nodes in a graph:

$$G = (N, E)$$

where $N = n_1, n_2, n_3, \ldots$

This is then implemented as a Network View:

$\theta = (\Pi, \ldots)$ where $\Pi = \pi_1, \pi_2, \pi_3, \ldots$

This approach contrasts to an alternative object-oriented approach which would require more machinery to do cross-layer copying.

Network Element Connections

Network Element Connections are implemented using edges. The set of Network Element Connections in a Network View corresponds to the set of edges in a graph.

$G = (N, E)$ where $E = e_1, e_2, e_3, \ldots$ where e_k represents a pair of nodes (n_1, n_2).

$\theta = (\Pi, E)$ where $E = \epsilon_1, \epsilon_2, \epsilon_3, \ldots$ where each ϵ_k represents a pair of Network Elements (π_i, π_j)

For reference, we provide the table of the Network Element Connections for the Simplified Small Internet Example Table C.1. These examples are provided in Appendix C.

Network Element Interfaces

The implementation of Network Element Interfaces has two components.

The set of Network Element Interfaces T consists of Network Element Interfaces τ_k where each Network Element Interface τ_k is defined as (π_i, α_j) where:

- π_i is the Network Element to which this Network Element Interface belongs.
- α_j is the unique identifier of this Network Element Interface within the Network Element π_i.

The constraints around α are discussed in Table D.1.3.

The set of Network Element Interface bindings has the form $B = b_1, b_2, b_3, \ldots$ where each Network Element Interface binding b_k is of the form $(\epsilon_i, \{(\pi_j, \tau_j), (\pi_k, \tau_k)\})$ where:

- ϵ_i is the Network Element Connection for which this binding applies
- τ_j is a binding of this Network Element Connection to Network Element Interface τ_j for Network Element π_j
- τ_k is a binding of this Network Element Connection to Network Element Interface τ_k for Network Element π_k

Recall that ϵ_i is of form (π_j, π_k), which implies that Network Element Connection ϵ_i connects Network Elements π_j and π_k.]

From this, we define the constraint that each of pi_l and π_m in the Network Element Interfaces must be present in the Network Element Connection (π_j, π_k).

For example $\pi_j = \pi_l$ and π_k, π_m or $\pi_j = \pi_m$ and π_k, π_n.

This implies that a Network Element Connection can only be bound to Network Element Interfaces if the interfaces are present on the Network Elements to which Network Element Connection connects.

An example of the Network Element Interfaces T for the Small Internet example is given in Table C.2, and an example of the Network Element Interface Bindings B for the Small Internet example is given in Table C.3.

LOGICAL TO INTERFACE ASSOCIATION This can be handled as a special case of the labels discussed in Section D.1.4. This can be handled by extending the label setting functions using utility functions

$$\mathrm{SET\_ASSOCIATED\_NEI}(\tau_i, \tau_j)$$

to associate τ_j on τ_i, and

$$\mathrm{GET\_ASSOCIATED\_NEI}(\tau_i)$$

to get the associated Network Element Interface on τ_i.

Identifiers

Network Elements and Network Element Interfaces have a unique *identifier*, which can be implemented as a special case of labelling. This identifier allows the Network Element and Network Element Interface to be accessed across Network Views.

Network Element Connection don't have a unique identifier, and can be referred by its endpoints, such as by the Network Elements. Optionally they can also be referred to by their Network Element Interfaces. This is shown in the functions in

D.1.4 *Label Implementation*

Label Sets

We now define the set of Labels, L which form part of the Network View $\theta = (\Pi, E, T, B, L)$.

L has four components, $L = (L_\Theta, L_\Pi, L_E, L_T)$ defined as:

- L_Θ is the set of labels for the Network View itself
- L_Π is the set of labels for Network Elements
- L_E is the set of labels for Network Element Connections
- L_T is the set of labels for Network Element Interfaces

NETWORK VIEW LABELS The set of Network View Labels is a set of tuple of key-value pairs, $L_\Theta = (k_1, v_1), (k_2, v_2), (k_3, v_3), \ldots$ where k_i is a key (the label) and v_i is the value for that label.

NETWORK ELEMENT LABELS The set of labels for Network Elements, Network Element Connections, and Network Element Interfaces could be represented in two ways: grouping by label and then by the element, or grouping by element and then the labels. We choose the latter as this aligns more closely to how the system would be implemented in a programming language. In a programming language, a typical approach would be to represent the element labels using a hash-map or dictionary of key-value pairs.

Therefore, we define L_Π, the set of labels for Network Elements as $L_\Pi = (\pi_1, l_{pi_1}), (\pi_2, l_{pi_2}), (\pi_3, l_{pi_3}), \ldots$ where l_{π_k} is the set of label key-value pairs for Network Element π_k.

I.e. $l_{\pi_k} = (k_1, v_1), (k_2, v_2), (k_3, v_3), \ldots$.

An example of the Network Element Labels L_Π for the Small Internet example is given in Figure C.4.

NETWORK ELEMENT CONNECTION LABELS Similarly, we define L_E, the set of labels for Network Element Connections as

$$L_E = (\epsilon_1, l_{\epsilon_1}), (\epsilon_2, l_{\epsilon_2}), (\epsilon_3, l_{\epsilon_3}), \ldots$$

where l_{ϵ_k} is the set of label key-value pairs for Network Element Connection ϵ_k.

NETWORK ELEMENT INTERFACE LABELS Finally, we define L_T, the set of labels for Network Element Interfaces as

$$L_T = (\tau_1, l_{tau_1}), (\tau_2, l_{tau_2}), (\tau_3, l_{tau_3}), \ldots$$

where l_{τ_k} is the set of label key-value pairs for Network Element Interface τ_k.

The label values are typically single integers or letters, but this is not a constraint. It is possible to have labels, where the values are key-value pairs. This can be used for instance to represent IP Address block allocations on the IP Address Network View. An example of the Network Element Labels, showing the Loopback IP addresses for the IP Address Network View of E the Small Internet example is given in Table C.4.

Label Access Functions

We define a series of label access functions which are used to return a specific label for a specific element. We define these labels access functions as λ with a label access function for each set of labels in L. Therefore $\lambda = (\lambda_\theta, \lambda_\Pi, \lambda_E, \lambda_T)$ where

- λ_θ is the label access function for the Network View
- λ_Π is the label access function for the Network Elements
- λ_E is the label access function for the Network Element Connections
- λ_T is the label access function for the Network Element Interfaces We now define each of these functions for accessing labels. We describe the setting of labels in Section D.2.8.

NETWORK VIEW LABEL ACCESS FUNCTIONS The Network View Label Access Function λ_θ is defined as $\lambda_\theta(l)$ where l is the label to be returned.

An example is $\lambda_\theta(\mathrm{color})$ which may return the value *blue*.

NETWORK ELEMENT LABEL ACCESS FUNCTIONS The Network Element Label Access Function λ_Π is defined as $\lambda_\Pi(\pi_i, l)$ where π_i is the Network Element that we wish to access label l of.

An example is $\lambda_\Pi(\theta_{physical}, \pi_1, \mathrm{asn})$, which for the Simplified Small Internet Example was described in Section 4.5 would return 1, the ASN of Network Element 1.

$\lambda_\Pi(\pi_2, \mathrm{asn})$ would return 20, the ASN of Network Element 2, and $\lambda_\Pi(\pi_9, \mathrm{asn})$ would return 100, the ASN of Network Element 9.

NETWORK ELEMENT CONNECTION LABEL ACCESS FUNCTIONS The Network Element Connection Label Access Function λ_E is defined as

$\lambda_E(\epsilon_i, l)$ where ϵ_i is the Network Element Connection that we wish to access label l of.

NETWORK ELEMENT INTERFACE LABEL ACCESS FUNCTIONS The Network Element Connection Label Access Function λ_T is defined as $\lambda_T(\tau_i, l)$ where τ_i is the Network Element Interfaces that we wish to access label l of.

Pass-Through Labels

Some labels are common between Network Views. We call these *Pass-Through Labels*. These are stored on the Abstract Network Model itself, rather than on an individual Network View. Examples of these *Pass-Through Labels* include the *asn* and *device_type* attributes.

Shortcut Functions

Shortcut functions are functions defined to simplify the syntax. They can be viewed in a similar fashion to syntactic sugar in programming languages. We present these here.

ROLE We can define a role function, ρ, for each of Network Elements, Network Element Connections, and Network Element Interfaces:

- $\rho_\Pi(\pi_i) = \lambda_\Pi(\pi_i, \text{role})$ to access the role of Network Element π_i.
- $\rho_E(\epsilon_i) = \lambda_E(\epsilon_i, \text{role})$ to access the role of Network Element Connection ϵ_i.
- $\rho_T(\tau_i) = \lambda_T(\tau_i, \text{role})$ to access the role of Network Element Interface τ_i.

ASN We can define an alias function for accessing the ASN label of a Network Element. $\lambda_{ASN}(\Theta, \pi_i) = \lambda_\Pi(\Theta, \pi_i, \text{asn})$ is the ASN Label Access Function. This does not need a subscript as the ASN label only has meaning on the set of Network Elements. This label is stored on the Abstract Network Model Θ, not on a Network View.

Network Element Groups

Some labels conceptually apply to groups of Network Elements. Our approach does not have Network Element groups as a structural component. Groups can be inferred by one or more labels on the Network Elements, during the Network View design process.

If a user wants to think of Network Elements as being part of a group, they would need to form the group based on the labels on the Network Elements.

There are situations however, where a user would want to store information that is associated with a group of Network Elements that have the same label set. One way to do this would be to repeat the same stored information on each Network Element that has the identical set of labels, but this would add unnecessary repetition.

An alternative to repetition is to use the unique set of labels that determine the grouping as the unique key in a hash-map.

An example is the IP Address block allocated to an ASN, which may be used in eBGP route advertisement at the border routers of an ASN.

There are two approaches to this. The first is storing a hash-map indexed by the label on the Network View itself, such as shown in Listing D.1. The second is forming an Intermediate Network View, with Pseudo-Nodes corresponding to the label, such as shown in Figure D.1.

```
{
  1: "10.0.1.0/24", 20: "10.0.2.0/24",
  20: "10.0.3.0/24", 30: "10.0.4.0/24",
  40: "10.0.5.0/24", 100: "10.0.6.0/24",
  200: "10.0.7.0/24", 300: "10.0.8.0/24",
}
```

Listing D.1: Example of grouping for IP Addresses

In our solution we use the first approach.

D.1.5 *Abstract Network Model*

The Abstract Network Model is the set of all Network Views. A Network View is defined as $\theta = (\Pi, E, T, B, L)$. A generalised Network View is then $\theta_i = (\Pi_i, E_i, T_i, B_i, L_i)$. This then allows us to define multiple Network Views, $\theta_1, \theta_2, \theta_3, \ldots$. The Abstract Network Model is then this set of all Network Views, $\Theta = \theta_1, \theta_2, \theta_3, \ldots$. We can *name* a Network View using subscript notation, such as $\theta_{physical}$, θ_{ospf}, *etc.*

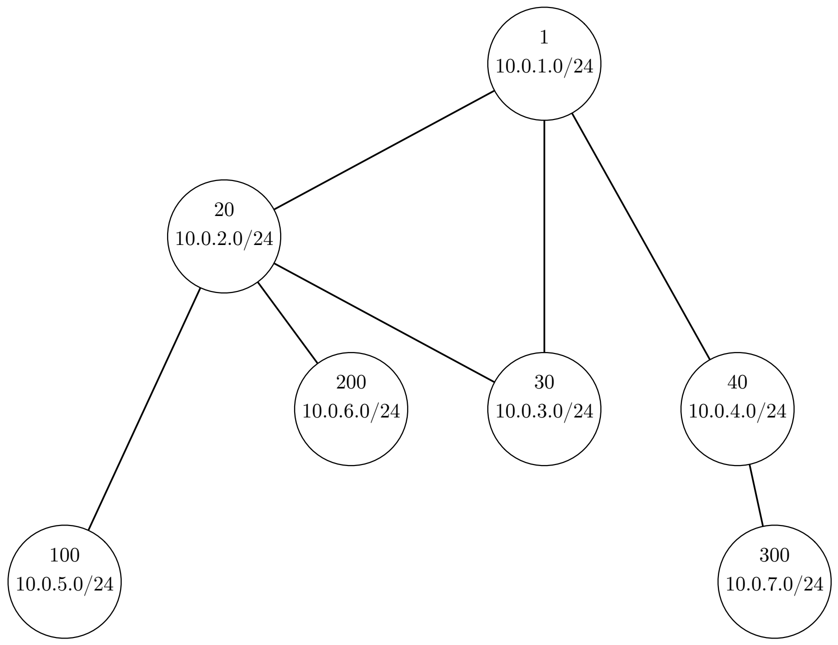


Figure D.1: Labels on Pseudo Network Elements representing Groups of Network Elements

D.1.6 *Shortcut functions for Network Views*

For notational convenience, we define the following shortcut functions to assist in explaining the Low-Level Primitives.

- $\Pi(\theta_i)$ which returns Π, the set of Network Elements, for a given Network View θ_i.
- $E(\theta_i)$ which returns E, the set of Network Element Connections, for a given Network View θ_i.
- $T(\theta_i)$ which returns T, the set of Network Element Interfaces, for a given Network View θ_i.

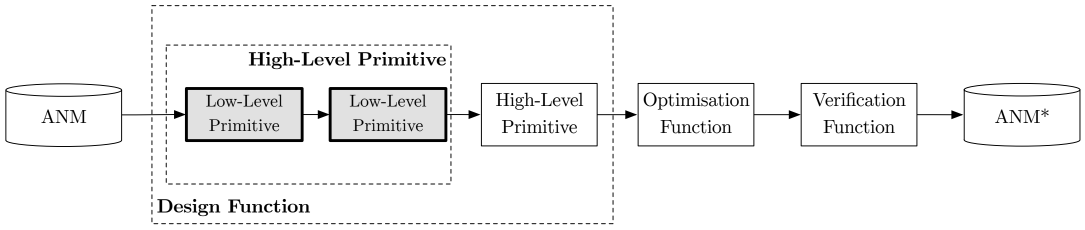
In the following sections we define some utility functions that give us information about the Network Elements, Network Element Connections, and Network Element Interfaces of Network Views.

D.2 LOW-LEVEL PRIMITIVES

D.2.1 *Introduction*

In Section D.1.6 we presented the design details for a number of Low-Level Primitives which can be used to retrieve elements or properties from a Network Whiteboard or Network View. These are read-only Low-Level Primitives.

In this section we present the design details for Low-Level Primitives that can create a Network View.



There are are four types of Low-Level Primitives used to create Network Views.

The first type *creates* individual Network Elements, Network Element Connections, and Network Element Interfaces.

The second types *adds* to the set of Network Elements, Network Element Connections, and Network Element Interfaces, for a given Network View. This could be adding a newly created element, or copying an element from the Network Whiteboard, or a different Network View.

The third type *removes* an element from the set of Network Elements, Network Element Connections, and Network Element Interfaces, for a given Network View. The removal type of Low-Level Primitive can used to prune a Network View after creation.

The final type *manipulates labels* for each of Network Elements, Network Element Connections, and Network Element Interfaces, and for labels on the Network Views themselves.

D.2.2 *Low-Level Primitives of Network Elements*

We now define some utility functions that give us information about the Network Elements.

Network Element Connections for Network Element

The Network Element Connections of a given Network Element is the set of all Network Element Connections which have a connection to the given Network Element.

In graph theory, this is the set of all edges incident to the node.

An example of NEC_FOR_NE for *Network Element 4* in the Physical Network View of the Small Internet example is shown in Figure D.2.

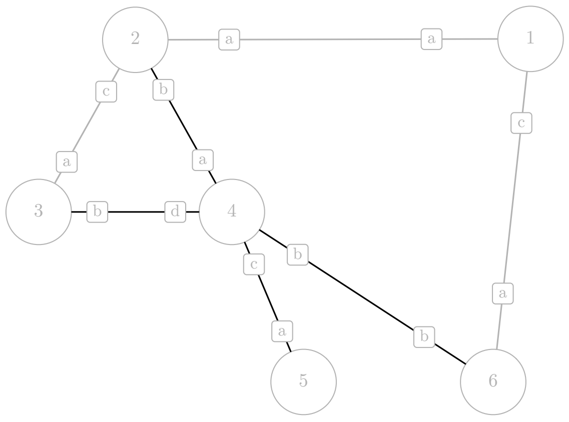


Figure D.2: NEC_FOR_NE for Network Element 4

Degree

The degree of a Network Element is defined as the number of Network Element Connections for that Network Element. This corresponds directly to the degree concept in graph theory.

An example of DEGREE of *Network Element 4* in the Physical Network View of the Small Internet example would be 4.

Presence

The Presence function is a boolean function, which returns True if a given Network Element is present in a given Network View.

The use of this function allows us to omit Network Elements from Network Views if they are not required, which can simplify Design Functions discussed in Chapter 5 and the compilation functions discussed in Chapter 6.

For instance, Network Elements with the role of *server* may not be present in the eBGP Network View. The presence function allows these to be filtered out.

Neighbour Network Elements for Network Element

The Neighbour Network Elements for a given Network Element is defined as the set of all Network Elements ofwhich have a Network Element Connection to that Network Element.

Informally it is the set of all Network Elements connected to the given Network Element.

An example of NEIGH_NE for *Network Element 4* in the Physical Network View of the Small Internet example is shown in Figure D.3.

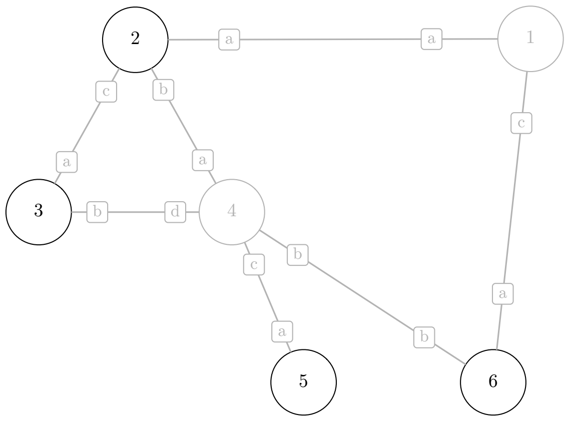


Figure D.3: NEIGH_NE for Network Element 4

Neighbour Network Element Interfaces for Network Element

The Neighbour Network Element Interfaces for a given Network Element is the set of all Network Element Interfaces in the given Network View that satisfy both of the following conditions:

- the Network Element Interface is bound to a Network Element Connection, which in turn is bound to a Network Element Interface of the given Network Element.
- the Network Element Interface does not belong to the given Network Element.

Informally, this is the set of all remote Network Element Interfaces connected to this Network Element.

An example of NEIGH_NEI_NE for *Network Element 4* in the Physical Network View of the Small Internet example is shown in Figure D.4.

Network Element Interfaces

The Network Element Interfaces for a given Network Element is the set of all Network Element Interfaces that belong to this Network Element.

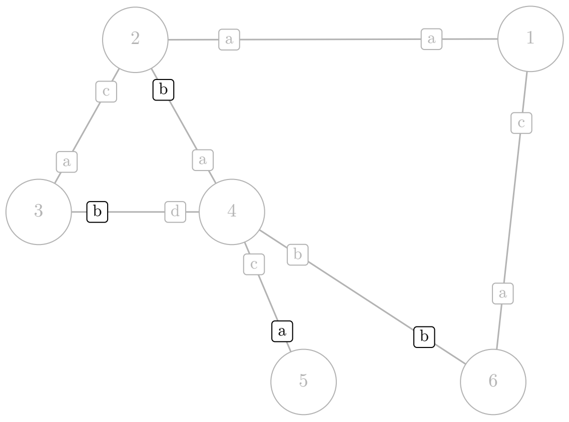


Figure D.4: NEIGH_NEI_NE for Network Element 4

This set is frequently used for iteration over the interfaces of a node. An example of NEI_FOR_NE for *Network Element 4* in the Physical Network View of the Small Internet example is shown in Figure D.14.

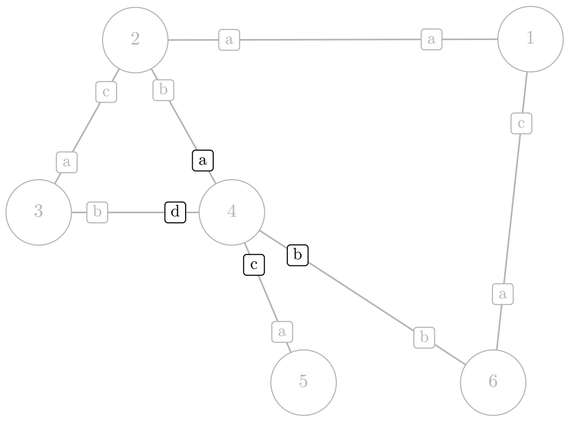


Figure D.5: NEI_FOR_NE for Network Element 4

Equality

The EQUALITY function is used to compare if two Network Elements are equal across Network Views. This compares the Network Element Identifier for both Network Elements, and returns True if they are equal. This can be used to check that two Network Elements, in different Network Views, refer to the same device.

Comparison and Sorting

We define an ordering function for an unordered set of elements, through the use of a sorting key operator, which maps a Network

Element to a unique value that can be used for sorting the set of elements.

This is used to order a set of Network Elements. This is used for instance in returning the (a,b) Network Elements from a Network Element Connection, since the Network Element Connection is defined as an *unordered* pair of Network Elements $\epsilon_i = \{u, v\} = \{v, u\}$.

We define a Sort Key function SORT_KEY that is used to sort Network Elements. We then use this to define a sorting function

$$\text{SORT}(\{\lambda_1, \lambda_2, \lambda_3, \ldots\}$$

which returns a sequence

$$\langle \lambda_i, \lambda_{i+1}, \lambda_{i+2}, \ldots)\rangle$$

such that

$$\text{SORT\_KEY}(lambda_i) < \text{SORT\_KEY}(lambda_{i+1})$$

for each pair.

D.2.3 *Low-Level Primitives of Network Element Connections*

We now define some utility functions that give us information about the Network Element Connections.

Network Elements for Network Element Connection

This Low-Level Primitive returns the Network Elements connected by a given Network Element Connection. An example of NE_FOR_NEC for a given Network Element Connection (highlighted in the figure) in the Physical Network View of the Small Internet example is shown in Figure D.6.

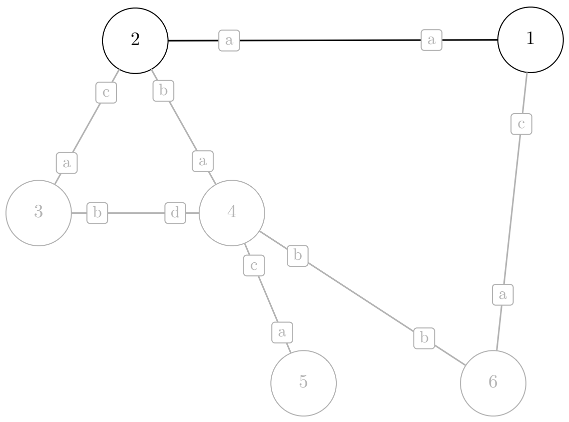


Figure D.6: NE_FOR_NEC result highlighted for Network Element 4

Network Element Interfaces for Network Element Connection

This Low-Level Primitive returns the Network Element Interfaces connected by a given Network Element Connection. An example of NEI_FOR_NEC for a given Network Element Connection (highlighted in the figure) in the Physical Network View of the Small Internet example is shown in Figure D.7.

Ordered Network Elements for Network Element Connection

Since our base graph is undirected, a Network Element Connection is defined as an unordered pair of (u, v, δ), we define a pair of functions that consistently return the same Network Element u or v, for the same Network Element Connection, ϵ, and δ is a unique *distinguisher* between Network Element Connections of the same $\{u, v\}$ pair (i.e for parallel Network Element Connections).

For notational convenience, we define the following functions:

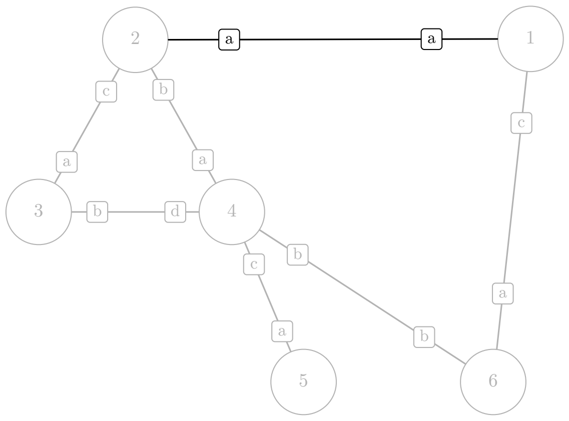


Figure D.7: NEI_FOR_NEC for the highlighted Network Element Connection

- $\Pi_u(\epsilon_i)$ which returns the *u* Network Element for the Network Element Connection ϵ_i
- $\Pi_v(\epsilon_i)$ which returns the *v* Network Element for the Network Element Connection ϵ_i.

Note that Network Element Connections are defined as an unordered pair of Network Elements. This simplifies the creation and comparison, particularly for the High-Level Primitives.

The Π_u and Π_v functions are therefore as shown in the pseudocode example in Algorithm E.21.

Other Network Element

This Low-Level Primitive returns the other Network Element for a given Network Element, and Network Element Connection. An example of NEC_OTHER_NE for a given Network Element Connection (highlighted in the figure), and Network Element 2 in the Physical Network View of the Small Internet example is shown in Figure D.8.

Other Network Element Interface

This Low-Level Primitive returns the other Network Element Interface for a given Network Element Interface and Network Element Connection.

$\mathtt{other}(e,(u,t_u))$ returns \\) returns \\)

An example of NEC_OTHER_NEI for a given Network Element Connection (highlighted in the figure), and Network Element Interface *a* of Network Element 2 in the Physical Network View of the Small Internet example is shown in Figure D.9.

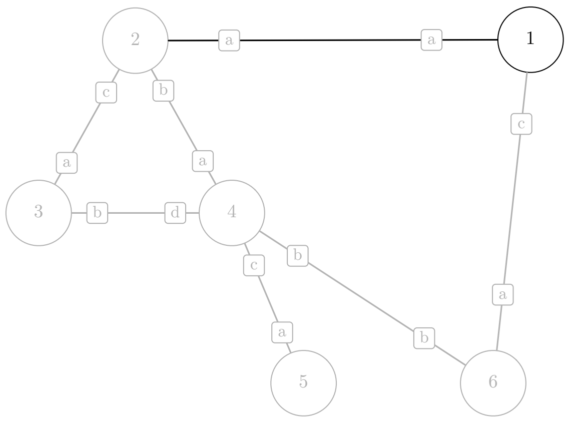


Figure D.8: nec_other_ne for the highlighted Network Element Connection and Network Element 2

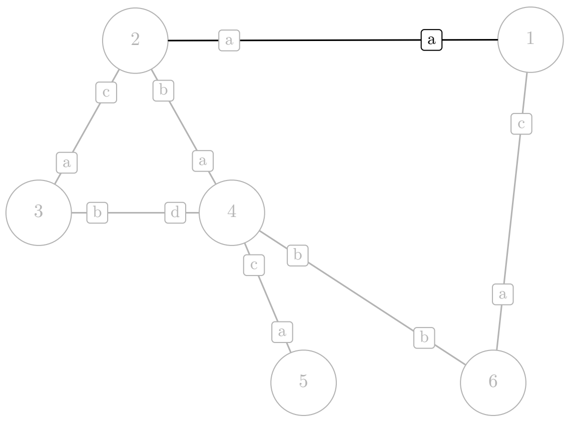


Figure D.9: nec_other_nei for the highlighted Network Element Connection and Network Element Interface *b* of Network Element 2

Distinguisher Function

To distinguish parallel Network Element Connections, each Network Element Connection also has an *index*. For visual clarity, if a Network View does not have any parallel Network Element Connections, then the distinguisher may be omitted from the figure. Parallel Network connections are shown in the diagram below in Figure D.10.

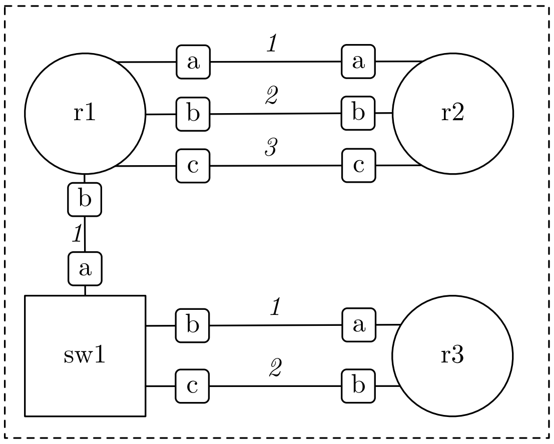


Figure D.10: Parallel Network Element Connections. Network Element Connection Distinguishers are shown by the italicised numbers above each parallel Network Element Connection.

We define an Network Element Connection Index Access function $E_\delta(\epsilon_i)$ which returns the index for Network Element Connection ϵ_i.

Network Element Connection by Network Element endpoints

This Low-level Primitive returns the set of Network Element Connections for a given pair of Network Elements. An example of NEC_BY_NE_NE for Network Elements 1 and 2 in the Physical Network View of the Small Internet example is shown in Figure D.11.

Parallel Network Element Connections

The Parallel Network Element Connections for a given Network Element Connection is defined as the set of all Network Element Connections in the given Network View, which connect the same pair of Network Elements.

It should be noted that the Parallel Network Element Connections are defined in terms of the Network Element pairs, not their Network Element Interface Bindings. This is because it is more common to have parallel connections between a node pair, such as two physical links between the same set of devices. However it is rare to have par-

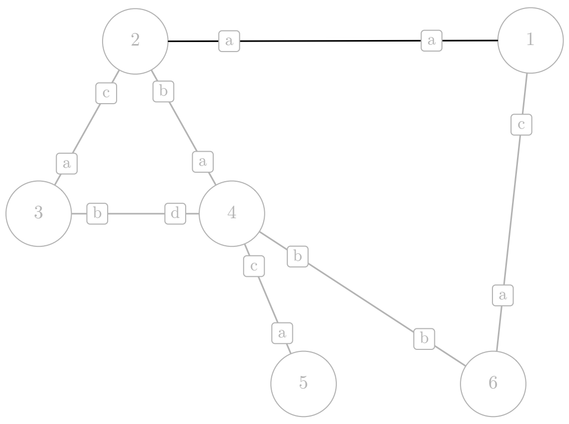


Figure D.11: NEC_BY_NE_NE for the Network Elements *1* and *2*

allel connections with the same Network Element Interface Bindings. The set of Network Element Connections sharing the same pair of Network Element Interfaces can be found by further refining the set of Parallel Network Element Connections.

$(NE_{src}, NEI_{src}, NE_{dst}, NEI_{dst})$

Detailed pseudocode for this example is found in Algorithm E.25.

Is Parallel

This Low-Level Primitive returns a boolean value depending on whether the given Network Element Connection has parallel Network Element Connections. A Network Element Connection is defined as Parallel if there exists at least one other Network Element Connection that connects the same pair of Network Elements. An example pseudocode definition is provided in Algorithm E.26.

For the Network View shown in Figure D.10, the results for IS_PARALLEL is as follows in Table D.2.

Similarly, we can define is parallel for a Network View if it contains at least one parallel Network Element Connection.

Table D.2: Return values for IS_PARALLEL for topology shown in Figure D.10

π_i	π_j	δ	IS_PARALLEL(π_i, π_j, δ)
r1	r2	1	True
r1	r2	2	True
r1	r2	3	True
r1	sw1	1	False
sw1	r3	1	True
sw1	r3	2	True

Equality

This primitive determines if two Network Element Connections are equivalent. Order is not important in determining equality. We consider two Network Element Connections to be equivalent if they connect the same pair of Network Element Connections. If there are parallel Network Element Connections in a Network View, then this can be represented using a Multigraph. For this case, we use the NetworkX Python Library [93] approach to compare, where we compare the Network Element at each end of the Network Element Connection, and the Network Element Connection distinguisher.

D.2.4 *Low-Level Primitives of Network Element Interfaces*

We now define some utility functions that give us information about the Network Element Interfaces.

Network Element Interface by Network Element Connection and Network Element

To determine the Network Element Interface, based on the Network Element and the Network Element Connection, this primitive can be used.

The binding access Low-Level Primitive is called NEI_BY_NEC_NE(ϵ_i, π_j). It return the Network Element Interface τ_k which is bound for the Network Element pi_j on Network Element Connection ϵ_i.

For instance for the Simplified Small Internet Example, as shown in Table C.3, for ϵ_1, the Network Element Connection between 1 and 2, NEI_BY_NEC_NE(ϵ_1, pi_1) would return π_1, τ_a, as shown in Figure D.12. NEI_BY_NEC_NE(ϵ_1, π_2) would return π_2, τ_a.

For another Network Element Connection, ϵ_2, between 1 and 6, NEI_BY_NEC_NE(ϵ_2, π_1) would return τ_c and NEI_BY_NEC_NE(ϵ_2, π_6) would return τ_a.

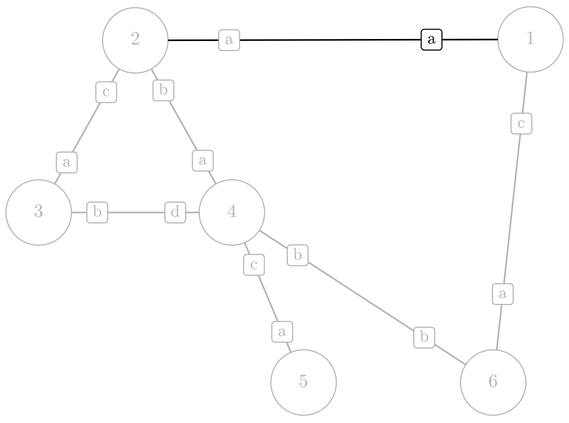


Figure D.12: NEI_BY_NEC_NE for the highlighted Network Element Connection and Network Element 2

Network Element Connection for Network Element Interface

This primitive returns all the Network Element Connections associated with a particular Network Element Interface. The formal of this query is provided in the pseudo-code example in Algorithm E.28.

An example of NEC_FOR_NEI for *Network Element Interface b* of *Network Element 4* the Physical Network View of the Small Internet example is shown in Figure D.13. This example is also valid for Network Element Interface *b* of Network Element *6*

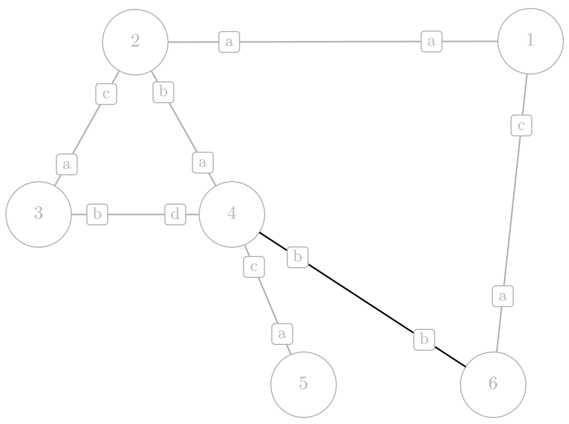


Figure D.13: NEC_FOR_NEI for Network Element Interface *b* of Network Element *4*

Ordered Network Element Interface for Network Element Connection

- $T_u(\epsilon_i)$ which returns the Network Element Interface that is bound for Network Element *u* in the Network Element Connection $\epsilon_i = \{u, v\}$
- $T_v(\epsilon_i)$ which returns the Network Element Interface that is bound for Network Element *v* in the Network Element Connection $\epsilon_i = \{u, v\}$

The T_u and T_v functions are therefore as shown in the pseudocode example in Algorithm E.29.

