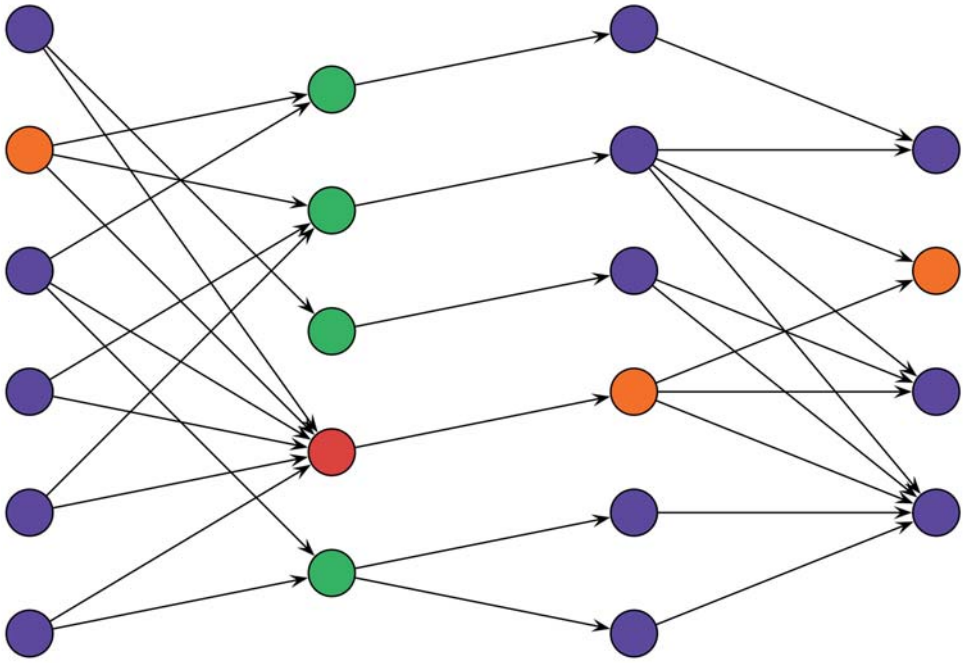
JYVÄSKYLÄ STUDIES IN COMPUTING
134

Tuukka Puranen

Metaheuristics Meet Metamodels

A Modeling Language and a Product Line Architecture for Route Optimization Systems



JYVÄSKYLÄN YLIOPISTO

JYVÄSKYLÄ STUDIES IN COMPUTING 134

Tuukka Puranen

Metaheuristics Meet Metamodels

A Modeling Language and a Product Line Architecture for Route Optimization Systems

Esitetään Jyväskylän yliopiston informaatioteknologian tiedekunnan suostumuksella julkisesti tarkastettavaksi yliopiston Agora-rakennuksen Lea Pulkkisen salissa marraskuun 4. päivänä 2011 kello 12.

Academic dissertation to be publicly discussed, by permission of the Faculty of Information Technology of the University of Jyväskylä, in the building Agora, Lea Pulkkinen hall, on November 4, 2011 at 12 o'clock noon.

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2011

Metaheuristics Meet Metamodels

A Modeling Language and a Product Line Architecture for Route Optimization Systems

JYVÄSKYLÄ STUDIES IN COMPUTING 134

Tuukka Puranen

Metaheuristics Meet Metamodels

A Modeling Language and a Product Line Architecture for Route Optimization Systems

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2011

Editors
Timo Männikkö
Department of Mathematical Information Technology, University of Jyväskylä
Pekka Olsbo, Ville Korkiakangas
Publishing Unit, University Library of Jyväskylä

Cover picture by Tuukka Puranen, based on figure on p. 152

URN:ISBN:978-951-39-4441-4
ISBN 978-951-39-4441-4 (PDF)

ISBN 978-951-39-4440-7 (nid.)
ISSN 1456-5390

Jyväskylä University Printing House, Jyväskylä 2011

ABSTRACT

Puranen, Tuukka
Metaheuristics Meet Metamodels. A Modeling Language and a Product Line Architecture for Route Optimization Systems.
Jyväskylä: University of Jyväskylä, 2011, 270 p.
(Jyväskylä Studies in Computing
ISSN 1456-5390; 134)
ISBN 978-951-39-4440-7 (nid.)
ISBN 978-951-39-4441-4 (PDF)
Finnish summary
Diss.

Transportation is an important human activity, and moving goods, information, and people is crucial to the functioning of our society. Computer-aided planning of these operations is being applied into practice and its benefits are evident. However, we do not yet fully know how to make software that can be applied cost effectively to solve today's heterogeneous set of routing problems. More specifically, we do not know how to manage the complexity of addressing the relevant aspects in logistic planning and solving the variety of different problem types arising in real-world routing. This inhibits the application of the latest results in operations research to real-world practice, which in turn prevents logistic operators benefiting from the most recent advances in computer-aided planning.

This thesis presents one approach for unifying routing problems and examines techniques for managing the inherent complexity of the domain. We suggest the construction of a higher-level model of vehicle routing and the application of model-driven software engineering practices for achieving an effective and systematic engineering approach for implementing vehicle routing systems.

In this thesis, we construct one higher-level model and present an implementation of such a model and the adjoining combinatorial optimization system. We argue that this approach allows a structured, unified view on vehicle routing models and metaheuristic optimization methods as well as decreases the effort needed in adapting the optimization system to different situations within the heterogeneous domain of vehicle routing. To validate our approach, a theoretical examination of the properties of the developed higher-level model is presented. The model is shown to be compatible with the state-of-the-art solution methodology. In addition, a number of routing problems from real-life cases and scientific benchmarks are modeled and solved using the developed system.

Keywords: vehicle routing problem, metaheuristic, metamodel, software architecture, product line, model-driven

Author

Tuukka Puranen
Department of Mathematical Information Technology
University of Jyväskylä
Finland

Supervisors

Professor Tommi Kärkkäinen
Department of Mathematical Information Technology
University of Jyväskylä
Finland

Professor Timo Tiihonen
Department of Mathematical Information Technology
University of Jyväskylä
Finland

Professor Pekka Neittaanmäki
Department of Mathematical Information Technology
University of Jyväskylä
Finland

Reviewers

Research Director, PhD Geir Hasle
Department of Optimisation
SINTEF
Norway

Professor Victor Zakharov
Faculty of Applied Mathematics and Control Processes
Saint-Petersburg State University
Russia

Opponent

Associate Professor, Dr. Wout Dullaert
Institute of Transport and Maritime Management
University of Antwerpen
Belgium

PREFACE

This thesis represents a culmination of research work conducted during 2009–2011 in a series of research projects. A new and young research group was formed at the University of Jyväskylä few years earlier, and we began working as a group on two simultaneous projects. The research behind this thesis sparked from these two projects where a number of differing vehicle routing cases needed to be modeled and solved. In addition, in one of the cases, an industry-strength implementation was required.

To cope with the requirements of the projects, we set out to solve the problems with as much reusable elements as possible. I was entrusted with the overall architecture of these systems and it quickly became apparent that a systematic approach to reuse was needed. We began building a product line of routing systems and, after some problems, typical in software engineering, completed them successfully.

The work of a software architect involves finding a balance between generality and specificity. The question is how to find the commonalities between different situations to avoid doing the same things more than once, and to accomplish this so that the applicability to individual situations is not compromised.

The question of generality versus specificity in software has been an interest of mine for quite some time, and I am grateful that I was offered the opportunity to work on the issue in my PhD. In many respects, the product line approach is an embodiment of this question, and the routing domain is a rewarding application area for testing the potential answers.

The generality versus specificity question will perhaps never be completely settled, but this work has made a young scientist realize something — and I believe there are researchers who agree with me. This realization might be the sole factor that led to the realization of this work as it exists today. It may well be, I observed, that sometimes it is beneficial to consider the larger context, to generalize, *a bit too far*; to refine the concepts too fine, only then, by necessity, to retract to the practical level. This level may, in the end, lie farther than was originally thought possible.

Indeed, local search may take us deep into the search space — but only to a certain depth. Sometimes what one needs is a *metaheuristic*.

ACKNOWLEDGEMENTS

This work has been supported by Tekes (The Finnish Funding Agency for Technology and Innovation) via TRANS-OPT and SCOPE projects, Ellen ja Artturi Nyyssösen säätiö, and COMAS graduate school. The work could not have been completed without the generous help of these entities, and this support is acknowledged with great gratitude.

It is a pleasure to thank the people who have made this work possible. I am grateful for the guidance given by my supervisors, Professors Tommi Kärkkäinen, Timo Tiihonen and Pekka Neittaanmäki. I cannot even begin to express how much I learned during the process, not only from research and writing, but also from supervising and teaching. It has been a pleasure to work with you all. I thank Professor Olli Bräysy for offering me the opportunity to work in the academia and especially in operations research. It has been a life-changing experience.

My special thanks go to the reviewers of this thesis, Research Director Geir Hasle and Professor Victor Zakharov for their careful examination and evaluation of my work. I am also grateful to Associate Professor Wout Dullaert for agreeing to be my opponent at the defense.

I emphasize that the usage of pronoun "we" in this work is intentional; I could not have completed this thesis without my fellow undergraduate and graduate students. I am grateful to Joni Brigatti, Antti Hallamäki, Pekka Hotokka, Antoine Kalmbach, Jukka Kemppainen, Antti Laitamäki, Jere Moilanen, and Jussi Rasku, who, among many other things, read drafts of this thesis, provided numerous fruitful discussions, and pointed out many of my embarrassing mistakes. I also wish to thank Jonathan Nussbaumer with whom I worked closely on the subject during his internship. A special mention must go to Jouko Nieminen who has taught me a lot during the short period we have worked together. His support in organizing the research activities made my thesis work possible.

I am thankful for the encouraging words from Professors Kaisa Miettinen and Tuomo Rossi. Also, a special mention should go to Antti-Juhani Kaijanaho whose invaluable comments made the Z-notation much more understandable. I also thank Doctor Yury Nikulin for his keen observations and suggestions.

I would like to express my gratitude to Professor Jean-François Cordeau with whom I worked during my PhD studies. I am grateful for the introduction to operations research and the guidance I received. And thank you for welcoming me to Montreal; the visit left a lasting impression on a young graduate student.

Finally, I am thankful to all my family and friends — you know who you are — sorry that I could not spend more time with you during this period. For my parents — who raised me, loved me, and supported me in everything I chose to do — I love you both. Pasi and Kaisa, this work is dedicated to you.

Jyväskylä, September 2011
Tuukka Puranen

LIST OF FIGURES

FIGURE 1 The structure of this dissertation. 28
FIGURE 2 An elementary classification of optimization problems. 53
FIGURE 3 The relationship of model accuracy and solution methodology performance. 55
FIGURE 4 A vehicle routing problem instance with 13 customers and four vehicles. 58
FIGURE 5 Different options for a comparative evaluation of an obtained solution. 65
FIGURE 6 Two conflicting criteria in designing metaheuristic search: intensification and diversification. 68
FIGURE 7 A relocate operator moving node 5. 71
FIGURE 8 A 2-opt operator replacing edges (1,5) and (2,6) with (1,2) and (5,6). 71
FIGURE 9 Processes and resulting products in software product line engineering. 91
FIGURE 10 The effort for developing *n* systems compared to product line engineering 94
FIGURE 11 A closed metamodeling architecture. 97
FIGURE 12 Basic concepts of model transformation. 98
FIGURE 13 An overview of the elements of the domain model visible to the model transformation engine. 107
FIGURE 14 An example of an acceptable mapping forming three routes within an *Ansatz* of a mapping and ordering problem. 114
FIGURE 15 An example of the results of two resource delta functions, travel time and distance, for usage on transitional projections. .. 117
FIGURE 16 An example of a PDP instance with two pickups and deliveries on one route. 124
FIGURE 17 An example of computing values for two capabilities and two resources on a route. 130
FIGURE 18 An example of checking constraints on the route length and time spent on a vehicle. 135
FIGURE 19 Modeling activity-specific time windows with zero capability and capability parents. 136
FIGURE 20 Modeling decision-dependent time windows with zero capability value and capability parents. 137
FIGURE 21 Modeling capability-dependent travel time. 139
FIGURE 22 An example of capability dependent transformations: a trailer affects the traveling speed of a vehicle. 143
FIGURE 23 Examples of groups of activities within an *Ansatz*. 144
FIGURE 24 Examples of different types of grouping constraints within a proper *Ansatz*. 147

FIGURE 25 An example of two resources, travel time and distance, and a profit value on a route. 148
FIGURE 26 A subset of modeling elements and their relationships in the modeling framework. 152
FIGURE 27 The modeling process and the two different metamodeling stacks within the framework. 154
FIGURE 28 A simplified domain model for CVRPs, PDPs, and VRPBs. 159
FIGURE 29 An example of a VRP model instance. 161
FIGURE 30 An example of a PDP model instance. 162
FIGURE 31 An example of a VRPB model instance modeled as a PDP with additional constraints. 164
FIGURE 32 An example of a relocate operation. 169
FIGURE 33 An example of a 3-opt operation. 170
FIGURE 34 Lexicographic search — the first step. 173
FIGURE 35 Lexicographic search — the second step. 174
FIGURE 36 Lexicographic search — the third step. 174
FIGURE 37 Concatenation of segments in constant time. 177
FIGURE 38 Layers within the product line architecture. 194
FIGURE 39 The module structure of the system. 195
FIGURE 40 Submodule level structures of the domain layer and an example application layer realization. 197
FIGURE 41 A simplified sequence of operations within the module structure. 198
FIGURE 42 The major elements of the VRP solver module. 200
FIGURE 43 A simplified sequence of operations within the `PopulateData` operation while providing geocoding and the shortest paths on the application layer. 204
FIGURE 44 A simplified sequence of operations within the `PopulateData` operation while employing optional road modules. 205
FIGURE 45 Modeling precedence constraints within a PDP instance. 207
FIGURE 46 Modeling two alternative deliveries within a PDP instance. 208
FIGURE 47 Modeling vehicle start and end activities. 213
FIGURE 48 Modeling driver, vehicle, equipment, and task compatibilities. 213
FIGURE 49 Modeling contamination rules. 216
FIGURE 50 Modeling trailer pickup and drop-off activities. 217
FIGURE 51 Modeling driver-task compatibilities. 218
FIGURE 52 Modeling multiple uses of vehicles. 218
FIGURE 53 Modeling breaks with predefined locations. 219
FIGURE 54 Modeling breaks with undefined locations. 219
FIGURE 55 Modeling breaks according to the UK legislation. 220
FIGURE 56 Modeling compartments with compatibilities within a PDP instance. 221
FIGURE 57 Modeling compartments in trailers within a PDP instance. 222
FIGURE 58 Modeling compartments within a heterogeneous fleet. 223
FIGURE 59 Modeling a locked sequence in the middle of a route. 224