Network Element Interfaces From Network Element

The $B_\Pi(\theta_i, \pi_j)$ Low-Level Primitive returns all Network Element Interfaces for π_j in $theta_i$. This is shown in the pseudo-code example in Algorithm E.30. An example of NEI_BY_NE for *Network Element 4* of the Physical Network View of the Small Internet example is shown in Figure D.14.

Degree

The DEGREE Low-Level Primitive for a Network Element Interface is defined as the number of Network Element Connections bound to the Network Element Interface.

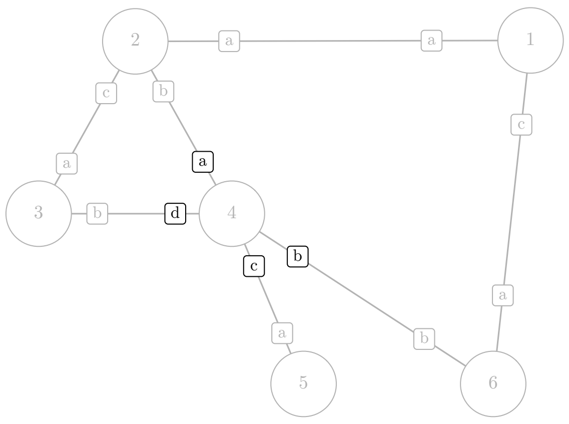


Figure D.14: NEI_BY_NE for Network Element 4

For each of the Network Views shown in the Small Internet example, the Network Element Interface DEGREE is 1. An example definition shown in the pseudocode example in Algorithm E.31.

Is Bound

This is a boolean Low-Level Primitive that can be used to check if a Network Element Interface is attached to any Network Element Connections.

This is simply a simple test if the degree of a Network Element Interface is greater than zero.

An example pseudo-code definition is provided in Algorithm E.32.

Parent Network Element

This Low-Level Primitive returns the Network Element for a given Network Element Interface. As Network Element Interfaces are referred to as a tuple of (Network Element, Network Element Interface identifier), this returns the Network Element referred to in the first part of the tuple.

Neighbour Network Element Interfaces for Network Element Interface

The Neighboring Network Element Interfaces for a given Network Element Interface, is defined as the set of of all Network Element Interfaces which share a Network Element Connection to the given Network Element Interface. Informally, it is the Interfaces which are connected to the Interface. An example of NEIGH_NEI_FOR_NEI for *Network Element Interface a* of *Network Element 2* the Physical Network View of the Small Internet example is shown in Figure D.15.

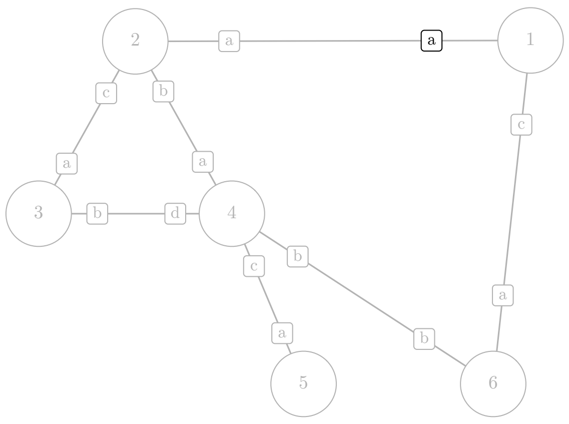


Figure D.15: NEIGH_NEI_FOR_NEI for Network Element Interface *a* of Network Element 2

Neighbour Network Elements for Network Element Interface

This is defined from the neighbor Network Element Interface and the Network Element Interface function on the λ_i of the (lambda, tau) pair for the Network Element Interface. An example of NEIGH_NE_FOR_NEI for *Network Element Interface a* of *Network Element 2* the Physical Network View of the Small Internet example is shown in Figure D.16.

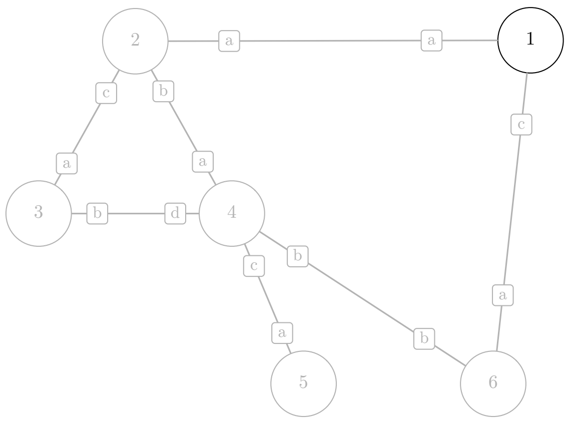


Figure D.16: NEIGH_NE_FOR_NEI for Network Element Interface *a* of Network Element 2

Network Element Connections

This Low-Level Primitive returns the Network Element Connections to which a Network Element Interface is bound. Typically this would be a single Network Element Connection per Network Element Interface, but this is not always the case. For instance, in higher-level protocol views, such as OSPF, created from exploding devices such as hubs or switches at lower layers, a Network Element Interface may have multiple Network Element Connections, representing a multi-point protocol interface.

An example of NEC_FOR_NEI for *Network Element Interface a* of *Network Element 2* of the Physical Network View of the Small Internet example is shown in Figure D.17. Note that this example would also be valid for *Network Element Interface a* of *Network Element 1*.

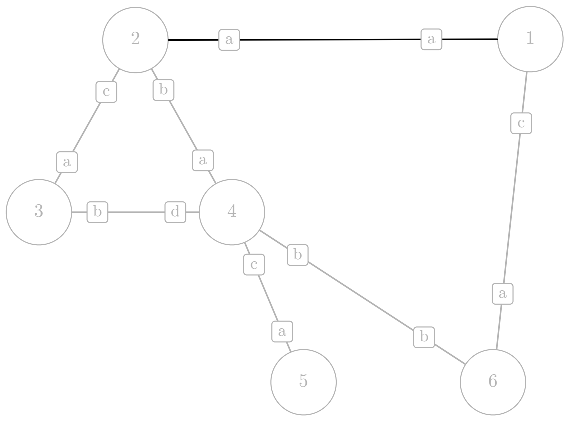


Figure D.17: NEC_FOR_NEI for Network Element Interface *a* of Network Element 2

D.2.5 *Creation*

These functions create a new Network Element, Network Element Connection, or Network Element Interface.

Indices

These functions are used internally for the addition function, to uniquely identify an element.

Recall $\Pi = \{\pi_1, \pi_2, \pi_3, \ldots\}$.

The index function for a given element, π_i, ϵ_j or τ_k returns the i value for a given set element.

- $index_\Pi(\pi_i)$ returns i for π_i.
- $index_E(\epsilon_i)$ returns i for ϵ_i.
- $index_T(\tau_i)$ returns i for τ_i.

We can then define the indices function for a set of elements:

- $indices_\Pi(\theta_i) = \{(index_\Pi(\pi_j)\forall\pi_j \in \Pi(\theta_i)\}$
- $indices_E(\theta_i) = \{(index_E(\epsilon_j)\forall\epsilon_j \in E(\theta_i)\}$
- $indices_T(\Pi_i, \pi_j) = \{(index_T(\tau_k)\forall\tau_k \in B_\Pi(\theta_i, \pi_j)\}$

These can also be defined by looking at the ANM labels for Pi and Tau.

Creation Primitives

Table D.3: Summary of *create* Low-Level Primitives

Function	Pseudo-Code	Description
$create\_\theta(\Theta)$	Creates a Network View θ_i within set Abstract Network Model Θ	
$create\_\pi(\Pi)$	Creates a Network Element π_i within set of Network Elements Π	
$create\_\epsilon(E)$	Creates a Network Element Connection ϵ_i within set of Network Element Connections E	
$create\_\tau\_\pi(T)$	Creates a Network Element Interface within set of Network Element Interfaces T	

D.2.6 *Add Primitives*

Once an element has been created then it can be added to the set of elements making up the Network View. This is typically done in a single step using a High-Level Primitive.

For each of the elements that can be created using the primitives in the previous section, we have an add primitive. These functions are described in more detail in Appendix F, and are summarised in Table D.4.

Once an element has been created then it can be added to the set of elements making up the Network View. This is typically done in a single step using a High-Level Primitive.

For each of the elements that can be created using the primitives in the previous section, we have an add primitive. These functions are described in more detail in Appendix E and are summarised in Table D.4.

Table D.4: Summary of *add* Low-Level Primitives

Function	Pseudo-Code	Description
$add\_\pi(\Pi, \pi_i)$	Algorithm E.35	Adds Network Element π to Network View
$next\_nec\_\delta(\theta_i, \pi_i, \pi_j)$	Algorithm E.35	Obtains next free distinguisher δ for Network Element pair π_i, π_j.
$add\_\epsilon(\pi_i, \pi_i, \delta)$	Algorithm E.37	Adds Network Element Connection ϵ to Network View
$add\_b(B, \epsilon_i)$	Algorithm E.38	Adds Network Element Binding B to Network Element Connection ϵ_i
$add\_\pi_t(T, \pi_i)$	Algorithm E.39)	Adds Network Element Interface to Network Element Interface Set T on Network Element π_i
$set\_binding(B, \epsilon_i, \pi_j, \tau_k)$	Algorithm E.41	Binds Network Element Interface τ_k for Network Element π_j on Network Element Connection ϵ_i

D.2.7 *Remove Primitives*

Occasionally we need to remove an element from a Network View after they have been created. In some situations a Network View can be generated by bulk addition of Network Elements and then pruning of some elements. This motivates the need for the removal primitives.

There are a series of low-level "internal" removal functions that remove a given element from a given set. There are a series of high-level primitives that are composed of these low-level internal functions but maintain structural integrity and perform clean-up of unused elements for consistency. These are summarised in Table D.5.

Table D.5: Summary of *remove* Low-Level Primitives

Function	Pseudo-Code	Description
$remove\_\pi(\Pi, \pi_i)$	Removes Network Element π_i from set of Network Elements Π	
$remove\_\epsilon(E, \epsilon_i)$	Removes Network Element ϵ_i from set of Network Element Connections E	
$remove\_\tau\_\pi(T, \pi_i)$	Removes Network Element Interface set for Network Element π_i from set of Network Element Interfaces T on Network View.	
$remove\_b(B, \epsilon_i)$	Removes Network Element Connection ϵ_i from set of Network Element Bindings B	
$remove\_\tau\_\tau(\pi_i, \tau_j)$	Removes Network Element Interface τ_j from set of Network Element Interfaces on Network Element π_i	

D.2.8 *Label Primitives*

When we create a Network View we need to create placeholders to store the labels, which are empty sets. Once the empty set is created, the labels are inserted. We also have remove function. These are shown in the Table D.6 and Table D.7. The *set* function for labels is used to store a specific value for a given label key. The set labels Low-Level Primitives are shown in Algorithm E.65.

Table D.6: Summary of *initialise label* Low-Level Primitives

Function	Pseudo-Code	Description
$add\_l\_\pi(L, \pi_i)$	Algorithm E.49	Initialises label set for Network Element π_i in Network View Label set L.
$add\_l\_\epsilon(L, \epsilon_i)$	Algorithm E.48	Initialises label set for Network Element Connection ϵ_i in Network View Label set L.
$add\_l\_t\_\pi(L, \pi_i)$	Algorithm E.50	Initialises label set for Network Element Interfaces for Network Element π_i in Network View Label set L.
$add\_l\_t\_\tau(L, \pi_i, \tau_j)$	Algorithm E.51	Initialises label set for Network Element Interfaces τ_j in Network Element Interfaces label set for Network Element π_i in Network View Label set L.

D.2.9 *Conclusion*

In this section we have outlined a number of Low-Level Primitives that obtain properties of or modify Network Views. These include creation of the Network View, adding elements to the Network View including Network Elements and Network Element Connections.

Network element interfaces are a requirement of Network Element Connections, and can either be created as logical Network Element Interfaces, or the physical Network Element Interfaces can be used, as the termination point of a Network Element Connection.

We also discussed how labels are added on these elements, and can be modified, in order to represent network configuration properties, that are then used in the compilers to generate device specific configurations.

Table D.7: Summary of *remove label* Low-Level Primitives

Function	Pseudo-Code	Description
$remove\_l\_\pi(L, \pi_i)$	Algorithm E.52	Removes label set for Network Element π_i in Network View Label set L.
$remove\_l\_\epsilon(L, \epsilon_i)$	Algorithm E.55	Removes label set for Network Element Connection ϵ_i in Network View Label set L.
$remove\_l\_\tau\_\pi(L, \pi_i, \tau_j)$	Algorithm E.53	Removes label set for Network Element Interfaces for Network Element π_i in Network View Label set L.
$remove\_l\_\tau\_\tau(L, \pi_i, \tau_j)$	Algorithm E.54	Removes label set for Network Element Interfaces τ_j in Network Element Interfaces label set for Network Element π_i in Network View Label set L.

In the next section we show how these Low-Level Primitives are assembled into High-Level Primitives.

D.3 HIGH-LEVEL PRIMITIVES

D.3.1 *Create Skeleton Network View*

A Network View is created in stages. The first stage is to create a skeleton, which is a placeholder for the remaining elements. In this step, we apply the create theta, which is an empty set of elements such as Network Elements, Network Element Connections, and labels. An important part contained in the theta placeholder is an empty label set.

D.3.2 *Network Element High-Level Primitives*

This High-Level Primitive adds the given Network Element to the set of Network Elements for the given Network View. It also creates the set of Network Element Interfaces for the Network Element, in the Network View, and the set of Labels for the Network Element and the Network Element Interfaces of the Network Element, on the Network View. Finally, it will set a role label for the Network Element, such as router or switch. This High-Level Primitive is used by both the Copy

Network Element, and Create Pseudo Network Element Primitives, discussed in the next section.

Copy Network Element

This High-Level Primitive copies a given Network Element from one Network View to another. In addition to copying the Network Element, the appropriate set for Network Element Interfaces of the Network Element, and labels for the Network Element and it's Network Element Interfaces, are also created on the target Network View.

Create Pseudo Network Element

This High-Level Primitive creates a new Pseudo Network Element. Pseudo Network Elements do not correspond to physical devices, but are used to help in the representation of network policy and network concepts.

D.3.3 *Network Element Interface High-Level Primitives*

Add Network Element Interface

Network Element Interfaces are used to represent the termination point of a Network Element Connection on a Network Element. This High-Level Primitive creates adds and copies Network Element Interfaces.

CREATE LOGICAL NETWORK ELEMENT INTERFACE In addition to physical interfaces, Network Views can contain Logical Network Element Interfaces, which are analogous to Pseudo-Nodes. These can include loopback interfaces.

Copy Network Element Interface

note that physical and loo are automatically copied across interfaces
For logical interfaces

Create Logical Network Element Interface

Cannot create a Network Element (as represents *physical* Network Element from Network Whiteboard), but can create Pseudo Network Element

Step 1: find identifier that's not used in *any* NV Step 2: create new Pseudo Network Element, *p*, with this identifier Step 3: return (Network View, *p*)

Set Network Element Interface Binding

Network Element Connections are bound to Network Element Interfaces. There are High-Level Primitives to do this.

D.3.4 *Network Element Connection High-Level Primitives*

High-Level Primitives exist for adding, copying and removing Network Element Connections.

D.3.5 *Set Label High-Level Primitives*

This High-Level Primitive is responsible for storing a label for a given key.

D.3.6 *Removal*

The *removal* High-Level Primitives take care of removing the entities from the Network View in a safe way so that the Network View maintains its structural integrity after the removal has taken place.

For example, to remove a Network Element Interface, we need to perform the following steps. The first step is to remove the Network Element Interface from the set of Network Element Interfaces, T. The next step is to unbind the Network Element Interface from the Network Element Connection. Finally, the Network Element Interface is removed from the set of labels for Network Element Interfaces.

E

DEFINITIONS OF PRIMITIVES

E.1 LOW-LEVEL PRIMITIVE DEFINITIONS FOR CHAPTER 4

Algorithm E.18 Definition for E_Π Low-level Primitive

```
function $E_\Pi((\theta_i, \pi_j))$
    $Y\{\epsilon_i \forall \epsilon_i \in E(\theta_i) \quad | \quad \pi_j \in E_\Pi(\epsilon_i)\}$
    return $Y$
end function
```

Algorithm E.19 Definition for DEGREE_Π Low-level Primitive

```
function $Degree_\Pi((\theta_i, \pi_j))$
    $Y \leftarrow E_\Pi(\theta_i, \pi_j, \tau_k)$    ▷ Network Element Connections for this Network Element
    return $|Y|$
end function
```

Algorithm E.20 Definition for Π_E Low-level Primitive

```
function $\Pi_E(\theta_i, \epsilon_j = (\{\lambda_i, \lambda_j\}, distinguisher))$
    return $\{\lambda_i, \lambda_j\}$
end function
```

Algorithm E.21 Definition for Π_u and Π_v Low-level Primitives

```
function $\Pi_u(\theta_i, \epsilon_i)$
    $X \leftarrow \Pi_E(\theta_i, \epsilon_i)$                         ▷ NEs $\{u, v\}$
    $S \leftarrow \textsc{sort}(X)$
     return $S_1$                         ▷ Return first sorted element
end function
function $\Pi_v(\theta_i, \epsilon_i)$
    $X \leftarrow \Pi_E(\theta_i, \epsilon_i)$                         ▷ NEs $\{u, v\}$
    $S \leftarrow \textsc{sort}(X)$
     return $S_2$                        ▷ Return second sorted element
end function
```

Algorithm E.22 Definition for other_Π Low-level Primitive

```
function $\text{other}_\Pi(\theta_i, \epsilon_j, \pi_k)$
    $u \leftarrow \Pi_u(\theta_i, \epsilon_j)$
    $v \leftarrow \Pi_v(\theta_i, \epsilon_j)$
    if $u = \pi_k$ then
     return $v$
    else
     return $u$
    end if
end function
```

Algorithm E.23 Definition for other_T Low-level Primitive

```
function $\text{other}_T(\theta_i, \epsilon_j, \pi_k, \tau_l)$
    $\pi_v \leftarrow \text{other}_\Pi(\theta_i, \epsilon_j, \pi_k)$                   ▷ Other NE
    $\tau_v \leftarrow E_T(\theta_i, \epsilon_j, \pi_v)$          ▷ NEI binding in NEC for other NE
     return $(\pi_v, \tau_v)$
end function
```

Algorithm E.24 Definition for NEC_BY_ENDPOINTS Low-level Primitive

```
function NEC_BY_NE_NE($\theta_i, \pi_j, \pi_k$)
    $X \leftarrow E_\Pi(\pi_j) \cap E_\Pi(\pi_k)$            ▷ All NECs incident to both NE
endpoints
     return $X$
end function
```

Algorithm E.25 Definition for PARALLEL Low-level Primitive

```
function PARALLEL($\theta_i, \epsilon_j$)
    $X \leftarrow \Pi(\epsilon_j)$                                   ▷ NEs for $\epsilon_j$
    $Y_1 \leftarrow (E(\theta_i))$                                   ▷ All NECs for NV $\theta_i$
    $Y_2 \leftarrow \{(\epsilon \forall \epsilon \in Y_1 \mid \Pi(\epsilon) \in X\}$    ▷ NECs incident to NEs of $\epsilon_j$
    $Y_3 \leftarrow Y_2 - \{\epsilon_j\}$                            ▷ Remove $\epsilon_j$ from set
     return $Y$
end function
```

Algorithm E.26 Definition for IS_PARALLEL_E Low-level Primitive

```
function IS_PARALLEL_E($\theta_i, \epsilon_j$)
    $Y \leftarrow$ PARALLEL($\theta_i, \epsilon_j$)
    if $|Y| > 0$ then
     return True
    else
     return True
    end if
end function
```

Algorithm E.27 Definition for E_T Low-level Primitive

```
function $E_T((\theta_i = (\Pi, E, T, B, L), \epsilon_j, \pi_k))$
    $bindings \leftarrow$ GET_BY_KEY($B, \epsilon_j$)                ▷ Bindings for this NEC
    $\tau_l \leftarrow$ GET_BY_KEY($bindings, \pi_k$)
     return $(\pi_k, \tau_l)$
end function
```

Algorithm E.28 Definition for E_T Low-level Primitive

```
function $NEI\_NEC((\lambda_i, \pi_j, \tau_k))$
    $Y \leftarrow E(\lambda_i)$                       ▷ Network Element Connections for $\lambda_i$
    $Y' \leftarrow \{(\epsilon_i \forall \epsilon_i \in Y | NEC\_NEI(\epsilon_i, \pi_j) = \tau_k\}$        ▷ Filter to only
Network Element Connections bound to $\tau_k$ for $\pi_j$
     return $Y'$
end function
```

Algorithm E.29 Definition for T_u and T_v Low-level Primitives

```
function $T_u(\epsilon_i)$
    $\pi_u \leftarrow \Pi\_u(\epsilon_i)$                ▷ First NE for NEC
    $x \leftarrow E_T(\theta_i, \epsilon_j, \pi_u)$      ▷ Bound NEI for first NE
     return $x$
end function
function $T_v(\epsilon_i)$
    $\pi_v \leftarrow \Pi\_v(\epsilon_i)$                ▷ First NE for NEC
    $x \leftarrow E_T(\theta_i, \epsilon_j, \pi_v)$      ▷ Bound NEI for second NE
     return $x$
end function
```

Algorithm E.30 Definition for B_π Low-level Primitive

```
function $B_\pi(\epsilon_i, \pi_j)$
    $X \leftarrow$ GET_BY_KEY$(B(\theta_i, \pi_j))$      ▷ Set of NEI for NE
    $Y = \{(\pi_j, \tau_k) \quad \forall \quad \tau_j \in X\}$
     return $Y$
end function
```

Algorithm E.31 Definition for $Degree_T$ Low-level Primitive

```
function $Degree_T((\theta_i, \pi_j, \tau_k))$
    $Y \leftarrow NEC\_NEI(\theta_i, \pi_j, \tau_k)$   ▷ Network Element Connections for
this Network Element Interface
     return $|Y|$
end function
```

Algorithm E.32 Definition for T_IS_BOUND Low-level Primitive

```
function E_IS_BOUND$(\theta_i, \pi_j, \tau_k)$
    $Y \leftarrow Degree_T(\theta_i, \pi_j, \tau_k)$
    if $|Y| > 0$ then
     return True
    else
     return True
    end if
end function
```

E.2 LOW-LEVEL PRIMITIVE DEFINITIONS FOR CHAPTER 5

E.2.1 *Add Primitives*

Algorithm E.33 Definition for CREATEθ Low-Level Primitive

```
function CREATEθ($\Theta$, name)
    $\Theta \leftarrow \Theta \cup \theta_{name}$
      return $\theta_{name}$
end function
```

Algorithm E.34 Definition for ADD_π Low-Level Primitive

```
function CREATE_π(($\Pi$)
      return $\Pi' = \Pi \cap \{\pi_i\}$
end function
```

Algorithm E.35 Definition for ADD_π Low-Level Primitive

```
function ADD_π($\Pi, \pi_i$)
      return $\Pi' = \Pi \cup \{\pi_i\}$
end function
```

Algorithm E.36 Definition for NEXT_NEC_δ Low-Level Primitive

```
function NEXT_NEC_δ($\theta_i, \pi_i, \pi_j$)
    $E \leftarrow$ NEC_BY_NE_NE$(\theta_i, \pi_i, \pi_j)$  ▷ NECs for this (NE, NE) pair
    $D \leftarrow \{E_\delta(\epsilon_i) \quad \forall \quad \epsilon_i \in E\}$        ▷ distinguishers on these NECs
    for $z \in \mathbb{Z}$ do
        if $z \notin D$ then                        ▷ The next free distinguisher
            $\delta \leftarrow z$ return $\delta$
        end if
    end for
end function
```

Algorithm E.37 Definition for ADD_ϵ Low-Level Primitive

```
function ADD_ϵ($E, \epsilon_i = (\pi_i, \pi_i, \delta)$)
      return $E' = E \cup \{\epsilon_i\}$
end function
```

Algorithm E.38 Definition for ADD_B Low-Level Primitive

```
function ADD_B($B, \epsilon_i$)
    return $B' = B \cup \{(\epsilon_i, \varnothing)\}$    ▷ Add and create empty set for NEI bindings
end function
```

Algorithm E.39 Definition for ADD_π_T Low-Level Primitive

```
function ADD_π_T($T, \pi_i$)
    $T' \leftarrow T \cup \{(\pi_i, \varnothing)\}$    ▷ Add with empty NEI set for NE
    return $T'$
end function
```

Algorithm E.40 Definition for ADD_T Low-Level Primitive

```
function ADD_T($\pi_i, \tau_j$)
    $T_{\pi_i} \leftarrow$ GET_BY_KEY($T, \pi_i$)    ▷ Current NEI set for NE
    $X' \leftarrow X \cup \{\tau_j\}$    ▷ add $\tau_j$ to NEI set
    $T' \leftarrow T - (\pi_i, T_{\pi_i})$    ▷ remove old NEI set for NE
    $T' \leftarrow T' \cup (\pi_i, X')$    ▷ Add new NEI set for NE
    return $T'$
end function
```

Algorithm E.41 Definition for SET_BINDING Low-Level Primitive

```
function SET_BINDING($B, \epsilon_i, \pi_j, \tau_k$)
    $X \leftarrow$ GET_BY_KEY($B, \epsilon_i$)    ▷ Bindings for this NEC
    $X' \leftarrow$ SET_BY_KEY($X, \pi_j, \tau_k$)
    $B' \leftarrow$ SET_BY_KEY($B, \epsilon_i, X'$)
    return $B'$
end function
```

Algorithm E.42 Definition for ADD_NEC_BY_NE_NE Low-Level Primitive

```
function ADD_NEC_BY_NE_NE($\theta_i, \pi_i, \pi_j$)
    $\tau_i \leftarrow$ NEC_BY_NE_NE($\theta_i, \pi_i, \pi_j$)
    $D \leftarrow \{E_\delta(\epsilon_i) \quad \forall \quad \epsilon_i \in E\}$
    for $z \in \mathbb{Z}$ do
        if $z \notin D$ then return $z$
        end if
    end for
end function
```

E.2.2 *Remove Primitives*

Algorithm E.43 Definition for REMOVE_π Low-Level Primitive

```
function REMOVE_π(($\Pi, \pi_i$)
    return $\Pi' = \Pi - \{\pi_i\}$
end function
```

Algorithm E.44 Definition for REMOVE_B Low-Level Primitive

```
function REMOVE_B(($B, \epsilon_i$)
    return $B' = B - \{(\epsilon_j, \{(\pi_i, \tau_i), (pi_j, \tau_j)\}) \in B | \epsilon_j = epsilon_i\}$
end function
```

Algorithm E.45 Definition for REMOVE_τ_π Low-Level Primitive

```
function REMOVE_τ_π($T, \pi_i$)
    $T_{\pi_i} = (\Pi_i, X) \leftarrow \{(\pi_j, \{\tau_1, \tau_2, \tau_3, \ldots\} \in T | \pi_j = \pi_i\}$    ▷ NEI set for NE $\pi_i$
    $T' \leftarrow T - T_{\pi_i}$    ▷ Remove old NEI set for $\pi_i$
    return $T'$
end function
```

Algorithm E.46 Definition for REMOVE_τ_τ Low-Level Primitive

```
function REMOVE_τ_τ(($\pi_i, \tau_j$)
    $T_{\pi_i} = (\Pi_i, X) \leftarrow \{(\pi_j, \{\tau_1, \tau_2, \tau_3, \ldots\} \in T | \pi_j = \pi_i\}$    ▷ NEI set for NE $\pi_i$
    $X' \leftarrow X - \tau_j$    ▷ Remove $\tau_j$ from set of NEI for NE $\pi_i$
    $T' \leftarrow T - T_{\pi_i}$    ▷ Remove old NEI set
    $T' \leftarrow T' \cap (\pi_i, X')$    ▷ Update with new NEI set
    return $T'$
end function
```

Algorithm E.47 Definition for REMOVE_ϵ Low-Level Primitive

```
function REMOVE_ϵ(($E, \epsilon_i$)
    return $E' = E - \{\epsilon_i\}$
end function
```

E.2.3 *Label Primitives*

Algorithm E.48 Definition for ADD_L_ϵ Low-Level Primitive

```
function ADD_L_ϵ($L = (L_\Theta, L_\Pi, L_E, L_T), \epsilon_i$)
    $L'_E = L_E \cup \{(\epsilon_i, \varnothing)\}$
    return $L' = (L_\Theta, L_\Pi, L'_E, L_T)$
end function
```

Algorithm E.49 Definition for ADD_L_π Low-Level Primitive

```
function ADD_L_π($L = (L_\Theta, L_\Pi, L_E, L_T), \pi_i$)
    $L'_\Pi = L_\Pi \cup \{(\pi_i, \varnothing)\}$
    return $L' = (L_\Theta, L'_\Pi, L_E, L_T)$
end function
```

Algorithm E.50 Definition for ADD_L_T_π Low-Level Primitive

```
function ADD_L_T_π($L = (L_\Theta, L_\Pi, L_E, L_T), \pi_i$)
    $L'_T \leftarrow L'_T \cup (\pi_i, \varnothing)$                ▷ Update with new NEI set
    return $L' = (L_\Theta, L_\Pi, L_E, L'_T)$
end function
```

Algorithm E.51 Definition for ADD_L_T_τ Low-Level Primitive

```
function ADD_L_T_τ($L = (L_\Theta, L_\Pi, L_E, L_T), \pi_i, \tau_j$)
    $X \leftarrow$ GET_BY_KEY($L_T, \pi_i$)                    ▷ NEI label set for NE
    $X' \leftarrow X \cup \{(\tau_j, \varnothing)\}$              ▷ add emtpty label set for $\tau_j$
    $L'_T \leftarrow L_T - L_{T_{\pi_i}}$                        ▷ remove old NEI label set
    $L'_T \leftarrow L'_T \cup (\pi_i, X')$                 ▷ Update with new NEI label set
    return $L' = (L_\Theta, L_\Pi, L_E, L'_T)$
end function
```

Algorithm E.52 Definition for REMOVE_L_π Low-Level Primitive

```
function REMOVE_L_π($L = (L_\Theta, L_\Pi, L_E, L_T), \pi_i$)
    $L'_\Pi = L_\Pi - \{(\pi_j, l_{\pi_i} \in L_E | \pi_j = \pi_i\}$
    return $L' = (L_\Theta, L'_\Pi, L_E, L_T)$
end function
```

Algorithm E.53 Definition for REMOVE_L_τ_π Low-Level Primitive

```
function REMOVE_L_τ_π($L = (L_\Theta, L_\Pi, L_E, L_T), \pi_i, \tau_j$)
    $L_{T_{\pi_i}} = (\Pi_i, X) \leftarrow \{(\pi_j, \{(\tau_1, L_{\tau_1}), (\tau_2, L_{\tau_2}), (\tau_3 L_{\tau_3})), \ldots\} \in L_T | \pi_j = \pi_i\}$    ▷ NEI label set for NE $\pi_i$
    $L'_T \leftarrow L_T - L_{T_{\pi_i}}$    ▷ Remove old NEI set
    return $L' = (L_\Theta, L_\Pi, L_E, L'_T)$
end function
```

Algorithm E.54 Definition for REMOVE_L_τ_τ Low-Level Primitive

```
function REMOVE_L_τ_τ($L = (L_\Theta, L_\Pi, L_E, L_T), \pi_i, \tau_j$)
    $L_{T_{\pi_i}} = (\Pi_i, X) \leftarrow \{(\pi_j, \{(\tau_1, L_{\tau_1}), (\tau_2, L_{\tau_2}), (\tau_3 L_{\tau_3})), \ldots\} \in L_T | \pi_j = \pi_i\}$    ▷ NEI label set for NE $\pi_i$
    $X' \leftarrow X - \{(\tau_k, L_{\tau_k}) \in X | \tau_k = \tau_i$    ▷ Remove $\tau_j$ from label set of NEI for NE $\pi_i$
    $L'_T \leftarrow L_T - L_{T_{\pi_i}}$    ▷ Remove old NEI set
    $L'_T \leftarrow L'_T \cap (\pi_i, X')$    ▷ Update with new NEI set
    return $L' = (L_\Theta, L_\Pi, L_E, L'_T)$
end function
```

Algorithm E.55 Definition for REMOVE_L_ϵ Low-Level Primitive

```
function REMOVE_L_ε($L = (L_\Theta, L_\Pi, L_E, L_T), \epsilon_i$)
    $L'_E = L_E - \{(\epsilon_j, l_{\epsilon_i} \in L_E | \epsilon_j = \epsilon_i\}$
    return $L' = (L_\Theta, L_\Pi, L'_E, L_T)$
end function
```

Algorithm E.56 Definitions for the SET_λ Low-Level Primitives

```
function SET_λ_Θ($L_\Theta$, $label$, $value$)
    $L'_\Theta \leftarrow$ SET_KEY($label$, $value$)
    return $L'_\Theta$
end function
function SET_λ_Π($L_\Pi$, $\pi_i$, $label$, $value$)
    $X \leftarrow$ GET_KEY($L_\Pi$, $\pi_i$)                        ▷ Labels for NE
    $X' \leftarrow$ SET_KEY($X$, $label$, $value$)               ▷ Set label to value for NE
    $L'_\Pi \leftarrow$ SET_KEY($L_\Pi$, $\pi_i$, $X'$)               ▷ Update labels for NE
    return $L'_\Pi$
end function
function SET_λ_E($L_E$, $\epsilon_i$, $label$, $value$)
    $X \leftarrow$ GET_KEY($L_E$, $\epsilon_i$)                       ▷ Labels for NEC
    $X' \leftarrow$ SET_KEY($X$, $label$, $value$)               ▷ Set label to value for NEC
    $L'_E \leftarrow$ SET_KEY($L_E$, $\epsilon_i$, $X'$)               ▷ Update labels for NEC
    return $L'_E$
end function
function SET_λ_T($L_T$, $\pi_i$, $\tau_j$, $label$, $value$)
    $X \leftarrow$ GET_KEY($L_T$, $\pi_i$)                        ▷ All NEI labels for NE
    $Y \leftarrow$ GET_KEY($X$, $\tau_j$)                         ▷ Labels for NEI in NE
    $Y' \leftarrow$ SET_KEY($Y$, $label$, $value$)               ▷ Set label to value for NEI
    $X' \leftarrow$ SET_KEY($X$, $\tau_j$, $Y'$)                    ▷ Update NEI labels for NE
    $L'_T \leftarrow$ SET_KEY($L_T$, $\pi_i$, $X'$)                ▷ Update NEI labels
    return $L'_T$
end function
```

E.3 HIGH-LEVEL PRIMITIVE DEFINITIONS

Algorithm E.57 Definition for GROUPING High-Level Primitive. Based on algorithm from NetworkX [76]

```
function GROUPING($\theta_i, X, label$)
    $labels \leftarrow \varnothing$
    for $x \in X$ do
        $\alpha \leftarrow \lambda_\Pi(\theta_i, x, label)$
        if $\alpha \notin labels$ then
            $labels \leftarrow labels \cup \{\alpha\}$
        end if
    end for
    $group \leftarrow \varnothing$
    for $\alpha \in labels$ do
        $Y \leftarrow \{x \in X \quad | \quad \lambda_\Pi(\theta_i, x, label) = \alpha\}$
        $group \leftarrow group \cup \{(\alpha, Y)\}$
    end for
end function
```

Algorithm E.58 Definition of BOUNDARY_NODES High-Level Primitive

```
function BOUNDARY_NODES($\theta_i, X$)
    $Y \leftarrow \varnothing$
    for $\pi_i \in X$ do
        $Y \leftarrow Y \cup$ NEIGHBOURS_$\pi(\pi_i)$
    end for
    $Y \leftarrow Y - X$                          ▷ Remove nodes inside boundary
    return $Y$
end function
```

Algorithm E.59 Definition for E_EQUIVALENT_Π High-level Primitive

```
function E_EQUIVALENT_Π($\theta_i, \epsilon_j, label$)
    $u \leftarrow \Pi_u(\theta_i, \epsilon_j)$
    $v \leftarrow \Pi_v(\theta_i, \epsilon_j)$
    if $\lambda_\Pi(\theta_i, u, label) = \lambda_\Pi(\theta_i, v, label)$ then
        return True
    else
        return False
    end if
end function
```

Algorithm E.60 Definition for E_EQUIVALENT_T High-level Primitive

```
function E_EQUIVALENT_T($\theta_i, \epsilon_j, label$)
    $u \leftarrow T_u(\theta_i, \epsilon_j)$
    $v \leftarrow T_v(\theta_i, \epsilon_j)$
    if $\lambda_T(\theta_i, u, label) = \lambda_T(\theta_i, v, label)$ then
        return True
    else
        return False
    end if
end function
```

Algorithm E.61 Definition of CONNECTED_NODES High-Level Primitive

```
function CONNECTED_NODES($\theta_i$)
    $G \leftarrow$ TO_GRAPH($\theta_i$) return GRAPH_CONNECTED_NODES($G$)
end function
```

Algorithm E.62 Definition of Role Access Functions High-Level Primitives

```
function $\Pi_{routers}(\theta_i)$
    $X \leftarrow \{\pi_i \in \Pi(\theta_i) \quad | \quad \rho(\pi_i) = \text{router}\}$
        return X
end function
function $\Pi_{switches}(\theta_i)$
    $X \leftarrow \{\pi_i \in \Pi(\theta_i) \quad | \quad \rho(\pi_i) = \text{switch}\}$
        return X
end function
function $\Pi_{servers}(\theta_i)$
    $X \leftarrow \{\pi_i \in \Pi(\theta_i) \quad | \quad \rho(\pi_i) = \text{server}\}$
        return X
end function
function $\Pi_{layer3}(\theta_i)$
    $X \leftarrow \Pi_{servers}(\theta_i) \cap \Pi_{servers}(\theta_i)$
        return X
end function
```

Algorithm E.63 Definition of TO_GRAPH High-Level Primitive

```
function TO_GRAPH($\theta_i$)
    $X \leftarrow \Pi(\theta_i)$                                    ▷ Nodes
    $Y \leftarrow \{(\Pi_u(\epsilon), \Pi_v(\epsilon) \forall \epsilon in E(\theta_i)\}$     ▷ Edges
    $G \leftarrow (X, Y)$
    return G
end function
```

Algorithm E.64 Definition of GRAPH_CONNECTED_NODES Function

```
function GRAPH_CONNECTED_NODES($G = (Nodes, Edges)$)
    $C \leftarrow \varnothing$
    $visited \leftarrow \varnothing$
    for $x \in Nodes$ do
        if $s \notin visited$ then
            $c \leftarrow$ BFS$(G, x)$                   ▷ Breadth-First Search from x
            $visited \leftarrow visited \cup c$          ▷ Updated visited nodes
            $C \leftarrow C \cap \{c\}$                 ▷ Store *set* of connected nodes
        end if
    end for
    return C                                         ▷ Return set of sets
end function
```

Algorithm E.65 Definition for SET_LABEL_π High-Level Primitive

```
function SET_LABEL_$\pi$($\theta_i = (\Pi_i, E_i, T_i, B_i, (L_\Theta, L_\Pi, L_E, L_T)), \pi_i, label, value$)
    $L'_\Pi \leftarrow$ SET_KEY$(label, value)$
    return $\theta_i = (\Pi_i, E_i, T_i, B_i, (L_\Theta, L_\Pi, L_E, L_T))$
end function
```

F

CODE FOR CHAPTER 6

F.1 OSPF EXAMPLE

F.1.1 *Setup*

```
import json
from collections import Counter, defaultdict

import autonetkit
import autonetkit.load.graphml as graphml
import netaddr
from autonetkit.ank import copy_attr_from, explode_nodes, groupby, split
from autonetkit.compilers.device import router_base
from jinja2 import Environment, Template
from netaddr import IPNetwork, IPSet

anm = autonetkit.NetworkModel()

with open("ospf.graphml") as fh:
    data = fh.read()

input_graph = graphml.load_graphml(data)

nodes = {}
for n, data in input_graph.nodes(data=True):
    nodes[n] = {
        "id": n,
        "asn": data["asn"],
        "x": int(data["x"] * 1.2),
        "y": int(data["y"] * 1.2),
    }


edges = []
for s, t, data in input_graph.edges(data=True):
    edges.append((s, t, data.get("src_area"), data.get(
        "dst_area"), data.get("src_cost"), data.get("dst_cost")))

g_in = anm.add_overlay("input")
for n, d in nodes.items():
    g_in.add_node(n, x=d["x"], device_type="router",
              y=d["y"], asn=1)

for edge in edges:
    src, dst, src_area, dst_area, src_cost, dst_cost = edge
    src = g_in.node(src)
    dst = g_in.node(dst)
    src_iface = src.add_interface()
    dst_iface = dst.add_interface()

    src_iface.set("area", int(src_area))
    src_iface.set("cost", int(src_cost))
    dst_iface.set("area", int(dst_area))
    dst_iface.set("cost", int(dst_cost))

    edge = g_in.add_edge(src_iface, dst_iface)

g_in.allocate_input_interfaces()
autonetkit.update_http(anm)

g_phy = anm['phy']
g_phy.add_nodes_from(g_in, retain=["asn", "device_type", "x", "y"])
g_phy.update(use_ipv4=True, host="localhost",
             platform="netkit", syntax="quagga")

g_phy.add_edges_from(g_in.edges())
autonetkit.update_http(anm)

g_l2 = anm.add_overlay("layer2")
g_l2.add_nodes_from(g_phy)
g_l2.add_edges_from(g_phy.edges())

# Split the point-to-point edges to add a collision domain
edges_to_split = [edge for edge in g_l2.edges()
                  if edge.src.is_l3device() and edge.dst.is_l3device()]

for edge in edges_to_split:
    edge.split = True  # mark as split for use in building nidb

split_created_nodes = split(g_l2, edges_to_split, id_prepend='bd_')

for node in split_created_nodes:
    # set midway x, y for plot
    neighs = node.neighbors()
    x = sum(neigh["phy"].get("x") for neigh in neighs) / len(neighs)
    y = sum(neigh["phy"].get("y") for neigh in neighs) / len(neighs)
    node.set("x", x)
    node.set("y", y)

    c = Counter(n.get("asn") for n in neighs)
    most_common_asn, _ = c.most_common(1)[0]
    node.set("asn", most_common_asn)

    node.set("broadcast_domain", True)
    node.set("device_type", "broadcast_domain")

autonetkit.update_http(anm)

g_l2_conn = anm.add_overlay("layer2_conn")
g_l2_conn.add_nodes_from(g_l2)
g_l2_conn.add_edges_from(g_l2.edges())

bc_nodes = g_l2.nodes(broadcast_domain=True)

explode_nodes(g_l2_conn, bc_nodes)

autonetkit.update_http(anm)

g_ospf = anm.add_overlay("ospf")
g_ospf.add_nodes_from(g_in.routers())
g_ospf.add_edges_from(e for e in g_in.edges()
                      if e.src.asn == e.dst.asn)
```

```
for node in g_ospf:
    node_areas = set()
    node.set("process_id", node.get("asn"))
    for interface in node.physical_interfaces():
        area = interface["input"].get("area")
        node_areas.add(area)
        interface.set("area", area)
        cost = interface["input"].get("cost")
        interface.set("cost", cost)

    lowest_area = sorted(node_areas)[0]
    node.loopback_zero.set("area", lowest_area)

autonetkit.update_http(anm)

g_ip = anm.add_overlay("ip")
g_ip.add_nodes_from(g_l2)
g_ip.add_edges_from(g_l2.edges())

bc_nodes = g_l2.nodes(broadcast_domain=True)

bc_attrs = ["broadcast_domain", "device_type", "asn"]
for attr in bc_attrs:
    copy_attr_from(g_l2, g_ip, attr, nbunch=bc_nodes)

# allocate loopback IPs
block = IPNetwork("10.0.0.0/16")
subnets = block.subnet(24)
loopback_allocations = {}
l3_nodes = g_ip.l3devices()
for asn, nodes in groupby("asn", l3_nodes):
    asn_block = subnets.next()
    loopback_allocations[asn] = asn_block
    hosts = asn_block.iter_hosts()

    for node in nodes:
        ip = hosts.next()
        node.set("loopback", ip)
        node.loopback_zero.set("ip", ip)
        node.loopback_zero.set("subnet", IPNetwork(ip))

# allocate infra IPs
block = IPNetwork("192.168.0.0/16")
subnets = block.subnet(24)
infra_allocations = {}
bc_nodes = g_ip.nodes(broadcast_domain=True)
for asn, nodes in groupby("asn", bc_nodes):
    asn_block = subnets.next()
    infra_allocations[asn] = asn_block
    ptp_subnets = asn_block.subnet(30)
    for node in nodes:
        prefix = ptp_subnets.next()
        node.set("subnet", prefix)

        hosts = prefix.iter_hosts()
        for neigh_iface in node.neighbor_interfaces():
            address = hosts.next()
            neigh_iface.set("ip", address)
            neigh_iface.set("subnet", prefix)
            ip_brief = str(address).replace("192.168.", "")
            neigh_iface.set("ip_brief", ip_brief)

autonetkit.update_http(anm)

for node in g_ospf:
    asn = node.get("asn")
    infra_block = infra_allocations.get(asn, [])
    lo_block = loopback_allocations.get(asn, [])
    adv_prefixes = IPSet(infra_block)
    adv_prefixes.update(lo_block)
    node.set("networks", adv_prefixes.iter_cidrs())
    node.set("router_id", node["ip"].loopback_zero.get("ip"))
```

Listing F.1: Source code for setup for OSPF configuration example

F.1.2 *Device Compilers and Platform Compiler*

```
class simple_router_compiler(object):

    def __init__(self, anm, nidb):
        self.anm = anm
        self.nidb = nidb

    def compile(self, node):
        interfaces = self.interfaces(node)
        ospf = self.ospf(node)
        return {
            "interfaces": interfaces,
            "ospf": ospf,
            "hostname": str(node),
        }

    def interfaces(self, node):
        pass

    def ospf(self, node):
        pass
```

Listing F.2: Simple Router Compiler for OSPF configuration example

```
class simple_quagga_compiler(simple_router_compiler):
    loopback_zero_id = "lo:1"  # on linux loopback zero is 127.0.0.1

    def interfaces(self, node):
        ifaces = []
        for interface in self.nidb["interfaces"].get(node):
            int_id = interface.get("id")
            ip = interface.get("ip")
            subnet = interface.get("subnet")
            cidr = "%s/%s" % (ip, subnet.prefixlen)
            ifaces.append({"cidr": cidr,
                           "id": int_id
                           })

        # add lo0
        lo0 = self.anm["ip"].node(node).loopback_zero
        ip = lo0.get("ip")
        subnet = lo0.get("subnet")
        cidr = "%s/%s" % (ip, subnet.prefixlen)
        lo_zero_id = lo0["phy"].get("id")
        ifaces.append({"cidr": cidr,
                       "id": lo_zero_id
                       })

        return ifaces

    def ospf(self, node):
        ospf_node = self.anm["ospf"].node(node)
        process_id = ospf_node.get("process_id")

        # now interfaces
        networks = []
        ospf_interfaces = []
        passive_interfaces = []

        lo0 = ospf_node.loopback_zero
        subnet = lo0["ip"].get("subnet")
        networks.append({
            "area": lo0.get("area"),
            "network": subnet.cidr
        })

        passive_interfaces.append({
            "id": self.loopback_zero_id,
        })

        for ospf_int in ospf_node.physical_interfaces():
            if not ospf_int.is_bound:
                continue

            ip_int = ospf_int["ip"]
            area = ospf_int.get("area")
            subnet = ip_int.get("subnet")
            networks.append({
                "network": subnet.cidr,
                "area": area
            })

            ospf_interfaces.append({
                "id": ospf_int["phy"].get("id"),
                "cost": ospf_int.get("cost")
            })

        return {
            "interfaces": ospf_interfaces,
            "passive_interfaces": passive_interfaces,
            "networks": networks
        }
```

Listing F.3: Simple Quagga Compiler for OSPF configuration example

```
class simple_ios_compiler(simple_router_compiler):
    loopback_zero_id = "Loopback0"

    def interfaces(self, node):
        ifaces = []
        phy_node = self.anm["phy"].node(node)
        for interface in phy_node.physical_interfaces():
            int_id = interface.get("id")
            ip_int = interface["ip"]
            ip = ip_int.get("ip")
            subnet = ip_int.get("subnet")

            ospf_int = interface["ospf"]
            ospf_cost = ospf_int.get("cost")

            ifaces.append({"ip": ip,
                           "id": int_id,
                           "ospf_cost": ospf_cost,
                           "shutdown": False,
                           "netmask": str(subnet.netmask)})

        interface = node.loopback_zero
        ip_int = interface["ip"]
        int_id = interface.get("id")
        ip = ip_int.get("ip")
        subnet = ip_int.get("subnet")
        ifaces.append({"ip": str(ip),
                       "id": self.loopback_zero_id,
                       "netmask": str(subnet.netmask)})
        return ifaces

    def ospf(self, node):
        # process interfaces
        networks = []
        passive_interfaces = []
        ospf_node = node["ospf"]
        process_id = ospf_node.get("process_id")
        router_id = ospf_node.get("router_id")
        for interface in ospf_node.physical_interfaces():
            area = interface.get("area")
            subnet = interface["ip"].get("subnet")
            networks.append({
                "area": area,
                "prefix": str(subnet.network),
                "hostmask": str(subnet.hostmask)
            })

        passive_interfaces.append({
            "id": self.loopback_zero_id,
        })

        lo0 = ospf_node.loopback_zero
        subnet = lo0["ip"].get("subnet")
        networks.append({
            "area": lo0.get("area"),
            "prefix": str(subnet.network),
            "hostmask": str(subnet.hostmask)
        })

        return {
            "networks": networks,
            "passive_interfaces": passive_interfaces}
```

Listing F.4: Simple IOS Compiler for OSPF configuration example

```
class simple_ios_xr_compiler(simple_router_compiler):
    loopback_zero_id = "Loopback0"

    def interfaces(self, node):
        ifaces = []
        phy_node = self.anm["phy"].node(node)
        for interface in phy_node.physical_interfaces():
            int_id = interface.get("id")
            ip_int = interface["ip"]
            ip = ip_int.get("ip")
            subnet = ip_int.get("subnet")
            ifaces.append({"ip": ip,
                           "id": int_id,
                           "shutdown": False,
                           "netmask": str(subnet.netmask)})

        interface = node.loopback_zero
        ip_int = interface["ip"]
        int_id = interface.get("id")
        ip = ip_int.get("ip")
        subnet = ip_int.get("subnet")
        ifaces.append({"ip": str(ip),
                        "id": self.loopback_zero_id,
                        "netmask": str(subnet.netmask)})

        return ifaces

    def ospf(self, node):
        # process interfaces
        networks = []
        passive_interfaces = []

        interfaces_by_area = defaultdict(list)

        ospf_node = node["ospf"]
        process_id = ospf_node.get("process_id")
        for interface in ospf_node.physical_interfaces():
            area = interface.get("area")
            cost = interface.get("cost")
            iface_id = interface["phy"].get("id")
            interfaces_by_area[area].append({
                "id": iface_id,
                "cost": cost
            })

        lo0 = ospf_node.loopback_zero
        loopback_zero_area = lo0.get("area")
        interfaces_by_area[loopback_zero_area].append({
            "id": self.loopback_zero_id,
            "passive": True
        })

        return {"interfaces_by_area": interfaces_by_area}
```

Listing F.5: Simple IOS-XR Compiler for OSPF configuration example

```
class simple_nxos_compiler(simple_router_compiler):
    loopback_zero_id = "Loopback0"

    def interfaces(self, node):
        ifaces = []
        phy_node = self.anm["phy"].node(node)

        ospf_process_id = node["ospf"].get("process_id")
        for interface in phy_node.physical_interfaces():
            int_id = interface.get("id")
            ip_int = interface["ip"]
            ip = ip_int.get("ip")
            subnet = ip_int.get("subnet")
            cidr = "%s/%s" % (ip, subnet.prefixlen)

            ospf_int = interface["ospf"]
            ospf_area = ospf_int.get("area")
            ospf_cost = ospf_int.get("cost")

            ifaces.append({"cidr": cidr,
                           "id": int_id,
                           "ospf_process_id": ospf_process_id,
                           "ospf_area": ospf_area,
                           "ospf_cost": ospf_cost,
                           "shutdown": False
                           })

        interface = node.loopback_zero
        ospf_area = interface["ospf"].get("area")
        ip_int = interface["ip"]
        int_id = interface.get("id")
        ip = ip_int.get("ip")
        subnet = ip_int.get("subnet")
        cidr = "%s/%s" % (ip, subnet.prefixlen)
        ifaces.append({"cidr": str(cidr),
                       "id": self.loopback_zero_id,
                       "ospf_process_id": ospf_process_id,
                       "ospf_area": ospf_area
                       })

        return ifaces

    def ospf(self, node):
        router_id = node["ospf"].get("router_id")
        return {"router_id": router_id}
```

Listing F.6: Simple NX-OS Compiler for OSPF configuration example

```
class simple_junos_compiler(simple_router_compiler):
    loopback_zero_id = "lo0"

    def interfaces(self, node):
        ifaces = []
        phy_node = self.anm["phy"].node(node)
        for interface in phy_node.physical_interfaces():
            int_id = interface.get("id")
            ip_int = interface["ip"]
            ip = ip_int.get("ip")
            subnet = ip_int.get("subnet")
            cidr = "%s/%s" % (ip, subnet.prefixlen)

            ifaces.append({"cidr": cidr,
                           "id": int_id
                           })

        interface = node.loopback_zero
        ip_int = interface["ip"]
        int_id = interface.get("id")
        ip = ip_int.get("ip")
        subnet = ip_int.get("subnet")
        cidr = "%s/%s" % (ip, subnet.prefixlen)
        ifaces.append({"cidr": str(cidr),
                       "id": self.loopback_zero_id
                       })

        return ifaces

    def ospf(self, node):
        # process interfaces
        networks = []
        passive_interfaces = []
        router_id = node["ospf"].get("router_id")

        interfaces_by_area = defaultdict(list)

        ospf_node = node["ospf"]
        process_id = ospf_node.get("process_id")
        for interface in ospf_node.physical_interfaces():
            area = interface.get("area")
            cost = interface.get("cost")
            iface_id = interface["phy"].get("id")
            interfaces_by_area[area].append({
                "id": iface_id,
                "metric": cost
            })
        lo0 = ospf_node.loopback_zero
        loopback_zero_area = lo0.get("area")
        interfaces_by_area[loopback_zero_area].append({
            "id": self.loopback_zero_id,
            "passive": True
        })

        return {"interfaces_by_area": interfaces_by_area}

nidb = {"hosts": {}, "labs": None, "interfaces": {}}
```

Listing F.7: Simple Junos Compiler for OSPF configuration example

```
class simple_platform_compiler(object):
    max_interfaces = 20

    def __init__(self, anm, nidb):
        self.anm = anm
        self.nidb = nidb

    def assign_interface_ids(self, node):
        target = node.get("target")
        if target == "quagga":
            prefix = "eth"
        elif target == "ios":
            prefix = "Ethernet0/"
        elif target == "ios_xr":
            prefix = "Ethernet0/"
        elif target == "nxos":
            prefix = "Ethernet0/"
        elif target == "junos":
            prefix = "Ethernet0/"

        for index, interface in enumerate(node.physical_interfaces()):
            id = "%s%s" % (prefix, index)
            interface.set("id", id)
            interface["phy"].set("id", id)

        node.loopback_zero["phy"].set("id", "lo:1")

    def setup_interfaces(self, node):
        ifaces = []

        for interface in node.physical_interfaces():
            ip_int = g_ip.interface(interface)
            ifaces.append({
                "id": interface.get("id"),
                "ip": ip_int.get("ip"),
                "subnet": ip_int.get("subnet")
            })

        return ifaces

    def compile(self):
        g_in = self.anm["input"]
        g_phy = self.anm["phy"]
        g_ip = self.anm["ip"]

        quagga_compiler = simple_quagga_compiler(self.anm, self.nidb)
        ios_compiler = simple_ios_compiler(self.anm, self.nidb)
        ios_xr_compiler = simple_ios_xr_compiler(self.anm, self.nidb)
        nxos_compiler = simple_nxos_compiler(self.anm, self.nidb)
        junos_compiler = simple_junos_compiler(self.anm, self.nidb)
        netkit_hosts = g_phy.nodes()
        for host in netkit_hosts:
            self.assign_interface_ids(host)
            self.nidb["interfaces"][host] = self.setup_interfaces(host)

        subnets = []
        for host in netkit_hosts:
            phy_node = g_phy.node(host)
            host_ifaces = host.physical_interfaces()
            host_ifaces = sorted(host_ifaces, key=lambda i: i.get("id"))

            for interface in host_ifaces:
                nk_id = interface.id.replace("eth", "")
                ip_int = g_ip.interface(interface)
                subnet = str(ip_int.get("subnet"))
                subnet = subnet.replace("/", ".")
                data = {"host": host, "interface_id": nk_id, "subnet":
                    subnet}
                subnets.append(data)

        taps = []

        routers = [n for n in netkit_hosts if n.is_router()]
        for node in routers:
            # and compile
            target = node["phy"].get("target")
            if target == "quagga":
                rtr_comp = quagga_compiler
            elif target == "ios":
                rtr_comp = ios_compiler
            elif target == "ios_xr":
                rtr_comp = ios_xr_compiler
            elif target == "nxos":
                rtr_comp = nxos_compiler
            elif target == "junos":
                rtr_comp = junos_compiler
            rtr_data = rtr_comp.compile(node)
            self.nidb["hosts"][node.label] = rtr_data

        lab = {"machines": netkit_hosts, "subnets": subnets, "taps": taps
            }

        self.nidb["labs"] = lab
```

Listing F.8: Simple Platform Compiler for OSPF configuration example

F.1.3 *Templates*

```
!
! zebra.conf
!
hostname {{node.hostname}}
{% for interface in node.interfaces %}
interface {{interface.id}}
  ip address {{interface.cidr}}
{% endfor %}
!
!
!
! ospfd.conf
!
hostname {{node.hostname}}
{% for interface in node.ospf.interfaces %}
interface {{interface.id}}
  ip ospf cost {{interface.cost}}
```

```
{% endfor %}
!
{% for interface in node.ospf.passive_interfaces %}
passive-interface {{interface.id}}
{% endfor %}
!
router ospf
  {% for network in node.ospf.networks %}
  network {{network.network}} area {{network.area}}
  {% endfor %}
!
!
```

Listing F.9: Template for Quagga for OSPF configuration example

```
!
hostname {{node.hostname}}
!
{% for interface in node.interfaces %}
interface {{interface.id}}
  ip address {{interface.ip}} {{interface.netmask}}
  {% if interface.ospf_cost%}
  ip ospf cost {{interface.ospf_cost}}
  {% endif %}
  {% if interface.shutdown == False%}
  no shutdown
  {% endif %}
{% endfor %}
!
router ospf {{node.ospf_process_id}}
{% for network in node.ospf.networks %}
  network {{network.prefix}} {{network.hostmask}} area {{network.area}}
{% endfor %}
{% for interface in node.ospf.passive_interfaces %}
  passive-interface {{interface.id}}
{% endfor %}
```

Listing F.10: Template for IOS for OSPF configuration example

```
!
hostname {{node.hostname}}
!
{% for interface in node.interfaces %}
interface {{interface.id}}
  ipv4 address {{interface.ip}} {{interface.netmask}}
  {% if interface.shutdown == False%}
  no shutdown
  {% endif %}
{% endfor %}
!
router ospf {{node.ospf_process_id}}
  {% for area, area_data in node.ospf.interfaces_by_area.iteritems() %}
  area {{area}}
    {% for entry in area_data %}
    interface {{entry.id}}
    {% if entry.cost%}
      cost {{entry.cost}}
    {% endif %}
    {% if entry.passive%}
      passive enable
    {% endif %}
    !
    {% endfor %}
  !
  {% endfor %}
```

Listing F.11: Template for IOS-XR for OSPF configuration example

```
!
hostname {{node.hostname}}
!
feature ospf
!
{% for interface in node.interfaces %}
interface {{interface.id}}
  ip address {{interface.cidr}}
  ip router ospf {{interface.ospf_process_id}} area {{interface.ospf_area
      }}
  ip ospf cost {{interface.ospf_cost}}
  {% if interface.shutdown == False%}
  no shutdown
  {% endif %}
{% endfor %}
!
router ospf {{node.ospf_process_id}}
  router-id {{node.ospf.router_id}}
```

Listing F.12: Template for NX-OS for OSPF configuration example

```
interfaces {
{% for interface in node.interfaces %}
 {{interface.id}} {
  unit 0 {
    family inet {
      address {{interface.cidr}};
    }
  }
}
{% endfor %}
protocols {
  ospf {
    {% for area, area_data in node.ospf.interfaces_by_area.iteritems() %}
    area {{area}} {
      {% for entry in area_data %}
      {{entry.id}} {
        {% if entry.metric %}
        metric {{entry.metric}};
        {% endif %}
        {% if entry.passive %}
        passive;
        {% endif %}
      }
      {% endfor %}
    }
  {% endfor %}
  }
}
```

Listing F.13: Template for Junos for OSPF configuration example

F.1.4 *Compiling and Rendering*

```
sim_plat = simple_platform_compiler(anm, nidb)
nidb_hosts = nidb.get("hosts")

templates = {}
```

```
# Setup Jinja2 environment
env = Environment(trim_blocks=True, lstrip_blocks=True)

templates["quagga"] = env.get_template("quagga.jinja2")
templates["ios"] = env.get_template("ios.jinja2")
templates["ios_xr"] = env.get_template("ios_xr.jinja2")
templates["nxos"] = env.get_template("nxos.jinja2")
templates["junos"] = env.get_template("junos.jinja2")


# Setup JSON encoder for custom data types
class AnkEncoder(json.JSONEncoder):

    def default(self, obj):
        if isinstance(obj, set):
            return str(obj)
        if isinstance(obj, netaddr.IPAddress):
            return str(obj)
        if isinstance(obj, netaddr.IPNetwork):
            return str(obj)

        return json.JSONEncoder.default(self, obj)


# Build for each target
targets = ["quagga", "ios", "ios_xr", "nxos", "junos"]
for target in targets:
    for node in g_phy:
        node.set("target", target)
    sim_plat.compile()
    # Print IDM for r3
    rtr_data = nidb_hosts["r3"]
    with open("%s_nidb.json" % target, "w") as fh:
        json.dump(rtr_data, fh, cls=AnkEncoder, indent=1, sort_keys=True)

    # Render r3 output using target template
    template = templates[target]
    with open("%s_output.txt" % target, "w") as fh:
        fh.write(template.render(node=rtr_data))
```

Listing F.14: Source code for compilation and rendering for OSPF configuration example

F.2 EXAMPLE TEMPLATES

These example templates show the process of generating the Device Configurations from a JSON version of the Intermediate Device Model. They are based on a simplified version of the Quagga configuration format.

F.2.1 *OSPF*

```
[
 {
  "area": 0,
  "cost": 1,
  "desc": "to r5",
  "network": "J/24"
 }, {
  "area": 0,
  "cost": 1,
  "desc": "to r6",
  "network": "K/24"
 }
]
```

Listing F.15: OSPF JSON

```
router ospf
{% for e in data %}
  network {{e.network}} area {{e.area}}
{% endfor %}
```

Listing F.16: OSPF Template

```
router ospf
network J/24 area 0
network K/24 area 0
```

Listing F.17: OSPF Output

F.2.2 *iBGP*

```
[
 {
  "asn": 20,
  "desc": "to r2",
  "neighbor": "B.1"
 }, {
  "asn": 20,
  "desc": "to r3",
  "neighbor": "B.2"
 }
]
```

Listing F.18: iBGP JSON

```
{% for e in data %}
  neighbor {{e.neighbor}} remote-as {{e.asn}}
{% endfor %}
```

Listing F.19: iBGP Template

```
neighbor B.1 remote-as 20
neighbor B.2 remote-as 20
```

Listing F.20: iBGP Output

F.2.3 *eBGP*

```
{
 "neighbors": [
  {
   "asn": 200,
   "desc": "to r5",
   "neighbor": "C.1"
  }, {
   "asn": 30,
   "desc": "to r6",
   "neighbor": "D.1"
  }
 ],
 "networks": [
  "B/24",
  "J/24",
  "K/24",
  "L/24"
 ]
}
```

Listing F.21: eBGP JSON

```
{% for e in data.networks %}
  network {{e}}
{% endfor %}
{% for e in data.neighbors %}
  neighbor {{e.neighbor}} remote-as {{e.asn}}
{% endfor %}
```

Listing F.22: eBGP Template

```
network B/24
network J/24
network K/24
network L/24
neighbor C.1 remote-as 200
neighbor D.1 remote-as 30
```

Listing F.23: eBGP Output

G

CODE FOR CHAPTER 7

```
hostname {{ hostname }}
password {{ zebra.pwd }}
enable password {{ zebra.pwd }}
{% for sr in zebra.static_routes %}
! {{ sr.description }}
ip route {{ sr.loopback }} {{ sr.
    next_hop }}
{% endfor %}
!
{% for i in interfaces %}
interface {{ i.id }}
  description {{i.desc}}
  {% if i.use_ipv4 %}
  ip address {{i.ipv4_cidr}}
  {% if i.ospf_cost %}
  ip ospf cost {{ i.ospf_cost }}
  {% endif %}
  {% if i.role=="loopback" %}
  ip address 127.0.0.1/8
  {% endif %}
  !
 {% endif %}
{% endfor %}
!
{% if ospf %}
router ospf
{% for o_link in ospf.ospf_links %}
  network {{ o_link.network }} area
       {{ o_link.area }}
{% endfor %}
  !
{% for pass_int in ospf.passive_
    interfaces %}
  passive-interface {{ pass_int.id
      }}
{% endfor %}
  !
  network {{ ospf.loopback_subnet
      }} area 0
{% endif %}
!
{% if bgp %}
router bgp {{ asn }}
  bgp router-id {{ loopback }}
  no synchronization
  {% for subnet in bgp.ipv4_adv_
      subnets %}
  network {{ subnet.cidr }}
  {% endfor %}
  ! ibgp
  {% for n in bgp.ibgp_neighs %}
    {% if loop.first %}
  ! ibgp peers
    {% endif %}
  ! {{ n.neighbor }}
  neighbor {{ n.loopback }} remote-
      as {{ n.asn }}
  neighbor {{ n.loopback }} update-
      source {{ loopback }}
  neighbor {{ n.loopback }} send-
      community
  neighbor {{ n.loopback }} next-
      hop-self
  {% endfor %}
  ! ebgp
  {% for n in bgp.ebgp_neighs%}
  ! {{ n.neighbor }}
  neighbor {{ n.dst_ip }} remote-as
       {{ n.asn }}
  neighbor {{ n.dst_ip }} update-
      source {{ n.src_int_ip }}
  neighbor {{ n.dst_ip }} send-
      community
  {% endfor %}
{% endif %}
!
{% if bgp.debug %}
debug bgp
debug bgp events
debug bgp updates
log file /var/log/zebra/bgpd.log
{% endif %}
!
log file /var/log/zebra/zebra.log
```

Listing G.1: Quagga Template Example

G.1 CODE FOR HOUSE EXAMPLE

G.1.1 *Design Functions*

```
from autonetkit.ank import explode_nodes

g_l2_conn = anm.add_overlay("layer2_conn")
g_l2_conn.add_nodes_from(g_l2)
g_l2_conn.add_edges_from(g_l2.edges())

bc_nodes = g_l2.nodes(broadcast_domain=True)

explode_nodes(g_l2_conn, bc_nodes)

autonetkit.update_http(anm)
```

Listing G.2: Layer 2 Connectivity Design Function

```
from autonetkit.ank import split, groupby, copy_attr_from
from netaddr import IPNetwork

g_ip = anm.add_overlay("ip")
g_ip.add_nodes_from(g_l2)
g_ip.add_edges_from(g_l2.edges())

bc_nodes = g_l2.nodes(broadcast_domain=True)

bc_attrs = ["broadcast_domain", "device_type", "asn"]
for attr in bc_attrs:
    copy_attr_from(g_l2, g_ip, attr, nbunch=bc_nodes)

# allocate loopback IPs
block = IPNetwork("10.0.0.0/16")
subnets = block.subnet(24)

# Dictionary to store per-AS loopback blocks
loopback_allocations = {}

l3_nodes = g_ip.l3devices()
for asn, nodes in groupby("asn", l3_nodes):
    # Get next /24 block for AS
    asn_block = subnets.next()
    # Record allocation
    loopback_allocations[asn] = asn_block
    hosts = asn_block.iter_hosts()

    for node in sorted(nodes):
        # Allocate the next IP to the node loopback zero
        ip = hosts.next()
        node.set("loopback", ip)
        node.loopback_zero.set("ip", ip)
        node.loopback_zero.set("subnet", IPNetwork(ip))

# allocate infra IPs
block = IPNetwork("192.168.0.0/16")
subnets = block.subnet(24)

# Dictionary to store per-AS infrastructure blocks
infra_allocations = {}

bc_nodes = g_ip.nodes(broadcast_domain=True)
for asn, nodes in groupby("asn", bc_nodes):
    # Get next /24 block for AS
    asn_block = subnets.next()
    # Record allocation
    infra_allocations[asn] = asn_block
    ptp_subnets = asn_block.subnet(30)
    for node in sorted(nodes):
        # Set the /30 onto the pseudo-node
        prefix = ptp_subnets.next()
        node.set("subnet", prefix)

        hosts = prefix.iter_hosts()
        for neigh_iface in sorted(node.neighbor_interfaces()):
            # Allocate the next IP to the node interface
            address = hosts.next()
            neigh_iface.set("ip", address)
            neigh_iface.set("subnet", prefix)

autonetkit.update_http(anm)
```

Listing G.3: IP Addressing Design Function

```
g_ospf = anm.add_overlay("ospf")
g_ospf.add_nodes_from(g_in.routers())
g_ospf.add_edges_from(e for e in g_in.edges()
                      if e.src.asn == e.dst.asn)

for node in g_ospf:
    for interface in node.physical_interfaces():
        interface.set("area", 0)

    node.loopback_zero.set("area", 0)

autonetkit.update_http(anm)
```