FIGURE 60 An example of situations during continuous planning..... 224
FIGURE 61 A result of compressing the route start sequence after activity 3 has been confirmed..... 225

LIST OF ALGORITHMS

ALGORITHM 1 Template of single solution metaheuristic. 69
ALGORITHM 2 Template of population metaheuristic. 69
ALGORITHM 3 Local search. 70
ALGORITHM 4 Simulated annealing. 72
ALGORITHM 5 Tabu search. 73
ALGORITHM 6 Iterated local search. 73
ALGORITHM 7 Guided local search. 74
ALGORITHM 8 Variable neighborhood descent. 75
ALGORITHM 9 Generating capacitated VRP model instance. 160
ALGORITHM 10 Generating capacity array. 160
ALGORITHM 11 Generating distance matrix 161
ALGORITHM 12 Generating PDP model instance. 162
ALGORITHM 13 Generating VRPB model instance 163
ALGORITHM 14 Generating time window. 180
ALGORITHM 15 Generating PDP benchmark instance 181
ALGORITHM 16 Generating HVRPTW instance. 183
ALGORITHM 17 Generating two-phase distribution problem instance. 185
ALGORITHM 18 Generating ship routing instance. 186
ALGORITHM 19 Generating multiple capacities on PDP model instance. ... 207
ALGORITHM 20 Generating alternative deliveries on PDP model instance. 208
ALGORITHM 21 Generating time windows on PDP model instance 209
ALGORITHM 22 Generating restriction on time on vehicle. 210
ALGORITHM 23 Generating route length constraint. 210
ALGORITHM 24 Generating heterogeneous fleet of vehicles 212
ALGORITHM 25 Generating compatibilities within heterogeneous fleet. 214
ALGORITHM 26 Generating contamination restrictions. 215
ALGORITHM 27 Generating trailers affecting costs and travel times. 216

LIST OF SYMBOLS

Spaces

$\mathbb{B}$	set of values $\{0, 1\}$
$\mathbb{M}$	space of values of decisions on mapping
$\mathbb{N}$	set of natural numbers $\{0, 1, \dots\}$
$\mathbb{O}$	space of values of decision on ordering
$\mathbb{R}$	set of real numbers
$\mathbb{S}$	search space
$\hat{\mathbb{S}}$	feasible region of search space

Functions

$\mathcal{C}(\cdot)$	cost function
$\mathcal{F}(\cdot)$	resource extension function
$\mathcal{N}(\cdot)$	neighborhood function
$\mathcal{R}(\cdot)$	constraint function

Sets and Tuples

A	set of compartments
C	set of customers
C^b	set of backhaul customers
C^d	set of delivery locations
C^l	set of linehaul customers
C^-	set of destinations of task
C^+	set of origins of task
C^p	set of pickup locations
D	set of depots
E	set of edges in graph
G	graph
K	set of vehicles
L	set of locations
O	set of vehicle start and end locations
O^+	set of vehicle starting locations
O^-	set of vehicle ending locations
P	path of vertices in graph
R	set of transportation requests
U	set of incompatibilities between orders and compartments
V	set of vertices in graph
Y	set of incompatibilities between orders

Variables

u	decision variable for compartment loading
v	decision variable for serving task
w	decision variable for service start time

x	decision variable for travel between locations
y	decision variable for number of stops
z	decision variable for load of vehicle

Constants

α	resource window lower bound
α^{τ}	time window start at location
β	resource window upper bound
β^{τ}	time window end at location
δ	demand at customer
η	profit of visit
Γ	resource vector
γ	resource
ϕ	vehicle capacity
ψ	cost of travel between locations
σ	fixed costs of vehicle
τ	travel time between locations

Z Notation

$\top$	actor end activity
$\bot$	actor start activity
$(\_, \_, \_)$	tuple of elements
$\{\_, \_, \_\}$	set of elements
$\langle \_, \_, \_ \rangle$	sequence of elements
$(\lambda \_ \bullet \_)$	lambda expression: from schema to result of expression
$(\mu \_ \bullet \_)$	mu expression: unique value of expression from schema
$(\mathbf{let} \_ == \_ \bullet \_)$	local definition of variables
$\forall \_ \bullet \_$	universal quantification
$\exists \_ \bullet \_$	existential quantification
$\mathbb{F} \_$	finite set
$\mathbb{P} \_$	power set
$\_ \leftrightarrow \_$	set of relations between sets
$\_ \rightarrow \_$	set of functions between sets
$\_ \nrightarrow \_$	set of partial functions between sets
$\mathrm{ran} \_$	range of relation
$\mathrm{dom} \_$	domain of relation
$\_^{\sim}$	inverse of relation
$\mathbb{Z}$	set of integers
$\mathbb{N}$	set of natural numbers $\{0, 1, \dots\}$
$\mathrm{seq} \_$	finite sequence of elements
$\mathrm{seq}_1 \_$	non-empty finite sequence of elements
$\# \_$	number of members in finite set
$\mathit{first} \_$	first component of ordered pair
$\mathit{second} \_$	second component of ordered pair
$\bigcup \_$	generalized union over set of sets

$\_ \oplus \_$	overriding members of set with members of another set
$\_ \lhd \!\!\lhd \_$	range of relation restricted to domain of another
$\_ \frown \_$	concatenation of segments
$\_ \upharpoonright \_$	extracting set from sequence based on set of indices
$\_ \fatsemi \_$	sequential composition of schemas
$\_ \gg \_$	piping result of schema into another

LIST OF ACRONYMS

ACO	ant colony optimization
AMPL	a mathematical programming language
AOSD	aspect-oriented software development
API	application programming interface
BB	branch-and-bound
CARP	capacitated arc routing problem
COP	context-oriented programming
CP	constraint programming
CVRP	capacitated vehicle routing problem
DARP	dial-a-ride problem
EA	evolutionary algorithm
ESPPRC	elementary shortest path problem with resource constraints
FSM	fleet size and mix problem
GA	genetic algorithm
GAMS	general algebraic modeling system
GDP	gross domestic product
GLS	guided local search
GPDP	general pickup and delivery problem
GUI	graphical user interface
HVRP	heterogeneous vehicle routing problem
ILS	iterative local search
IRP	inventory routing problem
LNS	large neighborhood search
LP	linear programming
LS	local search
MCPDP	multi-commodity pickup and delivery problem
MDA	model-driven architecture
MDE	model-driven engineering
MDSPL	model-driven software product line
MDVRP	multi-depot vehicle routing problem
MOF	meta object facility
MVRPB	mixed vehicle routing problem with backhauls
OMG	object management group
OVRP	open vehicle routing problem
OVRPPD	open vehicle routing problem with pickups and deliveries
PDP	pickup and delivery problem
PDPTW	pickup and delivery problem with time windows
PIM	platform independent model
PLA	product line architecture
PSM	platform-specific model
PVRP	periodic vehicle routing problem
RCP	resource-constrained path

REF	resource extension function
SA	simulated annealing
SCPDP	single-commodity pickup and delivery problem
SDVRP	site-dependent vehicle routing problem
SPL	software product line
SPPRC	shortest path problem with resource constraints
SS	scatter search
SWCPP	shortest weight-constrained path problem
TCO	total cost of ownership
TS	tabu search
TSP	traveling salesperson problem
UML	unified modeling language
VND	variable neighborhood descent
VNS	variable neighborhood search
VP	variation point
VRP	vehicle routing problem
VRPB	vehicle routing problem with backhauls
VRPC	vehicle routing problem with compartments
VRPPD	vehicle routing problem with pickups and deliveries
VRPSD	vehicle routing problem with split deliveries
VRPTW	vehicle routing problem with time windows

CONTENTS

ABSTRACT
PREFACE
ACKNOWLEDGEMENTS
LIST OF FIGURES
LIST OF ALGORITHMS
LIST OF SYMBOLS
LIST OF ACRONYMS
CONTENTS

1 INTRODUCTION 21
1.1 Background and Research Environment 21
1.2 Objectives and Scope 24
1.3 Research Approach 25
1.4 Contribution and Dissertation Structure 26

2 VEHICLE ROUTING PROBLEMS 30
2.1 Formulating Optimization Problems 30
2.2 Vehicle Routing Problem Variants 33
2.2.1 Traveling Salesperson Problem 33
2.2.2 Vehicle Routing Problem 34
2.2.3 Vehicle Routing Problem with Backhauls 36
2.2.4 Multi-depot Vehicle Routing Problem 37
2.2.5 Pickup and Delivery Problem 38
2.2.6 General Pickup and Delivery Problem 39
2.3 Vehicle Routing Problem Extensions 41
2.3.1 Time Windows and Quality of Service Constraints 41
2.3.2 Open Routing 43
2.3.3 Fleet Selection 44
2.3.4 Compartments 45
2.3.5 Multiple Uses of Vehicles 47
2.3.6 Complex Cost Structures and Dynamic Travel Times 47
2.3.7 Drivers, Trailers, Equipment, and Compatibilities 48
2.3.8 Periodic Routing 50
2.3.9 Further Extensions 50
2.4 Modeling Optimization Problems 52
2.4.1 Modeling and Optimization Process 52
2.4.2 Design Considerations in Model Encodings 56
2.4.3 Graph Encodings for Routing Problems 57
2.4.4 Constraint Satisfaction Models for Routing Problems 60
2.5 Summary of Routing Problem Models 61

3 VEHICLE ROUTING SOLUTION METHODOLOGY 63
3.1 Problem Complexity and Method Evaluation 63

3.2 Exact Methods ... 65
3.3 Heuristic Methods ... 67
3.4 Metaheuristic Methods ... 68
3.4.1 Principles of Metaheuristic Search ... 68
3.4.2 Local Search ... 70
3.4.3 Other Single Solution Metaheuristics ... 72
3.4.4 Population Metaheuristics ... 75
3.5 Unifying Elements in Solving Routing Problems ... 76

4 IMPLEMENTABILITY IN ROUTING SYSTEMS ... 79
4.1 Synthesis on Vehicle Routing Research ... 79
4.2 Routing Systems from Software Quality Viewpoint ... 82
4.3 Software Reuse ... 86
4.4 Large-scale Software Reuse Techniques ... 87
4.4.1 Software Frameworks ... 88
4.4.2 Software Product Lines ... 89
4.4.3 Product Line Architectures ... 91
4.4.4 Model-driven Engineering ... 95
4.4.5 Aspect and Context Oriented Techniques ... 100
4.5 Summary of Implementability Aspects of Routing Systems ... 101

5 MODELING FRAMEWORK ... 104
5.1 Overview ... 104
5.2 Domain Model ... 106
5.3 Routing Metamodel ... 108
5.3.1 Classification of Constraints ... 109
5.3.2 Decision Variables ... 113
5.3.3 Resources ... 117
5.3.4 Capabilities ... 122
5.3.5 Partial Resources ... 129
5.3.6 Stack Resources ... 138
5.3.7 Activity Groups ... 144
5.3.8 Implemented Model ... 147
5.4 Model Transformation ... 151
5.4.1 Introduction ... 151
5.4.2 Transformation Interface ... 153
5.4.3 Usage ... 158
5.5 Summary of Approach ... 164

6 OPTIMIZATION IN THE FRAMEWORK ... 165
6.1 Optimization Process ... 165
6.1.1 Expressing Local Search ... 165
6.1.2 Complexity of Evaluation ... 171
6.2 Preliminary Results ... 179
6.2.1 State of Implementation ... 179
6.2.2 Benchmark Instances ... 180

6.2.3 Real-life Instances 183
6.3 Implications of Model Characteristics 187
6.3.1 Effects of Problem Structure 187
6.3.2 Adequacy of Local Search Operators 189

7 PRODUCT LINE OF ROUTING SYSTEMS 192
7.1 System Design 192
7.1.1 Objectives and Approach 193
7.1.2 Overall Structure 194
7.1.3 Submodule Structure 195
7.1.4 VRP Solver Module 199
7.1.5 Variation 201
7.2 Model Variation 206
7.2.1 Elementary Constraints 206
7.2.2 Time Windows and Quality of Service 209
7.2.3 Situation-dependent Travel and Costs 210
7.2.4 Fleet, Crew, and Equipment Selection 211
7.2.5 Drivers and Legislation 217
7.2.6 Compartment Loading Decisions 221
7.2.7 Notes on Interactive Optimization 222
7.2.8 Contrast to Existing Modeling Approaches 225
7.3 Quality Attributes 226

8 CONCLUSION AND DISCUSSION 229
8.1 Conclusions 229
8.2 Applicability of Proposed Approach 230
8.3 Implications of Unifying Modeling Framework 233
8.4 Further Research 238

YHTEENVETO (FINNISH SUMMARY) 242

REFERENCES 243

APPENDIX 1 TYPE INDEX 261
APPENDIX 2 THE Z NOTATION: A PRIMER 264
APPENDIX 3 GENERIC STACK IN Z NOTATION 269
APPENDIX 4 DETAILED COMPUTATIONAL RESULTS 270

1 INTRODUCTION

This work is a combination of two disciplines, operations research and software engineering, more specifically combinatorial optimization and software architecture. More precisely, we consider vehicle routing and model-driven software product lines. A common thread linking the different disciplines and the different viewpoints is the notion of *implementability*; the ability to provide a concept that is both a suitable solution to the problem (vehicle routing) and efficiently realizable in practice (product line). This "dualism" poses challenges[1], but also provides opportunities for combining the recent advances in both fields. The author's background is in software engineering and this is reflected heavily in this thesis, despite the fact that its contributions remain primarily in the domain of operations research.

1.1 Background and Research Environment

The volume of logistic operations is substantial today, and a considerable amount of effort is put into its design. The reasons are apparent; in Finland alone, the total annual cost of logistic operations equals 34,7 billion euros [189], which translates on average 14,2% of turnover of Finnish companies, and forms 19% of Finnish gross domestic product (GDP). Similar figures of 10%–17% of GDP in other industrialized countries are also present. The Finnish Ministry of Transport and Communications concludes in their 2009 report on Finnish logistics that lowering the costs of logistic operations is the single most important target for development.

The planning of logistics is being centralized, and computer-aided design is being employed for achieving efficient operation. A notable amount of work is still being done by hand: computer systems do not generally automate the

[1] Not the least of which is, clearly, the extent of the material needed to provide a clear view of both domains. Although an attempt was made to keep the work concise, this fact translated to a slightly long thesis.

planning itself; they provide the designers with additional tools for accessing the necessary data. Recently, however, attention has been shifting towards more or less automated planning. In these cases a computer system would, somewhat independently of the user involvement, plan the logistic operations in some level of detail [163]. Implementation of these systems has been made possible by achievements in operations research, more specifically, in linear and combinatorial optimization [95].

A well-known and widely studied combinatorial optimization problem, the vehicle routing problem (VRP), is at the core of designing effective transportation. In VRP, a set of vehicles is to be optimally routed to visit a set of customers with known demands subject to capacity constraints on the vehicles. VRP has a number of extensions that consider for instance constraints on delivery times or vehicles of different sizes. Recently attention has been devoted to complex variants of VRP that comprise of several of these extensions. These problems are sometimes named "rich" VRPs.

Solving rich VRPs with optimization has proved to be a viable approach for real-life decision making in routing. The theoretical routing models have a strong connection to the practical design of logistic operations and their results can usually be applied into practice with moderate effort. The optimization approach also provides a possibility to surpass the human cognition in this design process: computers are beginning to handle a number of integrated decisions on magnitudes overwhelming the human designer, and optimization methods can arrive in efficient but unintuitive solutions typically not achievable by humans. In addition, the automated planning is especially suitable to dynamic situations where changes to plans must be made rapidly.

In general, solving rich vehicle routing problems has given promising results in improving the effectiveness of the logistic planning. Vehicle routing has been studied for decades but the amount of scientific attention is still growing. For example, Irnich [109] estimates that several hundred papers are being published under this topic annually. In these papers, researchers describe a wide variety of different cases and solution methods. They also list a number of benefits, which usually fall into one of the following categories: cost savings, more effective operations, environmental benefits, improvements in quality of service, and easier planning. Not surprisingly, these benefits have gotten attention from software industry. In February 2010, *OR/MS Today* [163] surveyed sixteen software vendors providing 22 products for vehicle routing optimization. The current software packages offer automated planning for variety of cases and claim to be able to solve practical problems of 1000 customers and 50 vehicles in less than ten minutes[2].

Recent advances in vehicle routing are, however, hardly a silver bullet [32]. There are a number of factors that inhibit the diffusion of optimization technologies into practice. In as early as 1995, one operations researcher experienced on the subject, Marshall Fisher [72], asked that if these algorithms are so effective, why are they not used by more companies. The answer at that time seemed to be

[2] The survey does not, however, discuss the *quality* of the solutions.

the lack of robustness; implementation was hard to transfer from one company to another and solution methods produced obviously unreasonable results in some occasions. Fisher predicted that exact optimization algorithms would offer the best promise for achieving robustness.

In the last fifteen years, despite the increase in computational power, exact algorithms have not improved enough to be able to solve problems within the magnitude of those found in real-life situations; and despite a number of new heuristic and metaheuristic solution methods developed during these years, the required robustness has not been widely achieved. The algorithm implementations are still hard to transfer from one environment to another, methods continue to produce unreasonable results from time to time, and they can be sensitive to nuances in parameters or data. In addition, insufficient modeling of the problem yields unusable results; the optimization models still omit elements present in practical situations, or include only those specifically needed at the moment. Models expressive enough for a wide variety of cases simply do not exist. This has a practical implication; the heterogeneity of the cases makes the implementation tedious and heavy customization a necessity as different cases have to be modeled and solved individually.

As producers of routing systems, Sörensen *et al.* [191] observe that "implementation of a commercial routing package typically requires a lot of manual work to be done". They note that since the set of problems addressed by the single software is quite heterogeneous, especially the optimization methodology requires a lot of manual tuning, and that [one of the challenges is] "inability to develop completely new methods each time a new problem is encountered ...[which] would generally require *rewriting large portions of the code base* of the algorithms used [emphasis added]". This imposes a requirement on the optimization methodology: it has to be flexible to allow customization. Currently, only a few approaches meet this criteria and software vendors have almost unanimously resorted to these algorithms. However, these methods do not represent the state of the art in performance and solution quality, and this makes the customization of the system an attractive alternative, and although modularization provides some possibilities for adapting the software to different cases, software vendors have not really been able to address the heterogeneity in the domain systematically and efficiently. Mass customization still has to give in for case-by-case customization.

To address these issues, there has been an effort to unify the research [109], and several approaches have taken place to achieve different objectives. We have witnessed a shift towards real-life applications with intricate properties, and this has brought on a unifying effort in modeling of complex side constraints, using, e.g., techniques from constraint programming and column generation. In addition, common concepts from solution methodology are being formulated both in metaheuristics [89], which has provided tools for optimization in large scale problems usually present in real-life applications, and on local search [80], which is a central component in vehicle routing algorithms. Adaptation mechanisms are being implemented in an effort to cope with the heterogeneity of the cases [174].

Parallelization and distribution of computing are some of the techniques used to enhance scalability. Due to recent advances, there seems to be a consensus on that the operations research community is able to solve the "basic situation" of vehicle routing routinely. Unfortunately, this is not yet quite enough.

1.2 Objectives and Scope

Within the domain of combinatorial optimization, we focus on vehicle routing for three reasons. First of all, the heterogeneity of the domain offers an interesting challenge and an opportunity for generalization. Secondly, the domain is conceptually relatively compact, well-defined, and understood in theory but has not been widely addressed from an implementation viewpoint. Thirdly, as combinatorial optimization in general requires tailoring of the search operators for solving the problem — unlike, e.g., in mixed integer programming — the management of the variation occurring in both models and algorithms is essential. These aspects make routing an interesting subdomain. While the results from this study should be applicable to other areas of combinatorial optimization, the approach does indeed attempt to exploit the particular characteristics of the routing domain. Increasing the scope may thus deteriorate the effectiveness of the approach, and because of this, in this work, we focus on this well-defined subdomain.

Our overall objective is to provide cost-efficient means for solving the design problems in routing and scheduling of both vehicles and crew. We argue that this can be achieved by constructing an optimization system capable of handling a wide variety of different types of routing problems and capable of adjusting its operations according to the particular problem instance under consideration. A key element in achieving this is *the ability to manage the complexity arising in the heterogeneous domain of vehicle routing and scheduling*. There are two devices we necessitate for attaining such a goal: a way to describe the different problems in a generic way, and machinery that is able to utilize this general description. In order to understand the requirements for such description, we need to examine the current state of vehicle routing research; and in order to build the machinery, we need to both examine the solution approaches from VRP domain and to employ the recent advances in software engineering. The generic description is realized by **a modeling language, a metamodel, and a modeling framework** of vehicle routing problems. The machinery is **a software product line** utilizing these constructs. These devices enable the system to be adapted, and in the future, *to adapt*, to different situations and problem types commonly encountered in route optimization. Furthermore, employing a software product line enables the reuse of, for instance, modeling and solution methodology and minimizes the need to rewrite the code base each time a new problem is encountered.

A central component of the software product line is the generic description of vehicle routing problems. This enables the system to vary modeling and solution constructs independently of each other thereby greatly increasing possibili-

ties for reuse. To construct this description, we envisioned a modeling framework with the following properties: I) a domain model or routing problems should be translatable to the generic optimization model; II) the generic model should be able to express most of the common variants of VRP and their extensions; III) the constructed optimization model should be solvable; and IV) the modeling system should support interactive optimization seamlessly. In order to be solvable, the model should not render the previous knowledge of solution methodology useless, and thus be able to utilize the existing optimization algorithms. In addition, the framework should have a structure that supports parallelization and distribution of the computing to enhance scalability of the system in the future.

In order to achieve adequately extensive reuse, we attempted to capture the deterministic VRP variants and extensions that tend to occur in practice. The "wide" set of problems addressed in this work includes, but is not limited to, vehicle routing, routing with pickups and deliveries, operations with multiple depots or backhauls, several quality of service limitations including time windows, driver properties and regulations, fleet selection, trailers and special equipment, service restrictions, preferences, multidimensional capacity, compartment loading decisions, complex cost structures, time dependent operation times, and interactive optimization extensions. These constructs will be discussed first as approached in the literature, and secondly in the context of the proposed modeling framework.

In contrast to much of the recent work in vehicle routing, in this dissertation we do not provide sufficient methodology for solving the introduced problems. Such a work would be far too extensive for a single thesis. However, as many of the specific variants have been solved in the literature individually, we outline a generic strategy for including the solution capabilities of these heterogeneous solution methods into the developed system. We demonstrate that the existing solution methodology is compatible with the presented approach, which should convince the reader that the presented approach is a viable option when addressing a heterogeneous set of problems or especially complex problem variants.

1.3 Research Approach

To address the issue of managing complexity in routing domain, we first analyzed prior VRP research from both modeling and solution methodology point of view, and subsequently framed the objectives listed in the previous section. We examined relevant techniques from software engineering and built a prototype of a software product line for attaining the objectives. To test our approach, we implemented several routing models and some well-known VRP algorithms, and performed an assessment of the viability of the product line approach by employing it in two research projects. Finally, we tested the optimization system on real-life problems and benchmark instances. During the course of the research we formulated the propositions we test in this thesis. The hypotheses are as follows:

I) it is possible to define a single modeling system capable of expressing a wide variety of VRP variants and extension without modifying the underlying neighborhood exploration system or algorithm implementations,

II) such a system can utilize the existing knowledge from the current state of the art algorithms and metaheuristics,

III) such a system can be embedded into a software product line in order to achieve reusability in both optimization models and optimization methodology, and

IV) within this product line, a single architectural *variation point* is able to express the modeled case.

The hypotheses are tested by I) defining a formal model and describing several rich VRP models motivated by real-life examples to verify expressiveness of the system, II) expressing the atomic building blocks of local-search algorithms within the defined framework and analyzing the computational complexity of the methods to verify applicability of the system, III) presenting a system with well-defined variation points in optimization models, solution methodology, and model representation to verify implementability of the system, and IV) expressing a number of routing problems by changing the variability object associated with the model variation point to verify modifiability of the system.

1.4 Contribution and Dissertation Structure

The main contributions of this thesis reside within the domain of vehicle routing, but a substantial contribution was also given to the domain of software engineering during the course of this research. The domain of vehicle routing is an interesting case study on the applicability and benefits of model-driven software product lines due to its inherent requirement of modeling and heterogeneity of the problem domain.

In this dissertation, we give the following contributions.

1. *A metamodel and a modeling language for describing rich vehicle routing problems.*

To employ a single modeling system capable of expressing a variety of VRP variants and extension without modifying algorithm implementations, we need a higher level model on which the algorithms can operate independently of the case-specific details. This metamodel and an adjoining modeling language forms the first contribution of this dissertation, This is, in fact, the first metamodel developed for routing problems the author is aware of. This metamodel attempts to capture the commonalities of the most common vehicle routing problems, and is shown to be compatible with recent unifying efforts in VRP research. In addition,

we analyze different aspects and elements found in rich vehicle routing problems. We argue that analyzing and categorizing the aspects of the developed model in a structured manner will broaden our knowledge of vehicle routing problem, provide us with better ways to include real-life aspects into vehicle routing problems, and provide a base for future solution methodology suitable for highly variable and complex routing problems. Moreover, the developed modeling language enables flexible usage of the developed approach, and is shown to be simple enough to be embedded into a working implementation. In fact, this is the first attempt to examine an implementation of a unified model of routing problems.

2. *A model-driven software product line architecture for vehicle routing systems.*

The second contribution of this dissertation is the application of the recent advances from software engineering into the context of routing systems. A model-driven software product line architecture is developed and is shown to enable reuse at multiple places in the architectural level as well as to simplify the incorporation of case-specific details into an implementation of routing systems. This is, as far as the author is aware, the first model-driven product line approach to routing systems. Moreover, the presented implementation provides, to the best of our knowledge, the first practical demonstration of application of a model transformation as an architectural variation point. In this work, we describe the implemented optimization system, and analyze its structure from product line architecture and variability management viewpoints. We also discuss software quality attributes in the context of vehicle routing software systems. It is noteworthy that from the viewpoint of system design and software architecture, the analysis of generalization of vehicle routing problems has largely been neglected in the previous work.

3. *A formal specification of optimization models and processes in vehicle routing.*

The third contribution of this thesis is a formal specification of optimization models and solution processes in vehicle routing. A formal specification language is used to describe the developed metamodel, but this specification can also be used to define the routing variants describable by the metamodel. A relatively extensive set of routing variants is described formally in this thesis, along with basic operations of the most commonly used solution methods. These formalizations provide a basis for complexity analysis of different models, reasoning about their properties, and proving their correctness from implementation viewpoint. As far as we know, this is the first description of structures within an optimization system in a formal specification language.

In addition to the primary contributions in this dissertation, a set of minor contributions is also given. Firstly, a minor contribution of this work is an attempt to form an overview of vehicle routing research from the viewpoint of software engineering and to provide a useful prediction on what will be the next challenges especially from the implementation perspective. In addition — although

the aim of this work was not the development of optimization algorithms — we provide some initial numerical results from existing scientific benchmarks. Moreover, for validating the full system against the requirements we used a number of real-life cases, including newspaper distribution and oil product transportation. The cases were modeled and solved using the developed system. Some remarks on findings from these cases are given.