Listing G.4: OSPF Design Function

```
from netaddr import IPSet

for node in g_ospf:
    asn = node.get("asn")
    # Get address blocks for this AS
    infra_block = infra_allocations.get(asn, [])
    lo_block = loopback_allocations.get(asn, [])
    # Create set of all networks for this node
    adv_prefixes = IPSet(infra_block)
    adv_prefixes.update(lo_block)
    # Store on the node
    node.set("networks", adv_prefixes.iter_cidrs())
```

Listing G.5: OSPF Advertise Design Function

```
g_ibgp = anm.add_overlay("ibgp")
g_ibgp.add_nodes_from(g_in.routers())
# Create cartesian product of nodes in same ASN
edges = [(s, t) for s in g_ibgp for t in g_ibgp
         if s != t and s.asn == t.asn]

for e in edges:
    # get source and destination nodes
    src, dst = e
    # Create BGP session termination points
    src_endpoint = src.add_interface(category="bgp_session")
    dst_endpoint = dst.add_interface(category="bgp_session")
    # Bind session termination point to loopback interface
    src_endpoint.set("bound_to", src.loopback_zero)
    dst_endpoint.set("bound_to", dst.loopback_zero)
    # add session to g_ibgp
    g_ibgp.add_edge(src_endpoint, dst_endpoint)

autonetkit.update_http(anm)
```

Listing G.6: iBGP Design Function

```
g_ebgp = anm.add_overlay("ebgp")
g_ebgp.add_nodes_from(g_in.routers())
edges = [e for e in g_in.edges()
         if e.src.asn != e.dst.asn]

for e in edges:
    # Obtain source and destination nodes
    src = g_ebgp.node(e.src)
    dst = g_ebgp.node(e.dst)
    # Create BGP session termination points
    src_endpoint = src.add_interface(category="bgp_session")
    dst_endpoint = dst.add_interface(category="bgp_session")
    # Bind session termination point to physical interface
    src_endpoint.set("bound_to", e.src_int)
    dst_endpoint.set("bound_to", e.dst_int)
    # add session to g_ebgp
    g_ebgp.add_edge(src_endpoint, dst_endpoint)

autonetkit.update_http(anm)
```

Listing G.7: eBGP Topology component of eBGP Design Function

```
# networks to advertise
from netaddr import IPSet

for node in g_ebgp:
    adv_prefixes = IPSet()
    if node.degree() == 0:
        # Not an eBGP speaker
        continue

    asn = node.get("asn")
    # Get address blocks for this AS
    infra_block = infra_allocations.get(asn, [])
    lo_block = loopback_allocations.get(asn, [])

    # Create set of all networks for this node
    adv_prefixes.update(infra_block)
    adv_prefixes.update(lo_block)
    # Store on the node
    node.set("networks", adv_prefixes.iter_cidrs())

autonetkit.update_http(anm)
```

Listing G.8: eBGP Network Advertisement component of eBGP Design Function

G.1.2 *Compiling*

```
from autonetkit.compilers.platform import platform_base
from netaddr import IPNetwork

nidb = {"hosts": {}, "labs": None, "interfaces": {}}


class simple_platform_compiler(object):
    max_interfaces = 20

    def __init__(self, anm, nidb):
        self.anm = anm
        self.nidb = nidb

    def assign_interface_ids(self, node):
        for index, interface in enumerate(node.physical_interfaces()):
            id = "eth%s" % index
            interface.set("id", id)
            interface["phy"].set("id", id)

        node.loopback_zero["phy"].set("id", "lo:1")

    def add_taps(self, netkit_hosts):
        tap_pool = IPNetwork("172.16.0.0/16")
        tap_hosts = tap_pool.iter_hosts()

        netkit_host_tap_ip = tap_hosts.next()  # for the tap vm
        tap_vm_ip = tap_hosts.next()  # for the tap vm

        tap_map = []

        for node in netkit_hosts:
            ifaces = {iface.get("id") for iface in node.
                physical_interfaces()}

            next_free_if = None
            for x in range(0, self.max_interfaces):
                if "eth%s" % x not in ifaces:
                    next_free_if_id = x
                    next_free_if = "eth%s" % x
                    break
            else:
                print "Warning: Reached interface limit for %s" % node
                continue

            iface = node.add_interface(id=next_free_if, desc=next_free_if
                )
            tap_ip = tap_hosts.next()
            iface.set("ip", tap_ip)
            iface.set("tap", True)

            tap_map.append(
                {"node": node,
                 "interface_id": next_free_if_id,
                 "tap_ip": tap_ip
                 })

            # and add to the interfaces for this host
            self.nidb["interfaces"][node].append({
                "id": next_free_if,
                "ip": tap_ip,
                "subnet": tap_pool
            })
        return {
            "linux_host": netkit_host_tap_ip,
            "tap_vm_ip": tap_vm_ip,
            "tap_hosts": tap_map
        }

    def setup_interfaces(self, node):
        ifaces = []

        for interface in node.physical_interfaces():
            ip_int = g_ip.interface(interface)
            ifaces.append({
                "id": interface.get("id"),
                "ip": ip_int.get("ip"),
                "subnet": ip_int.get("subnet")
            })

        return ifaces

    def compile(self):
        g_in = self.anm["input"]
        g_phy = self.anm["phy"]
        g_ip = self.anm["ip"]

        rtr_comp = simple_router_compiler(self.nidb, self.anm)
        netkit_hosts = g_in.nodes()

        for host in netkit_hosts:
            self.assign_interface_ids(host)
            self.nidb["interfaces"][host] = self.setup_interfaces(host)

        subnets = []
        for host in netkit_hosts:
            phy_node = g_phy.node(host)
            host_ifaces = host.physical_interfaces()
            host_ifaces = sorted(host_ifaces, key=lambda i: i.get("id"))

            for interface in host_ifaces:
                nk_id = interface.id.replace("eth", "")
                ip_int = g_ip.interface(interface)
                subnet = str(ip_int.get("subnet"))
                subnet = subnet.replace("/", ".")
                data = {"host": host, "interface_id": nk_id, "subnet":
                    subnet}
                subnets.append(data)

        taps = self.add_taps(netkit_hosts)

        routers = [n for n in netkit_hosts if n.is_router()]
        for node in routers:
            rtr_data = rtr_comp.compile(node)
            self.nidb["hosts"][node.label] = rtr_data
        lab = {"machines": netkit_hosts, "subnets": subnets, "taps": taps
            }
        self.nidb["labs"] = lab

sim_plat = simple_platform_compiler(anm, nidb)
sim_plat.compile()
```

Listing G.9: Create Platform Compiler

```
lab = nidb.get("labs")
```

```
nidb_hosts = nidb.get("hosts")
```

Listing G.10: NIDB Shortcuts

```
# Create Platform Compiler and Compile
sim_plat = simple_platform_compiler(anm, nidb)
sim_plat.compile()
```

Listing G.11: Compile Platform

```
from autonetkit.compilers.device import router_base


class simple_router_compiler(router_base.RouterCompiler):
    loopback_zero_id = "lo:1"  # on linux loopback zero is 127.0.0.1

    def compile(self, node):
        zebra = self.zebra(node)
        ssh = self.ssh(node)
        interfaces = self.interfaces(node)
        ospf = self.ospf(node)
        bgp = self.bgp(node)

        return {
            "zebra": zebra,
            "ssh": ssh,
            "interfaces": interfaces,
            "ospf": ospf,
            "bgp": bgp,
            "hostname": str(node),
            "asn": node.get("asn")
        }

    def zebra(self, node):
        password = "zebra"
        return {
            "password": password
        }

    def ssh(self, node):
        return {
            "use_key": False
        }

    def interfaces(self, node):
        ifaces = []
        for interface in self.nidb["interfaces"].get(node):
            int_id = interface.get("id")
            ip = interface.get("ip")
            subnet = interface.get("subnet")
            ifaces.append({"ip": ip,
                           "id": int_id,
                           "broadcast": subnet.broadcast,
                           "netmask": subnet.netmask
                           })

        lo0 = self.anm["ip"].node(node).loopback_zero
        ip = lo0.get("ip")
        subnet = lo0.get("subnet")
        lo_zero_id = lo0["phy"].get("id")
        ifaces.append({"ip": ip,
                       "id": lo_zero_id,
                       "broadcast": subnet.broadcast,
                       "netmask": subnet.netmask
```

```
                })
        return ifaces

    def ospf(self, node):
        ospf_node = self.anm["ospf"].node(node)

        process_id = ospf_node.get("process_id")

        # now interfaces
        networks = []

        lo0 = ospf_node.loopback_zero
        subnet = lo0["ip"].get("subnet")
        networks.append({
            "area": lo0.get("area"),
            "network": subnet.cidr
        })

        for ospf_int in ospf_node.physical_interfaces():
            if not ospf_int.is_bound:
                continue

            ip_int = ospf_int["ip"]
            area = ospf_int.get("area")
            subnet = ip_int.get("subnet")
            networks.append({
                "network": subnet.cidr,
                "area": area
            })

        return {
            "networks": networks
        }

    def bgp(self, node):
        ibgp_neighbors = []
        g_ibgp = self.anm["ibgp"]

        for session in g_ibgp.edges(node):
            dst = session.dst
            src_int = session.src_int
            src_bound_int = src_int.get("bound_to")
            update_source_ip = src_bound_int["ip"].get("ip")
            dst_int = session.dst_int
            dst_bound_int = dst_int.get("bound_to")
            neigh_ip = dst_bound_int["ip"].get("ip")
            desc = "Router %s" % dst
            data = {
                "update_source": update_source_ip,
                "neigh_ip": neigh_ip,
                "asn": dst.get("asn"),  # should be same as node's!
                "desc": desc
            }
            ibgp_neighbors.append(data)

        ebgp_neighbors = []
        g_ebgp = self.anm["ebgp"]
        ebgp_node = g_ebgp.node(node)
        for session in g_ebgp.edges(node):
            dst = session.dst
            src_int = session.src_int
            src_bound_int = src_int.get("bound_to")
            update_source_ip = src_bound_int["ip"].get("ip")
            dst_int = session.dst_int
            dst_bound_int = dst_int.get("bound_to")
            neigh_ip = dst_bound_int["ip"].get("ip")
            desc = "Router %s" % dst
            data = {
                "update_source": update_source_ip,
                "neigh_ip": neigh_ip,
                "asn": dst.get("asn"),
                "desc": desc
            }
            ebgp_neighbors.append(data)

        networks = ebgp_node.get("networks") or []

        lo0 = self.anm["ip"].node(node).loopback_zero
        router_id = lo0.get("ip")

        return {
            "ibgp_neighbors": ibgp_neighbors,
            "ebgp_neighbors": ebgp_neighbors,
            "networks": networks,
            "router_id": router_id
        }
```

Listing G.12: Specify Device Compiler

G.1.3 *Templates*

```
from jinja2 import Template, Environment
env = Environment(trim_blocks=True, lstrip_blocks=True)
```

Listing G.13: Setup Environment

```
!
hostname {{node.hostname}}
password zebra
enable password zebra
!
router bgp {{node.asn}}
{% for network in node.bgp.networks %}
network {{network}}
{% endfor %}
{% for neigh in node.bgp.ibgp_neighbors %}
! {{neigh.neigh_ip}}
neighbor {{neigh.neigh_ip}} remote-as {{neigh.asn}}
neighbor {{neigh.neigh_ip}} update-source {{neigh.update_source}}
neighbor {{neigh.neigh_ip}} description {{neigh.desc}}
neighbor {{neigh.neigh_ip}} next-hop-self
{% endfor %}
!
{% for neigh in node.bgp.ebgp_neighbors %}
! {{neigh.neighbor}}
neighbor {{neigh.neigh_ip}} remote-as {{neigh.asn}}
neighbor {{neigh.neigh_ip}} update-source {{neigh.update_source}}
neighbor {{neigh.neigh_ip}} description {{neigh.desc}}
{% endfor %}
!
!
log file /var/log/zebra/bgpd.log
!
debug bgp
debug bgp events
debug bgp filters
debug bgp fsm
debug bgp keepalives
debug bgp updates
!
```

Listing G.14: BGP Template

```
# daemons file
zebra={{ 'yes' if node.zebra else 'no' }}
bgpd={{ 'yes' if node.bgp else 'no' }}
ripd={{ 'yes' if node.rip else 'no' }}
ospfd={{ 'yes' if node.ospf else 'no' }}
ospf6d={{ 'yes' if node.ospf6 else 'no' }}
ripngd={{ 'yes' if node.ripng else 'no' }}
isisd={{ 'yes' if node.isis else 'no' }}
```

Listing G.15: Daemons Template

```
!
hostname {{node.hostname}}
password zebra
enable password zebra
!
!
router ospf
    {% for network in node.ospf.networks %}
    network {{network.network}} area {{network.area}}
    {% endfor %}
!
log file /var/log/zebra/ospfd.log
!
```

Listing G.16: OSPF Template

```
{% if node.zebra %}
hostname {{node.hostname}}
password {{node.zebra.password}}
enable password {{node.zebra.password}}
banner motd file /etc/quagga/motd.txt
{% for static_route in node.zebra.static_routes %}
! {{static_route.description}}
ip route {{static_route.loopback}} {{static_route.next_hop}}
{% endfor %}
!

log file /var/log/zebra/zebra.log
{% endif %}
```

Listing G.17: Zebra Template

```
hostname {{node.hostname}}
```

Listing G.18: Hostname Template

```
{% for i in node.interfaces %}
/sbin/ifconfig {{i.id}} {{i.ip}} netmask {{i.netmask}} broadcast {{i.
    broadcast}} up
{% endfor %}
route del default
/sbin/ifconfig lo 127.0.0.1 up
/etc/init.d/ssh start
/etc/init.d/hostname.sh
{% if node.zebra %}
```

```
/etc/init.d/zebra start
{% endif %}
{% if node.ssh.use_key %}
chown -R root:root /root
chmod 755 /root
chmod 755 /root/.ssh
chmod 644 /root/.ssh/authorized_keys
{% endif %}
/etc/init.d/inetd restart
echo pts/0 >> /etc/securetty
echo pts/1 >> /etc/securetty
echo pts/2 >> /etc/securetty
echo pts/3 >> /etc/securetty
echo pts/4 >> /etc/securetty
echo pts/5 >> /etc/securetty
echo pts/6 >> /etc/securetty
```

Listing G.19: Startup Template

```
root:$1$LB0MUyNC$4y8CpiP0PjymXi.p.M8fg/:14558:0:99999:7:::
daemon:*:14219:0:99999:7:::
bin:*:14219:0:99999:7:::
sys:*:14219:0:99999:7:::
sync:*:14219:0:99999:7:::
games:*:14219:0:99999:7:::
man:*:14219:0:99999:7:::
lp:*:14219:0:99999:7:::
mail:*:14219:0:99999:7:::
news:*:14219:0:99999:7:::
uucp:*:14219:0:99999:7:::
proxy:*:14219:0:99999:7:::
www-data:*:14219:0:99999:7:::
backup:*:14219:0:99999:7:::
list:*:14219:0:99999:7:::
irc:*:14219:0:99999:7:::
gnats:*:14219:0:99999:7:::
nobody:*:14219:0:99999:7:::
libuuid:!:14219:0:99999:7:::
bind:*:14219:0:99999:7:::
messagebus:*:14219:0:99999:7:::
dnsmasq:*:14219:0:99999:7:::
Debian-exim:!:14219:0:99999:7:::
freerad:*:14219:0:99999:7:::
statd:*:14219:0:99999:7:::
sshd:*:14219:0:99999:7:::
pdns:!:14219:0:99999:7:::
proftpd:!:14219:0:99999:7:::
ftp:*:14219:0:99999:7:::
quagga:*:14219:0:99999:7:::
snmp:*:14219:0:99999:7:::
snort:*:14219:0:99999:7:::
telnetd:*:14219:0:99999:7:::
uml-net:*:14219:0:99999:7:::
xorp:*:14219:0:99999:7:::
guest:nlbMnI.VQBS3s:14219:0:99999:7:::
```

Listing G.20: Shadow Template

```
UseDNS no
LogLevel DEBUG
```

Listing G.21: SSH Config Template

```
machines = "{{ lab.machines|join(', ') }}"

{% for entry in lab.subnets %}
{{entry.host}}[{{entry.interface_id}}]={{entry.subnet}}
{% endfor %}

{% for entry in lab.taps.tap_hosts %}
{{entry.node}}[{{entry.interface_id}}]=tap,{{lab.taps.linux_host}},{{
    entry.tap_ip}}
{% endfor %}
```

Listing G.22: Lab Template

G.1.4 *Rendering*

```
templates = {
    "quagga":
    {
        "bgpd": bgp_template,
        "ospfd": ospf_template,
        "zebra": zebra_template,
    },
        "etc":
    {
        "hostname": etc_host_template,
        "shadow": etc_shadow_template,
    },
    "ssh_config": etc_ssh_config_template,
    "startup": startup_template,
    "zebra_daemons": daemons_template,

}
```

Listing G.23: Setup Templates

```
import zipfile
import os

archive_path = "lab.zip"
archive = zipfile.ZipFile(archive_path, mode='w')

lab_data = lab_template.render(lab=lab)
archive.writestr("lab.conf", lab_data)

for name, data in sorted(nidb_hosts.items()):
    dest = "%s.startup" % name
    template = templates["startup"]
    archive.writestr(dest, template.render(node=data))

    zebra_path = "%s/etc/zebra/" % name
    for fname, template in templates["quagga"].items():
        rendered = template.render(node=data)
        dest = os.path.join(zebra_path, "%s.conf" % fname)
        archive.writestr(dest, rendered)

    dest = os.path.join(zebra_path, "daemons")
    template = templates["zebra_daemons"]
    archive.writestr(dest, template.render(node=data))

    etc_dir = os.path.join(name, "etc")

    dest = os.path.join(etc_dir, "hostname")
    template = templates["etc"]["hostname"]
    archive.writestr(dest, template.render(node=data))

    dest = os.path.join(etc_dir, "shadow")
    template = templates["etc"]["shadow"]
    archive.writestr(dest, template.render())

    dest = os.path.join(etc_dir, "ssh", "ssh_config")
    template = templates["ssh_config"]
    archive.writestr(dest, template.render())

archive.close()
```

Listing G.24: Render Lab

G.1.5 *Results*

```
 r3# sh ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

B>* 10.0.0.0/24 [200/0] via 10.0.0.4 (recursive via 192.168.0.18, eth1),
    00:04:56
O>* 10.0.0.1/32 [110/20] via 192.168.0.5, eth0, 00:06:12
O>* 10.0.0.2/32 [110/30] via 192.168.0.5, eth0, 00:06:05
  *                      via 192.168.0.18, eth1, 00:06:05
O   10.0.0.3/32 [110/10] is directly connected, lo, 00:06:57
C>* 10.0.0.3/32 is directly connected, lo
O>* 10.0.0.4/32 [110/20] via 192.168.0.18, eth1, 00:06:05
B>* 10.0.1.0/24 [200/0] via 10.0.0.4 (recursive via 192.168.0.18, eth1),
    00:04:56
C>* 127.0.0.0/8 is directly connected, lo
C>* 172.16.0.0/16 is directly connected, eth2
B>* 192.168.0.0/24 [200/0] via 10.0.0.4 (recursive via 192.168.0.18, eth1
    ), 00:04:56
O>* 192.168.0.0/30 [110/20] via 192.168.0.5, eth0, 00:06:12
O   192.168.0.4/30 [110/10] is directly connected, eth0, 00:06:57
C>* 192.168.0.4/30 is directly connected, eth0
O>* 192.168.0.8/30 [110/20] via 192.168.0.18, eth1, 00:06:05
O   192.168.0.16/30 [110/10] is directly connected, eth1, 00:06:05
C>* 192.168.0.16/30 is directly connected, eth1
```

Listing G.25: SHOW IP ROUTE from *r3*

```
r3# show ip ospf neighbor

Neighbor ID Pri State   Dead Time  Address       Interface          RXmtL
     RqstL DBsmL
10.0.0.1 1 Full/Backup  33.886s    192.168.0.5   eth0:192.168.0.6   0
        0     0
10.0.0.4 1 Full/DR      35.501s    192.168.0.18  eth1:192.168.0.17  0
        0     0
```

Listing G.26: SHOW IP OSPF NEIGHBOR from *r3*.

```
r5# show ip bgp summary
BGP router identifier 10.0.1.1, local AS number 2
RIB entries 5, using 320 bytes of memory
Peers 2, using 5032 bytes of memory

Neighbor      V   AS MsgRcvd MsgSent   TblVer  InQ OutQ Up/Down  State/
    PfxRcd
192.168.0.13  4   1      30      33        0    0    0 00:00:43        2
```

```
192.168.0.21  4   1      29      32        0    0    0 00:26:29        2

Total number of neighbors 2
```

Listing G.27: SHOW IP BGP SUMMARY from *r5*

```
hostname r5:~# traceroute 10.0.0.1
traceroute to 10.0.0.1 (10.0.0.1), 64 hops max, 40 byte packets
 1  192.168.0.13    (192.168.0.13)      0 ms  0 ms  0 ms
 2  10.0.0.1        (10.0.0.1)          0 ms  0 ms  0 ms
```

Listing G.28: Output of TRACEROUTE from *r5* to *r1*

H

CODE FOR CHAPTER 8 CASE STUDIES

H.1 INTRODUCTION

In this appendix we provide the code for the case studies shown in Chapter 8.

H.2 CODE FOR SIMPLIFIED SMALL INTERNET CASE STUDY

H.2.1 *Design Functions*

```
import autonetkit.load.graphml as graphml
with open("small_internet.graphml") as fh:
    data = fh.read()

input_graph = graphml.load_graphml(data)
nodes = {}
for n, data in input_graph.nodes(data=True):
    nodes[n] = {
        "id": n,
        "asn": data["asn"],
        "x": int(data["x"] * 1.2),
        "y": int(data["y"] * 1.2),
    }

edges = input_graph.edges()
```

Listing H.1: Load GraphML

```
import autonetkit
anm = autonetkit.NetworkModel()
g_in = anm.add_overlay("input")

nodes = {
    "as20r2":  {"y": -22,  "x": 30, "id": "as20r2",   "asn": 20},
    "as20r3":  {"y": -94,  "x": 102, "id": "as20r3",  "asn": 20},
    "as200r1": {"y": 102,  "x": 237, "id": "as200r1", "asn": 200},
    "as20r1":  {"y": -22,  "x": 174, "id": "as20r1",  "asn": 20},
    "as30r1":  {"y": 102,  "x": 389, "id": "as30r1",  "asn": 30},
    "as40r1":  {"y": 102,  "x": 541, "id": "as40r1",  "asn": 40},
    "as1r1":   {"y": -167, "x": 415, "id": "as1r1",   "asn": 1},
    "as100r1": {"y": 232,  "x": 30,  "id": "as100r1", "asn": 100},
    "as100r2": {"y": 232,  "x": -98, "id": "as100r2", "asn": 100},
    "as100r3": {"y": 304,  "x": -41, "id": "as100r3", "asn": 100},
    "as300r4": {"y": 322,  "x": 619, "id": "as300r4", "asn": 300},
    "as300r2": {"y": 232,  "x": 619, "id": "as300r2", "asn": 300},
    "as300r3": {"y": 322,  "x": 506, "id": "as300r3", "asn": 300},
    "as300r1": {"y": 232,  "x": 506, "id": "as300r1", "asn": 300}
}

for n, d in nodes.items():
    g_in.add_node(n, x=d["x"], device_type="router",
                  y=d["y"], asn=d["asn"])

for node in g_in:
    print node.x, node.y
```

Listing H.2: Create Network Whiteboard

```
edges = [("as20r2",  "as20r1"),  ("as20r2",  "as100r1"),
         ("as20r3",  "as20r2"),  ("as20r1",  "as30r1"),
         ("as20r1",  "as20r3"),  ("as300r4", "as300r2"),
         ("as300r3", "as300r4"), ("as300r1", "as300r3"),
         ("as40r1",  "as300r2"), ("as1r1",   "as20r3"),
         ("as1r1",   "as40r1"),  ("as100r1", "as20r1"),
         ("as100r1", "as100r3"), ("as100r2", "as100r1"),
         ("as100r3", "as100r2"), ("as200r1", "as20r1"),
         ("as30r1",  "as1r1")]

g_in.add_edges_from(edges)
g_in.allocate_input_interfaces()
```

Listing H.3: Add Edges to Network Whiteboard

```
autonetkit.update_http(anm)
```

Listing H.4: Update Visualisation

```
g_phy = anm['phy']
g_phy.add_nodes_from(g_in, retain=["asn", "device_type", "x", "y"])
g_phy.update(use_ipv4=True, host="localhost",
             platform="netkit", syntax="quagga")

g_phy.add_edges_from(g_in.edges())
autonetkit.update_http(anm)
```

Listing H.5: Create Physical Network View

```
from autonetkit.ank import split
from collections import Counter

g_l2 = anm.add_overlay("layer2")
g_l2.add_nodes_from(g_phy)
g_l2.add_edges_from(g_phy.edges())

# Split the point-to-point edges to add a collision domain
edges_to_split = [edge for edge in g_l2.edges()
                  if edge.src.is_l3device() and edge.dst.is_l3device()]

for edge in edges_to_split:
    edge.split = True  # mark as split for use in building nidb

split_created_nodes = split(g_l2, edges_to_split, id_prepend='bd_')

for node in split_created_nodes:
    # set midway x, y for plot
    neighs = node.neighbors()
    x = sum(neigh["phy"].get("x") for neigh in neighs) / len(neighs)
    y = sum(neigh["phy"].get("y") for neigh in neighs) / len(neighs)
    node.set("x", x)
    node.set("y", y)

    c = Counter(n.get("asn") for n in neighs)
    most_common_asn, _ = c.most_common(1)[0]
    node.set("asn", most_common_asn)

    node.set("broadcast_domain", True)
    node.set("device_type", "broadcast_domain")


autonetkit.update_http(anm)
```

Listing H.6: Create Layer 2 Network View

```
from autonetkit.ank import explode_nodes

g_l2_conn = anm.add_overlay("layer2_conn")
g_l2_conn.add_nodes_from(g_l2)
g_l2_conn.add_edges_from(g_l2.edges())

bc_nodes = g_l2.nodes(broadcast_domain=True)

explode_nodes(g_l2_conn, bc_nodes)

autonetkit.update_http(anm)
```

Listing H.7: Create Layer 2 Connectivity Network View

```
from autonetkit.ank import split, groupby, copy_attr_from
from netaddr import IPNetwork

g_ip = anm.add_overlay("ip")
g_ip.add_nodes_from(g_l2)
g_ip.add_edges_from(g_l2.edges())

bc_nodes = g_l2.nodes(broadcast_domain=True)

bc_attrs = ["broadcast_domain", "device_type", "asn"]
for attr in bc_attrs:
    copy_attr_from(g_l2, g_ip, attr, nbunch=bc_nodes)


# allocate loopback IPs
block = IPNetwork("10.0.0.0/16")
subnets = block.subnet(24)

loopback_allocations = {}

l3_nodes = g_ip.l3devices()
for asn, nodes in groupby("asn", l3_nodes):
    asn_block = subnets.next()
    loopback_allocations[asn] = asn_block
    hosts = asn_block.iter_hosts()

    for node in nodes:
        ip = hosts.next()
        node.set("loopback", ip)
        node.loopback_zero.set("ip", ip)
        node.loopback_zero.set("subnet", IPNetwork(ip))

# allocate infra IPs
```

```
block = IPNetwork("192.168.0.0/16")
subnets = block.subnet(24)

infra_allocations = {}

bc_nodes = g_ip.nodes(broadcast_domain=True)
for asn, nodes in groupby("asn", bc_nodes):
    asn_block = subnets.next()
    infra_allocations[asn] = asn_block
    ptp_subnets = asn_block.subnet(30)
    for node in nodes:
        prefix = ptp_subnets.next()
        node.set("subnet", prefix)

        hosts = prefix.iter_hosts()
        for neigh_iface in node.neighbor_interfaces():
            address = hosts.next()
            neigh_iface.set("ip", address)
            neigh_iface.set("subnet", prefix)


autonetkit.update_http(anm)
```

Listing H.8: Create IP Network View

```
g_ospf = anm.add_overlay("ospf")
g_ospf.add_nodes_from(g_in.routers())
g_ospf.add_edges_from(e for e in g_in.edges()
                      if e.src.asn == e.dst.asn)

for node in g_ospf:
    for interface in node.physical_interfaces():
        interface.set("area", 0)

    node.loopback_zero.set("area", 0)

autonetkit.update_http(anm)
```

Listing H.9: Create OSPF Network View

```
from netaddr import IPSet

for node in g_ospf:
    asn = node.get("asn")
    # Get address blocks for this AS
    infra_block = infra_allocations.get(asn, [])
    lo_block = loopback_allocations.get(asn, [])
    # Create set of all networks for this node
    adv_prefixes = IPSet(infra_block)
    adv_prefixes.update(lo_block)
    # Store on the node
    node.set("networks", adv_prefixes.iter_cidrs())
```

Listing H.10: OSPF Networks

```
g_ibgp = anm.add_overlay("ibgp")
g_ibgp.add_nodes_from(g_in.routers())
edges = [(s, t) for s in g_ibgp for t in g_ibgp
         if s != t and s.asn == t.asn]

for e in edges:
    src, dst = e
    src_endpoint = src.add_interface(category="bgp_session")
    dst_endpoint = dst.add_interface(category="bgp_session")
    src_endpoint.set("bound_to", src.loopback_zero)
    dst_endpoint.set("bound_to", dst.loopback_zero)

    g_ibgp.add_edge(src_endpoint, dst_endpoint)

autonetkit.update_http(anm)
```

Listing H.11: Create iBGP Network View

```
g_ebgp = anm.add_overlay("ebgp")
g_ebgp.add_nodes_from(g_in.routers())
edges = [e for e in g_in.edges()
         if e.src.asn != e.dst.asn]

for e in edges:
    # Obtain source and destination nodes
    src = g_ebgp.node(e.src)
    dst = g_ebgp.node(e.dst)
    # Create BGP session termination points
    src_endpoint = src.add_interface(category="bgp_session")
    dst_endpoint = dst.add_interface(category="bgp_session")
    # Bind session termination point to physical interface
    src_endpoint.set("bound_to", e.src_int)
    dst_endpoint.set("bound_to", e.dst_int)
    # add session to g_ebgp
    g_ebgp.add_edge(src_endpoint, dst_endpoint)

autonetkit.update_http(anm)
```

Listing H.12: Create eBGP Network View

```
# networks to advertise
from netaddr import IPSet

for node in g_ebgp:
    adv_prefixes = IPSet()
    if node.degree() == 0:
        # Not an eBGP speaker
        continue

    asn = node.get("asn")
    # Get address blocks for this AS
    infra_block = infra_allocations.get(asn, [])
    lo_block = loopback_allocations.get(asn, [])

    # Create set of all networks for this node
    adv_prefixes.update(infra_block)
    adv_prefixes.update(lo_block)
    # Store on the node
    node.set("networks", adv_prefixes.iter_cidrs())

autonetkit.update_http(anm)
```

Listing H.13: eBGP Networks

H.2.2 *Compilation and Rendering*

Netkit Platform Compiler

```
# Create Platform Compiler and Compile
sim_plat = simple_platform_compiler(anm, nidb)
sim_plat.compile()
```

Listing H.14: Netkit Platform Compiler

Quagga Device Compiler

```
from autonetkit.compilers.device import router_base


class simple_router_compiler(router_base.RouterCompiler):
    loopback_zero_id = "lo:1"  # on linux loopback zero is 127.0.0.1

    def compile(self, node):
        zebra = self.zebra(node)
        ssh = self.ssh(node)
        interfaces = self.interfaces(node)
        ospf = self.ospf(node)
        bgp = self.bgp(node)

        return {
            "zebra": zebra,
            "ssh": ssh,
            "interfaces": interfaces,
            "ospf": ospf,
            "bgp": bgp,
            "hostname": str(node),
            "asn": node.get("asn")
        }

    def zebra(self, node):
        password = "zebra"
        return {
            "password": password
        }

    def ssh(self, node):
        return {
            "use_key": False
        }

    def interfaces(self, node):
        # Append attributes to the interface, rather than add a stanza
        ifaces = []
        for interface in self.nidb["interfaces"].get(node):
            int_id = interface.get("id")
            ip = interface.get("ip")
            subnet = interface.get("subnet")
            ifaces.append({"ip": ip,
                           "id": int_id,
                           "broadcast": subnet.broadcast,
                           "netmask": subnet.netmask
                           })

        # add lo0
        lo0 = self.anm["ip"].node(node).loopback_zero
        ip = lo0.get("ip")
        subnet = lo0.get("subnet")
        lo_zero_id = lo0["phy"].get("id")
        ifaces.append({"ip": ip,
                       "id": lo_zero_id,
                       "broadcast": subnet.broadcast,
                       "netmask": subnet.netmask
                       })

        return ifaces

    def ospf(self, node):
        ospf_node = self.anm["ospf"].node(node)

        process_id = ospf_node.get("process_id")

        networks = []

        lo0 = ospf_node.loopback_zero
        subnet = lo0["ip"].get("subnet")
        networks.append({
            "area": lo0.get("area"),
            "network": subnet.cidr
        })

        for ospf_int in ospf_node.physical_interfaces():
            if not ospf_int.is_bound:
                continue

            ip_int = ospf_int["ip"]
            area = ospf_int.get("area")
            subnet = ip_int.get("subnet")
            networks.append({
                "network": subnet.cidr,
                "area": area
            })

        return {
            "networks": networks
        }

    def bgp(self, node):
        ibgp_neighbors = []
        g_ibgp = self.anm["ibgp"]

        for session in g_ibgp.edges(node):
            dst = session.dst
            src_int = session.src_int
            src_bound_int = src_int.get("bound_to")
            update_source_ip = src_bound_int["ip"].get("ip")
            dst_int = session.dst_int
            dst_bound_int = dst_int.get("bound_to")
            neigh_ip = dst_bound_int["ip"].get("ip")
            desc = "Router %s" % dst
            data = {
                "update_source": update_source_ip,
                "neigh_ip": neigh_ip,
                "asn": dst.get("asn"),  # should be same as node's!
                "desc": desc
            }
            ibgp_neighbors.append(data)

        ebgp_neighbors = []
        g_ebgp = self.anm["ebgp"]
        ebgp_node = g_ebgp.node(node)
        for session in g_ebgp.edges(node):
            dst = session.dst
            src_int = session.src_int
            src_bound_int = src_int.get("bound_to")
            update_source_ip = src_bound_int["ip"].get("ip")
            dst_int = session.dst_int
            dst_bound_int = dst_int.get("bound_to")
            neigh_ip = dst_bound_int["ip"].get("ip")
            desc = "Router %s" % dst
            data = {
                "update_source": update_source_ip,
                "neigh_ip": neigh_ip,
                "asn": dst.get("asn"),
                "desc": desc
            }
            ebgp_neighbors.append(data)

        networks = ebgp_node.get("networks") or []
        lo0 = self.anm["ip"].node(node).loopback_zero
        router_id = lo0.get("ip")

        return {
            "ibgp_neighbors": ibgp_neighbors,
            "ebgp_neighbors": ebgp_neighbors,
            "networks": networks,
            "router_id": router_id
        }
```