As we will observe, optimization can be divided into three main aspects, modeling, solving, and implementation. Mathematical modeling provides a description, optimization algorithms provide solution methods, and software implementation enables practical use. This tripartite structure is reflected also in the structure of this dissertation and illustrated in Figure 1. The vertical axis cov-

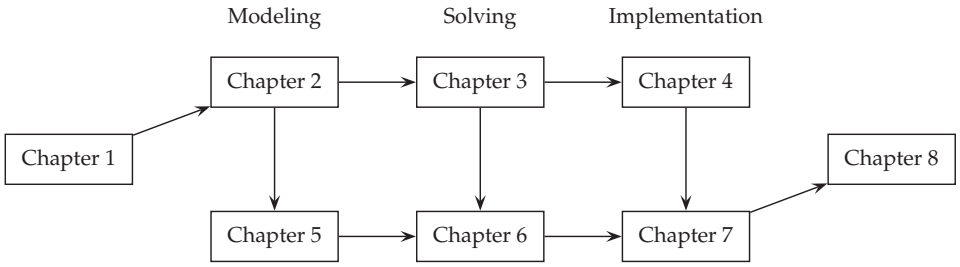


FIGURE 1 The structure of this dissertation.

ers the dimension of theoretical and constructive content. As usual, we proceed with providing context and theoretical background in the chapters at the top, and later construct our contributions on that basis in the chapters at the bottom. On the horizontal axis we see the three aspects. The chapters on the left consider primarily the aspects of modeling, the chapters in the middle, the solution methodology, and the chapters on the right, the implementation.

The structure of this dissertation, in detail, is as follows: first in **Chapter 2**, we briefly introduce the basic concepts in vehicle routing, and review in necessary detail the current variants of vehicle routing models and their extensions. Although mathematical programming formulations are given, this chapter is more illustrative than definitive. **Chapter 3** provides an overview of both exact and approximate methodology in solving vehicle routing problems. **Chapter 4** summarizes the research on vehicle routing and subsequently brings the routing problems and solution methodology to the context of implementation by examining the related work in the field of software engineering. This chapter presents more definitive conceptualizations than the previous two, and provides a basis for formal representations of both the modeling constructs and system design in subsequent chapters. In **Chapter 5**, we attempt to capture the synthesis formulated in the previous chapter, and present two formal models and a modeling framework for describing vehicle routing problems. Furthermore, we discuss the elements found in VRP models and analyze their properties. In **Chapter 6**, we examine the optimization process within the developed modeling framework, and

Chapter 7 provides details of the implementation of such a framework. In addition, we analyze the developed system from the quality attribute viewpoint. **Chapter 8** draws conclusions on the proposed approach, discusses the wider implications of the approach on vehicle routing research, and provides topics for further studies.

As this work is a combination of operations research and software engineering, different parts of this thesis may attract different audiences. For a reader with a background in operations research and an interest in modeling of routing problems, we recommend concentrating on Chapters 2, 5, and 6. For a reader with more interest in software engineering and architecture, we propose glancing through the end of Chapter 2 and reading Chapters 4, 5, and 7.

2 VEHICLE ROUTING PROBLEMS

"Looking back, I think it was more difficult to see what the problems were than to solve them."

— CHARLES DARWIN

In this chapter, we review the current state of the modeling of vehicle routing problems by providing an overview on several subtopics. The overview is given by highlighting some of the first papers on the subject, some recent developments, and where possible, by providing a pointer to a more comprehensive survey. This chapter is structured as follows: in Section 2.1 we introduce the basic concepts of formulating combinatorial optimization problems, in Section 2.2 describe several of the key variants of vehicle routing problems, and in Section 2.3 we review the work done on more complex routing problems with additional constraints, decisions, and properties. In Section 2.4, we discuss the practicalities of modeling the optimization problems, which is essential for enabling efficient solving of the problems; and finally in Section 2.5, we provide a brief summary.

2.1 Formulating Optimization Problems

Optimization is a tool for decision making. Many of these decisions involve designing structures or processes, and as we deal with increasingly complex problems still wanting to approach them with analytical means, optimization has become more widespread in many fields of engineering, management, and science. The classical process of decision making consists of three steps: *stating the problem*, *solving the problem*, and *applying the solution* into practice. When employing an optimization approach, the problem statement can further be divided into two phases: *formulating the problem* and *modeling the problem*.

Formulating the problem involves identifying the design questions of the problem, determining the criteria of the design on which to evaluate different solutions, and expressing the rules of an acceptable solution. Formulating the

problem may be done in varying degrees of formalism, depending on the case-specific requirements. Inexact formulations may be as simple as prose specifications of the goals and rules of the problem. In contrast, an example of an exact and definitive description is the formulation by *mathematical programming*, a technique for expressing analytical problems using algebraic formalism. The programming consists of mathematical relationships, which include, for instance, equations, inequalities, and logical dependencies [211]. Essential concepts in this formulation are *decision variables*, *objective function*, and *constraints*. These concepts correspond to the real-life questions we attempt to answer, the criteria on which to evaluate a solution, and the rules of an acceptable solution, respectively.

Formulation of a problem in exact mathematical terms may be a complex undertaking, and, as noted in [211], some have in fact questioned the applicability of optimization in decision making. This is an especially acute question in real-life operations, where multiple conflicting criteria, uncertainty about every aspect of the reality, inaccuracies in available data, aspects which are difficult or impossible to quantify, and dynamism during the actual operations should nevertheless be taken into account while making robust decisions (which are itself often ill-defined) for the long-term good of the decision maker (which is even more often ill-defined).

These decisions must be made with or without the help of the available tools. As mentioned, optimization is one such tool, and can be seen as a device for providing one or many *suggestions* for a solution. The results should usually not be taken as a final and complete solution to the problem, and the human decision maker should be kept in the process to provide both the needed accuracy and to enhance robustness of the planning. In practice, this can be achieved by, e.g., incorporating an interactive interface for manipulating and examining the models and the results of the optimization during the decision making. The soft nature of the decision making can be considered by, e.g., increasing the complexity of the objective(s) of the optimization. In this work we (partially) address these aspects by incorporating some of the interactivity requirements and modeling of soft constraints. This work needs to be continued towards multi-criteria optimization in the future. Although we consider here only deterministic problems, some aspects of uncertainty can also be addressed with these types of systems. Decreasing sensitivity to stochastic elements may be done through manipulation of data: for example, defining loose enough time windows can enhance robustness and this can — again — be left for the user of the optimization tool. As to the criticism on the unquantifiable data, Williams [211] notes that "many decisions concerning unquantifiable concepts, however they are made, involve an implicit quantification which cannot be avoided. Making such a quantification in a mathematical model seems more honest as well as scientific". This sentiment seems indeed reasonable.

Even with all the aforementioned aspects included, an optimization formulation (or in fact any other model) cannot capture all the aspects relevant to the decision maker. In practice the formulation of the problem begins by taking into account only the most relevant of its aspects. In routing, these include the distri-

bution of tasks to vehicles to form efficient tours that adhere to the most relevant constraints (e.g., vehicle load capacities, service time windows). As formulations gradually become more complex, they (hopefully) model the reality *accurately enough*.

With these remarks, we invite the discussion on the limitations of analytical approaches to decision making and, more broadly, those of operations research. In this work we limit ourselves to note that despite the shortcomings of the mathematical idealizations often used, mathematical modeling and optimization do have an impressive record on enhancing various operations, including routing.

To provide a solid base for discussing more complex problems, we begin by formulating a number of idealized routing problems frequently dealt with in scientific literature. In Sections 2.2–2.3, we present a number of problem formulations using mathematical programming. Such formulations take the following general form:

$$\min \mathcal{C}(s) \tag{1}$$

s.t.

$$s \in \hat{\mathbb{S}} \subseteq \mathbb{S}, \tag{2}$$

where $\mathbb{S}$ is the set of all solutions, $s \in \mathbb{S}$ a solution instance, $\mathcal{C} : \mathbb{S} \to \mathbb{R}$ the objective function that maps each solution s to a real value, and $\hat{\mathbb{S}}$ is the set of feasible solutions described by the problem constraints. The task is to find the values of decision variables occurring in both the objective and constraints, such that the objective value is minimized[1] and the equations, inequalities, and logical dependencies in constraints are respected.

As said, these sections discuss the problems formally for communicating the exact meaning of different variants, thus addressing the phase of *formulating* the problem. In Section 2.4, we discuss the *modeling* of optimization problems in more detail. Thus, we consider here only the problem statement in a mathematical form. Although we are aware that, for instance, the traveling salesperson problem (TSP) has several representations, for example the *permutation encoding* [195], we employ a unified approach: the problem statement is intended to be generalized from TSP into a general pickup and delivery problem (GPDP) to illustrate the differences and commonalities between the problems. One of the most used representations in the literature is the mixed integer linear programming formulation, and although many constraints presented here are of the linear form, we do not limit ourselves to linear representations. In fact, when solving routing problems, often the problem formulation is given in a linear form, but the metaheuristic solution methodology operates on a nonlinear graph representation. We discuss also the different representations in detail in Section 2.4.

Where possible, the following conventions are used: first, the objective function is formulated, followed by completeness and consistency constraints, governing, e.g., proper visits and properties of routes. Next we introduce the flow constraints governing the traversal between locations, and finally present the restrictions on values of decision variables.

[1] We may assume, without loss of generality, a minimization problem.

2.2 Vehicle Routing Problem Variants

Vehicle routing problem has been applied to numerous different cases, which vary slightly on their assumptions of the problem characteristics. We have selected some of the VRP variants based on their structure. The choice is not arbitrary, but nevertheless is not meant to be an exact classification, and indeed, many other variants can be defined. The sampling is illustrative rather than definitive, and covers the most distinguishable problem types.

2.2.1 Traveling Salesperson Problem

The most fundamental routing problem is the classic TSP, in which a salesperson must visit a set of locations exactly once and return to his original location. In this problem the objective is to minimize the total distance traveled. Despite its simple formulation, the TSP is still a relevant problem in many areas, including logistic planning. When designing routes for, e.g, individual couriers or for finding an optimal route in predefined distribution areas, the TSP still plays a role due to the effective solution methods [3] developed.

Although strictly speaking the TSP is not usually considered a VRP variant, its definition forms a basis for the VRP formulation. Note that since we employ semantics of VRP, the representation of the TSP is a bit unconventional. For instance, the flow constraints and decisions on sequence numbers are not typically used in a TSP formulation. Moreover, in these formulations, two locations can have travel costs of zero (denoting a distance of zero). Formally, the TSP is defined using a set of locations $L = \{0, \ldots, n\}$. The model contains two sets of decision variables, x_{ij} which have value 1 if and only if the salesperson travels from location $i \in L$ to location $j \in L$ and 0 otherwise; and y_i which have value indicating the sequence number of location i at the route. Moreover, traveling from i to j has a cost $\psi_{ij} \geq 0$ associated to it. Finally, let $\mathbb{B}$ denote the set of values $\{0, 1\}$. It is notable that in the formulations used in Sections 2.2 and 2.3, we assume that *it is possible to travel between any pair of locations*. This means that the resulting graph is a *full graph*. Using this notation, the problem can be stated as

$$\min \sum_{i \in L} \sum_{j \in L} \psi_{ij} x_{ij} \tag{3}$$

s.t.

$$\sum_{j \in L} x_{ij} = 1, \quad \forall i \in L, \tag{4}$$

$$\sum_{i \in L} x_{ij} - \sum_{i \in L} x_{ji} = 0, \quad \forall j \in L, \tag{5}$$

$$x_{ij}(y_i + 1 - y_j) = 0, \quad \forall i \in L, \forall j \in L \setminus \{0\}, \tag{6}$$

$$y_0 = 0, \tag{7}$$

$$x_{ij} \in \mathbb{B}, \quad \forall i \in L, \forall j \in L, \tag{8}$$

$$y_i \in \mathbb{N}, \quad \forall\, i \in L. \tag{9}$$

The objective function (3) minimizes the total costs of travel. TConstraint set (4) states that each customer has to be visited exactly once, and constraint set (5) states that the salesperson leaves a customer if and only if it entered it. Constraint set (6) ensures that the sequence numbers on the route are consistent. It also follows from this constraint that subtours are not allowed. Constraint (7) states that the sequence number of the first[2] node is zero. Constraint set (8) is a set of integrality constraints, and constraint set (9) states that the sequence numbers are natural numbers.

Note that the subtour elimination constraints are non-linear. Although the discussion of linearity is of importance, we do not present the linearized form of these constraints here as our aim is not to solve the problems using (mixed integer) linear programming solvers. Instead we refer to, for example, work by Cordeau *et al.* [43]. In general, this linearization can be achieved by introducing an exponential number of linear constraints.

Mathematically, the traveling salesperson problem is related to the question of a Hamiltonian circuit in a graph. This question goes back to Kirkman and Hamilton in 1856. The actual mathematical roots of the traveling salesperson problem are obscure, but references to an informal definition can be found as early as 1832 [186]. A number of other variants have since been studied, and these include, for instance, a precedence-constrained TSP, e.g., in [10], a TSP with backhauls, e.g., in [87], and a TSP with pickups and deliveries, e.g., in [65].

Some of the most effective heuristic methods for solving the TSP include the well-known Lin–Kernighan algorithm [136] and its subsequent modifications. For an in-depth view on a state-of-the-art TSP solution methodology we refer to book by Applegate *et al.* [3].

2.2.2 Vehicle Routing Problem

The vehicle routing problem, or the capacitated vehicle routing problem (CVRP), is probably the most studied combinatorial optimization problem. The VRP generalizes the TSP and can be characterized as h traveling salesperson problem, where $h > 1$. Thus we define a set of salespersons, or vehicles, K where $|K| = h$, and note that each vehicle $k \in K$ is identical having a capacity limit of ϕ.

The VRP has a wide range of applications. Usual examples include management of distribution of consumer goods such as dairy products, scheduling shipments, planning courier services, and routing of postal distribution. Other examples include school bus routing, planning of maintenance routes, designing operations within warehouses, and designing forestry inspection tours. In fact, almost any problem where a group of people has to perform spatially distributed tasks, is a candidate for a vehicle routing problem.

Similarly to the TSP, the VRP can be formulated using a set of locations, where each vehicle starts and ends at the same location, often called *depot* and

[2] We may assume, without loss of generality, that the route starts at node 0.

denoted here as d. In the VRP, we define a set of customers $C = \{1, \ldots, n\}$, and set $L = C \cup \{d\}$. Moreover, each customer $i \in C$ has a demand δ_i associated with it. As in the traveling salesperson problem, we have two decision variables x_{ij}^k, now indexed also with $k \in K$ having value 1 if and only if vehicle k travels from location i to location j and 0 otherwise; and y_i^k denoting the sequence number of location i at the route of vehicle k. The problem can be stated as

$$\min \sum_{k \in K} \sum_{i \in L} \sum_{j \in L} \psi_{ij} x_{ij}^k \tag{10}$$

s.t.

$$\sum_{k \in K} \sum_{j \in L} x_{ij}^k = 1, \quad \forall i \in C, \tag{11}$$

$$\sum_{i \in C} \delta_i \sum_{j \in L} x_{ij}^k \leq \phi, \quad \forall k \in K, \tag{12}$$

$$\sum_{j \in L} x_{dj}^k = 1, \quad \forall k \in K, \tag{13}$$

$$\sum_{i \in L} x_{ij}^k - \sum_{i \in L} x_{ji}^k = 0, \quad \forall j \in L, \forall k \in K, \tag{14}$$

$$\sum_{i \in L} x_{id}^k = 1, \quad \forall k \in K, \tag{15}$$

$$x_{ij}^k (y_i^k + 1 - y_j^k) = 0, \quad \forall i \in L, \forall j \in L, \forall k \in K, \tag{16}$$

$$y_d^k = 0, \quad \forall k \in K, \tag{17}$$

$$x_{ij}^k \in \mathbb{B}, \quad \forall i \in L, \forall j \in L, \forall k \in K. \tag{18}$$

$$y_i^k \in \mathbb{N}, \quad \forall i \in L, \forall k \in K. \tag{19}$$

The objective function (10) minimizes the total costs generated by all the vehicles. Constraint sets (11), (14), (16), (17), (18), and (19) have an additional index for each vehicle $k \in K$ in the problem and correspond to constraint sets (4), (5), (6), (7), (8), and (9), respectively. In addition, we define a constraint set (12) which states that each vehicle should not exceed its capacity, and constraint sets (13) and (15) which state that each vehicle should start its route from the depot and end their route at it, respectively.

The described formulation requires each location to be visited. If there are a limited number of vehicles at the depot, a feasible solution may not exist. Alternatively to assuming an unlimited fleet, we may relax the problem by modifying constraint set (11) and the objective function (10) to allow but penalize unvisited locations. In this context, it may be more natural to formulate a maximization problem, where each visit has a profit associated to it. We may formulate a maximization problem by observing that $\min f = \max -f$, for any function f. Let η_i be the profit of visiting a location $i \in L$. The problem can be stated as constraint sets (12)–(19), constraint set (21), and objective function (20).

$$\max \sum_{k \in K} \sum_{i \in L} \sum_{j \in L} \eta_i x_{ij}^k - \sum_{k \in K} \sum_{i \in L} \sum_{j \in L} \psi_{ij} x_{ij}^k. \tag{20}$$

$$\sum_{k \in K} \sum_{j \in L} x_{ij}^k \leq 1, \quad \forall i \in C. \tag{21}$$

The objective function (20) maximizes the profits collected by visiting the different locations. By constraint set (21) one vehicle at most visits each location. Note also that this formulation allows prioritization of customers by introducing different profits to different locations according to their priority.

The vehicle routing problem was first formulated in 1959 by Dantzig and Ramser [54] as the "truck dispatching problem". During the following 50 years of study, a significant amount of work has been devoted to the subject, and one could argue that the VRP has grown to a class of problems [130]. For a more in-depth view on the VRP research, we refer to books by Toth and Vigo [202] and Golden *et al.* [95], and a recent taxonomic review by Eksioglu *et al.* [66].

2.2.3 Vehicle Routing Problem with Backhauls

While the capacitated vehicle routing problem models either the distribution *or* collection of goods, in many cases both of these operations are performed by the same operator. For instance, a company distributing beverages to grocery stores is in fact many times responsible for collecting the empty bottles for refilling or recycling. The distribution part of the operation is referred to as *linehauling* and the collection part as *backhauling*. In practice, however, these operations cannot be freely mixed; many of the trucks used in transportation are rear-loaded, and in these types of vehicles, a simultaneous loading and unloading of goods is slow, unpractical, or impossible. Also, it is more likely that the linehaul customers prefer an early delivery, and the backhaul customers, a late pickup. Thus, in a vehicle routing problem with backhauls (VRPB), we divide the problem into two parts: linehauls and backhauls, where all the linehauls must occur before any of the backhauls. In addition, a route is not allowed to consist only of backhaul customers. Note, however, that when the precedence restriction does not apply and the linehauls and backhauls can be freely mixed, we need additional modeling constructs. We introduce this variant, the pickup and delivery problem (PDP), in Section 2.2.5.

The VRPB is a generalization of the VRP with additional constraints for precedence. While the VRPB could be considered as two separate routing problems, solving the integrated problem can yield additional savings. This is best illustrated by considering the fact that starting the backhauls from the last linehaul customer is likely to be a good approach, but this cannot be taken into account in the formulation of the second routing problem in any way; all trucks are assumed to start from the central depot in both phases. However, when applicable, integration of backhauls and linehauls is a standard practice in the industry and the need for an integrated model is apparent.

Formally, let C be the set of customers and L the set of locations. We define a set of linehaul customers $C^l \subseteq C$ and a set of backhaul customers $C^b \subset C$ such that $C^l \cup C^b = C$ and $C^l \cap C^b = \emptyset$. Note that if both linehaul and backhaul operation are required at the same place, this is modeled as two customers in C with travel

costs of zero between them. The model consists of the objective (10), constraint sets (11)–(19), and constraint sets (22)–(23) described below.

$$y_i^k - y_j^k \leq 0, \quad \forall\, i \in C^l, \forall\, j \in C^b, \forall\, k \in K, \tag{22}$$

$$y_i^k > 1, \quad \forall\, i \in C^b, \forall\, k \in K. \tag{23}$$

Constraint set (22) states that each linehaul must occur before any backhaul. Constraint set (23) ensures that at least one linehaul customer is served before serving any backhaul customers.

Deif and Bodin [57] appear to have been among the first to formulate the integrated VRPB in 1984, and the problem has since been addressed a number of times, for example, in Toth and Vigo [201]. Recent research includes that of Ropke and Pisinger [175] who consider a number of VRPB variants simultaneously, and apply a unified heuristic for all the problem variants presented in their work; and Brandão [26] who applies a metaheuristic search scheme to the problem. For a more in depth view on the VRPB we refer to the recent survey by Parragh *et al.* [161].

2.2.4 Multi-depot Vehicle Routing Problem

When a company operates on a geographically broad area or whenever the size of the fleet increases, the number of terminals (depots) tends to increase. And the larger the operator, the larger the potential in integrated planning. This increases the need to consider a number of vehicle terminals simultaneously and to coordinate operations between them. Although in the VRP all vehicles are considered identical, they may yet reside in different sites, and selecting the depot each customer is served from is a relevant design problem. The multi-depot vehicle routing problem (MDVRP) addresses this issue by splitting the fleet into several parts, or by assuming an unlimited fleet in every depot and choosing the appropriate set of vehicles.

The MDVRP formulation is rather simple, and does not take into account inter-depot transportation nor pickups at multiple depots. The problem is divided into several subproblems that interact only by an assignment of a customer from a depot to another — in other words, assigning a customer from one vehicle to another whenever those vehicles reside at different depots. For several pickups at multiple sites, such as some customers requiring picking up newspapers from two different printing plants before distributing them, one needs yet again constructs from the PDP introduced in Section 2.2.5. For inter-depot transportation, one needs both the PDP formulation and additional precedence and dependency constraints.

To define the MDVRP formally, instead of a single depot d, we introduce a set of depots $D = \{d_1, \ldots, d_m\}$. We assume an unlimited number of vehicles at each depot. We now set $L = C \cup D$. The MDVRP consists of the objective function (10) and constraint sets (11), (12), (14), and (16)–(19), as well as constraint sets

(24)–(26) given below.

$$\sum_{d\in D}\sum_{i\in C} x_{di}^{k} = 1, \quad \forall k \in K \tag{24}$$

$$\sum_{d\in D}\sum_{i\in C} x_{id}^{k} = 1, \quad \forall k \in K \tag{25}$$

$$\sum_{i\in C} x_{di}^{k} - \sum_{j\in C} x_{jd}^{k} = 0, \quad \forall k \in K, \forall d \in D \tag{26}$$

Constraint sets (24) and (25) state that each vehicle must start at a single depot and end its route to a single depot, respectively. Constraint set (26) ensures that each vehicle returns to the depot it started at.

The MDVRP was first considered by Tillman [198], Tillman and Hering [199], and Cassidy and Bennet [35]. The MDVRP has been studied, e.g., by Renaud *et al.* [170] and Cordeau *et al.* [46] who both use a metaheuristic solution approach. Crevier *et al.* [50] studied an interesting variant where vehicles are allowed to replenish at intermediate depots during their route. Cordeau *et al.* [44], and, more recently, Vidal *et al.* [206] considered periodic multi-depot vehicle routing problems.

2.2.5 Pickup and Delivery Problem

Designing transportation for delivery and collection is only one aspect of logistics design. In the previous sections we highlighted some of the variants of delivery and collection problems, but almost as important is the family of problems usually referred to as the pickup and delivery problem (PDP). These kinds of problems where the task is to transport goods or passengers between given origins and destinations, differ from the previously discussed variants in that the goods or passengers do not reside in any central location. These problems arise naturally in dial-a-ride or courier services, and door-to-door transportation, for example, in the municipal school bus service.

We distinguish three types of pickup and delivery problems that have been covered in the literature (see, for instance, Cordeau *et al.* [47]). One is the single-commodity pickup and delivery problem (SCPDP) in which one type of goods is either picked at or delivered to each customer. A typical example of this is transportation of cash between bank offices. The second variant considered is the mixed vehicle routing problem with backhauls (MVRPB), where no restrictions on mixing linehaul and backhaul customers are defined. A standard VRPB can therefore be seen as a special case of PDP. The third variant is the multi-commodity pickup and delivery problem (MCPDP) in which a unique commodity has to be picked at each pickup location and transported to a delivery location. If the distinction to the three cases is made, the third variant, which is the most general of these, is often referred to as the vehicle routing problem with pickups and deliveries (VRPPD).

Note that the function describing the load of a vehicle is no longer monotonic since a route involves a number of pickups and deliveries. A vehicle can — in the absence of other constraints — therefore serve any number of requests.

However, ensuring the vehicle load on every location requires introduction of another decision variable z_i^k which denotes the load of vehicle $k \in K$ after location i. In addition we define a set of pickup customer locations $C^p \subset C$ and a set of delivery customer locations $C^d \subset C$, such that $C^p \cup C^d = C$ and $C^p \cap C^d = \emptyset$. Let $C^p = \{1, \ldots, n\}$, and $C^d = \{n+1, \ldots, 2n\}$. Each task is then associated with a pickup location i and a corresponding delivery location $i+n$. Finally, let $\delta_i = -\delta_{n+i} \; \forall \, i \in C$.

The VRPPD model consists of the objective function (10), constraint set (11), constraint sets (13)–(15), constraint set (18), as well as constraint sets (27)–(31) defined below.

$$\sum_{i \in C} x_{ij}^k - \sum_{i \in C} x_{n+i,j}^k = 0, \quad \forall \, j \in C^p, \forall \, k \in K \tag{27}$$

$$x_{ij}^k (z_i^k + \delta_j - z_j^k) = 0, \quad \forall \, i \in C, \forall \, j \in C, \forall \, k \in K \tag{28}$$

$$z_i^k \leq \phi, \quad \forall \, i \in L, \forall \, k \in K \tag{29}$$

$$z_d^k = 0, \quad \forall \, k \in K \tag{30}$$

$$z_i^k \in \mathbb{R} \geq 0, \quad \forall \, i \in L, \forall \, k \in K \tag{31}$$

Constraint set (27) states that each delivery location is visited by the same vehicle visiting the corresponding pickup location, constraint set (28) ensures consistency of vehicle load variables, constraint sets (29) states that maximum capacity of a vehicle cannot be exceeded, and constraint sets (30) and (31) ensure that the vehicle load has proper initial and intermediate values, respectively.

Additional VRPPD variants have been introduced, for example, by Maciek *et al.* in [156] where the authors consider a problem with the possibility of splitting the loads such that the delivery of a certain order is completed in multiple trips. For a recent survey on PDPs, we refer to that of Parragh *et al.* [162].

2.2.6 General Pickup and Delivery Problem

The general pickup and delivery problem was introduced by Savelsbergh and Sol [180]. The authors introduced a more general variant of VRPPD to deal with practical issues arising in vehicle routing. This was partly motivated by their observation of the fact that although the pickup and delivery problem is theoretically at least as interesting and in practice as important as the vehicle routing problem, it had received far less attention. Fifteen years from their observation, the situation seems to prevail. It may be that the classic CVRP attracts more research due its simple yet elusive nature. The PDP is just — some could say — an annoyingly complex version of the former.

Two major differences separate the GPDP from the VRPPD. First of all, in the former, each vehicle starts and ends its route at an arbitrary, but predefined, point. This variant arises naturally in situations where there are multiple depots, an additional external fleet that can be hired, and most importantly, whenever dynamic planning occurs. Dynamic planning is necessary if situation, that is, the routing problem, changes while vehicles are on route. This can happen for a number of reasons; a transportation request may become available in real-time, or a

sudden delay or a breakdown may require redesign of the plan. In this situation, the problem has to be solved without any notion of depot, since all vehicles are scattered over the planning area. The second difference is the fact that each transportation request may consist of more than one part. In this situation, either a number of pickups has to be made to collect a set of goods and transport them to a destination or vice versa. This situation can arise, for example, in construction industry, where each construction site requires a number of raw-materials from different factories and warehouses.