Listing H.15: Quagga Device Compiler

H.2.3 *Launching*

```
ubuntu@ip-172-31-14-185:~/lab$ lstart -o --con0=none -p5

======================== Starting lab ===========================
Lab directory: /home/ubuntu/lab
Version:       <unknown>
Author:        <unknown>
Email:         <unknown>
Web:           <unknown>
Description:
<unknown>
=================================================================
You chose to use parallel startup.
Starting "as100r3"...
Starting "as1r1"...
Starting "as100r1"...
Starting "as100r2"...
Starting "as200r1"...
Starting "as20r1"...
Starting "as20r2"...
Starting "as20r3"...
Starting "as300r1"...
Starting "as300r2"...
Starting "as300r3"...
Starting "as300r4"...
Starting "as30r1"...
Starting "as40r1"...

The lab has been started.
=================================================================
```

Listing H.16: Launching Lab

H.2.4 *Simulation Results*

```
as20r1# sh ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

C>* 127.0.0.0/8 is directly connected, lo
C>* 172.16.0.0/16 is directly connected, eth5
B>* 172.16.0.0/24 [200/0] via 172.16.1.2 (recursive via 192.168.1.1, eth4
    ), 00:02:04
O>* 172.16.1.1/32 [110/20] via 192.168.1.5, eth3, 00:02:17
O>* 172.16.1.2/32 [110/20] via 192.168.1.1, eth4, 00:02:17
O   172.16.1.3/32 [110/10] is directly connected, lo, 00:03:03
C>* 172.16.1.3/32 is directly connected, lo
B>* 172.16.2.0/24 [20/0] via 192.168.1.17, eth2, 00:02:34
B>* 172.16.3.0/24 [200/0] via 172.16.1.2 (recursive via 192.168.1.1, eth4
    ), 00:02:04
B>* 172.16.4.0/24 [20/0] via 192.168.1.13, eth1, 00:02:57
B>* 172.16.5.0/24 [20/0] via 192.168.4.1, eth0, 00:02:57
B>* 172.16.6.0/24 [200/0] via 172.16.1.2 (recursive via 192.168.1.1, eth4
    ), 00:01:55
B>* 192.168.0.0/24 [200/0] via 172.16.1.2 (recursive via 192.168.1.1,
    eth4), 00:02:04
O   192.168.1.0/30 [110/10] is directly connected, eth4, 00:03:03
C>* 192.168.1.0/30 is directly connected, eth4
O   192.168.1.4/30 [110/10] is directly connected, eth3, 00:03:03
C>* 192.168.1.4/30 is directly connected, eth3
O>* 192.168.1.8/30 [110/20] via 192.168.1.1, eth4, 00:02:17
  *                         via 192.168.1.5, eth3, 00:02:17
C>* 192.168.1.12/30 is directly connected, eth1
C>* 192.168.1.16/30 is directly connected, eth2
B>* 192.168.2.0/24 [200/0] via 172.16.1.2 (recursive via 192.168.1.1,
    eth4), 00:02:04
B>* 192.168.3.0/24 [20/0] via 192.168.1.13, eth1, 00:02:57
B>* 192.168.4.0/24 [20/0] via 192.168.4.1, eth0, 00:02:57
C>* 192.168.4.0/30 is directly connected, eth0
B>* 192.168.5.0/24 [200/0] via 172.16.1.2 (recursive via 192.168.1.1,
    eth4), 00:01:55
```

Listing H.17: *show ip route* output for *as20r1*

```
as1r1# sh ip bgp summary
BGP router identifier 172.16.0.1, local AS number 1
RIB entries 25, using 1600 bytes of memory
Peers 3, using 7548 bytes of memory

Neighbor      V  AS MsgRcvd MsgSent  TblVer  InQ OutQ Up/Down  State/
    PfxRcd
192.168.0.1   4  20       6      11       0    0    0 00:01:44        7
192.168.0.5   4  30       8      10       0    0    0 00:01:16        7
192.168.2.5   4  40       4       9       0    0    0 00:01:17        4
```

Listing H.18: *show ip bgp summary* output for *as1r1*

```
as20r1# sh ip bgp summary
BGP router identifier 10.0.1.3, local AS number 20
RIB entries 25, using 1600 bytes of memory
Peers 5, using 12 KiB of memory

Neighbor      V   AS MsgRcvd MsgSent  TblVer  InQ OutQ Up/Down  State/
    PfxRcd
10.0.1.1      4   20      15      19       0    0    0 00:12:20        4
```

```
10.0.1.2      4   20      20      23       0    0    0 00:12:24        8
192.168.1.13  4  100      17      26       0    0    0 00:14:23        2
192.168.1.17  4   30      19      23       0    0    0 00:14:07        7
192.168.4.1   4  200      17      26       0    0    0 00:14:20        2

Total number of neighbors 5
```

Listing H.19: *show ip bgp summary* output for *as20r1*label

```
as300r1:~# traceroute lo:1.as100r2
traceroute to lo:1.as100r2 (lo:1.as100r2), 64 hops max, 40 byte packets
 1  eth0.as300r3 (eth0.as300r3)  0 ms  0 ms  0 ms
 2  eth1.as300r4 (eth1.as300r4)  1 ms  0 ms  0 ms
 3  eth0.as300r2 (eth0.as300r2)  0 ms  1 ms  0 ms
 4  eth1.as40r1  (eth1.as40r1)   1 ms  1 ms  1 ms
 5  eth2.as1r1   (eth2.as1r1)    1 ms  1 ms  1 ms
 6  eth1.as20r3  (eth1.as20r3)   1 ms  1 ms  1 ms
 7  eth2.as20r2  (eth2.as20r2)   1 ms  1 ms  1 ms
 8  eth2.as100r1 (eth2.as100r1)  1 ms  1 ms  1 ms
 9  lo:1.as100r2 (lo:1.as100r2)  1 ms  1 ms  1 ms
```

Listing H.20: Post-processed output of TRACEROUTE from *as300r1* to *as100r2*

H.3 CODE FOR COMPLETE SMALL INTERNET CASE STUDY

H.3.1 *Network Whiteboard*

```
import autonetkit
anm = autonetkit.NetworkModel()
```

Listing H.21: Initialisation

```
import autonetkit.load.graphml as graphml
from collections import namedtuple

with open("small_internet_complete.graphml") as fh:
    data = fh.read()

input_graph = graphml.load_graphml(data)
nodes = {}
for n, data in input_graph.nodes(data=True):
    nodes[n] = {
        "id": n, "asn": data["asn"], "device_type": data["device_type"],
        "x": int(data["x"] * 1.5), "y": int(data["y"] * 1.5),
    }

edge_tuple = namedtuple('Edge',
    ["src", "dst", "subnet", "src_id", "dst_id", "src_ip", "dst_ip"])

edges = []
for s, t, data in input_graph.edges(data=True):
    edges.append((s, t, data.get("src_id"), data.get("dst_id"),
        data.get("subnet"), data.get("src_ip"), data.get("dst_ip"),
        data.get("bgp_policy")))
```

Listing H.22: Load Nodes and Edges

```
import autonetkit
anm = autonetkit.NetworkModel()
g_in = anm.add_overlay("input")

for n, d in nodes.items():
    g_in.add_node(n, x=d["x"], device_type=d["device_type"],
                  y=d["y"], asn=d["asn"])

# mark routers to simulate
for n in g_in:
    if n.is_router():
        n.set("simulate", True)

# map roles
for node in g_in:
    asn = node.get("asn")
    node.set("bgp_role", bgp_roles[asn])

# map redist policy
for node in g_in:
    asn = node.get("asn")
    pol = bgp_redist_policy.get(asn)
    if pol == "ibgp":
        node.set("ibgp", True)
    elif pol == "redistribute_igp":
        node.set("redistribute_bgp_to_igp", True)
```

Listing H.23: Load Network Whiteboard nodes

```
for edge in edges:
    src, dst, src_id, dst_id, subnet, src_ip, dst_ip, bgp_policy = edge
    src = g_in.node(src)
    dst = g_in.node(dst)
    src_iface = src.add_interface()
    dst_iface = dst.add_interface()

    prefix = prefix_split = []

    if subnet != None:
        prefix, prefix_len = subnet.split("/")
        prefix_split = prefix.split(".")

    if src_id:
        src_iface.set("id", src_id)
        src_iface.set("description", src_id)
    if src_ip != None:
        src_prefix = prefix_split[0:3]
        src_prefix.append(src_ip)
        src_prefix = ".".join(src_prefix)
        src_iface.set("ip", src_prefix)
        src_iface.set("subnet", subnet)
    if dst_id:
        dst_iface.set("id", dst_id)
        dst_iface.set("description", dst_id)
    if dst_ip != None:
        dst_prefix = prefix_split[0:3]
        dst_prefix.append(dst_ip)
        dst_prefix = ".".join(dst_prefix)
        dst_iface.set("ip", dst_prefix)
        dst_iface.set("subnet", subnet)

    edge = g_in.add_edge(src_iface, dst_iface)

    if bgp_policy != None:
        edge.set("bgp_policy", bgp_policy)

g_in.allocate_input_interfaces()
autonetkit.update_http(anm)
```

Listing H.24: Load Network Whiteboard edges

H.3.2 *Design Functions*

```
g_phy = anm['phy']
g_phy.add_nodes_from(g_in, retain=["asn", "device_type", "x", "y"])
g_phy.update(platform="netkit", syntax="quagga")

g_phy.add_edges_from(g_in.edges())
```

Listing H.25: Create Physical Network View

```
def neigh_ave_xy(nodes):
    x = sum(neigh["phy"].get("x") for neigh in neighs) / len(neighs)
    y = sum(neigh["phy"].get("y") for neigh in neighs) / len(neighs)
    return (x, y)
```

Listing H.26: Sum Neighbor Averages

```
from autonetkit.ank import split
from collections import Counter
```

```
g_l2 = anm.add_overlay("layer2")
g_l2.add_nodes_from(g_phy)
g_l2.add_edges_from(g_phy.edges())

# Split the point-to-point edges to add a collision domain
edges_to_split = [edge for edge in g_l2.edges()
                  if edge.src.is_l3device() and edge.dst.is_l3device()]

for edge in edges_to_split:
    edge.split = True  # mark as split for use in building nidb

split_created_nodes = split(g_l2, edges_to_split, id_prepend='bd_')

for node in split_created_nodes:
    # set midway x, y for plot
    neighs = node.neighbors()
    x, y = neigh_ave_xy(neighs)
    node.set("x", x)
    node.set("y", y)

    neigh_asns = {n.get("asn") for n in neighs}
    if len(neigh_asns) == 1:
        node.set("asn", neigh_asns.pop())
    else:
        node.set("asn", None)  # inter-AS link

    c = Counter(n.get("asn") for n in neighs)
    most_common_asn, _ = c.most_common(1)[0]
    node.set("broadcast_domain", True)
    node.set("device_type", "broadcast_domain")

for switch in g_l2.switches():
    pn_id = "bc_" + switch.get("id")
    bc = g_l2.add_node(pn_id)
    g_phy.add_node(bc)

    bc.set("broadcast_domain", True)
    bc.set("device_type", "broadcast_domain")
    bc.set("asn", switch.get("asn"))

    for edge in switch.edges():
        dst = edge.dst
        dst_int = edge.dst_int
        src_int = bc.add_interface()
        g_l2.add_edge(src_int, dst_int)

    neighs = bc.neighbors()
    x, y = neigh_ave_xy(neighs)
    bc.set("x", x)
    bc.set("y", y)
    g_l2.remove_node(switch)

autonetkit.update_http(anm)
```

Listing H.27: Create Layer 2 Network View

```
from autonetkit.ank import explode_nodes

g_l2_conn = anm.add_overlay("layer2_conn")
g_l2_conn.add_nodes_from(g_l2)
g_l2_conn.add_edges_from(g_l2.edges())
bc_nodes = g_l2.nodes(broadcast_domain=True)
explode_nodes(g_l2_conn, bc_nodes)
```

```
autonetkit.update_http(anm)
```

Listing H.28: Create Layer 2 Connectivity Network View

```
from autonetkit.ank import split, groupby, copy_attr_from,
     copy_int_attr_from
from netaddr import IPNetwork, IPAddress
from collections import defaultdict

g_ip = anm.add_overlay("ip")
g_ip.add_nodes_from(g_l2)
g_ip.add_edges_from(g_l2.edges())

bc_nodes = g_l2.nodes(broadcast_domain=True)

copy_int_attr_from(g_in, g_ip, "ip")
copy_int_attr_from(g_in, g_ip, "subnet")

bc_attrs = ["broadcast_domain", "device_type", "asn"]
for attr in bc_attrs:
    copy_attr_from(g_l2, g_ip, attr, nbunch=bc_nodes)

infra_allocations = defaultdict(list)
host_allocations = defaultdict(list)

bc_nodes = g_ip.nodes(broadcast_domain=True)
for asn, nodes in groupby("asn", bc_nodes):
    nodes = list(nodes)  # generator to list
    for bc in nodes:
        neigh_ips = set()
        neigh_subnets = set()
        for neigh_int in bc.neighbor_interfaces():
            neigh_ip = neigh_int.get("ip")

            if neigh_ip and neigh_ip != None:
                neigh_ip = IPAddress(neigh_ip)
                neigh_int.set("ip", neigh_ip)
                neigh_ips.add(neigh_ip)
            neigh_sn = neigh_int.get("subnet")
            if neigh_sn and neigh_sn != None:
                neigh_sn = IPNetwork(neigh_sn)
                neigh_int.set("subnet", neigh_sn)
                neigh_subnets.add(neigh_sn)

        if len(neigh_subnets) == 1:
            sn = neigh_subnets.pop()
            bc.set("subnet", sn)
        else:
            print "Error: Subnet mismatch for %s" % bc

        if asn is None:
            # inter-AS links
            pass
        else:
            infra_allocations[asn].append(bc.get("subnet"))
            if any(n.is_server() for n in bc.neighbors()):
                host_allocations[asn].append(bc.get("subnet"))

autonetkit.update_http(anm)
```

Listing H.29: Create IP Network View

```
g_rip = anm.add_overlay("rip")
g_rip.add_nodes_from(g_in.routers())
g_rip.add_edges_from(e for e in g_l2_conn.edges()
                    if e.src.asn == e.dst.asn)
```
Listing H.30: Create RIP Network View

```
# redistribute connected prefixes
for node in g_rip:
    if node.degree() == 0:
        continue
    l2_neighs = node["layer2_conn"].neighbors()
    if any(n.is_server() for n in l2_neighs):
        node.set("redistribute_connected", True)

autonetkit.update_http(anm)
```
Listing H.31: Set RIP Redistribution

```
# rip networks
from netaddr import IPSet

for node in g_rip:
    asn = node.get("asn")
    ip_host = host_allocations.get(asn, [])
    adv_prefixes = IPSet(ip_host)
#     adv_prefixes.update(ip_host)
    node.set("networks", adv_prefixes.iter_cidrs())
```
Listing H.32: Set RIP Advertisement Networks

```
g_ibgp = anm.add_overlay("ibgp")
g_ibgp.add_nodes_from(g_in.routers())

ibgp_routers = set(g_in.routers(ibgp=True))

edges = [e for e in g_l2_conn.edges()
         if e.src in ibgp_routers and e.dst in ibgp_routers
         and e.src.asn == e.dst.asn]

for e in edges:
    src = g_ibgp.node(e.src)
    dst = g_ibgp.node(e.dst)
    src_endpoint = src.add_interface(category="bgp_session")
    dst_endpoint = dst.add_interface(category="bgp_session")
    src_endpoint.set("bound_to", e.src_int)
    dst_endpoint.set("bound_to", e.dst_int)

    g_ibgp.add_edge(src_endpoint, dst_endpoint)

autonetkit.update_http(anm)
```
Listing H.33: Create iBGP Network View

```
g_ebgp = anm.add_overlay("ebgp")
g_ebgp.add_nodes_from(g_in.routers())
edges = [e for e in g_in.edges()
         if e.src.asn != e.dst.asn]

for e in edges:
    # Obtain source and destination nodes
    src = g_ebgp.node(e.src)
    dst = g_ebgp.node(e.dst)
    # Create BGP session termination points
    src_endpoint = src.add_interface(category="bgp_session")
    dst_endpoint = dst.add_interface(category="bgp_session")
    # Bind session termination point to physical interface
    src_endpoint.set("bound_to", e.src_int)
    dst_endpoint.set("bound_to", e.dst_int)
    # add session to g_ebgp
    g_ebgp.add_edge(src_endpoint, dst_endpoint)

autonetkit.update_http(anm)
```

Listing H.34: Create eBGP Network View

```
# now apply ebgp policies
from autonetkit.ank import copy_attr_from, copy_edge_attr_from
copy_attr_from(g_in, g_ebgp, "bgp_role")
copy_edge_attr_from(g_in, g_ebgp, "bgp_policy")

def bgp_rank(role):
    ranks = {"b": 1, "p": 2, "c": 3}
    return ranks.get(role, -1)
```

Listing H.35: Apply eBGP Policies

```
# networks to advertise
from netaddr import IPSet

default_route = IPNetwork("0.0.0.0/0")

for node in sorted(g_ebgp):
    adv_prefixes = set()
    if node.degree() == 0:
        continue

    asn = node.get("asn")
    ip_infra = infra_allocations.get(asn, [])
    ip_host = host_allocations.get(asn, [])

    # and infra links
    node_role = node.get("bgp_role")
    if node_role == "b":
        # backbone, advertise default route also
        adv_prefixes.add(default_route)

    for edge in node.edges():
        src_int = edge.src_int
        dst = edge.dst
        dst_role = dst.get("bgp_role")
        advertise = False
        node_rank = bgp_rank(node_role)
        dst_rank = bgp_rank(dst_role)

        if ((node_rank < dst_rank)
                or (node_rank == dst_rank and node.get("asn") < dst.get("
                    asn"))):
            # get physical interface session is bound to
            bound_int = src_int.get("bound_to")
            subnet = bound_int["ip"].get("subnet")

            # upstream of destination, advertise prefix
            adv_prefixes.add(subnet)

    adv_prefixes.update(ip_infra)
    adv_prefixes.update(ip_host)
    if node_role == "c":
        # customer, advertise as a /16
        adv_prefixes = {x.supernet(16)[0] if x.prefixlen > 16 else x
                        for x in adv_prefixes}

    node.set("networks", adv_prefixes)

# and apply load share policy
for asn, policy in bgp_load_share_policy.items():
    if policy != True:
        continue

    asn_ebgp_nodes = [n for n in g_ebgp.nodes(asn=asn) if n.degree() > 0]
    # only load share common prefixes
    common_networks = set()
    networks = [n.get("networks") for n in asn_ebgp_nodes]
    asn_prefixes = set.intersection(*networks)
    # now split and allocate
    if len(asn_ebgp_nodes) != 2:
        # Future work to extend to support splitting across more than 2
            nodes
        break

    for prefix in asn_prefixes:
        # assume /16
        split_prefixes = prefix.subnet(17)
        for node in sorted(asn_ebgp_nodes):
            node_adv_prefixes = node.get("networks")
            node_adv_prefixes.add(split_prefixes.next())
            node.set("networks", node_adv_prefixes)


# for customer, also store the advertised prefix to form the peer asXIn
# prefix-list
for node in sorted(g_ebgp):
    node_role = node.get("bgp_role")
    if node.degree() > 0 and node_role == "c":
        node_adv_prefixes = node.get("networks")
        for peer in node.neighbors():
            cust_asn_prefixes = peer.get("customer_asn_prefixes")
            if not cust_asn_prefixes:
                cust_asn_prefixes = {}

            cust_asn_prefixes[node.get("asn")] = node_adv_prefixes

            # and store result
            peer.set("customer_asn_prefixes", cust_asn_prefixes)

autonetkit.update_http(anm)
```

Listing H.36: Add eBGP Prefix Advertisements

```
    backbones = set(g_ebgp.nodes(bgp_role="b"))
    providers = set(g_ebgp.nodes(bgp_role="p"))
    customers = set(g_ebgp.nodes(bgp_role="c"))


    def appendPlIn(iface, policy):
        pol = iface.get("plIn") or []
        if policy not in pol:
            pol.append(policy)
        iface.set("plIn", pol)


    def appendPlOut(iface, policy):
        pol = iface.get("plOut") or []
        if policy not in pol:
            pol.append(policy)
        iface.set("plOut", pol)


    def appendRmIn(iface, policy):
        pol = iface.get("rmIn") or []
        if policy not in pol:
            pol.append(policy)
        iface.set("rmIn", pol)


    def appendRmOut(iface, policy):
        pol = iface.get("rmOut") or []
        if policy not in pol:
            pol.append(policy)
        iface.set("rmOut", pol)


    def appendRmDo(iface, policy):
        pol = iface.get("rmDo") or []
        if policy not in pol:
            pol.append(policy)
        iface.set("rmDo", pol)

    for node in backbones:
        for edge in node.edges():
            dst = edge.dst
            src_int = edge.src_int
            dst_int = edge.dst_int

            if dst in backbones:
                pass  # b-b link

            elif dst in providers:
                src_int.set("default_originate", True)
                appendPlIn(src_int, "acceptAny")
                appendPlOut(src_int, "defaultOut")
                pass  # b -> p

            elif dst in customers:
                pass  # invalid b -> c

    for node in providers:
        prefix_lists = {}
        for edge in node.edges():
            dst = edge.dst
            src_int = edge.src_int
            dst_int = edge.dst_int

            if dst in backbones:
                pass  # p-b link

            elif dst in providers:
                # p -> p
                appendRmDo(src_int, "dontUseMe")
                appendRmOut(src_int, "dontUseMe")
                appendPlOut(src_int, "defaultOut")

            elif dst in customers:
                # p -> c
                src_int.set("default_originate", True)
                appendPlOut(src_int, "defaultOut")
                dst_asn = dst.get("asn")
                plist_name = "as%sIn" % dst_asn
                prefix_lists[plist_name] = dst_asn
                appendPlIn(src_int, plist_name)

        node.set("pl_to_create", prefix_lists)

    for node in customers:
        for edge in node.edges():
            dst = edge.dst
            src_int = edge.src_int
            dst_int = edge.dst_int

            if dst in backbones:
                pass  # invalid c-b link

            elif dst in providers:
                # c -> p
                if edge.get("bgp_policy"):
                    edge_pol = edge.get("bgp_policy")
                    if edge_pol == "backup":
                        appendRmOut(src_int, "metricOut")
                        appendRmIn(src_int, "localPrefIn")

                appendPlIn(src_int, "defaultIn")
                appendPlOut(src_int, "mineOutOnly")

            elif dst in customers:
                pass  # invalid c -> c
```

Listing H.37: Apply per-node eBGP policies

```
prefix_list_library = {
    "defaultOut": [("permit", "0.0.0.0/0")],
    "defaultOk": [("permit", "0.0.0.0/0")],
    "defaultIn": [("permit", "0.0.0.0/0")],
    "defaultOut": [("permit", "0.0.0.0/0")],
    "acceptAny": [("permit", "any")]
}

# Create prefix lists
for node in sorted(g_ebgp):
    cust_asn_prefixes = node.get("customer_asn_prefixes")
    asn = node.get("asn")
    asn_host_prefixes = host_allocations.get(asn, [])
    pl_to_create = node.get("pl_to_create") or {}

    prefix_lists = {}
    pls = set()

    endpoints = node.interfaces(category="bgp_session")
    for ep in endpoints:
        pls.update(p for p in ep.get("plIn") or [])
        pls.update(p for p in ep.get("plOut") or [])

    for pl in pls:
        # lookup
        if pl in prefix_list_library:
            prefix_lists[pl] = prefix_list_library[pl]
        else:
            if pl == "mineOutOnly":
                prefix_lists[pl] = [("permit", str(prefix))
                                    for prefix in node.get("networks")]
            else:
                pass
    for pl_name, dst_asn in pl_to_create.items():
        cust_prefixes = cust_asn_prefixes[dst_asn]
        prefix_lists[pl_name] = [("permit", str(prefix))
                                 for prefix in cust_prefixes]

    node.set("prefix_lists", prefix_lists)
    node.set("prefix_list_keys", prefix_lists.keys())

autonetkit.update_http(anm)
```

Listing H.38: Create Prefix Lists

```
# route maps

for node in g_ebgp:
    asn = node.get("asn")
    route_maps = {}
    access_lists = {}

    endpoints = node.interfaces(category="bgp_session")
    rms = set()
    for ep in endpoints:
        rms.update(p for p in ep.get("rmIn") or [])
        rms.update(p for p in ep.get("rmOut") or [])
        rms.update(p for p in ep.get("rmDo") or [])

    for rm in rms:
        if rm == "dontUseMe":
            route_maps[rm] = [("prepend", [asn, asn, asn])]
        elif rm == "localPrefIn":
            route_maps[rm] = [("local-preference", 90)]
        elif rm == "metricOut":
            route_maps[rm] = [
                ("match-ip-address", "myAggregate"),
                ("metric", 10)]

            # This could be parameterised further
            myAggregate = []
            for prefix in infra_allocations[asn]:
                myAggregate.append(("permit", str(prefix)))
            access_lists["myAggregate"] = myAggregate

        else:
            print "ERROR: Unknown route map", rm

    node.set("route_maps", route_maps)
    node.set("access_lists", access_lists)

autonetkit.update_http(anm)
```

Listing H.39: Create Route Maps

```
# routing protocol interactions
for node in g_rip:
    redist = node["input"].get("redistribute_bgp_to_igp")
    if not redist:
        continue

    if node["ebgp"].degree() > 0:
        node.set("redistribute_bgp", True)

autonetkit.update_http(anm)
```

Listing H.40: Set Routing Protocol Redistribution

H.3.3 *Templates*

```
router rip {{node.rip.process_id}}
    {% if node.rip.redistribute_connected %}
redistribute connected
    {% endif %}
    {% if node.rip.redistribute_bgp %}
redistribute bgp
    {% endif %}
    {% for network in node.rip.networks %}
network {{network.cidr}}
    {% endfor %}
!
```

Listing H.41: RIP Template

```
!
hostname bgpd
password zebra
enable password zebra
!
router bgp {{node.asn}}
{% for network in node.bgp.networks %}
network {{network}}
{% endfor %}
{% for neigh in node.bgp.ibgp_neighbors %}
!
neighbor {{neigh.neigh_ip}} remote-as {{neigh.asn}}
neighbor {{neigh.neigh_ip}} description {{neigh.desc}} (iBGP)
{% endfor %}
!
{% for neigh in node.bgp.ebgp_neighbors %}
! {{neigh.neighbor}}
neighbor {{neigh.neigh_ip}} remote-as {{neigh.asn}}
neighbor {{neigh.neigh_ip}} description {{neigh.desc}} (eBGP)
{% if neigh.default_originate %}
neighbor {{neigh.neigh_ip}} default-originate
{% endif %}
    {% for plName in neigh.plOut %}
neighbor {{neigh.neigh_ip}} prefix-list {{plName}} out
    {% endfor %}
    {% for rmName in neigh.rmDo %}
neighbor {{neigh.neigh_ip}} default-originate route-map {{rmName}}
    {% endfor %}
    {% for rmName in neigh.rmOut %}
neighbor {{neigh.neigh_ip}} route-map {{rmName}} out
    {% endfor %}
    {% for plName in neigh.plIn %}
neighbor {{neigh.neigh_ip}} prefix-list {{plName}} in
    {% endfor %}
    {% for rmName in neigh.rmIn %}
neighbor {{neigh.neigh_ip}} route-map {{rmName}} in
    {% endfor %}
{% endfor %}
!
!
{% for name, data in node.bgp.prefix_lists.items() %}
    {% for permitDeny, prefix in data %}
ip prefix-list {{name}} {{permitDeny}} {{prefix}}
    {% endfor %}
!
{% endfor %}
!
```

```
{% for name, data in node.bgp.route_maps.items() %}
route-map {{name}} permit 10
    {% for keyword, value in data %}
{{keyword}} {{value}}
    {% endfor %}
!
{% endfor %}
!
{% for name, data in node.bgp.access_lists.items() %}
    {% for permitDeny, prefix in data %}
access-list {{name}} {{permitDeny}} {{prefix}}
    {% endfor %}
!
{% endfor %}
!
log file /var/log/zebra/bgpd.log
!
debug bgp
debug bgp events
debug bgp filters
debug bgp fsm
debug bgp keepalives
debug bgp updates
!
```

Listing H.42: BGP Template

H.3.4 *Compilation and Rendering*

Netkit Platform Compiler

```
# Create Platform Compiler and Compile
sim_plat = simple_platform_compiler(anm, nidb)
sim_plat.compile()
```

Listing H.43: Netkit Platform Compiler

Quagga Device Compiler

```
from autonetkit.compilers.device import router_base

class simple_router_compiler(router_base.RouterCompiler):

    def compile(self, node):
        zebra = self.zebra(node)
        ssh = self.ssh(node)
        interfaces = self.interfaces(node)
        rip = self.rip(node)
        bgp = self.bgp(node)

        return {
            "zebra": zebra,
            "ssh": ssh,
            "interfaces": interfaces,
            "rip": rip,
            "bgp": bgp,
            "hostname": str(node),
            "asn": node.get("asn")
        }

    def zebra(self, node):
        password = "zebra"
        return {
            "password": password
        }

    def ssh(self, node):
        return {
            "use_key": False
        }

    def interfaces(self, node):
        ifaces = []
        for interface in self.nidb["interfaces"].get(node):
            print "iface is", interface
            int_id = interface.get("id")
            ip = interface.get("ip")
            subnet = interface.get("subnet")
            ifaces.append({"ip": ip,
                           "id": int_id,
                           "broadcast": subnet.broadcast,
                           "netmask": subnet.netmask})

        return ifaces

    def rip(self, node):
        rip_node = self.anm["rip"].node(node)
        networks = rip_node.get("networks") or []
        redistribute_connected = rip_node.get("redistribute_connected")
            or False
        redistribute_bgp = rip_node.get("redistribute_bgp") or False

        return {
            "networks": networks,
            "redistribute_connected": redistribute_connected,
            "redistribute_bgp": redistribute_bgp,
        }

    def process_route_maps(self, route_maps):
        # maps from generic route-map command to Quagga-specific syntax
        retval = {}

        lookup_table = {
            "local-preference": "set local-preference",
            "match-ip-address": "match ip address",
            "metric": "set metric"
        }

        for name, data in route_maps.items():
            processed = []
            for action, value in data:
                if action in lookup_table:
                    processed.append((lookup_table[action], value))
                elif action == "prepend":
                    formatted_value = " ".join(str(v) for v in value)
                    processed.append(("set as-path prepend",
                        formatted_value))
                else:
                    print "Unknown action: %s" % action

            retval[name] = processed

        return retval

    def bgp(self, node):
        ibgp_neighbors = []
        g_ibgp = self.anm["ibgp"]

        for session in g_ibgp.edges(node):
            dst = session.dst
            dst_int = session.dst_int
            dst_bound_int = dst_int.get("bound_to")
            neigh_ip = dst_bound_int["ip"].get("ip")
            desc = "Router %s" % dst
            data = {
                "neigh_ip": neigh_ip,
                "asn": dst.get("asn"),
                "desc": desc
            }
            ibgp_neighbors.append(data)

        ebgp_neighbors = []
        g_ebgp = self.anm["ebgp"]
        ebgp_node = g_ebgp.node(node)
        for session in g_ebgp.edges(node):
            plIn = session.src_int.get("plIn") or []
            plOut = session.src_int.get("plOut") or []
            rmIn = session.src_int.get("rmIn") or []
            rmOut = session.src_int.get("rmOut") or []
            rmDo = session.src_int.get("rmDo") or []
            default_originate = session.src_int.get(
                "default_originate") or False

            dst = session.dst
            dst_int = session.dst_int
            dst_bound_int = dst_int.get("bound_to")
            neigh_ip = dst_bound_int["ip"].get("ip")
            desc = "Router %s" % dst
            data = {
                "neigh_ip": neigh_ip,
                "asn": dst.get("asn"),
                "desc": desc,
                "default_originate": default_originate,
                "plIn": plIn,
                "plOut": plOut,
                "rmIn": rmIn,
                "rmOut": rmOut,
                "rmDo": rmDo
            }
            ebgp_neighbors.append(data)

        networks = ebgp_node.get("networks") or []
        prefix_lists = ebgp_node.get("prefix_lists")
        route_maps = self.process_route_maps(ebgp_node.get("route_maps"))
        access_lists = ebgp_node.get("access_lists")
        return {
            "ibgp_neighbors": ibgp_neighbors,
            "ebgp_neighbors": ebgp_neighbors,
            "networks": networks,
            "prefix_lists": prefix_lists,
            "route_maps": route_maps,
            "access_lists": access_lists
        }
```

Listing H.44: Quagga Device Compiler

Example Router Configurations

```
!
hostname bgpd
password zebra
enable password zebra
!
router bgp 30
network 11.0.0.8/30
network 30.3.3.0/24
!
!
neighbor 11.0.0.9 remote-as 300
neighbor 11.0.0.9 description Router as300r1 (eBGP)
neighbor 11.0.0.9 default-originate
neighbor 11.0.0.9 prefix-list defaultOut out
neighbor 11.0.0.9 prefix-list as300In in
!
neighbor 11.0.0.26 remote-as 1
neighbor 11.0.0.26 description Router as1r1 (eBGP)
!
neighbor 11.0.0.17 remote-as 20
neighbor 11.0.0.17 description Router as20r1 (eBGP)
neighbor 11.0.0.17 default-originate route-map dontUseMe
neighbor 11.0.0.17 route-map dontUseMe out
!
!
ip prefix-list as300In permit 200.1.128.0/17
!
ip prefix-list defaultOut permit 0.0.0.0/0
!
!
route-map dontUseMe permit 10
set as-path prepend 30 30 30
!
!
!
log file /var/log/zebra/bgpd.log
!
debug bgp
debug bgp events
debug bgp filters
debug bgp fsm
debug bgp keepalives
debug bgp updates
!
```

Listing H.45: BGPd configuration for *as30r1* from Complete Small Internet Case Study