Formally, for each transportation request $r \in R$, a total load of δ_r has to be transported from a set of origins $C_r^+ \subset C$ to a set of destinations $C_r^- \subset C$, such that $C^+ = \bigcup_{r \in R} C_r^+$, $C^- = \bigcup_{r \in R} C_r^-$, and $C^- \cup C^+ = C$, where R is the set of transportation requests. Each load is divided as follows: $\delta_r = \sum_{i \in C_r^+} \delta_i = -\sum_{i \in C_r^-} \delta_i$. Moreover, each vehicle $k \in K$ has a starting location o_k^+, and an ending location o_k^-. Let $O^+ = \{o_k^+ \mid k \in K\}$, $O^- = \{o_k^- \mid k \in K\}$, and $O = O^+ \cup O^-$. Let $L = O \cup C$. Finally, let v_r^k be a decision variable equal to 1 if and only if request $r \in R$ is assigned to vehicle $k \in K$ and 0 otherwise. The problem consists of the objective function (10), constraint sets (14), (18), (28), (29), and (31), as well as constraint sets (32)–(37) given below.

$$\sum_{k \in K} v_r^k = 1, \quad \forall r \in R, \tag{32}$$

$$\sum_{j \in L} x_{ij}^k = \sum_{j \in L} x_{ji}^k = v_r^k, \quad \forall r \in R, i \in C_r^+ \cup C_r^-, k \in K, \tag{33}$$

$$\sum_{i \in L \setminus \{o_k^+\}} x_{o_k^+ i}^k = 1, \quad \forall k \in K, \tag{34}$$

$$\sum_{i \in L \setminus \{o_k^-\}} x_{i o_k^-}^k = 1, \quad \forall k \in K, \tag{35}$$

$$z_{o_k^+} = 0, \quad \forall k \in K, \tag{36}$$

$$v_r^k \in \mathbb{B}, \quad \forall r \in R, \forall k \in K. \tag{37}$$

Constraint set (32) ensures each transportation request is assigned to exactly one vehicle. Constraint set (33) states that a vehicle must visit a location if it is an origin or a destination of a request assigned to that particular vehicle. By constraint sets (34) and (35) each vehicle starts and ends at the correct location. Constraint set (36) ensures a proper vehicle load value at the starting location and (37) is a set of integrality constraints.

It is evident that the GPDP is the most general of the formulations given in this chapter so far. In fact, each variant presented here can be expressed as a special case of GPDP. Savelsbergh and Sol [180] described three problem variants in terms of their formulation: the pickup and delivery problem, the dial-a-ride problem (DARP) and the vehicle routing problem. We will mention the DARP in Section 2.3.1, when we discuss time window and quality of service constraints. Other variants can be characterized as follows:

The pickup and delivery problem can be defined using the GPDP by setting $|O| = 1$ and $|C_r^+| = |C_r^-| = 1 \; \forall r \in R$. In addition, we let r^+ be the unique element of C_r^+, r^- be the unique element of C_r^-, and o the unique element of O.

The mixed vehicle routing problem with backhauls can be formulated by setting $|O| = 1$, $|C_r^+| = |C_r^-| = 1 \; \forall r \in R$, and $(r^+ = o \wedge r^- \neq o) \vee (r^+ \neq o \wedge r^- = o) \; \forall r \in R$. If mixing of linehauls and backhauls is prohibited, we need to impose the precedence constraints (22)–(23).

The multi-depot vehicle routing problem can be formulated by setting $|O| > 1$, $|C_r^+| = |C_r^-| = 1 \; \forall r \in R$, and $C^+ = O$ or $C^- = O$.

The capacitated vehicle routing problem can be formulated by setting $|O| = 1$, $|C_r^+| = |C_r^-| = 1 \; \forall r \in R$, and $C^+ = O$ or $C^- = O$.

The traveling salesperson problem can be formulated by setting $|O| = 1$, $|C_r^+| = |C_r^-| = 1 \; \forall r \in R$, $C^+ = O$ or $C^- = O$, and $|K| = 1$.

Although the GPDP can be used to express a wide variety of problems, lifting assumptions naturally yields more difficult problems. The general pickup and delivery problem, and to most extent VRPPDs, are more difficult to solve than VRPs partly due to their lack of concept of *location closeness*. In the VRP, if locations are geographically close to each other, they are likely to be served by the same vehicle. In PDPs, this is not certain. This has also a direct consequence to solution evaluation: in the CVRP — and the TSP for that matter — solutions can be evaluated by looking at their geographical structure, but an optimal solution to a PDP may look incomprehensible to the human eye.

2.3 Vehicle Routing Problem Extensions

In the previous section, we introduced a set of routing problem variants which differ by their structural properties, number of constraints, and decision variables. In this section, we discuss a set of *extensions*. From our point of view, extensions differ from variants in that each variant is usually dealt in literature as such, and additional extensions are in many cases added to the problem. This distinction is not exact, since, for instance, backhauls could be interpreted as an extension for the VRP — or, in fact, for the TSP or the PDP. We will address this distinction between different types of problems in the subsequent chapters when we utilize our unified model, which allows us to analyze the differences on the structural level.

2.3.1 Time Windows and Quality of Service Constraints

Capacitated vehicle routing is inherently a spatial problem. However, it is apparent that the nature of logistic operations requires a temporal dimension. Vehicles and drivers have schedules, customers have to be served at specific time, and there are limitations on how long goods or passengers are allowed to stay on a vehicle (quality of service). Well-known examples of these kinds of problems in-

clude postal services, courier services, meal transportation, school bus routing, and in general, situations in which the receiver of the order has to be present at the delivery. Nontrivial quality of service constraints arise naturally especially in school bus routing where each schoolchild should not stay on the vehicle more than a predefined amount of time typically depending on the total distance between home and school.

One notable problem that has been dealt with in the literature as a separate variant is the dial-a-ride problem. In the DARP each pickup and delivery consists of one or more passengers traveling together, and thus every order has a demand equal to the capacity of the vehicle. In these problems, and to some extend in the PDP in general, time windows and quality of service restrictions tend also to be tight. This is due to the fact that when serving individual passengers, waiting and being late are not tolerated. These characteristics make the case different, although modeling the problem is quite straightforward.

To deal with the temporal aspects of routing problems, the notion of time was introduced along with adjoining constraints. Time constrained problems, especially the vehicle routing problem with time windows (VRPTW), are often considered a separate variant, but since it can be combined with almost every other problem type, we chose to refer to it as an extension. It can be considered one of the most important ones, and the amount of study devoted to this extension matches its importance.

To extend the problems formally, we define an additional set of decision variables. We denote the service start time for vehicle k at location i with w_i^k. Let α_i^τ and β_i^τ be the time window start and end times for location i, respectively. Finally, let τ_{ij} denote the travel time between locations i and j, including any service (stop) time at location i. The VRPTW can be defined by the objective function (10), constraint sets (11)–(15), (18), and constraint sets (38)–(41):

$$x_{ij}^k(w_i^k + \tau_{ij} - w_j^k) = 0, \quad \forall\, i \in L, \forall\, j \in L, \forall\, k \in K, \tag{38}$$

$$\alpha_i^\tau \leq w_i^k \leq \beta_i^\tau, \forall\, i \in L, \forall\, k \in K, \tag{39}$$

$$w_d^k = 0, \forall\, k \in K, \tag{40}$$

$$w_i^k \in \mathbb{R} \geq 0, \forall\, i \in L, \forall\, k \in K. \tag{41}$$

Constraint set (38) ensures consistency of the service time decision variable, by constraint set (39) the time windows are respected at each location, and constraint sets (40) and (41) ensure proper initial and intermediate values for the service time decision variables.

Time windows have been combined with practically every problem variant, including the VRPB and the PDP. We note that in the mathematical programming formulations presented, the decision variable y becomes redundant in the VRPB and the PDP when we introduce time windows. In fact, more generally, any precedence constraint can be stated using constraints on the starting time at locations, and this practice is widely used.

Among the first to work on temporally oriented problems were Dantzig and Fulkerson [55] in 1954, who considered a scheduling problem of oil tankers.

These aspects were later applied to vehicle routing, and among the most notable work has been that of Solomon [190]. More recently, Azi *et al.* [6] and Moccia *et al.* [145] considered time windowed variants and applied exact and metaheuristic approaches to the problems, respectively. Interestingly, Cordeau *et al.* [44] note that they applied methodology not specifically developed for temporally oriented problems and achieved good results. This highlights the fact that time windows are a natural extension to the base problems. The pickup and delivery problem with time windows (PDPTW) was considered among the first by Dumas *et al.* [64].

One recent and notable extension to time windowed problems is the routing with multiple time windows, where there are a number of alternative time windows in which a customer can be served or a vehicle used. This has been dealt, for example, by Xu *et al.* [216], who considered a pickup and delivery problem with several real-life aspects, including multiple time windows. Furthermore, a more general case was considered by Ibaraki *et al.* [107], who solved a problem where time windows could be violated by a certain cost. This extension is usually referred to as the problem with soft time windows. For a more extensive overview of time constrained routing and scheduling we refer to work of Desaulniers *et al.* [59].

2.3.2 Open Routing

In every variant from the vehicle routing problem to the pickup and delivery problem, each vehicle has had predefined starting and ending points. There are cases, however, where the routes start from predefined locations, but do not have an explicit end. This is often referred to as the open vehicle routing problem (OVRP). There are two notable cases where this setting is appropriate: if, after serving each customer, the vehicle has to return to the depot using exactly the same route it traveled, the route can be considered from the depot to the last customer; and in case of an external fleet that has to be rented to, for instance, compensate fluctuating demand, the vehicles are typically charged by the kilometers driven, starting from the first stop (depot) and ending at the last (the last customer of the route). The OVRP has, therefore, a practical relevance, despite its apparent similarity to the VRP.

From the modeling perspective, the OVRP and the VRP are almost identical. The only difference is that we do not consider costs for returning to the depot in the objective function. The problem can be defined by the objective function (10), constraint sets (11)–(18), and setting $\psi_{id}^k = 0$, $\forall\, i \in C$, $\forall\, k \in K$ in (10).

We are not aware of any comprehensive review on open routing; in fact only a handful of papers have been published, and most of them quite recently. Brandão [25] mentions that the first to consider open vehicle routing problem was Schrage [185], and the first to solve Bodin *et al.* [21] in 1981 and 1983, respectively. After that, Sariklis and Powell [179], and Brandão [25] applied heuristic and metaheuristic solution methods to the problem, respectively. It should also be noted that Pisinger and Ropke [164] considered OVRP among others in their

general solution framework. More recently, the OVRP has attracted attention at least by Panagiotis *et al.* [172], Fleszar *et al.* [73], and Zachariadis and Kiranoudis [217]. These papers apply different types of metaheuristic search to the problem. For a list of articles on the OVRP, we refer to the literature review section in the recent paper by Repoussis *et al.* [171].

The conducted studies indicate that even though the OVRP is very similar to the VRP in terms of modeling, the solution methodology has to be adapted, or at least have some adaptation mechanisms, to solve the problem most effectively. This highlights the reason the routing problem variants are typically considered separately. In addition, we note that we are not aware of any articles considering the open vehicle routing problem with pickups and deliveries (OVRPPD). In these problems one would have to serve a set of customers involving both pickup and delivery, but not to return to a central depot.

2.3.3 Fleet Selection

So far we have assumed a fleet that consists of identical vehicles — an assumption not valid on every article already referred. In practice, a typical fleet consists of a heterogeneous set of vehicles, fit to serve different purposes. The problem is to design both the routes and select the fleet to serve the customers with. Typically, the vehicles differ in capacity, cost, speed, and availability. Examples of reasons to utilize a non-homogeneous fleet include the need for special equipment in some vehicles, the cost effectiveness of large vehicles with restrictions for service locations for instance due to their size, and the flexibility of allocation of capacity due to fluctuating demands.

In the literature, the combined design problem of fleet and routes is referred to as the fleet size and mix problem (FSM) and the heterogeneous vehicle routing problem (HVRP) [11], depending on whether an unlimited or a limited number of vehicles is considered, respectively. Moreover, the models may or may not consider fixed costs for vehicles and individual travel costs. In the general case, the fleet consists of a set of vehicles, where each vehicle $k \in K$ has a specific cost structure with fixed costs σ^k for using and variable costs ψ^k for traveling with the vehicle, and a loading capacity ϕ^k. Note that a similar indexing with time windows can be done for heterogeneous vehicle routing with time constraints (FSMTW and HVRPTW). The HVRP with fixed and vehicle dependent travel costs can be defined by the objective function (42), constraint sets (11), (13)–(18), and constraint set (43):

$$\min \sum_{k \in K} \sigma^k \sum_{i \in C} x_{di}^k + \sum_{k \in K} \sum_{i \in L} \sum_{j \in L} \psi_{ij}^k x_{ij}^k, \tag{42}$$

$$\sum_{i \in C} \delta_i \sum_{j \in L} x_{ij}^k \leq \phi^k, \quad \forall k \in K. \tag{43}$$

The objective function (42) minimizes the fixed and variable costs by all vehicles. Constraint set (43) ensures that vehicles do not exceed their maximum capacity.

According to Bräysy et. al [29], among the first to study the FSM were Golden *et al.* [94], who considered both fixed and variable costs of vehicles. The time windowed variant was recently formulated by Liu and Shen [137] and further studied, e.g., by Bräysy et. al [29]. More recently, for example, Brandão [27] and Euchi and Chabchoub [68] have applied a metaheuristic search scheme to the FSM. For a more comprehensive review on routing a heterogeneous fleet of vehicles, we refer to work by Baldacci *et al.* [11].

2.3.4 Compartments

The variants and extensions considered so far have assumed a single capacity limit type and a uniform space within the vehicles. In many cases, in addition to assigning the delivered goods to vehicles, there is a need to consider the actual loading of the trucks that have separate sections, or compartments, for different types of goods. Derigs *et al.* [58] give a definition for a vehicle routing problem with compartments (VRPC), which extends the capacitated vehicle routing problem in several ways: firstly, it considers demand for multiple inhomogeneous products rather than a homogeneous product; secondly, a vehicle consists of multiple compartments rather than one; thirdly, all goods delivered on a tour must be assigned to a compartment; fourthly, certain product pairs must not be loaded together into the same compartments; and fifthly, certain products may not be loaded into certain compartments.