H.3.5 *Simulation Results*

```
as1r1# sh ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

B>* 11.0.0.0/30 [20/0] via 11.0.0.21, eth2, 00:01:07
B>* 11.0.0.4/30 [20/0] via 11.0.0.21, eth2, 00:00:37
B>* 11.0.0.8/30 [20/0] via 11.0.0.25, eth1, 00:01:07
B>* 11.0.0.12/30 [20/0] via 11.0.0.29, eth0, 00:01:06
B>* 11.0.0.16/30 [20/0] via 11.0.0.21, eth2, 00:00:37
C>* 11.0.0.20/30 is directly connected, eth2
C>* 11.0.0.24/30 is directly connected, eth1
```

```
C>* 11.0.0.28/30 is directly connected, eth0
B>* 11.0.0.32/30 [20/0] via 11.0.0.21, eth2, 00:00:37
B>* 20.1.1.0/24 [20/0] via 11.0.0.21, eth2, 00:01:07
B>* 30.3.3.0/24 [20/0] via 11.0.0.25, eth1, 00:01:07
B>* 40.4.4.0/24 [20/0] via 11.0.0.29, eth0, 00:01:06
B>* 100.1.0.0/16 [20/0] via 11.0.0.21, eth2, 00:00:37
C>* 127.0.0.0/8 is directly connected, lo
B>* 200.1.0.0/16 [20/0] via 11.0.0.25, eth1, 00:01:07
B>* 200.1.0.0/17 [20/0] via 11.0.0.25, eth1, 00:01:07
B>* 200.1.128.0/17 [20/0] via 11.0.0.29, eth0, 00:01:06
B>* 200.2.0.0/16 [20/0] via 11.0.0.21, eth2, 00:00:37

as20r1# sh ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

B>* 0.0.0.0/0 [200/0] via 11.0.0.22 (recursive via 20.1.1.3, eth2),
    00:06:06
B   11.0.0.0/30 [200/0] via 20.1.1.2, eth2, 00:06:06
R>* 11.0.0.0/30 [120/2] via 20.1.1.2, eth2, 00:06:12
C>* 11.0.0.4/30 is directly connected, eth1
C>* 11.0.0.16/30 is directly connected, eth3
R>* 11.0.0.20/30 [120/2] via 20.1.1.3, eth2, 00:06:12
C>* 11.0.0.32/30 is directly connected, eth0
C>* 20.1.1.0/24 is directly connected, eth2
B>* 100.1.0.0/16 [200/0] via 11.0.0.1 (recursive via 20.1.1.2, eth2),
    00:06:06
C>* 127.0.0.0/8 is directly connected, lo
B>* 200.2.0.0/16 [20/0] via 11.0.0.33, eth0, 00:06:08

BGP table version is 0, local router ID is 11.0.0.18
Status codes: s suppressed, d damped, h history, * valid, > best, i -
    internal,
              r RIB-failure, S Stale, R Removed
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*  0.0.0.0          11.0.0.17                              0 20 20 20 20
    i
*>                  11.0.0.26                              0 1 i
*> 11.0.0.8/30      0.0.0.0                  0         32768 i
*> 30.3.3.0/24      0.0.0.0                  0         32768 i
*> 200.1.0.0/16     11.0.0.9                 0             0 300 i
*> 200.1.0.0/17     11.0.0.9                 0             0 300 i

as300r1# sh ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

B>* 0.0.0.0/0 [20/0] via 11.0.0.10, eth0, 00:23:15
C>* 11.0.0.8/30 is directly connected, eth0
C>* 127.0.0.0/8 is directly connected, lo
C>* 200.1.0.0/18 is directly connected, eth1
R>* 200.1.64.0/18 [120/3] via 200.1.0.2, eth1, 00:23:16
R>* 200.1.128.0/17 [120/2] via 200.1.0.2, eth1, 00:23:16
as300r1#

bgpd# sh ip bgp
BGP table version is 0, local router ID is 11.0.0.17
Status codes: s suppressed, d damped, h history, * valid, > best, i -
    internal,
              r RIB-failure, S Stale, R Removed
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
```

```
*  0.0.0.0          11.0.0.18                              0 30 30 30 30
    i
*>i                 11.0.0.22                    100      0 1 i
*>i11.0.0.0/30      20.1.1.2                0    100      0 i
*> 11.0.0.4/30      0.0.0.0                 0         32768 i
*> 11.0.0.16/30     0.0.0.0                 0         32768 i
*> 11.0.0.32/30     0.0.0.0                 0         32768 i
* i20.1.1.0/24      20.1.1.3                0    100      0 i
* i                 20.1.1.2                0    100      0 i
*>                  0.0.0.0                 0         32768 i
*>i100.1.0.0/16     11.0.0.1                0    100      0 100 i
*> 200.2.0.0/16     11.0.0.33               0             0 200 i

as300r4# sh ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

R>* 0.0.0.0/0 [120/2] via 200.1.64.1, eth0, 00:24:59
C>* 127.0.0.0/8 is directly connected, lo
R>* 200.1.0.0/18 [120/2] via 200.1.128.1, eth1, 00:25:05
C>* 200.1.64.0/18 is directly connected, eth0
C>* 200.1.128.0/17 is directly connected, eth1
```

H.4 CODE FOR VLAN CASE STUDY

H.4.1 *Network Whiteboard*

```
import autonetkit.load.graphml as graphml
with open("vlans.graphml") as fh:
    data = fh.read()

input_graph = graphml.load_graphml(data)
nodes = {}

switches = {4, 5, 8, 10, 14}

for n, data in input_graph.nodes(data=True):
    if int(n) in switches:
        device_type = "switch"
    else:
        device_type = "router"

    nodes[n] = {
        "id": n,
        "asn": data["asn"],
        "device_type": device_type,
        "x": int(data["x"] * 1.2),
        "y": int(data["y"] * 1.2),
    }

edges = []
for s, t, data in input_graph.edges(data=True):
    edges.append((s, t, data.get("label")))
```

Listing H.46: Prepare Network Whiteboard

```
import autonetkit
anm = autonetkit.NetworkModel()
g_in = anm.add_overlay("input")

nodes = {
    "1":  {"y": 96,   "x": 522,  "device_type": "router", "asn": 1},
    "2":  {"y": 70,   "x": 927,  "device_type": "router", "asn": 1},
    "3":  {"y": 536,  "x": 114,  "device_type": "router", "asn": 1},
    "4":  {"y": 536,  "x": 522,  "device_type": "switch", "asn": 1},
    "5":  {"y": 530,  "x": 927,  "device_type": "switch", "asn": 1},
    "6":  {"y": 530,  "x": 1306, "device_type": "router", "asn": 1},
    "7":  {"y": 962,  "x": 114,  "device_type": "router", "asn": 1},
    "8":  {"y": 962,  "x": 522,  "device_type": "switch", "asn": 1},
    "9":  {"y": 959,  "x": 927,  "device_type": "router", "asn": 1},
    "10": {"y": 959,  "x": 1306, "device_type": "switch", "asn": 1},
    "11": {"y": 959,  "x": 1705, "device_type": "router", "asn": 1},
    "12": {"y": 1388, "x": 529,  "device_type": "router", "asn": 1},
    "13": {"y": 1388, "x": 927,  "device_type": "router", "asn": 1},
    "14": {"y": 1388, "x": 1316, "device_type": "switch", "asn": 1}
}

for n, d in nodes.items():
    g_in.add_node(n, x=d["x"] / 2,
                  device_type=d["device_type"],
                  y=d["y"] / 2, asn=d["asn"])
```

Listing H.47: Create Network Whiteboard

```
edges = [("11", "10", "2"), ("13", "8", "1"),
         ("13", "14", "2"), ("12", "8", "3"),
         ("14", "10", None), ("1", "4", "1"),
         ("3", "4", "1"), ("2", "5", "3"),
         ("4", "5", None), ("7", "8", "2"),
         ("6", "10", "1"), ("6", "5", "2"),
         ("9", "10", "1"), ("9", "8", "1"),
         ("8", "4", None)]

for s, t, vlan in edges:
    src = g_in.node(s)
    dst = g_in.node(t)
    src_iface = src.add_interface()
    dst_iface = dst.add_interface()

    if vlan:
        dst_iface.set("vlan", vlan)

    g_in.add_edge(src_iface, dst_iface)
```

Listing H.48: Add Edges to Network Whiteboard

H.4.2 *Design Functions*

```
g_phy = anm['phy']
g_phy.add_nodes_from(g_in, retain=["asn", "device_type", "x", "y"])
g_phy.update(platform="netkit", syntax="quagga")

g_phy.add_edges_from(g_in.edges())
```

Listing H.49: Create Physical Network View

```
from autonetkit.ank import copy_int_attr_from, connected_subgraphs
from collections import defaultdict
from itertools import count
bc_node_id = count()

g_vlan = anm.add_overlay("vlan")
g_vlan.add_nodes_from(g_phy)
g_vlan.add_edges_from(g_phy.edges())
autonetkit.update_http(anm)
```

Listing H.50: Create VLAN Network View

```
from autonetkit.ank import unwrap_nodes, unwrap_graph, wrap_edges,
    wrap_nodes
import networkx as nx
import autonetkit.log as log
from itertools import count


def aggregate_nodes(nm_graph, nodes, retain=None):
    """Combines connected into a single node,
    Note: retain is for the interfaces..."""
    pseudo_nodes = []
    if retain is None:
        retain = []
    try:
        retain.lower()
        retain = [retain]  # was a string, put into list
    except AttributeError:
        pass  # already a list
    nodes = list(unwrap_nodes(nodes))
    graph = unwrap_graph(nm_graph)
    subgraph = graph.subgraph(nodes)
    if not len(subgraph.edges()):
        pass
    total_added_edges = []
    if graph.is_directed():
        component_nodes_list = nx.strongly_connected_components(subgraph)
    else:
        component_nodes_list = nx.connected_components(subgraph)

    pseudo_node_id = count()
    for component_nodes in component_nodes_list:
        if len(component_nodes) > 1:
            component_nodes = [nm_graph.node(n)
                               for n in component_nodes]

            nodes_to_remove = list(component_nodes)
            pn_id = "pn" + str(pseudo_node_id.next())
            base = nm_graph.add_node(pn_id)
            base.set("device_type", "null")
            pseudo_nodes.append(base)

            base_wrapped = nm_graph.node(base)

            external_edges = []
            for node in nodes_to_remove:
                external_edges += [e for e in node.edges() if e.dst
                                   not in component_nodes]

            edges_to_add = []
            for edge in external_edges:
                dst = edge.dst
                dst_int = edge.dst_int

                prev_src_int = edge.src_int
                base_src_int = base.add_interface()
                for key in retain:
                    base_src_int.set(key, prev_src_int.get(key))

                new_edge = nm_graph.add_edge(base_src_int, dst_int)
            nm_graph.remove_nodes_from(nodes_to_remove)

            neighs = base.neighbors()
            x = sum(neigh["phy"].get("x") for neigh in neighs) / len(
                neighs)
            y = sum(neigh["phy"].get("y") for neigh in neighs) / len(
                neighs)
            base.set("x", x)
            base.set("y", y)

    return pseudo_nodes
```

Listing H.51: Aggregate Nodes High-Level Design Primitive

```
copy_int_attr_from(g_in, g_vlan, "vlan")

switches = g_vlan.switches()

pseudo_nodes = aggregate_nodes(g_vlan, switches, retain=["vlan"])
autonetkit.update_http(anm)
```

Listing H.52: Aggregate VLANs

```
for p in pseudo_nodes:

    # group interfaces
    vlan_edges = defaultdict(list)
    for edge in p.edges():
        vlan = edge.src_int.get("vlan")
        if vlan:
            vlan_edges[vlan].append(edge)

    # now create a new pseudo node for this vlan group
    for vlan, edges in vlan_edges.items():
        bc_id = "bc" + str(bc_node_id.next())
        bc = g_vlan.add_node(bc_id, device_type="broadcast_domain")
        bc.set("broadcast_domain", True)

        g_phy.add_node(bc)

        for edge in edges:
            bc_iface = bc.add_interface()
            dst_iface = edge.dst_int
            g_vlan.add_edge(bc_iface, dst_iface)

        neighs = bc.neighbors()
        x = sum(neigh["phy"].get("x") for neigh in neighs) / len(neighs)
        y = sum(neigh["phy"].get("y") for neigh in neighs) / len(neighs)
        bc.set("x", x)
        bc.set("y", y)

    g_vlan.remove_node(p)

autonetkit.update_http(anm)
```

Listing H.53: Form VLAN Broadcast Domains

```
from autonetkit.ank import split
from autonetkit.ank import copy_int_attr_from, connected_subgraphs

g_switch = anm.add_overlay("switches")
g_switch.add_nodes_from(g_in.switches())
g_switch.add_edges_from(g_phy.edges())

copy_int_attr_from(g_in, g_switch, "vlan")

for edge in g_switch.edges():
    edge.src_int.set("trunking", True)
    edge.dst_int.set("trunking", True)

# and copy the VLANs for explicitly adding to each switch for VTP
connected = connected_subgraphs(g_switch)
for conn in connected:
    vlans = set()
    for node in conn:
        vlans.update(i.get("vlan") for i in node.physical_interfaces()
                     if i.get("vlan"))
    # now store on the nodes
    for node in conn:
        node.set("vlans", list(vlans))
autonetkit.update_http(anm)
```

Listing H.54: Create Switches Network View

```
from autonetkit.ank import split
from collections import Counter
```

```
g_l2 = anm.add_overlay("layer2")
g_l2.add_nodes_from(g_vlan, retain="broadcast_domain")
g_l2.add_edges_from(g_vlan.edges())

# Split the point-to-point edges to add a collision domain
edges_to_split = [edge for edge in g_l2.edges()
                  if edge.src.is_l3device() and edge.dst.is_l3device()]

for edge in edges_to_split:
    edge.split = True  # mark as split for use in building nidb

split_created_nodes = split(g_l2, edges_to_split, id_prepend='bd_')

for node in split_created_nodes:
    # set midway x, y for plot
    neighs = node.neighbors()
    x = sum(neigh["phy"].get("x") for neigh in neighs) / len(neighs)
    y = sum(neigh["phy"].get("y") for neigh in neighs) / len(neighs)
    node.set("x", x)
    node.set("y", y)

    c = Counter(n.get("asn") for n in neighs)
    most_common_asn, _ = c.most_common(1)[0]
    node.set("asn", most_common_asn)

    node.set("broadcast_domain", True)
    node.set("device_type", "broadcast_domain")

autonetkit.update_http(anm)
```

Listing H.55: Create Layer 2 Network View

```
from autonetkit.ank import explode_nodes

g_l2_conn = anm.add_overlay("layer2_conn")
g_l2_conn.add_nodes_from(g_l2)
g_l2_conn.add_edges_from(g_l2.edges())
bc_nodes = g_l2.nodes(broadcast_domain=True)
explode_nodes(g_l2_conn, bc_nodes)
```

Listing H.56: Create Layer 2 Connectivity Network View

```
from autonetkit.ank import split, groupby, copy_attr_from
from netaddr import IPNetwork

g_ip = anm.add_overlay("ip")
g_ip.add_nodes_from(g_l2, retain="device_type")
g_ip.add_edges_from(g_l2.edges())

bc_nodes = g_l2.nodes(broadcast_domain=True)

bc_attrs = ["broadcast_domain", "asn"]
for attr in bc_attrs:
    copy_attr_from(g_l2, g_ip, attr, nbunch=bc_nodes)

# allocate loopback IPs
block = IPNetwork("172.16.0.0/16")
subnets = block.subnet(24)

loopback_allocations = {}

l3_nodes = g_ip.l3devices()
for asn, nodes in groupby("asn", l3_nodes):
    asn_block = subnets.next()
    loopback_allocations[asn] = asn_block
    hosts = asn_block.iter_hosts()

    for node in nodes:
        ip = hosts.next()
        node.set("loopback", ip)

# allocate infra IPs
block = IPNetwork("192.168.0.0/16")
subnets = block.subnet(24)

infra_allocations = {}

bc_nodes = g_ip.nodes(broadcast_domain=True)
for asn, nodes in groupby("asn", bc_nodes):
    asn_block = subnets.next()
    infra_allocations[asn] = asn_block
    ptp_subnets = asn_block.subnet(27)
    for node in nodes:
        prefix = ptp_subnets.next()
        node.set("subnet", prefix)
        hosts = prefix.iter_hosts()
        for neigh_iface in node.neighbor_interfaces():
            address = hosts.next()
            neigh_iface.set("ip", address)
            neigh_iface.set("subnet", prefix)

autonetkit.update_http(anm)
```

Listing H.57: Create IP Network View

```
g_ospf = anm.add_overlay("ospf")
g_ospf.add_nodes_from(g_in.routers())
g_ospf.add_edges_from(e for e in g_l2_conn.edges()
                      if e.src.asn == e.dst.asn)
autonetkit.update_http(anm)
```

Listing H.58: Create OSPF Network View

H.4.3 *Compilation and Rendering*

VIRL Platform Compiler

```
class simple_platform_compiler(object):

    def __init__(self, anm, nidb):
        self.anm = anm
        self.nidb = nidb

    def assign_router_interface_ids(self, node):
        node.loopback_zero.set("id", "Loopback0")
        for index, interface in enumerate(node.physical_interfaces()):
            id = "GigabitEthernet0/%s" % (index + 1)
            interface.set("id", id)
            interface.set("brief_id", id.replace("GigabitEthernet", ""))

    def assign_switch_interface_ids(self, node):
        for index, interface in enumerate(node.physical_interfaces()):
            offset_index = index + 1
            quotient = offset_index / 4  # python integer division
            remainder = offset_index % 4
            id = "GigabitEthernet%s/%s" % (quotient, remainder)
            interface.set("id", id)
            interface.set("brief_id", id.replace("GigabitEthernet", ""))

    def setup_interfaces(self, node):
        ifaces = []
        copy_ip_info = node.is_l3device()

        for interface in node.physical_interfaces():
            if copy_ip_info:
                ip_int = interface["ip"]
                ip = ip_int.get("ip")
                subnet = ip_int.get("subnet")
            else:
                ip = subnet = None

            ifaces.append({
                "id": interface.get("id"),
                "ip": ip,
                "subnet": subnet
            })

        return ifaces

    def compile(self):
        g_in = self.anm["input"]
        g_phy = self.anm["phy"]
        g_ip = self.anm["ip"]

        rtr_comp = simple_router_compiler(self.anm, self.nidb)
        switch_comp = simple_switch_compiler(self.anm, self.nidb)

        virl_hosts = g_phy.nodes()

        for host in g_phy.routers():
            self.assign_router_interface_ids(host)
            self.nidb["interfaces"][host] = self.setup_interfaces(host)

        for host in g_phy.switches():
            self.assign_switch_interface_ids(host)
            self.nidb["interfaces"][host] = self.setup_interfaces(host)
        templates = {}

        for node in g_phy.routers():
            rtr_data = rtr_comp.compile(node)
            self.nidb["hosts"][node.label] = rtr_data
            templates[node.label] = router_template

        for node in g_phy.switches():
            switch_data = switch_comp.compile(node)
            self.nidb["hosts"][node.label] = switch_data
            templates[node.label] = switch_template

        lab = {"machines": virl_hosts, "subnets": subnets}

        self.nidb["labs"] = lab
        self.nidb["templates"] = templates

sim_plat = simple_platform_compiler(anm, nidb)
sim_plat.compile()
```

Listing H.59: VIRL Platform Compiler

IOSv Router Compiler

```
class simple_router_compiler(object):

    def __init__(self, anm, nidb):
        self.anm = anm
        self.nidb = nidb

    def compile(self, node):
        interfaces = self.interfaces(node)
        ospf = self.ospf(node)

        hostname = str(node)
        if hostname.isdigit():
            hostname = "sw" + hostname

        return {
            "interfaces": interfaces,
            "ospf": ospf,
            "hostname": hostname,
            "asn": node.get("asn")
        }

    def interfaces(self, node):
        ifaces = []
        phy_node = self.anm["phy"].node(node)
        for interface in phy_node.physical_interfaces():
            int_id = interface.get("id")
            ip_int = interface["ip"]
            ip = ip_int.get("ip")
            subnet = ip_int.get("subnet")

            ospf_int = interface["ospf"]
            ospf_cost = ospf_int.get("cost")

            ifaces.append({"ip": ip,
                           "id": int_id,
                           "physical": True,
                           "ospf_cost": ospf_cost,
                           "broadcast": subnet.broadcast,
                           "netmask": subnet.netmask})
        interface = node.loopback_zero
        ip_int = interface["ip"]
        int_id = interface.get("id")
        ip = ip_int.get("ip")
        subnet = ip_int.get("subnet")
        ifaces.append({"ip": ip,
                       "id": int_id,
                       "physical": False,
                       "broadcast": subnet.broadcast,
                       "netmask": subnet.netmask})

        return ifaces

    def ospf(self, node):
        # process interfaces
        networks = []
        ospf_node = node["ospf"]
        process_id = ospf_node.get("process_id")
        for interface in ospf_node.physical_interfaces():
            area = interface.get("area")
            subnet = interface["ip"].get("subnet")
            networks.append({
                "area": area,
                "prefix": subnet.network,
                "hostmask": subnet.hostmask
            })

        lo0 = ospf_node.loopback_zero
        subnet = lo0["ip"].get("subnet")
        networks.append({
            "area": lo0.get("area"),
            "prefix": subnet.network,
            "hostmask": subnet.hostmask
        })

        return {"networks": networks,
                "process_id": process_id}
```

Listing H.60: VIRL IOSv Router Compiler

IOSvL2 Switch Compiler

```
class simple_switch_compiler(object):

    def __init__(self, anm, nidb):
        self.anm = anm
        self.nidb = nidb

    def compile(self, node):
        interfaces = self.interfaces(node)
        vlans = self.vlans(node)

        return {
            "interfaces": interfaces,
            "vlans": vlans,
            "hostname": str(node),
            "asn": node.get("asn")
        }

    def interfaces(self, node):
        ifaces = []
        phy_node = self.anm["phy"].node(node)
        data_interfaces = phy_node.physical_interfaces()
        for interface in data_interfaces:
            int_id = interface.get("id")
            switches_iface = self.anm["switches"].interface(interface)

            if switches_iface.get("trunking"):
                trunking = True
            else:
                trunking = False

            vlan = switches_iface.get("vlan")

            ifaces.append({
                "id": int_id,
                "trunking": trunking,
                "vlan": vlan
            })

        return ifaces

    def vlans(self, node):
        switch_node = self.anm["switches"].node(node)
        retval = switch_node.get("vlans")
        return retval
```

Listing H.61: VIRL IOSvL2 Switch Compiler

Templates

```
!
hostname {{node.hostname}}
boot-start-marker
boot-end-marker
!
no aaa new-model
!
ip cef
!
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
no service config
enable password cisco
ip classless
ip subnet-zero
no ip domain lookup
line vty 0 4
 transport input ssh telnet
 exec-timeout 720 0
 password cisco
 login
line con 0
 password cisco
!
interface GigabitEthernet0/0
  description OOB Management
  ! Configured on launch
  no ip address
  duplex auto
  speed auto
  no shutdown
!
{% for interface in node.interfaces %}
interface {{interface.id}}
  ip address {{interface.ip}} {{interface.netmask}}
  {% if interface.physical %}
  duplex auto
  speed auto
  no shutdown
  {% endif %}
!
{% endfor %}
!
router ospf {{node.ospf.process_id}}
  log-adjacency-changes
  passive-interface Loopback0
{% for network in node.ospf.networks %}
  network {{network.prefix}} {{network.hostmask}} area {{network.area}}
{% endfor %}
!
end
```

Listing H.62: VIRL IOSv Router Template

```
!
version 15.2
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
service compress-config
no service config
enable password cisco
ip classless
ip subnet-zero
no ip domain lookup
!
line vty 0 4
transport input ssh telnet
exec-timeout 720 0
password cisco
login
!
line con 0
password cisco
!
hostname {{node.hostname}}
!
boot-start-marker
boot-end-marker
!
no aaa new-model
!
ip cef
no ipv6 cef
!
spanning-tree mode pvst
spanning-tree extend system-id
!
vlan internal allocation policy ascending
!
vtp domain test.lab
vtp mode transparent
{% for vlan in node.vlans %}
vlan {{vlan}}
{% endfor %}
!
interface Loopback0
  description Loopback
!
interface GigabitEthernet0/0
  description Mapped to Vlan1 for management
  ! Configured on launch
  switchport mode access
  no shutdown
!
{% for interface in node.interfaces %}
interface {{interface.id}}
  {% if interface.trunking %}
  switchport trunk encapsulation dot1q
  switchport mode trunk
  {% elif interface.vlan %}
  switchport mode access
  switchport access vlan {{interface.vlan}}
  {% endif %}
  no shutdown
!
{% endfor %}
interface Vlan1
  description OOB Management
  ! Configured on launch
  no ip address
!
ip forward-protocol nd
!
no ip http server
no ip http secure-server
!
control-plane
!
end
```

Listing H.63: VIRL IOSvL2 Switch Template

Packaging

```
from collections import defaultdict

import xml.etree.ElementTree as ET
topology = ET.Element('topology')
topology.set("xmlns", "http://www.cisco.com/VIRL")
topology.set("xmlns:xsi", "http://www.w3.org/2001/XMLSchema-instance")
topology.set("schemaVersion", "0.9")
topology.set("xsi:schemaLocation",
             "http://www.cisco.com/VIRL https://raw.github.com/CiscoVIRL/
                 schema/v0.9/virl.xsd")

node_indices = {}
iface_indices = defaultdict(dict)

node_index = 0
for node in sorted(g_phy):
    if node.is_switch():
        subtype = "IOSvL2"
    elif node.is_router():
        subtype = "IOSv"
    else:
        continue

    nodeElem = ET.SubElement(topology, "node")
    nodeName = node.get("label")
    nodeElem.set("type", "SIMPLE")
    nodeElem.set("subtype", subtype)

    nodeExtensions = ET.SubElement(nodeElem, "extensions")
    configEntry = ET.SubElement(nodeExtensions, "entry")
    configEntry.set("key", "config")
    configEntry.set("type", "string")
    configEntry.text = configs[nodeName]
    nodeElem.set("name", node.get("label"))

    location = "%s,%s" % (node.get("x"), node.get("y"))
    nodeElem.set("location", location)
    node_indices[nodeName] = node_index
    node_index = node_index + 1

    for iface_index, iface in enumerate(node.physical_interfaces()):
        ifaceElem = ET.SubElement(nodeElem, "interface")
        ifaceName = str(iface.get("id"))
        ifaceElem.set("name", ifaceName)
        ifaceElem.set("id", str(iface_index))

        iface_indices[nodeName][ifaceName] = iface_index

for edge in g_phy.edges():
    src_node = edge.src
    dst_node = edge.dst
    src_iface = edge.src_int
    dst_iface = edge.dst_int

    src_node_index = node_indices[src_node.get("label")] + 1
    src_iface_index = iface_indices[src_node.get("label")][
        src_iface.get("id")] + 1
    src_string = "/virl:topology/virl:node[%s]/virl:interface[%s]" % (
        src_node_index, src_iface_index)

    dst_node_index = node_indices[dst_node.get("label")] + 1
    dst_iface_index = iface_indices[dst_node.get("label")][
        dst_iface.get("id")] + 1
    dst_string = "/virl:topology/virl:node[%s]/virl:interface[%s]" % (
        dst_node_index, dst_iface_index)

    connection = ET.SubElement(topology, "connection")
    connection.set("src", src_string)
    connection.set("dst", dst_string)

# http://stackoverflow.com/questions/15356641
from io import BytesIO
xml_string = ET.tostring(topology, encoding="UTF-8")
import xml.dom.minidom
xml = xml.dom.minidom.parseString(xml_string)
pretty_xml_as_string = xml.toprettyxml(encoding="UTF-8")
with open("output.virl", "w") as fh:
    fh.write(pretty_xml_as_string)
```

Listing H.64: VIRL XML Packaging

H.4.4 *Simulation Results*

```
sw8#sh vlan

VLAN Name                             Status    Ports
---- -------------------------------- ---------
    ------------------------------
1    default                          active    Gi0/0
2    VLAN0002                         active    Gi0/1, Gi1/0
3    VLAN0003                         active    Gi0/3
4    VLAN0004                         active    Gi0/2
1002 fddi-default                     act/unsup
1003 token-ring-default               act/unsup
1004 fddinet-default                  act/unsup
1005 trnet-default                    act/unsup
```

Listing H.65: Output of SHOW VLAN for *sw8*

```
r2#sh ip route
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS
           level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static
           route
       o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP
       a - application route
       + - replicated route, % - next hop override, p - overrides from
           PfR

Gateway of last resort is not set

      10.0.0.0/8 is variably subnetted, 2 subnets, 2 masks
C        10.255.0.0/16 is directly connected, GigabitEthernet0/0
L        10.255.1.173/32 is directly connected, GigabitEthernet0/0
      172.16.0.0/32 is subnetted, 2 subnets
C        172.16.0.2 is directly connected, Loopback0
O        172.16.0.8 [110/2] via 192.168.0.66, 00:09:51, GigabitEthernet0
    /1
      192.168.0.0/24 is variably subnetted, 2 subnets, 2 masks
C        192.168.0.64/27 is directly connected, GigabitEthernet0/1
L        192.168.0.65/32 is directly connected, GigabitEthernet0/1
```

Listing H.66: Output of SH IP ROUTE for *r2*

```
r9#sh ip route
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS
           level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static
           route
       o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP
       a - application route
       + - replicated route, % - next hop override, p - overrides from
           PfR

Gateway of last resort is not set
```

```
      10.0.0.0/8 is variably subnetted, 2 subnets, 2 masks
C        10.255.0.0/16 is directly connected, GigabitEthernet0/0
L        10.255.1.179/32 is directly connected, GigabitEthernet0/0
      172.16.0.0/32 is subnetted, 7 subnets
O        172.16.0.1 [110/2] via 192.168.0.33, 00:09:05, GigabitEthernet0
    /2
O        172.16.0.3 [110/2] via 192.168.0.34, 00:09:05, GigabitEthernet0
    /2
O        172.16.0.4 [110/2] via 192.168.0.129, 00:08:48, GigabitEthernet0
    /1
O        172.16.0.5 [110/3] via 192.168.0.129, 00:08:48, GigabitEthernet0
    /1
C        172.16.0.6 is directly connected, Loopback0
O        172.16.0.7 [110/3] via 192.168.0.36, 00:09:05, GigabitEthernet0
    /2
O        172.16.0.9 [110/2] via 192.168.0.36, 00:09:05, GigabitEthernet0
    /2
      192.168.0.0/24 is variably subnetted, 6 subnets, 2 masks
O        192.168.0.0/27
           [110/2] via 192.168.0.129, 00:08:48, GigabitEthernet0/1
C        192.168.0.32/27 is directly connected, GigabitEthernet0/2
L        192.168.0.35/32 is directly connected, GigabitEthernet0/2
O        192.168.0.96/27
           [110/2] via 192.168.0.36, 00:09:05, GigabitEthernet0/2
C        192.168.0.128/27 is directly connected, GigabitEthernet0/1
L        192.168.0.130/32 is directly connected, GigabitEthernet0/1
```

Listing H.67: Output of sh ip route for *r9*

```
r9#sh ip route
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS
           level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static
           route
       o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP
       a - application route
       + - replicated route, % - next hop override, p - overrides from
           PfR

Gateway of last resort is not set

      /8 is variably subnetted, 2 subnets, 2 masks
C        /16 is directly connected, GigE0/0
L        /32 is directly connected, GigE0/0
      /32 is subnetted, 7 subnets
O        Lo0.r1 [110/2] via GigE0/1.r1, 00:09:05, GigE0/2
O        Lo0.r3 [110/2] via GigE0/1.r3, 00:09:05, GigE0/2
O        Lo0.r6 [110/2] via GigE0/1.r6, 00:08:48, GigE0/1
O        Lo0.r7 [110/3] via GigE0/1.r6, 00:08:48, GigE0/1
C        Lo0.r9 is directly connected, Lo0
O        Lo0.r11 [110/3] via GigE0/1.r13, 00:09:05, GigE0/2
O        Lo0.r13 [110/2] via GigE0/1.r13, 00:09:05, GigE0/2
      /24 is variably subnetted, 6 subnets, 2 masks
O        /27
           [110/2] via GigE0/1.r6, 00:08:48, GigE0/1
C        /27 is directly connected, GigE0/2
L        GigE0/2.r9/32 is directly connected, GigE0/2
O        /27
           [110/2] via GigE0/1.r13, 00:09:05, GigE0/2
C        /27 is directly connected, GigE0/1
```

```
L        GigE0/1.r9/32 is directly connected, GigE0/1
```

Listing H.68: Output of sh ip route for *r9*, after post-processing

H.5 CODE FOR LARGE-SCALE CASE STUDY

H.5.1 *Network Whiteboard*

```
import autonetkit.load.graphml as graphml
with open("interconnect.graphml") as fh:
    data = fh.read()

input_graph = graphml.load_graphml(data)
nodes = {}
for n, data in input_graph.nodes(data=True):
    nodes[n] = {
        "id": n,
        "asn": data["asn"],
        "x": float(data.get("Longitude", 0)),
        "y": -1 * float(data.get("Latitude", 0)),
        "Network": data["Network"]
    }


edges = input_graph.edges()
```

Listing H.69: Load Source Graphml

```
import autonetkit
anm = autonetkit.NetworkModel()
g_in = anm.add_overlay("input")

index = 0
for n, d in nodes.items():
    g_in.add_node(n, x=d["x"], device_type="router",
                  network=d["Network"],
                  y=d["y"], asn=d["asn"])
```

Listing H.70: Add Nodes to Network Whiteboard

```
for src, dst in edges:
    src = src.replace(" ", "_")
    dst = dst.replace(" ", "_")
    if src in g_in and dst in g_in:
        g_in.add_edge(src, dst)
```

Listing H.71: Add Edges to Network Whiteboard

```
for n in sorted(g_in):
    if int(n.get("x")) == 0 or int(n.get("y")) == 0:
        neighs = n.neighbors()
        x = sum(neigh["input"].get("x") for neigh in neighs) / len(neighs
            )
        y = sum(neigh["input"].get("y") for neigh in neighs) / len(neighs
            )
        n.set("x", x)
        n.set("y", y)

# normalise x, y
overall_scale = 80
for n in g_in:
    n.set("x", overall_scale * n.get("x"))
    n.set("y", overall_scale * n.get("y"))
```

```
node_x = [n.get('x') for n in g_in]
node_y = [n.get('y') for n in g_in]
min_x = min(node_x)
max_x = max(node_x)
min_y = min(node_y)
max_y = max(node_y)

x_scale = 1 * (max_x - min_x)
y_scale = 1 * (max_y - min_y)
overall_scale = max(x_scale, y_scale)
for n in g_in:
    n.set("x", n.get("x") - min_x)
    n.set("y", n.get("y") - min_y)
```

Listing H.72: Normalise Locations

H.5.2 *Design Functions*

```
g_phy = anm['phy']
g_phy.add_nodes_from(g_in, retain=["asn", "device_type", "x", "y"])
g_phy.update(platform="netkit", syntax="quagga")

g_phy.add_edges_from(g_in.edges())
```

Listing H.73: Create Physical Network View

```
from autonetkit.ank import split
from collections import Counter

g_l2 = anm.add_overlay("layer2")
g_l2.add_nodes_from(g_phy)
g_l2.add_edges_from(g_phy.edges())

# Split the point-to-point edges to add a collision domain
edges_to_split = [edge for edge in g_l2.edges()
                  if edge.src.is_l3device() and edge.dst.is_l3device()]

for edge in edges_to_split:
    edge.split = True  # mark as split for use in building nidb

split_created_nodes = split(g_l2, edges_to_split, id_prepend='bd_')

for node in split_created_nodes:
    # set midway x, y for plot
    neighs = node.neighbors()
    x = sum(neigh["phy"].get("x") for neigh in neighs) / len(neighs)
    y = sum(neigh["phy"].get("y") for neigh in neighs) / len(neighs)
    node.set("x", x)
    node.set("y", y)

    c = Counter(n.get("asn") for n in neighs)
    most_common_asn, _ = c.most_common(1)[0]
    node.set("asn", most_common_asn)

    node.set("broadcast_domain", True)
    node.set("device_type", "broadcast_domain")
```

Listing H.74: Create Layer 2 Network View

```
from autonetkit.ank import explode_nodes

g_l2_conn = anm.add_overlay("layer2_conn")
g_l2_conn.add_nodes_from(g_l2)
g_l2_conn.add_edges_from(g_l2.edges())
bc_nodes = g_l2.nodes(broadcast_domain=True)
explode_nodes(g_l2_conn, bc_nodes)
```