These types of considerations are commonplace, for example, in oil transportation with oil tankers, modern waste collection, grocery distribution, and chemical transportation. In oil tankers, a number of compartments have to be used, since multiple different products are transported simultaneously and cannot be freely mixed. In addition, in some cases there are complex rules of contamination and compatibility between the products and they have to be considered separately for each compartment. Maritime transportation is, however, an extensive topic by itself and we do not consider it further here; instead we refer to a comprehensive survey on maritime routing by Christiansen *et al.* [38]. In waste collection, a number of different waste products, such as biowaste, glass, and metal, have to be transported separately, but perhaps within a single vehicle. When frozen goods are mixed with regular ones within a shipment, there is a need to consider the capacity of the cold-storage separately. Chemical transportations are somewhat similar to oil transportations, both of which can also be transported on road by tank lorries.

If we are allowed to split orders of different products into multiple deliveries at a customer, we may still assume without loss of generality that each customer orders a single product: multiple orders for a single customer can be modeled by multiple customer nodes with a zero distance between them. Formally, let u_{ai}^{k} be the decision variable for loading an order for customer $i \in C$ to compartment $a \in A$ with a capacity of ϕ_a at vehicle $k \in K$ where A is the set of compartments available at each (identical) vehicle. Furthermore, let $U \subseteq C \times A$ be the set of incompatibilities between orders and compartments, and $Y \subseteq C \times C$

the set of incompatibilities between orders. The VRPC can be defined by the objective function (10), constraint sets (11)–(19), and constraint sets (44)–(49) below:

$$\sum_{i \in C} u_{ai}^k \delta_i \leq \phi_a, \quad \forall\, a \in A, k \in K, \tag{44}$$

$$\sum_{k \in K} \sum_{a \in A} u_{ai}^k = 1, \quad \forall\, i \in C, \tag{45}$$

$$\sum_{i \in L} \sum_{j \in C} x_{ij}^k - \sum_{a \in A} \sum_{j \in C} u_{aj}^k = 0, \quad \forall\, k \in K, \tag{46}$$

$$u_{ai}^k = 0, \quad \forall (i,a) \in U, k \in K, \tag{47}$$

$$u_{ai}^k + u_{aj}^k \leq 1, \quad \forall (i,j) \in Y, k \in K, \tag{48}$$

$$u_{ai}^k \in \mathbb{B}, \quad \forall\, a \in A, k \in K, i \in C. \tag{49}$$

Constraint set (44) ensures that the load at each compartment does not exceed its capacity and constraint set (45) makes sure that each order is assigned to only one compartment. By constraint set (46) each order is assigned to a compartment of a vehicle that visits the corresponding customer. Constraint sets (47) and (48) impose restrictions on incompatible pairs of compartments and customers, and customers and customers, respectively. Finally, (49) is the set of integrality constraints.

Note that this formulation allows definition of incompatibilities between individual orders (customers) instead of products. This situation may arise if orders for different customers may not be mixed, even though they represent the same product. These constraints can be present for instance in oil tanker routing due to inaccuracies in measuring amounts loaded and unloaded at the harbors. We assumed in this formulation that the different products ordered by a customer can be split into multiple deliveries, but if this is not the case, we may define sets of delivery locations $C_g \subseteq C$ that represent a single customer g requesting set of orders. These can be grouped to be served consecutively by a single vehicle by imposing constraints (50) and (51) below:

$$\sum_{j \in C_g} x_{ij}^k = |C_g|, \quad \forall\, k \in K, \forall\, i \in L. \tag{50}$$

$$|y_i - y_j| \leq |C_g|, \quad \forall\, i, j \in C_g. \tag{51}$$

Constraint set (50) ensures that orders in a group are served by the same vehicle and constraint set (51) ensures that orders in a group reside successively on a route.

Compartments have received far less attention than many other vehicle routing extensions. Recent research on routing with compartments includes that of El Fallahi *et al.* [69] and Muyldermans and Pang [149], who considered vehicles of several compartments dedicated to one product and compared single-compartment waste collection to a multi-compartment version, respectively. For a recent review on the vehicle routing problem with compartments we refer to work by Derigs *et al.* [58], who also gave the VRPC a general formulation and designed a metaheuristic solution methodology to solve the problem.

2.3.5 Multiple Uses of Vehicles

Whenever operations are planned, there is a time span in which the planning occurs. The notion of *planning horizon* refers to the time from the first activity that is planned to the last activity that is planned. These can be, for example, the departure of the first vehicle from the depot and the arrival of the last vehicle to it, respectively. However, sometimes the given fleet cannot serve all the customers within the planning horizon using a single tour for each vehicle due to limited capacity. On the other hand, customers may have time windows that permit some of the vehicles to serve the remaining after a refill or emptying at the depot. This situation can be considered as a vehicle routing problem with multiple trips and can arise especially in operations where the loads at the customers are considerable compared to the capacity of vehicles.

We observe that the cases involving multiple trips often include constraints for total route lengths, additional breaks for drivers, and complex cost structures in the objective function. We will examine these issues in detail in the subsequent sections.

There are a limited number of studies devoted to multi-trip routing. Recent work involving multi-trip planning include those of Brandão and Mercer [28], [24], who considered legal limits on the drivers' schedules, determined distances in the road network, and included into the objective the costs of drivers, fuel, fleet maintenance and hired vehicles. More recently, Ren *et al.* [169] examined and solved a case involving multiple shifts and overtime considerations. They had to extend the multi-trip model since their trips occur in a single shift, and overtime had to be considered only for the last trip of a vehicle. We are not aware of a comprehensive review on multi-trip vehicle routing problems.

2.3.6 Complex Cost Structures and Dynamic Travel Times

Many of the variants described assume a fairly simple objective function. These functions are usually defined by the distances, the travel times, or the travel costs of the problem. In addition, sometimes a fixed cost on vehicles is applied. In practice, the cost structure is more complex. These types of situations arise especially when employing a (partially) rented fleet where the wages depend on multiple factors. In general, the most common costs relate to that of travel, that of route structure, and the selection of customers.

The cost of travel may vary significantly between vehicles. Vehicles may have an hourly wage plus a cost from mileage, and both of these rates can vary according to the time of day. This applies especially when the wages of drivers are considered. A typical example includes the different wages for overtime. In addition, the cost of travel may also be affected by the existence of other payments; tolls, for example, may have a considerable effect on the cost structure of a routing, especially when these payments are set according to the time of the day (e.g., due to rush hours). Finally, the special equipment and usage of trailers may impact the cost of travel.

The structure of the solution may affect the total costs in several ways. For instance, we may need to impose a lower limit on shift length (for example four hours). Such a limit dictates the minimum that must be paid regardless of the actual length. In addition, the balance between route lengths is important in practical routing. This affects especially long term planning: no driver hired to work full days would want to work only for five hours repeatedly. Typically, this is solved by penalizing the imbalance on route lengths. Note also that the soft time windows mentioned in Section 2.3.1 are implemented within the objective function since they penalize too early or late an arrival with an additional cost.

Finally, if customer selection is part of the problem, that is, the fleet is not large enough to serve all the customers and some has to be left unserved, one approach is to add a profit for each customer according to its priority and maximize the total earnings. This is also included in the objective function of the problem and increases the complexity of the problem.

In some areas, traffic can be a major aspect affecting the design of vehicle based logistics. This is natural due to changes in travel times because of, for example, distinct rush hours. Although these phenomena can relatively easily be taken into account by a human dispatcher, modeling them for optimization has been tedious [110]. Rush hours are, however, not the only factor affecting the length of the operations. In some postal distribution tasks, the time of the day considerably affects the service times at customers: it is faster to handle packets in daylight than in nighttime. Furthermore, to accommodate river transportation in some cities, not all the bridges remain open around the clock. All of these aspects cause changes in the problem structure within the planning horizon and result in dynamic properties for the problem. In practice, in these types of problems there are decisions that affect the validity of other decisions, and these feedback loops make the problem more difficult to solve.

Examples of studies involving complex cost structures arising especially in crew scheduling include those of Vance *et al.* [204] and Gamache *et al.* [82]. Modeling rush hours themselves is complex, and has been subject to studies also outside the vehicle routing research. Dynamic travel times have been recently studied, for example, by Van Woensel *et al.* [212] and Lecluyse *et al.* [134]. We are not aware of work reviewing VRP research involving complex objective functions.

2.3.7 Drivers, Trailers, Equipment, and Compatibilities

In reality, different drivers have different properties, such as constraints on handling different equipment, a license to drive a given vehicle, or knowledge on how to serve a certain customer. The inclusion of the mapping of vehicles on drivers creates additional design decisions. Moreover, different drivers may have differing availability according to rest regulations. Planning schedules for drivers is in practice more complex than designing their routes during the day. A number of legislative regulations on driver shifts, breaks, and competency requirements exist, and they must be conformed to. Typical examples include two breaks and a lunch during a workday, ensuring enough sleep in long-range cargo transporta-

tion, and a number of rest days during a given two-week period. These regulations also govern, e.g., the amount of allowed overtime.

Brandão and Mercer [28] note that in the UK "a driver cannot drive for more than 4.5 hours consecutively without having a break of 45 min. This break can be taken in a single period, two periods (one of 30 min and another of 15 min) or three periods of 15 min." As they note, this constraint is conceptually easy to consider, but very complicated to implement. They considered the breaks by increasing the traveling time to accommodate the breaks in the final solution. In contrast, Rochart and Semet [173] considered the situation in Switzerland where "after 5.5 hours of uninterrupted work, or 4 hours of uninterrupted driving time, the driver must take at least a one-hour break" and considered two breaks by day, one 30-minute break in the morning and one 60-minute break at lunch time by introducing additional fictitious customers. More recently, Goel [93] introduced a model conforming to the recent regulations introduced by the European Union. The model considers the maximum daily driving time between two rest periods, the maximum driving between two breaks or rest periods, the time required for daily rest, the time required for breaks, the maximum time after the end of a rest period until which driver shall have taken a new rest, the maximum weekly driving time between two weekly rest periods, and the maximum time after the end of weekly rest until which a new weekly rest period shall start.

Another design dimension emerges when the fleet includes a set of additional equipment, the most common being trailers. Designing from where to get or where to leave the trailers may be part of the problem. In addition, some customers may not be served while driving with a trailer unit, due to space restrictions. Driving with a trailer may also affect the speed of the vehicle as well as the service times needed at customers. At some customer sites, other special equipment may be required which may not be available on all vehicles, such as in the ready-mixed concrete delivery problem described by Schmid *et al.* [184], where a special pump or a conveyor belt was attached to some of the vehicles. General compatibilities between customer sites and vehicles are considered in a problem variant often referred to as the site-dependent vehicle routing problem (SDVRP) where, unlike in the HVRP, a set of heterogeneous vehicles but with identical costs have to be routed so that customer-vehicle incompatibilities are respected. A recent example of a case considering multiple complex aspects simultaneously, a case study by Ceselli *et al.* [36], includes constraints for multiple capacities; time windows; incompatibilities between goods, depots, vehicles, and customers; the maximum route length and duration; the upper limit on consecutive driving hours; compulsory breaks; an optional delivery replaced by an outside courier; splitting deliveries; and a possibility for open routing. Moreover, they consider an objective function which considers a system of fees on route which depend on the locations visited, the distance traveled, the load of the vehicle, and the number of stops along the route. Indeed, their work highlights the complexity of real-life routing.

The different side constraints in the vehicle routing models are very heterogeneous and appear frequently in case studies, which, quite naturally, describe

several additional real-life inspired rules. Due to the richness of the case studies, a conceptual review on different aspects would be difficult, and we are not aware of such work.

2.3.8 Periodic Routing

In periodic routing, the task is to design a set of routes over a planning period of several days. In classical vehicle routing, the planning horizon consists of a single day in this sense, whereas the periodic vehicle routing problem (PVRP) considers schedules that have dependencies between the days. In practice, this means that each customer needs to be visited periodically, for example every three days, and thus the time of the previous visit constrains the point of time in which the next can be done. Moreover, each customer may have a different frequency, that is, a *visit pattern*. Typical examples include visit patterns of *twice a week*, *every second day*, and *Monday, Wednesday, Saturday*. Periodic routing can also be seen as a special case of multiple time windows, where each time window consists of a single day, and additional constraints are defined on the visits to ensure proper visit patterns. Cordeau *et al.* [46] observed interestingly that the multi-depot vehicle routing problem is equivalent to the periodic vehicle routing problem under certain assumptions.

Applications of periodic routing arise naturally in waste collection; different replenish tasks, such as filling ATMs and vending machines; maintenance routes, such as elevator repairs; and different forms of security services. In practice, the routes can be designed using classical vehicle routing problem models by fixing the delivery schedules manually before solving the resulting set of routing problems. But as also in many other cases, solving the integrated problem yields larger potential for improvement.

According to [77] the PVRP was first formulated by Beltrami and Bodin [16] in 1974, who applied periodic routing to waste collection. Later, Cordeau *et al.* [46] and Drummond *et al.* [63] applied metaheuristic solution algorithms to the problem. For a more comprehensive view on periodic routing and its recent advances we refer to a survey by Francis *et al.* [77].

2.3.9 Further Extensions

There are a number of variants and extensions we do not consider in this work, but we mention some of the notable ones. These include *arc routing*, *vehicle routing with split deliveries*, *inventory routing*, *stochastic routing*, *routing with inter-tour dependencies*, and *multi-objective routing*.

In **arc routing**, instead of visiting a set of nodes in a network, the goal is to visit a set of arcs. The CVRP counterpart of arc routing is the capacitated arc routing problem (CARP), in which the problem is to service a set of streets in a street network using fleet of capacity constrained vehicles located at the central depot. Examples of applications of arc routing include street sweeping, snow-plowing, spreading salt or sand in the winter, and collection of waste. In general,

any CVRP instance in which a number of tasks reside within the same street is an attractive candidate for the CARP. For a recent review on capacitated arc routing problem we refer to a survey by Wøhlk [213].

The **vehicle routing problem with split deliveries (VRPSD)**, unlike classical vehicle routing problems, allows each customer to be visited by more than one vehicle, that is, splitting the deliveries. Early work on the VRPSD was motivated by the fact that additional savings can be achieved by allowing delivery in multiple parts. Moreover, these kinds of instances arise naturally when the demand at customers is larger than the capacity of the vehicles available. For a survey on the split delivery vehicle routing we refer to that of Archetti and Speranza [4].