Listing H.75: Create Layer 2 Connectivity Network View

```
from autonetkit.ank import split, groupby, copy_attr_from
from netaddr import IPNetwork

g_ip = anm.add_overlay("ip")
g_ip.add_nodes_from(g_l2)
g_ip.add_edges_from(g_l2.edges())

bc_nodes = g_l2.nodes(broadcast_domain=True)
```

```
bc_attrs = ["broadcast_domain", "device_type", "asn"]
for attr in bc_attrs:
    copy_attr_from(g_l2, g_ip, attr, nbunch=bc_nodes)


# allocate loopback IPs
block = IPNetwork("172.16.0.0/16")
subnets = block.subnet(24)

loopback_allocations = {}

l3_nodes = g_ip.l3devices()
for asn, nodes in groupby("asn", l3_nodes):
    asn_block = subnets.next()
    loopback_allocations[asn] = asn_block
    hosts = asn_block.iter_hosts()

    for node in nodes:
        ip = hosts.next()
        node.set("loopback", ip)

# allocate infra IPs
block = IPNetwork("10.0.0.0/8")
subnets = block.subnet(16)

infra_allocations = {}

bc_nodes = g_ip.nodes(broadcast_domain=True)
for asn, nodes in groupby("asn", bc_nodes):
    asn_block = subnets.next()
    infra_allocations[asn] = asn_block
    ptp_subnets = asn_block.subnet(30)
    for node in nodes:
        prefix = ptp_subnets.next()
        node.set("subnet", prefix)

        hosts = prefix.iter_hosts()
        for neigh_iface in node.neighbor_interfaces():
            address = hosts.next()
            neigh_iface.set("ip", address)
            neigh_iface.set("subnet", prefix)
```

Listing H.76: Allocate IP Addresses

```
g_ospf = anm.add_overlay("ospf")
g_ospf.add_nodes_from(g_in.routers())
g_ospf.add_edges_from(e for e in g_l2_conn.edges()
                      if e.src.asn == e.dst.asn)
```

Listing H.77: Create OSPF Network View

```
g_ibgp = anm.add_overlay("ibgp")
g_ibgp.add_nodes_from(g_in.routers())
```

Listing H.78: Create iBGP Network View

```
from autonetkit.ank import groupby, unwrap_graph
import networkx as nx

graph = unwrap_graph(g_ibgp)

for asn, nodes in groupby("asn", g_ibgp):
    nodes = list(nodes)
    subgraph = graph.subgraph(n.id for n in nodes)
    central = nx.betweenness_centrality(subgraph)
    sorted_central = sorted(central, key=lambda k: central[k])
    rr_size = min(int(len(nodes) / 3), 3)
    most_central = sorted_central[:rr_size]

    rrs = {g_ibgp.node(r) for r in most_central}
    for rr in rrs:
        rr.set("ibgp_role", "RR")

    clients = set(nodes) - rrs
    for client in clients:
        client.set("ibgp_role", "Client")
```

Listing H.79: Allocate iBGP Roles

```
from autonetkit.ank import groupby, unwrap_graph
import networkx as nx

for asn, nodes in groupby("asn", g_ibgp):
    nodes = list(nodes)

    # Form set of route reflectors
    rrs = set(n for n in nodes if n.get("ibgp_role") == "RR")
    # Form set of route reflector clients
    clients = set(n for n in nodes if n.get("ibgp_role") == "Client")
    # Form tuples of (route reflector, client)
    rr_to_c = [(r, c) for r in rrs for c in clients]
    edges = []

    for src, dst in rr_to_c:
        # Create iBGP session termination points
        src_endpoint = src.add_interface(category="bgp_session")
        dst_endpoint = dst.add_interface(category="bgp_session")
        # Specify direction for route-reflection peering
        src_endpoint.set("direction", "down")
        dst_endpoint.set("direction", "up")
        # Create session between termination points
        edges.append((src_endpoint, dst_endpoint))

    # Form tuples of (route reflector, route reflector)
    rr_to_rr = [(r1, r2) for r1 in rrs for r2 in rrs if r1 != r2]
    for src, dst in rr_to_rr:
        # Create iBGP session termination points
        src_endpoint = src.add_interface(category="bgp_session")
        dst_endpoint = dst.add_interface(category="bgp_session")
        # Specify direction for route-reflection peering
        src_endpoint.set("direction", "over")
        dst_endpoint.set("direction", "over")
        # Create session between termination points
        edges.append((src_endpoint, dst_endpoint))

    # Add all sessions
    g_ibgp.add_edges_from(edges)
```

Listing H.80: Create iBGP Connectivity

```
rrs = g_ibgp.nodes(ibgp_role="RR")
from autonetkit.ank_messaging import highlight
highlight(nodes=rrs)
```

Listing H.81: Highlight Route Reflectors in Visualisation

```
g_ebgp = anm.add_overlay("ebgp")
g_ebgp.add_nodes_from(g_in.routers())
edges = [e for e in g_l2_conn.edges()
         if e.src.asn != e.dst.asn]

for e in edges:
    # Obtain source and destination nodes
    src = g_ebgp.node(e.src)
    dst = g_ebgp.node(e.dst)
    # Create BGP session termination points
    src_endpoint = src.add_interface(category="bgp_session")
    dst_endpoint = dst.add_interface(category="bgp_session")
    # add session to g_ebgp
    g_ebgp.add_edge(src_endpoint, dst_endpoint)

ebgp_nodes = [n for n in g_ebgp if n.degree() > 0]
highlight(nodes=ebgp_nodes)
```

Listing H.82: Create eBGP Network View

```
from netaddr import IPSet

for node in g_ebgp:
    adv_prefixes = IPSet()
    if node.degree() == 0:
        continue

    asn = node.get("asn")
    # Get infrastructure address blocks for this AS
    ip_infra = infra_allocations.get(asn, [])
    # Store on the node
    node.set("networks", [str(ip_infra)])
```

Listing H.83: BGP Network Advertisements

```
from autonetkit.ank import unwrap_graph
import networkx as nx
from random import choice
from collections import Counter

graph = unwrap_graph(g_phy)
nodes = graph.nodes()


def ntwrk(p):
    return g_in.node(p).get("network")


def unique_networks(networks):
    networks = list(networks)
    c = Counter(networks)
    retval = []
    previous = None
    for n in networks:
        if n != previous:
            retval.append(n)
        previous = n

    return ["%s (%s)" % (n, c[n]) for n in retval]

paths = []

path_nodes = set()
```

```
for _ in range(3):
    src = choice(nodes)
    dst = choice(nodes)
    path = nx.shortest_path(graph, src, dst)
    print "%s to %s" % (src, dst)
    print " -> ".join(path)
    print " -> ".join(unique_networks(ntwrk(p) for p in path))
    print
    paths.append(path)
    path_nodes.add(src)
    path_nodes.add(dst)
    path_nodes.update(list(g_phy.node(p) for p in path))

print "p nodes", path_nodes

g_analysis = anm.add_overlay("analysis")
g_analysis.add_nodes_from(g_phy)
g_analysis.add_edges_from(g_phy.edges())
for n in g_analysis:
    if n in path_nodes:
        n.set("path_label", n.id)

from autonetkit.ank_messaging import highlight
highlight(paths=paths)
```

Listing H.84: Example of Paths

```
Telsiai to Castilla_Y_Leon
Telsiai -> Klaipeda -> Kaunas -> LT -> PL -> DE -> FR -> ES -> Nacional
    -> Castilla_Y_Leon
LITNET -> GEANT -> RedIris

Vaxjo to Breda
Vaxjo -> Jonkoping -> Linkoping -> Stockholm_2603 -> Copenhagen -> DK ->
    NL -> Amsterdam -> Delft -> Rotterdam -> Dordrecht -> Breda
SUNET -> NORDUnet -> GEANT -> SURFnet

Fe to Celje
Fe -> BO -> MI-1 -> IT -> AT -> Ljubljana -> Kamnik -> Celje
GARR -> GEANT -> ARNES
```

Listing H.85: Example analysis of paths in NREN1400 topology

H.5.3 *Netkit*

Netkit Platform Compiler

```
# Create Platform Compiler and Compile
sim_plat = simple_platform_compiler(nidb, anm, "localhost")
sim_plat.compile()
```

Listing H.86: Quagga Device Compiler

Quagga Device Compiler

```
from autonetkit.compilers.device import router_base


class simple_router_compiler(router_base.RouterCompiler):

    def compile(self, node):
        interfaces = self.interfaces(node)
        ospf = self.ospf(node)
        bgp = self.bgp(node)

        return {
            "interfaces": interfaces,
            "ospf": ospf,
            "bgp": bgp,
        }

    def interfaces(self, node):
        ifaces = []
        ip_node = self.anm['ip'].node(node)
        for interface in node.physical_interfaces():
            ip_int = ip_node.interface(interface)
            ip = ip_int.get("ip")
            netmask = ip_int.get("subnet").netmask
            ifaces.append({"ip": ip, "netmask": netmask})

        return ifaces

    def ospf(self, node):
        ospf_node = self.anm["ospf"].node(node)
        networks = ospf_node.get("networks") or []
        redistribute_connected = ospf_node.get("redistribute_connected")
            or []
        redistribute_bgp = ospf_node.get("redistribute_bgp") or []

        return {
            "networks": networks,
            "redistribute_connected": redistribute_connected,
            "redistribute_bgp": redistribute_bgp,
        }

    def bgp(self, node):
        ibgp_neighbors = []
        g_ibgp = self.anm["ibgp"]

        for session in g_ibgp.edges(node):
            dst = session.dst
            dst_int = session.dst_int
            dst_bound_int = dst_int.get("bound_to")
            neigh_ip = dst_bound_int["ip"].get("ip")
            desc = "Router %s" % dst
            data = {
```

```
            "neigh_ip": neigh_ip,
            "asn": dst.get("asn"),
            "desc": desc
        }
        ibgp_neighbors.append(data)

    ebgp_neighbors = []
    g_ebgp = self.anm["ebgp"]
    ebgp_node = g_ebgp.node(node)
    for session in g_ebgp.edges(node):
        dst = session.dst
        dst_int = session.dst_int
        dst_bound_int = dst_int.get("bound_to")
        neigh_ip = dst_bound_int["ip"].get("ip")
        desc = "Router %s" % dst
        data = {
            "neigh_ip": neigh_ip,
            "asn": dst.get("asn"),
            "desc": desc
        }
        ebgp_neighbors.append(data)

    networks = ebgp_node.get("networks") or []
    prefix_lists = ebgp_node.get("prefix_lists") or []
    route_maps = ebgp_node.get("route_maps") or []
    return {
        "ibgp_neighbors": ibgp_neighbors,
        "ebgp_neighbors": ebgp_neighbors,
        "networks": networks,
        "prefix_lists": prefix_lists,
        "route_maps": route_maps
    }
```

Listing H.87: Quagga Device Compiler

H.5.4 *C-BGP*

C-BGP Platform Compiler

```
class cbgp_compiler(object):

    def __init__(self, anm, nidb):
        self.anm = anm
        self.nidb = nidb

    def physical(self):
        # Physical connectivity
        data = {}
        g_phy = self.anm["phy"]
        for asn, nodes in groupby("asn", g_phy):
            # Do on a per-AS basis for visual clarity
            data[asn] = {"nodes": [], "links": []}
            nodes = list(nodes)
            for node in nodes:
                lo_ip = str(node["ip"].loopback_zero.get("ip"))
                data[asn]["nodes"].append(lo_ip)

            # use subgraph to form asn-only subgraph
            # note this also could be formed from iBGP graph
            asn_subgraph = g_phy.subgraph(nodes)
            for edge in asn_subgraph.edges():
                src = edge.src
                dst = edge.dst
                src_ip = src["ip"].loopback_zero.get("ip")
                dst_ip = dst["ip"].loopback_zero.get("ip")
                link = (src_ip, dst_ip)
                data[asn]["links"].append(link)

        return data

    def inter_domain(self):
        # Compiler inter-domain links
        data = []
        for edge in self.anm["ebgp"].edges():
            src = edge.src
            dst = edge.dst
            src_ip = src["ip"].loopback_zero.get("ip")
            dst_ip = dst["ip"].loopback_zero.get("ip")
            link = (src_ip, dst_ip)
            data.append(link)
        return data

    def igp(self):
        # Setup IGP links within an AN
        data = {}
        g_ospf = self.anm["ospf"]
        for asn, nodes in groupby("asn", g_ospf):
            # Do on a per-AS basis for visual clarity
            data[asn] = {"nodes": [], "links": []}
            nodes = list(nodes)
            for node in nodes:
                lo_ip = str(node["ip"].loopback_zero.get("ip"))
                data[asn]["nodes"].append(lo_ip)

            # use subgraph to form asn-only subgraph
            # note this also could be formed from iBGP graph
            asn_subgraph = g_phy.subgraph(nodes)
            for edge in asn_subgraph.edges():
                src = edge.src
                dst = edge.dst
                src_ip = src["ip"].loopback_zero.get("ip")
                dst_ip = dst["ip"].loopback_zero.get("ip")
                link = (src_ip, dst_ip)
                data[asn]["links"].append(link)

        return data

    def bgp(self):
        # Setup BGP routers and sessions
        g_phy = self.anm["phy"]
        g_ibgp = self.anm["ibgp"]
        g_ebgp = self.anm["ebgp"]

        routers = {}
        peers = {}

        for asn, nodes in groupby("asn", g_phy):
            nodes = list(nodes)
            routers[asn] = []
            peers[asn] = {}
            prefixes = {}
            for node in nodes:
                if node in g_ebgp:
                    lo_ip = str(node["ip"].loopback_zero.get("ip"))
                    routers[asn].append(lo_ip)
                if node["ebgp"].degree():
                    peers[asn][lo_ip] = {"ebgp": []}

                for session in node["ebgp"].edges():
                    # Sessions from this eBGP router
                    src_endpoint = session.src_int
                    direction = src_endpoint.get("direction")

                    peer_ip = session.dst["ip"].loopback_zero.get("ip")
                    peer = {
                        "domain": session.dst.get("asn"),
                        "next_hop_self": True,
                        "up": True,
                        "ip": peer_ip
                    }

                    peers[asn][lo_ip]["ebgp"].append(peer)

            for node in g_ebgp:
                # Networks to advertise for this eBGP router
                lo_ip = str(node["ip"].loopback_zero.get("ip"))
                networks = node.get("networks")
                if networks:
                    prefixes[lo_ip] = networks

        data = {
            "routers": routers,
            "peers": peers,
            "prefixes": prefixes
        }

        return data

    def compile(self):
        # Compile each component
        self.nidb["physical"] = self.physical()
        self.nidb["inter_domain"] = self.inter_domain()
        self.nidb["igp"] = self.igp()
        self.nidb["bgp"] = self.bgp()
```

Listing H.88: C-BGP Platform Compiler

C-BGP Template

```
{% for asn, asn_data in idm.physical.iteritems() %}
# "physical" topology for {{asn}}
    {% for node in asn_data.nodes %}
net add node {{node}}
    {% endfor %}
    {% for src, dst in asn_data.links %}
net add link {{src}} {{dst}}
    {% endfor %}

{% endfor %}

# Interdomain links
{% for src, dst in idm.inter_domain %}
net add link {{src}} {{dst}}
{% endfor %}

# Static routes for interdomain links
{% for src, dst in idm.inter_domain %}
net node {{src}} route add --oif={{dst}} {{dst}}/32 1
net node {{dst}} route add --oif={{src}} {{src}}/32 1
{% endfor %}

{% for asn, asn_data in idm.physical.iteritems() %}
# IGP topology for {{asn}}
net add domain {{asn}} igp
    {% for node in asn_data.nodes %}
net node {{node}} domain {{asn}}
    {% endfor %}
    {% for src, dst in asn_data.links %}
net link {{src}} {{dst}} igp-weight --bidir 1
    {% endfor %}
net domain {{asn}} compute

{% endfor %}

{% for asn, asn_data in idm.bgp.routers.iteritems() %}
# BGP routers in {{asn}}
    {% for node in asn_data %}
bgp add router {{asn}} {{node}}
    {% endfor %}
bgp domain {{asn}} full-mesh

{% endfor %}


{% for asn, asn_data in idm.bgp.peers.iteritems() %}
# BGP routers for {{asn}}
    {% for node, node_data in asn_data.iteritems() %}
bgp router {{node}}
    {% for peer in node_data.ebgp %}
    add peer {{peer.domain}} {{peer.ip}}
    {% if peer.next_hop_self %}
    peer {{peer.ip}} next-hop-self
    {% endif %}
    {% if peer.up %}
    peer {{peer.ip}} up
    {% endif %}
    {% endfor %}
    exit

    {% endfor %}

{% endfor %}

# Originate prefixes
{% for node, prefixes in idm.bgp.prefixes.iteritems() %}
    {% for prefix in prefixes %}
bgp router {{node}} add network {{prefix}}
    {% endfor %}
{% endfor %}

sim run
```

Listing H.89: C-BGP Template

Simulation Results

```
# "physical" topology for 3221
net add node 192.168.25.8
net add node 192.168.25.1
net add node 192.168.25.2
net add node 192.168.25.3
net add node 192.168.25.7
net add node 192.168.25.10
net add node 192.168.25.12
net add node 192.168.25.5
net add node 192.168.25.11
net add node 192.168.25.4
net add node 192.168.25.9
net add node 192.168.25.6
net add link 192.168.25.8 192.168.25.3
net add link 192.168.25.8 192.168.25.9
net add link 192.168.25.2 192.168.25.12
net add link 192.168.25.6 192.168.25.12
net add link 192.168.25.3 192.168.25.5
net add link 192.168.25.4 192.168.25.12
net add link 192.168.25.4 192.168.25.5
net add link 192.168.25.10 192.168.25.12
net add link 192.168.25.12 192.168.25.1
net add link 192.168.25.12 192.168.25.11
net add link 192.168.25.12 192.168.25.5
net add link 192.168.25.7 192.168.25.5


# Interdomain links
net add link 192.168.9.19 192.168.18.6
net add link 192.168.16.34 192.168.38.15
net add link 192.168.13.42 192.168.38.17
net add link 192.168.13.42 192.168.38.7
net add link 192.168.38.1 192.168.28.5
net add link 192.168.33.2 192.168.21.28
net add link 192.168.19.24 192.168.38.14
net add link 192.168.22.30 192.168.38.2
net add link 192.168.1.15 192.168.18.21
net add link 192.168.36.1 192.168.12.7
net add link 192.168.38.5 192.168.23.2
net add link 192.168.31.6 192.168.18.4

# Static routes for interdomain links
net node 192.168.9.19 route add --oif=192.168.18.6 192.168.18.6/32 1
net node 192.168.18.6 route add --oif=192.168.9.19 192.168.9.19/32 1
net node 192.168.16.34 route add --oif=192.168.38.15 192.168.38.15/32 1
net node 192.168.38.15 route add --oif=192.168.16.34 192.168.16.34/32 1
net node 192.168.13.42 route add --oif=192.168.38.17 192.168.38.17/32 1
net node 192.168.38.17 route add --oif=192.168.13.42 192.168.13.42/32 1
net node 192.168.13.42 route add --oif=192.168.38.7 192.168.38.7/32 1
net node 192.168.38.7 route add --oif=192.168.13.42 192.168.13.42/32 1
net node 192.168.38.1 route add --oif=192.168.28.5 192.168.28.5/32 1
net node 192.168.28.5 route add --oif=192.168.38.1 192.168.38.1/32 1
net node 192.168.33.2 route add --oif=192.168.21.28 192.168.21.28/32 1
net node 192.168.21.28 route add --oif=192.168.33.2 192.168.33.2/32 1
net node 192.168.19.24 route add --oif=192.168.38.14 192.168.38.14/32 1

# IGP topology for 21274
net add domain 21274 igp
net node 192.168.39.1 domain 21274
net node 192.168.39.4 domain 21274
net node 192.168.39.2 domain 21274
net node 192.168.39.6 domain 21274
net node 192.168.39.3 domain 21274
net node 192.168.39.5 domain 21274
net link 192.168.39.1 192.168.39.6 igp-weight --bidir 1
net link 192.168.39.4 192.168.39.6 igp-weight --bidir 1
net link 192.168.39.2 192.168.39.6 igp-weight --bidir 1
net link 192.168.39.5 192.168.39.6 igp-weight --bidir 1
net link 192.168.39.6 192.168.39.3 igp-weight --bidir 1
net domain 21274 compute


# BGP routers in 12687
bgp add router 12687 192.168.35.1
bgp add router 12687 192.168.35.8
bgp add router 12687 192.168.35.2
bgp add router 12687 192.168.35.7
bgp add router 12687 192.168.35.15
bgp add router 12687 192.168.35.17
bgp add router 12687 192.168.35.18
bgp add router 12687 192.168.35.9
bgp add router 12687 192.168.35.14
bgp add router 12687 192.168.35.3
bgp add router 12687 192.168.35.10
bgp add router 12687 192.168.35.4
bgp add router 12687 192.168.35.11
bgp add router 12687 192.168.35.5
bgp add router 12687 192.168.35.6
bgp add router 12687 192.168.35.13
bgp add router 12687 192.168.35.19
bgp add router 12687 192.168.35.12
bgp add router 12687 192.168.35.16
bgp domain 12687 full-mesh
bgp add router 2108 192.168.15.7


# BGP routers for 137
bgp router 192.168.0.42
    add peer 20965 192.168.38.10
    peer 192.168.38.10 next-hop-self
    peer 192.168.38.10 up
    exit

bgp router 192.168.0.40
    add peer 20965 192.168.38.10
    peer 192.168.38.10 next-hop-self
    peer 192.168.38.10 up
    exit

bgp router 192.168.32.8 add network 192.168.32.0/24
bgp router 192.168.38.11 add network 10.35.0.0/16
bgp router 192.168.38.11 add network 192.168.38.0/24
bgp router 192.168.38.12 add network 10.35.0.0/16
bgp router 192.168.38.12 add network 192.168.38.0/24
bgp router 192.168.16.34 add network 10.15.0.0/16
bgp router 192.168.16.34 add network 192.168.16.0/24
bgp router 192.168.18.4 add network 10.17.0.0/16
bgp router 192.168.18.4 add network 192.168.18.0/24
bgp router 192.168.0.42 add network 10.0.0.0/16
bgp router 192.168.0.42 add network 192.168.0.0/24
bgp router 192.168.18.6 add network 10.17.0.0/16
bgp router 192.168.18.6 add network 192.168.18.0/24

sim run
```

Listing H.90: Extract of C-BGP configuration file

```
cbgp> bgp router 192.168.6.17 show rib *
*> 10.0.0.0/16  192.168.38.18   0   4294967295  20965 137   i
*> 10.1.0.0/16  192.168.38.18   0   4294967295  20965 2603 224  i
*> 10.2.0.0/16  192.168.38.18   0   4294967295  20965 378   i
*> 10.3.0.0/16  192.168.38.18   0   4294967295  20965 559   i
*> 10.4.0.0/16  192.168.38.18   0   4294967295  20965 680   i
*> 10.5.0.0/16  192.168.38.18   0   4294967295  20965 766   i
i> 10.6.0.0/16  192.168.6.17    0   0   null    i
*> 10.7.0.0/16  192.168.38.18   0   4294967295  20965 1103  i
*> 10.8.0.0/16  192.168.38.18   0   4294967295  20965 1213  i
*> 10.9.0.0/16  192.168.38.18   0   4294967295  20965 2603 1835 i
*> 10.10.0.0/16 192.168.38.18   0   4294967295  20965 1853  i
*> 10.11.0.0/16 192.168.38.18   0   4294967295  20965 1930  i
*> 10.12.0.0/16 192.168.38.18   0   4294967295  20965 1955  i
*> 10.13.0.0/16 192.168.38.18   0   4294967295  20965 1967  i
*> 10.14.0.0/16 192.168.38.18   0   4294967295  20965 2107  i
*> 10.15.0.0/16 192.168.38.18   0   4294967295  20965 2108  i
*> 10.16.0.0/16 192.168.38.18   0   4294967295  20965 2200  i
*> 10.17.0.0/16 192.168.38.18   0   4294967295  20965 2602  i
*> 10.18.0.0/16 192.168.38.18   0   4294967295  20965 2603  i
*> 10.19.0.0/16 192.168.38.18   0   4294967295  20965 2607  i
*> 10.20.0.0/16 192.168.38.18   0   4294967295  20965 2611  i
*> 10.21.0.0/16 192.168.38.18   0   4294967295  20965 2614  i
*> 10.22.0.0/16 192.168.38.18   0   4294967295  20965 2847  i
*> 10.23.0.0/16 192.168.38.18   0   4294967295  20965 2852  i
*> 10.24.0.0/16 192.168.38.18   0   4294967295  20965 3058  i
*> 10.25.0.0/16 192.168.38.18   0   4294967295  20965 3221  i
*> 10.26.0.0/16 192.168.38.18   0   4294967295  20965 3268  i
*> 10.27.0.0/16 192.168.38.18   0   4294967295  20965 5379  i
*> 10.28.0.0/16 192.168.38.18   0   4294967295  20965 5408  i
*> 10.29.0.0/16 192.168.38.18   0   4294967295  20965 5538  i
*> 10.30.0.0/16 192.168.38.18   0   4294967295  20965 6802  i
*> 10.31.0.0/16 192.168.38.18   0   4294967295  20965 2603 8624 i
*> 10.32.0.0/16 192.168.38.18   0   4294967295  20965 9112  i
*> 10.33.0.0/16 192.168.38.18   0   4294967295  20965 2614 9199 i
*> 10.34.0.0/16 192.168.38.18   0   4294967295  20965 12687 i
*> 10.35.0.0/16 192.168.38.18   0   4294967295  20965 1955 13092    i
*> 10.36.0.0/16 192.168.38.18   0   4294967295  20965 2603 15474    i
*> 10.37.0.0/16 192.168.38.18   0   4294967295  20965   i
*> 10.38.0.0/16 192.168.38.18   0   4294967295  20965 21274 i
*> 192.168.0.0/24   192.168.38.18   0   4294967295  20965 137   i
*> 192.168.1.0/24   192.168.38.18   0   4294967295  20965 2603 224  i
*> 192.168.2.0/24   192.168.38.18   0   4294967295  20965 378   i
*> 192.168.3.0/24   192.168.38.18   0   4294967295  20965 559   i
*> 192.168.4.0/24   192.168.38.18   0   4294967295  20965 680   i
*> 192.168.5.0/24   192.168.38.18   0   4294967295  20965 766   i
i> 192.168.6.0/24   192.168.6.17    0   0   null    i
*> 192.168.7.0/24   192.168.38.18   0   4294967295  20965 1103  i
*> 192.168.8.0/24   192.168.38.18   0   4294967295  20965 1213  i
*> 192.168.9.0/24   192.168.38.18   0   4294967295  20965 2603 1835 i
*> 192.168.10.0/24  192.168.38.18   0   4294967295  20965 1853  i
*> 192.168.11.0/24  192.168.38.18   0   4294967295  20965 1930  i
*> 192.168.12.0/24  192.168.38.18   0   4294967295  20965 1955  i
*> 192.168.13.0/24  192.168.38.18   0   4294967295  20965 1967  i
*> 192.168.14.0/24  192.168.38.18   0   4294967295  20965 2107  i
*> 192.168.15.0/24  192.168.38.18   0   4294967295  20965 2108  i
*> 192.168.16.0/24  192.168.38.18   0   4294967295  20965 2200  i
*> 192.168.17.0/24  192.168.38.18   0   4294967295  20965 2602  i
*> 192.168.18.0/24  192.168.38.18   0   4294967295  20965 2603  i
*> 192.168.19.0/24  192.168.38.18   0   4294967295  20965 2607  i
*> 192.168.20.0/24  192.168.38.18   0   4294967295  20965 2611  i
*> 192.168.21.0/24  192.168.38.18   0   4294967295  20965 2614  i
*> 192.168.22.0/24  192.168.38.18   0   4294967295  20965 2847  i
*> 192.168.23.0/24  192.168.38.18   0   4294967295  20965 2852  i
*> 192.168.24.0/24  192.168.38.18   0   4294967295  20965 3058  i
*> 192.168.25.0/24  192.168.38.18   0   4294967295  20965 3221  i
*> 192.168.26.0/24  192.168.38.18   0   4294967295  20965 3268  i
*> 192.168.27.0/24  192.168.38.18   0   4294967295  20965 5379  i
*> 192.168.28.0/24  192.168.38.18   0   4294967295  20965 5408  i
*> 192.168.29.0/24  192.168.38.18   0   4294967295  20965 5538  i
*> 192.168.30.0/24  192.168.38.18   0   4294967295  20965 6802  i
*> 192.168.31.0/24  192.168.38.18   0   4294967295  20965 2603 8624 i
*> 192.168.32.0/24  192.168.38.18   0   4294967295  20965 9112  i
*> 192.168.33.0/24  192.168.38.18   0   4294967295  20965 2614 9199 i
*> 192.168.34.0/24  192.168.38.18   0   4294967295  20965 12046 i
*> 192.168.35.0/24  192.168.38.18   0   4294967295  20965 12687 i
*> 192.168.36.0/24  192.168.38.18   0   4294967295  20965 1955 13092    i
*> 192.168.37.0/24  192.168.38.18   0   4294967295  20965 2603 15474    i
*> 192.168.38.0/24  192.168.38.18   0   4294967295  20965   i
*> 192.168.39.0/24  192.168.38.18   0   4294967295  20965 21274 i
```

Listing H.91: C-BGP bgp show rib result for London node

```
cbgp> net node 192.168.6.17 show rt *
10.0.0.0/16 192.168.38.18   --- 0   BGP
10.1.0.0/16 192.168.38.18   --- 0   BGP
10.2.0.0/16 192.168.38.18   --- 0   BGP
10.3.0.0/16 192.168.38.18   --- 0   BGP
10.4.0.0/16 192.168.38.18   --- 0   BGP
10.5.0.0/16 192.168.38.18   --- 0   BGP
10.7.0.0/16 192.168.38.18   --- 0   BGP
10.8.0.0/16 192.168.38.18   --- 0   BGP
10.9.0.0/16 192.168.38.18   --- 0   BGP
10.10.0.0/16    192.168.38.18   --- 0   BGP
10.11.0.0/16    192.168.38.18   --- 0   BGP
10.12.0.0/16    192.168.38.18   --- 0   BGP
10.13.0.0/16    192.168.38.18   --- 0   BGP
10.14.0.0/16    192.168.38.18   --- 0   BGP
10.15.0.0/16    192.168.38.18   --- 0   BGP
10.16.0.0/16    192.168.38.18   --- 0   BGP
10.17.0.0/16    192.168.38.18   --- 0   BGP
10.18.0.0/16    192.168.38.18   --- 0   BGP
10.19.0.0/16    192.168.38.18   --- 0   BGP
10.20.0.0/16    192.168.38.18   --- 0   BGP
10.21.0.0/16    192.168.38.18   --- 0   BGP
10.22.0.0/16    192.168.38.18   --- 0   BGP
10.23.0.0/16    192.168.38.18   --- 0   BGP
10.24.0.0/16    192.168.38.18   --- 0   BGP
10.25.0.0/16    192.168.38.18   --- 0   BGP
10.26.0.0/16    192.168.38.18   --- 0   BGP
10.27.0.0/16    192.168.38.18   --- 0   BGP
10.28.0.0/16    192.168.38.18   --- 0   BGP
10.29.0.0/16    192.168.38.18   --- 0   BGP
```

```
10.30.0.0/16    192.168.38.18   --- 0   BGP
10.31.0.0/16    192.168.38.18   --- 0   BGP
10.32.0.0/16    192.168.38.18   --- 0   BGP
10.33.0.0/16    192.168.38.18   --- 0   BGP
10.34.0.0/16    192.168.38.18   --- 0   BGP
10.35.0.0/16    192.168.38.18   --- 0   BGP
10.36.0.0/16    192.168.38.18   --- 0   BGP
10.37.0.0/16    192.168.38.18   --- 0   BGP
10.38.0.0/16    192.168.38.18   --- 0   BGP
192.168.0.0/24  192.168.38.18   --- 0   BGP
192.168.1.0/24  192.168.38.18   --- 0   BGP
192.168.2.0/24  192.168.38.18   --- 0   BGP
192.168.3.0/24  192.168.38.18   --- 0   BGP
192.168.4.0/24  192.168.38.18   --- 0   BGP
192.168.5.0/24  192.168.38.18   --- 0   BGP
192.168.6.1/32  0.0.0.0 192.168.6.10    2   IGP
192.168.6.2/32  0.0.0.0 192.168.6.10    2   IGP
192.168.6.3/32  0.0.0.0 192.168.6.3 1   IGP
192.168.6.4/32  0.0.0.0 192.168.6.10    2   IGP
192.168.6.5/32  0.0.0.0 192.168.6.5 1   IGP
192.168.6.6/32  0.0.0.0 192.168.6.10    3   IGP
192.168.6.7/32  0.0.0.0 192.168.6.10    3   IGP
192.168.6.8/32  0.0.0.0 192.168.6.3 2   IGP
192.168.6.9/32  0.0.0.0 192.168.6.9 1   IGP
192.168.6.10/32 0.0.0.0 192.168.6.10    1   IGP
192.168.6.11/32 0.0.0.0 192.168.6.11    1   IGP
192.168.6.12/32 0.0.0.0 192.168.6.14    2   IGP
192.168.6.13/32 0.0.0.0 192.168.6.10    2   IGP
192.168.6.14/32 0.0.0.0 192.168.6.14    1   IGP
192.168.6.15/32 0.0.0.0 192.168.6.10    2   IGP
192.168.6.16/32 0.0.0.0 192.168.6.16    1   IGP
192.168.6.18/32 0.0.0.0 192.168.6.10    3   IGP
192.168.6.19/32 0.0.0.0 192.168.6.10    3   IGP
192.168.6.20/32 0.0.0.0 192.168.6.9 2   IGP
                0.0.0.0 192.168.6.16    2   IGP
192.168.6.21/32 0.0.0.0 192.168.6.21    1   IGP
192.168.6.22/32 0.0.0.0 192.168.6.9 3   IGP
                0.0.0.0 192.168.6.16    3   IGP
192.168.6.23/32 0.0.0.0 192.168.6.10    2   IGP
192.168.6.24/32 0.0.0.0 192.168.6.10    2   IGP
192.168.6.25/32 0.0.0.0 192.168.6.10    3   IGP
192.168.6.26/32 0.0.0.0 192.168.6.9 2   IGP
192.168.6.27/32 0.0.0.0 192.168.6.10    4   IGP
192.168.6.28/32 0.0.0.0 192.168.6.10    4   IGP
192.168.6.29/32 0.0.0.0 192.168.6.10    3   IGP
192.168.7.0/24  192.168.38.18   --- 0   BGP
192.168.8.0/24  192.168.38.18   --- 0   BGP
192.168.9.0/24  192.168.38.18   --- 0   BGP
192.168.10.0/24 192.168.38.18   --- 0   BGP
192.168.11.0/24 192.168.38.18   --- 0   BGP
192.168.12.0/24 192.168.38.18   --- 0   BGP
192.168.13.0/24 192.168.38.18   --- 0   BGP
192.168.14.0/24 192.168.38.18   --- 0   BGP
192.168.15.0/24 192.168.38.18   --- 0   BGP
192.168.16.0/24 192.168.38.18   --- 0   BGP
192.168.17.0/24 192.168.38.18   --- 0   BGP
192.168.18.0/24 192.168.38.18   --- 0   BGP
192.168.19.0/24 192.168.38.18   --- 0   BGP
192.168.20.0/24 192.168.38.18   --- 0   BGP
192.168.21.0/24 192.168.38.18   --- 0   BGP
192.168.22.0/24 192.168.38.18   --- 0   BGP
192.168.23.0/24 192.168.38.18   --- 0   BGP
192.168.24.0/24 192.168.38.18   --- 0   BGP
192.168.25.0/24 192.168.38.18   --- 0   BGP
192.168.26.0/24 192.168.38.18   --- 0   BGP
```

```
192.168.27.0/24 192.168.38.18   --- 0   BGP
192.168.28.0/24 192.168.38.18   --- 0   BGP
192.168.29.0/24 192.168.38.18   --- 0   BGP
192.168.30.0/24 192.168.38.18   --- 0   BGP
192.168.31.0/24 192.168.38.18   --- 0   BGP
192.168.32.0/24 192.168.38.18   --- 0   BGP
192.168.33.0/24 192.168.38.18   --- 0   BGP
192.168.34.0/24 192.168.38.18   --- 0   BGP
192.168.35.0/24 192.168.38.18   --- 0   BGP
192.168.36.0/24 192.168.38.18   --- 0   BGP
192.168.37.0/24 192.168.38.18   --- 0   BGP
192.168.38.18/32    0.0.0.0 192.168.38.18   1   STATIC
192.168.38.0/24 192.168.38.18   --- 0   BGP
192.168.39.0/24 192.168.38.18   --- 0   BGP
```

Listing H.92: C-BGP NODE SHOW RT result for London node

```
cbgp> bgp router 192.168.38.9 show rib *
*> 10.0.0.0/16  192.168.38.10   0   4294967295  137 i
*> 10.1.0.0/16  192.168.38.8    0   4294967295  2603 224    i
*> 10.2.0.0/16  192.168.2.8 0   4294967295  378 i
*> 10.3.0.0/16  192.168.38.3    0   4294967295  559 i
*> 10.4.0.0/16  192.168.4.10    0   4294967295  680 i
*> 10.5.0.0/16  192.168.38.12   0   4294967295  766 i
*> 10.6.0.0/16  192.168.38.18   0   4294967295  786 i
*> 10.7.0.0/16  192.168.38.11   0   4294967295  1103    i
*> 10.8.0.0/16  192.168.38.18   0   4294967295  1213    i
*> 10.9.0.0/16  192.168.38.8    0   4294967295  2603 1835   i
*> 10.10.0.0/16 192.168.38.14   0   4294967295  1853    i
*> 10.11.0.0/16 192.168.38.18   0   4294967295  1930    i
*> 10.12.0.0/16 192.168.38.4    0   4294967295  1955    i
*> 10.13.0.0/16 192.168.38.7    0   4294967295  1967    i
*> 10.14.0.0/16 192.168.38.14   0   4294967295  2107    i
*> 10.15.0.0/16 192.168.38.4    0   4294967295  2108    i
*> 10.16.0.0/16 192.168.38.15   0   4294967295  2200    i
*> 10.17.0.0/16 192.168.17.3    0   4294967295  2602    i
*> 10.18.0.0/16 192.168.38.8    0   4294967295  2603    i
*> 10.19.0.0/16 192.168.38.14   0   4294967295  2607    i
*> 10.20.0.0/16 192.168.38.11   0   4294967295  2611    i
*> 10.21.0.0/16 192.168.38.7    0   4294967295  2614    i
*> 10.22.0.0/16 192.168.38.2    0   4294967295  2847    i
*> 10.23.0.0/16 192.168.38.5    0   4294967295  2852    i
*> 10.24.0.0/16 192.168.24.1    0   4294967295  3058    i
*> 10.25.0.0/16 192.168.38.13   0   4294967295  3221    i
*> 10.26.0.0/16 192.168.38.1    0   4294967295  3268    i
*> 10.27.0.0/16 192.168.38.17   0   4294967295  5379    i
*> 10.28.0.0/16 192.168.38.1    0   4294967295  5408    i
*> 10.29.0.0/16 192.168.38.2    0   4294967295  5538    i
*> 10.30.0.0/16 192.168.38.17   0   4294967295  6802    i
*> 10.31.0.0/16 192.168.38.8    0   4294967295  2603 8624   i
*> 10.32.0.0/16 192.168.38.16   0   4294967295  9112    i
*> 10.33.0.0/16 192.168.38.7    0   4294967295  2614 9199   i
*> 10.34.0.0/16 192.168.38.16   0   4294967295  12687   i
*> 10.35.0.0/16 192.168.38.4    0   4294967295  1955 13092  i
*> 10.36.0.0/16 192.168.38.8    0   4294967295  2603 15474  i
i> 10.37.0.0/16 192.168.38.9    0   0   null    i
*> 10.38.0.0/16 192.168.38.16   0   4294967295  21274   i
*> 192.168.0.0/24   192.168.38.10   0   4294967295  137 i
*> 192.168.1.0/24   192.168.38.8    0   4294967295  2603 224    i
*> 192.168.2.0/24   192.168.2.8 0   4294967295  378 i
*> 192.168.3.0/24   192.168.38.3    0   4294967295  559 i
*> 192.168.4.0/24   192.168.4.10    0   4294967295  680 i
*> 192.168.5.0/24   192.168.38.12   0   4294967295  766 i
*> 192.168.6.0/24   192.168.38.18   0   4294967295  786 i
*> 192.168.7.0/24   192.168.38.11   0   4294967295  1103    i
*> 192.168.8.0/24   192.168.38.18   0   4294967295  1213    i
*> 192.168.9.0/24   192.168.38.8    0   4294967295  2603 1835   i
*> 192.168.10.0/24  192.168.38.14   0   4294967295  1853    i
*> 192.168.11.0/24  192.168.38.18   0   4294967295  1930    i
*> 192.168.12.0/24  192.168.38.4    0   4294967295  1955    i
*> 192.168.13.0/24  192.168.38.7    0   4294967295  1967    i
*> 192.168.14.0/24  192.168.38.14   0   4294967295  2107    i
*> 192.168.15.0/24  192.168.38.4    0   4294967295  2108    i
*> 192.168.16.0/24  192.168.38.15   0   4294967295  2200    i
*> 192.168.17.0/24  192.168.17.3    0   4294967295  2602    i
*> 192.168.18.0/24  192.168.38.8    0   4294967295  2603    i
*> 192.168.19.0/24  192.168.38.14   0   4294967295  2607    i
*> 192.168.20.0/24  192.168.38.11   0   4294967295  2611    i
*> 192.168.21.0/24  192.168.38.7    0   4294967295  2614    i
*> 192.168.22.0/24  192.168.38.2    0   4294967295  2847    i
*> 192.168.23.0/24  192.168.38.5    0   4294967295  2852    i
*> 192.168.24.0/24  192.168.24.1    0   4294967295  3058    i
*> 192.168.25.0/24  192.168.38.13   0   4294967295  3221    i
*> 192.168.26.0/24  192.168.38.1    0   4294967295  3268    i
*> 192.168.27.0/24  192.168.38.17   0   4294967295  5379    i
*> 192.168.28.0/24  192.168.38.1    0   4294967295  5408    i
*> 192.168.29.0/24  192.168.38.2    0   4294967295  5538    i
*> 192.168.30.0/24  192.168.38.17   0   4294967295  6802    i
*> 192.168.31.0/24  192.168.38.8    0   4294967295  2603 8624   i
*> 192.168.32.0/24  192.168.38.16   0   4294967295  9112    i
*> 192.168.33.0/24  192.168.38.7    0   4294967295  2614 9199   i
*> 192.168.34.0/24  192.168.38.10   0   4294967295  12046   i
*> 192.168.35.0/24  192.168.38.16   0   4294967295  12687   i
*> 192.168.36.0/24  192.168.38.4    0   4294967295  1955 13092  i
*> 192.168.37.0/24  192.168.38.8    0   4294967295  2603 15474  i
i> 192.168.38.0/24  192.168.38.9    0   0   null    i
*> 192.168.39.0/24  192.168.38.16   0   4294967295  21274   i
```

Listing H.93: C-BGP BGP SHOW RIB result for DE.GEANT node

I

CODE FOR SPECULATIVE FUTURE WORK

```
import autonetkit.ank as ank_utils


def validate(anm):
    tests_passed = True
    tests_passed = validate_ipv4(anm) and tests_passed

    validate_ibgp(anm)
    validate_igp(anm)
    check_for_selfloops(anm)
    all_nodes_have_asn(anm)

    return tests_passed


def check_for_selfloops(anm):
    # checks each overlay for selfloops
    for overlay in anm:
        selfloop_count = overlay._graph.number_of_selfloops()
        if selfloop_count > 0:
            print "%s has %s self-loops" % (overlay, selfloop_count)


def all_nodes_have_asn(anm):
    g_phy = anm['phy']
    for node in g_phy.l3devices():
        if node.asn is None:
            print "No ASN set for physical device %s" % node


def validate_ibgp(anm):
    import networkx as nx
    if not anm.has_overlay("ibgp"):
        return  # no ibgp v4  - eg if ip addressing disabled

    g_ibgp_v4 = anm['ibgp']

    for asn, devices in ank_utils.groupby("asn", g_ibgp_v4):
        asn_subgraph = g_ibgp_v4.subgraph(devices)
        graph = asn_subgraph._graph
        # get subgraph
        if not nx.is_strongly_connected(graph):
            return False
        else:
            return True


def validate_igp(anm):
    import networkx as nx
    if not anm.has_overlay("igp"):
        return  # no ibgp

    g_igp = anm['ospf']

    for asn, devices in ank_utils.groupby("asn", g_igp):
        if asn is None:
            continue
        asn_subgraph = g_igp.subgraph(devices)
        graph = asn_subgraph._graph
        if not nx.is_connected(graph):
            return False
        else:
            return True


def all_same(items):
    # based on http://stackoverflow.com/q/3787908
    return all(x == items[0] for x in items)


def all_unique(items):
    # based on http://stackoverflow.com/q/3787908
    seen = set()
    return not any(i in seen or seen.add(i) for i in items)


def duplicate_items(items):
    unique = set(items)
    counts = {i: items.count(i) for i in unique}
    return [i for i in counts if counts[i] > 1]

def validate_ipv4(anm):
    # TODO: make this generic to also handle IPv6
    if not anm.has_overlay("ip"):
        return False

    g_ip = anm['ip']
    tests_passed = True

    # check globally unique ip addresses
    all_ints = [i for n in g_ip.l3devices()
                for i in n.physical_interfaces()
                if i.is_bound]  # don't include unbound interfaces
    all_int_ips = [i.ip_address for i in all_ints if i.ip_address]

    if all_unique(all_int_ips):
        pass
    else:
        tests_passed = False
        duplicates = duplicate_items(all_int_ips)
        duplicate_ips = set(duplicate_items(all_int_ips))
        duplicate_ints = [n for n in all_ints
                          if n.ip_address in duplicate_ips]
        duplicates = ", ".join("%s: %s" % (i.node, i.ip_address)
                               for i in duplicate_ints)

    for bc in g_ip.nodes("broadcast_domain"):
        if not bc.allocate:
            return False
            continue

        neigh_ints = list(bc.neighbor_interfaces())
        neigh_ints = [i for i in neigh_ints if i.node.is_l3device()]
        neigh_int_subnets = [i.get("subnet") for i in neigh_ints]
        if all_same(neigh_int_subnets):
```

```
            pass
        else:
            subnets = ", ".join("%s: %s" % (i.node, i.subnets)
                                for i in neigh_int_subnets)
            tests_passed = False
            print "Different subnets on %s. %s" % (bc, subnets)

        ip_subnet_mismatches = [i for i in neigh_ints
                                if i.ip_address not in i.subnet]
        if len(ip_subnet_mismatches):
            tests_passed = False
            mismatches = ", ".join("%s not in %s on %s" %
                                   (i.ip_address, i.subnet, i.node)
                                   for i in ip_subnet_mismatches)
            print "Mismatched IP subnets: %s" % mismatches
        else:
            print "All subnets match"

        neigh_int_ips = [i.ip_address for i in neigh_ints]
        if all_unique(neigh_int_ips):
            print "All interface IP addresses are unique"
            duplicates = duplicate_items(neigh_int_ips)
        else:
            tests_passed = False
            duplicate_ips = set(duplicate_items(neigh_int_ips))
            duplicate_ints = [n for n in neigh_ints
                              if n.ip_address in duplicate_ips]
            duplicates = ", ".join("%s: %s" % (i.node, i.ip_address)
                                   for i in duplicate_ints)
            print "Duplicate IP addresses: %s" % duplicates

    if tests_passed:
        print "All IP tests passed."
    else:
        print "Some IP tests failed."

    return tests_passed
```

Listing I.1: Source code for validation example

BIBLIOGRAPHY

[1] M. Agrawal, S. R. Bailey, A. Greenberg, J. Pastor, P. Sebos, S. Seshan, K. van der Merwe, and J. Yates, “Routerfarm: towards a dynamic, manageable network edge,” Sep. 2006. DOI: 10.1145/1162638.1162639.

[2] C Alaettinoglu, C Villamizar, E Gerich, D Kessens, D Meyer, T Bates, D Karrenberg, and M Terpstra, “Routing policy specification language (rpsl),” *Internet Engineering Task Force, RFC 2280*, Jun. 1999. DOI: 10.17487/RFC2280.

[3] Ansible, *Ansible is simple it automation*, https://www.ansible.com, Accessed 2017-01-13. [Online]. Available: https://www.ansible.com.

[4] J. Arnoldus, M. van den Brand, A Serebrenik, and J. J. Brunekreef, *Code Generation with Templates*, ser. Atlantis Studies in Computing. Paris: Springer Science & Business Media, May 2012, vol. 1, ISBN: 9491216562. DOI: 10.2991/978-94-91216-56-5.

[5] AutoNetkit Documentation, *Junosphere quickstart guide — autonetkit*, Accessed 2017-01-13. [Online]. Available: https://pythonhosted.org/AutoNetkit/quickstart.html.

[6] E. Banks, *Why network engineers are sick of sdn – and what vendors can do about it*, Nov. 2012. [Online]. Available: http://packetpushers.net/why-network-engineers-are-sick-of-sdn-and-what-vendors-can-do-about-it.

[7] G. D. Battista, M. Rimondini, and G. Sadolfo, “Monitoring the status of mpls vpn and vpls based on bgp signaling information,” *IEEE/IFIP Network Operations and Management Symposium (NOMS) 2012*, pp. 237–244, 2012, ISSN: 1542-1201. DOI: 10.1109/NOMS.2012.6211904.

[8] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, “Don’t mind the gap,” *SIGCOMM ’16*, 2016. DOI: 10.1145/2934872.2934909.

[9] S Bellovin and R Bush, “Configuration management and security,” *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 3, pp. 268–274, 2009. DOI: 10.1109/JSAC.2009.090403.

[10] H Boehm, A. Feldmann, O. Maennel, C. Reiser, and R. Volk, “Design and realization of an as-wide inter-domain routing policy,” in *NANOG 34*, Seattle, Washington, 2005.

[11] M. Bostock, V. Ogievetsky, and J. Heer, “D3: data-driven documents,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2301–2309, 2011. DOI: 10.1109/TVCG.2011.185.

[12] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall, “Graphml progress report structural layer proposal,” in *Graph Drawing*, Springer, Sep. 2001, pp. 501–512, ISBN: 978-3-540-43309-5. DOI: 10.1007/3-540-45848-4_59.

[13] D. Buchmann, D. Jungo, and U. Ultes-Nitsche, “A role model to cope with the complexity of network configuration,” *International Network Optimization Conference 2007*, Apr. 2007.

[14] A. Burke, *Cisco virl : autonetkit*, Accessed 2017-01-13, Jun. 2014. [Online]. Available: https://networkinferno.net/cisco-virl-autonetkit.

[15] D Caldwell, S Lee, S Sen, and J Yates, “Gold standard auditing for router configurations,” *Local and Metropolitan Area Networks (LANMAN), 2010 17th IEEE Workshop on*, 2010. DOI: 10.1109/LANMAN.2010.5507163.

[16] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford, “The cutting edge of ip router configuration,” vol. 34, no. 1, p. 21, 2004. DOI: 10.1145/972374.972379.

[17] Cesnet.cz, *Netopeer - remote configuration system using netconf protocol*, Accessed 2017-01-13. [Online]. Available: https://code.google.com/p/netopeer/.

[18] B. Chapman, “Automating network configuration,” *NANOG 49*, Jun. 2010. [Online]. Available: http://www.netomata.com/tools/ncg.

[19] X Chen, Z. Mao, and J Van der Merwe, “Shadownet: a platform for rapid and safe network evolution,” *Proceedings of the 2009 conference on USENIX Annual technical conference*, pp. 3–3, 2009.

[20] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe, “Decor: Declaritive network management and operation,” PRESTO ’09, pp. 67–72, 2009. DOI: 10.1145/1592631.1592647. [Online]. Available: http://doi.acm.org/10.1145/1592631.1592647.

[21] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe, “Declarative configuration management for complex and dynamic networks,” Nov. 2010. DOI: 10.1145/1921168.1921176. [Online]. Available: http://dl.acm.org/citation.cfm?id=1921176.

[22] X. Chen, Z. M. Mao, and J. Van der Merwe, “Pacman: a platform for automated and controlled network operations and configuration management,” 2009. DOI: 10.1145/1658939.1658971.

[23] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, “Planetlab: an overlay testbed for broad-coverage services,” *ACM SIGCOMM Computer Communication Review*, pp. 3–12, Jul. 2003. DOI: 10.1145/956993.956995.

[24] *Create network models with cml’s autonetkit*, Accessed 2017-01-13. [Online]. Available: http://blog.ipspace.net/2013/10/create-network-models-with-cmls.html.

[25] G Di Battista, M Patrignani, M Pizzonia, F Ricci, and M Rimondini, “Netkit-lab bgp: small-internet,” Tech. Rep. [Online]. Available: http://wiki.netkit.org/index.php/Labs_Official.

[26] J. Duerig, R. Ricci, J. Byers, and J. Lepreau, “Automatic ip address assignment on network topologies,” Tech. Rep. Flux Technical Note FTN–2006–02, Feb. 2006. [Online]. Available: https://www.cs.utah.edu/flux/papers/ipassign-ftn2006-02.pdf.

[27] J. Edelman, *Automated network diagrams with schprokits & autonetkit*. [Online]. Available: http://jedelman.com/home/automated-network-diagrams-with-schprokits-autonetkit/.

[28] B. Edgeworth, A. Foss, and R. G. Rios, *IP Routing on Cisco IOS, IOS XE, and IOS XR*, ser. An Essential Guide to Understanding and Implementing IP Routing Protocols. Pearson Education, Dec. 2014, ISBN: 1587144239.

[29] K Elbadawi and J Yu, “Improving network services configuration management,” pp. 1–6, 2011. DOI: 10.1109/ICCCN.2011.6006050.

[30] K. Elbadawi and J. Yu, “High level abstraction modeling for network configuration validation,” *GLOBECOM 2010 - 2010 IEEE Global Communications Conference*, pp. 1–6, 2010, ISSN: 1930-529X. DOI: 10.1109/GLOCOM.2010.5683110.

[31] W Enck, P McDaniel, S Sen, and P Sebos, “Configuration management at massive scale: system design and experience,” *USENIX '07*, 2007. [Online]. Available: https://www.usenix.org/events/usenix07/tech/full_papers/enck/enck.pdf.

[32] R Enns, M Bjorklund, J Schoenwaelder, and A Bierman, “Network configuration protocol (netconf),” *Internet Engineering Task Force, RFC 6241*, 2011, ISSN: 2070-1721. DOI: 10.17487/RFC6241.

[33] Ethan Banks, *@ecbanks*, Jul. 2016. [Online]. Available: https://twitter.com/ecbanks/status/625812864264208385.

[34] N. Feamster and H. Balakrishnan, “Detecting bgp configuration faults with static analysis,” *NSDI '05: 2nd Symposium on Networked Systems Design & Implementation*, May 2005.

[35] A Feldmann, “Netdb: ip network configuration debugger/database,” *AT&T Software Symposium*, 1999.

[36] B Fortz, J Rexford, and M Thorup, “Traffic engineering with traditional ip routing protocols,” *IEEE Communications Magazine*, vol. 40, no. 10, pp. 118–124, 2002. DOI: 10.1109/MCOM.2002.1039866.

[37] B. D. Freeman, “Network configuration management,” in, Guide to Reliable Internet Services and Applications, 2010, ISBN: 1848828284. DOI: 10.1007/978-1-84882-828-5.

[38] J. George and A. Shaikh, *Better management of large-scale, heterogeneous networks*, Nanog 64, 2015. [Online]. Available: https://www.nanog.org/sites/default/files//meetings/NANOG64/1011/20150604_George_Sdn_In_The_v1.pdf.

[39] V. Gill and M. Shields, “Programatic networks - autogen,” pp. 1–36, 2008.

[40] Google, *Textfsm - python module for parsing semi-structured text into python tables. - google project hosting*, 2013. [Online]. Available: https://code.google.com/p/textfsm/.

[41] J Gottlieb, A Greenberg, J Rexford, and J. Wang, “Automated provisioning of bgp customers,” *IEEE network*, vol. 17, no. 6, pp. 44–55, Nov. 2003. DOI: 10.1109/MNET.2003.1248660.

[42] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, “Evolve or die,” *SIGCOMM '16*, 2016. DOI: 10.1145/2934872.2934891.

[43] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, “Reproducible network experiments using container-based emulation,” *CoNEXT '12*, Dec. 2012. DOI: 10.1145/2413176.2413206.

[44] S Hares, *Summary of i2rs use case requirements (internet draft)*. [Online]. Available: http://www.ietf.org/id/draft-hares-i2rs-usecase-reqs-summary-00.txt.

[45] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau, “Large-scale virtualization in the emulab network testbed.,” *USENIX '08: 2008 USENIX Annual Technical Conference*, pp. 113–128, 2008.

[46] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," *WREN '09: Proceedings of the 1st ACM workshop on Research on enterprise networking*, Aug. 2009. DOI: 10.1145/1592681.1592683.

[47] X Huang and T Wolf, "A next generation internet state management framework," *Proceedings of the 2007 ACM CoNEXT conference*, 2007.

[48] G. Huston, *Leaking routes*, Accessed 2017-01-13, 2012. [Online]. Available: http://www.potaroo.net/ispcol/2012-03/leaks.html.

[49] Juniper Networks, Inc., "What's behind network downtime?" Sunnyvale, California, Tech. Rep. 200249-004, May 2008.

[50] ——, "Junosphere user guide," Juniper Networks, Inc., Tech. Rep., 2011.

[51] Z. Kerravala, "As the value of enterprise networks escalates, so does the need for configuration management," The Yankee Group, Tech. Rep., 2004.

[52] Knight, Simon, *Using python to design, configure, and measure large-scale networks*, Pycon Australia 2013, Hobart, Australia, Jul. 2013. [Online]. Available: https://www.dropbox.com/s/0je11t4q56qupc6/pycon13_autonetkit.pdf.

[53] S Knight, H. X. Nguyen, N Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, Oct. 2011. DOI: 10.1109/JSAC.2011.111002.

[54] S. Knight, "Automated configuration and measurement of emulated networks with autonetkit," 2013. DOI: 10.1145/2486001.2491692.

[55] S. Knight, A. Jaboldinov, O. Maennel, I. Phillips, and M. Roughan, "Autonetkit: simplifying large scale, open-source network experimentation," pp. 97–98, Aug. 2012. DOI: 10.1145/2342356.2342378.

[56] S. Knight, H. Nguyen, N. Falkner, and M. Roughan, "Realistic network topology construction and emulation from multiple data sources," The University of Adelaide, Tech. Rep., 2012. [Online]. Available: http://topology-zoo.org/publications/eu_nren_tech/eu_nren_tech.html.

[57] S. Knight, H. Nguyen, O. Maennel, I. Phillips, N. Falkner, R. Bush, and M. Roughan, "An automated system for emulated network experimentation," *CoNEXT '13*, 2013. DOI: 10.1145/2535372.2535378.

[58] B. Lantz, B. O'Connor, and B. O'Connor, "A mininet-based virtual testbed for distributed sdn development," pp. 365–366, Sep. 2015. DOI: 10.1145/2829988.2790030.

[59] G. Lauer, R. Irwin, C. Kappler, and I. Nishioka, "Distributed resource control using shadowed subgraphs.," pp. 43–48, 2013. DOI: 10.1145/2535372.2535410.

[60] S. Lee, T. Wong, and H. S. Kim, *To Automate or Not to Automate: On the Complexity of Network Configuration*. IEEE, 2008, ISBN: 978-1-4244-2075-9. DOI: 10.1109/ICC.2008.1072.

[61] B. Lewis and M. Peterson, "Network automation: ansible 101," *RIPE 71*, Nov. 2015.

[62] L. Lhotka, "Xml schema for router configuration data: an annotated dtd," Tech. Rep. 2/2003, Apr. 2003. [Online]. Available: http://archiv.cesnet.cz/doc/techzpravy/2003/netopeer-dtd/.

[63] S Litkowski, R Shakir, L Tomotaki, and K DSouza, *Yang data model for l3vpn service delivery*. [Online]. Available: https://tools.ietf.org/pdf/draft-ietf-l3sm-l3vpn-service-model-02.pdf.

[64] J. Lobo and V. Pappas, “C2: the case for a network configuration checking language,” *2008 IEEE Workshop on Policies for Distributed Systems and Networks - POLICY*, pp. 29–36, 2008. DOI: 10.1109/POLICY.2008.45.

[65] D. Maker, *Day 2 with virl: my early impressions.* Accessed 2017-01-13. [Online]. Available: https://gns3.com/discussions/day-2-with-virl-my-early-impress.

[66] D. Maltz, J. Zhan, G Hjalmtysson, A. Greenberg, J Rexford, G Xie, and H. Zhang, “Structure preserving anonymization of router configuration data,” *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 3, pp. 349–358, 2009. DOI: 10.1109/JSAC.2009.090410.

[67] M. Matuska, “Metaconfiguration of the computer network,” Tech. Rep., Dec. 2004. [Online]. Available: http://archiv.cesnet.cz/doc/techzpravy/2004/metaconfig/metaconfig.pdf.

[68] J. McClurg, H. Hojjat, P. C ern y, and N. Foster, “Efficient synthesis of network updates,” *SIGPLAN Not.*, vol. 50, no. 6, pp. 196–207, Jun. 2015. DOI: 10.1145/2813885.2737980.

[69] A Medem, R Teixeira, N Feamster, and M Meulle, “Joint analysis of network incidents and intradomain routing changes,” *2010 International Conference on Network and Service Management (CNSM)*, pp. 198–205, 2010.

[70] S. Morris, “Ntt global ip network configuration tools overview,” Feb. 2012. [Online]. Available: http://www.nanog.org/meetings/nanog54/presentations/Tuesday/Morris.pdf.

[71] D. P. D. Moss, *Netaddr*, Accessed 2017-01-13. [Online]. Available: http://netaddr.readthedocs.io/en/latest/.

[72] S Narain, “Network configuration management via model finding,” 2005. [Online]. Available: http://www.usenix.org/event/lisa05/tech/full_papers/narain/narain_html/.

[73] S Narain, R Talpade, and G Levin, “Network configuration validation,” English, in *Guide to Reliable Internet Services and Applications*, Springer, 2010, pp. 277–316, ISBN: 978-1-84882-827-8. DOI: 10.1007/978-1-84882-828-5_9.

[74] S. Narain, G. Levin, S. Malik, and V. Kaul, “Declarative infrastructure configuration synthesis and debugging,” *Journal of Network and Systems Management*, vol. 16, no. 3, pp. 235–258, 2008. DOI: 10.1007/s10922-008-9108-y.

[75] Neo Technology, Junisphere’s innovative Business eReality *solution built on neo4j*. [Online]. Available: http://info.neotechnology.com/rs/neotechnology/images/Junisphere.pdf.

[76] NetworkX Developers, *Networkx.algorithms.components.connected — networkx 1.10 documentation*, Oct. 2015. [Online]. Available: https://networkx.github.io/documentation/latest/_modules/networkx/algorithms/components/connected.html#connected_components.

[77] H. Nguyen, M. Roughan, S. Knight, N. Falkner, O. Maennel, and R. Bush, “How to build complex, large-scale emulated networks,” *TridentCom 2010*, pp. 3–18, May 2010. DOI: 10.1007/978-3-642-17851-1_1.

[78] J. Obstfeld, *Developer: have you discovered virl yet? | cisco communities*, Feb. 2016. [Online]. Available: https://communities.cisco.com/community/developer/blog/2016/02/09/have-you-discovered-virl-yet.

[79] J. Obstfeld, S. Knight, E. Kern, Q. S. Wang, T. Bryan, and D. Bourque, "Virl: the virtual internet routing lab," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 577–578, Aug. 2014. DOI: 10.1145/2740070.2631463. [Online]. Available: http://doi.acm.org/10.1145/2740070.2631463.

[80] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" *4th Usenix Symposium on Internet Technologies and Systems (USITS '03)*, 2003.

[81] M. Oswalt. (Mar. 2016). Network configuration templates using jinja2. Accessed 2017-01-13.

[82] T. J. Parr, "Enforcing strict model-view separation in template engines," *WWW '04: The 13th international conference on World Wide Web*, pp. 224–233, May 2004. DOI: 10.1145/988672.988703.

[83] E Parsonage, H. Nguyen, R Bowden, S Knight, N Falkner, and M Roughan, "Generalized graph products for network design and analysis," *Network Protocols (ICNP), 2011 19th IEEE International Conference on*, pp. 79–88, 2011. DOI: 10.1109/ICNP.2011.6089084.

[84] M. Pizzonia and M. Rimondini, "Netkit: network emulation for education," *Software: Practice and Experience*, n/a–n/a, 2014. DOI: 10.1002/spe.2273.

[85] I. Poese, B. Frank, S. Knight, N. Semmler, and G. Smaragdakis, "Padis emulator: an emulator to evaluate cdn-isp collaboration," p. 81, 2012. DOI: 10.1145/2377677.2377691.

[86] C. Prakash, J. Lee, Y. Turner, *et al.*, "Pga: using graphs to express and automatically reconcile network policies," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 29–42, Aug. 2015. DOI: 10.1145/2785956.2787506.

[87] *Quagga software routing suite*, Accessed 2017-01-13. [Online]. Available: http://www.nongnu.org/quagga/.

[88] B Quoitin and S. Uhlig, "Modeling the routing of an autonomous system with c-bgp," *IEEE network*, vol. 19, no. 6, pp. 12–19, 2005. DOI: 10.1109/MNET.2005.1541716.

[89] L Sanchez, K McCloghrie, and J Saperia, "Requirements for configuration management of ip-based networks," *Internet Engineering Task Force, RFC 3139*, 2001. DOI: 10.17487/RFC3139.

[90] D. Schmidt and V. Gill, *Automated configuration management*, NANOG 55.

[91] Schulman, Jeremy, *Jinja2 - 105 - dealing with data*, Accessed 2017-01-13, Feb. 2015. [Online]. Available: https://vimeo.com/120182263.

[92] J. Schulman, *Hosts data csv example*, Feb. 2015. [Online]. Available: https://raw.githubusercontent.com/jeremyschulman/demo_host_csv_template_render/master/hosts_data.csv.

[93] D. A. Schult and P Swart, "Exploring network structure, dynamics, and function using networkx," 2008.

[94] M Shaw, "What makes good research in software engineering?" *International Journal on Software Tools for Technology ...*, 2002. DOI: 10.1007/s10009-002-0083-4.

[95] S. Shenker, *The future of networking, and the past of protocols*, Accessed 2017-01-13, 2011. [Online]. Available: http://opennetsummit.org/talks/shenker-tue.pdf.

[96] R. Sherman, *Jinja2 templates for network automation*, Mar. 2015. [Online]. Available: http://vulgarpython.com/jinja2-templates-for-network-automation/.

[97] Y. W. Sung, S Rao, S Sen, and S Leggett, "Extracting network-wide correlated changes from longitudinal configuration data," pp. 111–121, 2009. DOI: 10.1007/978-3-642-00975-4_11.

[98] Y.-W. E. Sung, X. Sun, S. G. Rao, G. G. Xie, and D. A. Maltz, "Towards systematic design of enterprise networks," *IEEE/ACM Transactions on Networking*, vol. 19, no. 3, pp. 695–708, 2011. DOI: 10.1109/TNET.2010.2089640.

[99] P. J. Taylor, "Specification of policy languages for network routing protocols in the bellman-ford family," PhD thesis, zaynar.co.uk. [Online]. Available: http://zaynar.co.uk/misc/thesis.pdf.

[100] M. E. Tozal and K. Sarac, "Network layer internet topology construction," Feb. 2011, Accessed 2017-01-13. [Online]. Available: http://www.caida.org/workshops/isma/1102/slides/aims1102_ksarac_mtozal.pdf.

[101] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage, "California fault lines: Understanding the causes and impact of network failures," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 315–326, Aug. 2010, ISSN: 0146-4833. DOI: 10.1145/1851275.1851220. [Online]. Available: http://doi.acm.org/10.1145/1851275.1851220.

[102] B. Ujcich, K. C. Wang, B. Parker, and D. Schmiedt, "Thoughts on the internet architecture from a modern enterprise network outage," in *2012 IEEE Network Operations and Management Symposium*, 2012, pp. 494–497. DOI: 10.1109/NOMS.2012.6211939.

[103] VIRL-Open, ***Mar15 webinar l2 managed.virl***, Accessed 2017-01-13, Apr. 2015. [Online]. Available: https://github.com/VIRL-Open/sample-topologies.

[104] L Vanbever, G Pardoen, and O Bonaventure, "Towards validated network configurations with ncguard," *2008 IEEE Internet Network Management Workshop (INM)*, pp. 1–6, 2008. DOI: 10.1109/INETMW.2008.4660329.

[105] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure, "Seamless network-wide igp migrations," *SIGCOMM '11*, 2011. DOI: 10.1145/2018436.2018473.

[106] C. Vicente, *Automatically build, test and deploy your network configurations*, NANOG 63, 2015. [Online]. Available: https://www.nanog.org/sites/default/files/monday_general_autobuild_vicente_63.28.pdf.

[107] A. Voellmy and P. Hudak, "Nettle: A language for configuring routing networks," *Domain-Specific Languages: IFIP TC 2 Working Conference, DSL 2009 Oxford, UK, July 15-17, 2009 Proceedings*, W. M. Taha, Ed., pp. 211–235, 2009. DOI: 10.1007/978-3-642-03034-5_11.

[108] V. Voloshin, *Introduction to Graph Theory*. Nova Science Pub Incorporated, Jan. 2009, ISBN: 9781606923740.

[109] J. Wang, *The Virl Book*, ser. A Step-By-Step Guide Using Cisco Virtual Internet Routing Lab. Speak Network Solutions, Sep. 2016, ISBN: 9780692784365.

[110] J. M. Yates and Z Ge, "Network management: fault management, performance management, and planned maintenance," *Guide to Reliable Internet Services and Applications*, 2010. DOI: 10.1007/978-1-84882-828-5.

[111] D Yeung, Y Qu, J Zhang, I Chen, and L. A. (). Rfc draft: Yang data model for ospf protocol draft-ietf-ospf-yang-04, (visited on Jan. 13, 2017).

[112] E. W. Zegura, K. L. Calvert, and S Bhattacharjee, "How to model an internetwork," pp. 594–602, 1996. DOI: `10.1109/INFCOM.1996.493353`.