Inventory routing is a complex extension of vehicle routing. In this variant, not only routing of vehicles but also inventory control decisions have to be made. In the inventory routing problem (IRP), given a set of customers that have an inventory and a fleet of capacitated vehicles, the problem is to decide when and how much to deliver to each customer and which routing to use. The objective is to minimize the total costs resulting from transportation and inventory holding, so that the inventories do not deplete. In many cases, there is a trade-off between inventory holding costs and delivery costs; larger deliveries cost less to transport, but more to store and vice versa. Finding the balance between the two is the key design decision considered by the IRP. For a more detailed introduction to inventory routing we refer to the recent work by Bertazzi *et al.* [18].

Stochastic elements may occur in every aspect of vehicle routing. The demands may be subject to uncertainty, which occurs, for instance, in refilling gas stations and transporting home heating oil. Travel and service times may also change unexpectedly due to traffic conditions as described recently by Lecluyse *et al.* [134]. One other major area of uncertainty is the arrival of new orders, and this is known as the dynamic vehicle routing problem where planning occurs while the vehicles are on route. An introduction and a review on recent research on dynamic routing is given by Larsen *et al.* [132]. Typically, the dial-a-ride problems also belong in this category. Furthermore, in many cases, frequent customers are routed *a priori*, that is, before exact deliveries are known, which means that customers on the route may or may not receive an order that day. This variant is sometimes referred to as an a priori routing problem. A priori routing is reviewed by Campbell and Thomas in [34].

Inter-tour dependencies arise when two or more vehicles have to coordinate their operations jointly. In practice, restrictions on loading stations or limited availability of processing capability at the return depot can lead to a situation where feasibility of a route depends on the arrival time and load of each vehicle. These instances can be modeled using inter-tour constraints. Recent work addressing this issue includes that of Hempsch and Irnich [104] who use a giant tour representation for the routing problem. In this representation, each individual route is considered a part of a single "giant" route, and inter-tour resources, such as processing capability, are considered as a global resource. Moreover, they used a resource-constrained path formulation, which we discuss in the subsequent sections.

Multi-objective routing is concerned with multiple conflicting criteria (objectives) on which to evaluate the routes. Many of the problems encountered in the industry are multi-objective in nature and exhibit this directly in the cost function. But in addition to the different cost factors, minimization of aspects such as waiting time, walking distance, the perceived risk, the total route length, and the number of constraint violations may be present and need to be taken into account. For a recent review on multi-objective routing we refer to that of Jozefowiez *et al.* [119].

2.4 Modeling Optimization Problems

The previous sections presented a number of problem formulations. Now, formulation of a problem is a prerequisite for modeling it, and while we could use the given mathematical programming formulations also as *models*, this is usually not the best approach for routing problems as we will observe in Chapter 3. This section discusses the modeling aspect of vehicle routing in more detail and builds the bridge towards solving VRPs.

2.4.1 Modeling and Optimization Process

Where *formulating* the problem consists of identifying the problem in reality, that is, stating the questions that need answers from the decision maker, **modeling** involves transforming the required aspects of reality into a mathematical abstraction that *can be operated on by algorithmic means*. The mathematical abstraction, or model, can originate from a similar problem described in the literature, can be described using well known techniques, or in some cases can be described by a new modeling technique. Usually it is the case that existing models will suffice and the problem is described formally, for example, by using an algebraic modeling language such as a mathematical programming language (AMPL) or a general algebraic modeling system (GAMS) [120].

Different taxonomies have been proposed for optimization models, and one classification is to divide them into the following four categories: mathematical programming models, combinatorial optimization models, constraint satisfaction models, and nonanalytic models (see, e.g., Talbi [195]). Vehicle routing problems are a subset of combinatorial optimization problems, which have their roots in combinatorial analysis. The definition of combinatorial analysis is best given by Lawler [133] as *the mathematical study of the arrangement, grouping, ordering, or selection of discrete objects, usually finite in number*. Therefore, *combinatorial optimization* can be characterized as the act of searching the best of such arrangement, grouping, ordering, or selection. In this work, we are concerned with certain types of combinatorial optimization problems. To illustrate this, an elementary and somewhat simplified classification of optimization problems is depicted in Figure 2.

Optimization problems can first be divided into deterministic and stochas-

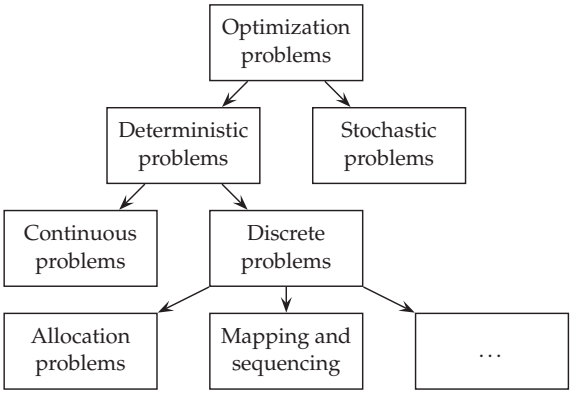


FIGURE 2 An elementary classification of optimization problems.

tic. Stochastic problems involve some form of uncertainty in the objective, constraints, or decision variables. This stochasticity does not, however, refer to random elements in the solution methodology; this classification is based solely on problem characteristics. In contrast, deterministic problems contain no elements of uncertainty. The second major criterion is the division into continuous and discrete problems. Whereas continuous problems may contain real-valued decision variables in the objective, constraints, or both, discrete problems have strictly discrete values[3], which, as we have observed, leads in many cases into enumerative solution approaches. Discrete problems are then further divided into different types, including, for instance, combinatorial optimization problems such as allocation problems, sequencing problems [211], and especially those involving decisions of both *mapping and ordering*[4]. In these problems, all the decisions are encoded into mapping a set of discrete elements to another set of discrete elements, and forming an ordering of the former for each of the latter. This class of deterministic, discrete mapping and ordering problems is our primary concern in this work.

Note that *scheduling problems* are also clearly in this class of mapping and ordering problems. They contain inherent differences in the problem structure; most notably lacking a spatial dimension, and because of this typically attract different solution methods. There is, however, no reason from the modeling viewpoint to differentiate between routing and scheduling problems in a generic setting.

The requirement that a model must be operable by algorithms distinguishes

[3] Constraint and objective *functions* may still be continuous, linear or nonlinear.

[4] The term mapping is used here interchangeably with allocation, and the term ordering, with sequencing.

models from formulations. When modeling the problem, we need to consider additional aspects and it is, generally, not enough to build a naive, albeit exact, mathematical statement of the problem[5]. As the elementary classification suggested, a class of combinatorial optimization problems can be seen as decisions to map and order discrete elements. This abstract description of a combinatorial optimization problem does not yet consider the two central issues of modeling: *encoding* and *data*.

On one hand, collecting and eliciting the **data** describing the real-life situation subject to optimization is crucial to the quality of the solution. Although input data is not considered a part of the model, gathering, sanitizing, and processing data for the model can be seen as a part of the *modeling process*. Missing or invalid data mischaracterizes the problem and optimization may result in a solution to the wrong problem. On the other hand, **encoding**, or problem representation, is needed to describe how the problem is seen by the algorithms. A problem formulation has a number of alternative encodings, typically divided into two main classes: *linear encodings* such as binary, permutation, or mixed integer; and *nonlinear encodings*, such as trees and graphs. Not all encodings are equally good, however. There are at least three characteristics required from a problem representation: *completeness*, *connectivity*, and *efficiency* [195]. Firstly, the encoding must be complete, i.e., able to represent all the possible solutions to the problem. Secondly, the representation must yield a search path for the search algorithms between any two solutions, which ensures the possibility of finding an optimal solution. Finally, the representation must be efficient, that is, easily manipulatable by the search operators. These properties greatly affect the third phase of decision making — solving the formulated problem.

Solving combinatorial optimization problems is done by traversing the set $\mathbf{S}$. The set $\mathbf{S}$ is defined by the values of decision variables of the problem. The objective is, therefore, to find optimal values for the decision variables by evaluating the value of their combinations. More specifically, we must find $s^* \in \hat{\mathbf{S}}$ s.t. $\mathcal{C}(s^*) \leq \mathcal{C}(s) \quad \forall s \in \hat{\mathbf{S}}$. When the solution is obtained, the encoding has to be decoded into a concrete solution to the problem. In vehicle routing, this means constructing the actual routes within the road network from the obtained result.

Decoding of a solution to an optimization problem is made possible by providing a *mapping* between the decoded and encoded solutions. Such a mapping can be classified according to its cardinality; a mapping can be one-to-one, one-to-many or many-to-one, depending on how many instances correspond to each other in both representations. One-to-one mapping is the most used, since it is straightforward to translate the encoded solution back to the original. Sometimes, however, such a mapping cannot be constructed and one-to-many or many-to-one mapping has to be used. In one-to-many mapping, the same solution can be represented by several encodings, and this redundancy increases the size of the search space. In contrast, in many-to-one mappings, several solutions are represented by the same encoding, meaning that some information on the solution

[5] This is an idealized process; in practice, formulation and modeling processes may be interleaved in an iterative fashion.

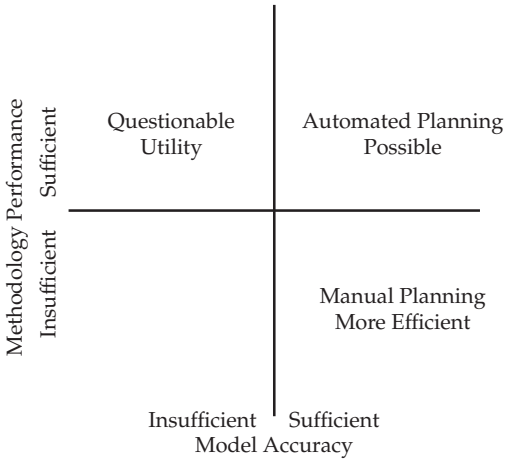


FIGURE 3 The relationship of model accuracy and solution methodology performance.

is excluded from the encoding. This decreases the size of the search space, but the decoding of such solutions requires more effort. [195] An example of one-to-many encoding is the vehicle routing problem where vehicles are identical: no matter which vehicle actually serves which route within the problem encoding, the solution remains the same. In contrast, an example of many-to-one encoding is the TSP, due to its rotational invariance: one encoding represents multiple solutions, each starting at a different location.

The final step of the decision making process involves **applying** the obtained solution into practice. In an ideal world, the constructed model would contain all the necessary information of the real-world problem, and the result of the optimization would be directly applicable to practice. However, usually many of the details have to be omitted, for example, due to the computational complexity of solving the problem. Thus, the decision making using optimization involves finding a balance between the model accuracy and the optimization solution methodology performance. Typically, the improvements in one degrade the quality of the other. This relationship between modeling and solving is depicted in Figure 3. When the modeling is sufficient in a sense that the result of the optimization can be applied to practice with little or no modifications, the optimization system describes the reality with enough accuracy. The performance and the robustness of the solution methodology can be regarded sufficient when they outperform the human decision maker in quality. Both of these criteria need to be met for the system to be useful. Although the exact problem statement and correct data are prerequisites for a correct problem description, in this work we are especially interested in describing the problem to algorithms in an efficient way. It is often the *design of the encoding* that plays the key role in defining both the modeling accuracy and the solution methodology performance.

2.4.2 Design Considerations in Model Encodings

Encoding is the element connecting the solution process into the problem formulation: on one hand, we must be able to express the problem accurately enough, and on the other hand be able to efficiently operate on the problem algorithmically. The aim is to find somehow "meaningful" constructs in both the modeling and solving points of view. From the algorithm perspective, the search for meaningful constructs is concerned with the efficiency of traversing the search space. Now, the design of *search algorithms* is concerned primarily with designing the *moves* (the set of possible moves of an algorithm defines its *search neighborhood*) in the search space. In contrast, the design of encoding is concerned with the *evaluation of the constraints and the objective function as computationally efficiently as possible* to make these moves "easy". These two aspects largely define the efficiency of traversal of the search space.

The traversal of the search space is performed by altering the values of the decision variables. It is, however, noteworthy that a single decision variable is typically limited by only a subset of constraints of the problem. This implies that given a search operator that alters the values of a (possibly small) subset of the decision variables at a time, only a subset of constraints have to be evaluated when moving in the search space. This partial evaluation makes the traversal of the search space efficient in terms of computational complexity, and, in fact, in many cases the complexity of the constraint evaluation defines the computational complexity of the whole solution process itself.

As the values of the decision variables form the search space of the problem, it is essentially a product space of the ranges of those variables. Constraints, on the other hand, are evaluated by computing the values of the *constraint functions* in the point of the search space, thus forming a transformation from the search space into another space. In this context, this co-domain is referred to as a *constraint space*. In order to examine the structure of the constraints, it is useful to think of them as consisting of two parts: firstly, from the constraint function from the search space into the constraint space, and secondly, from a set of dependencies governing the permissible values of these functions, i.e., *rules*. These rules define the feasible region in the constraint space. As the *objective* is also a function in its own right, we may form a dimension for its values in the constraint space. This dimension is not restricted by the rules, but instead is used to measure the value of the current point in the search space. This generalization is also useful from a practical point of view; the evaluation of the objective function may be equally important to the performance of the system as the evaluation of constraints, and this way we do not have to consider the objective separately.

The number of constraints, their nature, and interconnectedness define the computational complexity of evaluating the feasibility of a move. Typically, the more global the constraint — that is, more decision variables constrained — the more computationally demanding the evaluation. The constraints are evaluated by calculating the values of the constraint functions in the point in the search space, and evaluating their results against the rules defined on these values. As

mentioned, these rules can be equations, inequalities, and logical dependencies. Consequently, the efficiency of constraint evaluation depends on the assumptions that can be made of the structure of the constraints.

In addition to regular constraints, there may also be functions whose sole purpose is to enhance the efficiency of the constraint evaluation. That is, functions that do not alter the feasible region in the search space, but instead change the topology of the constraint space in a useful way. An introduction of the notion of *slack* on time and capacity is an example of such functions. These functions compute the distance to the "edge" of the feasible region and introduce additional dimensions to the constraint space. These dimensions may be traversed easily to check whether we would leave the feasible region by applying a given move.

As mentioned, in addition to the efficiency of constraint and objective evaluation, there is another side to the performance of the solution process from the encoding point of view: the design of search neighborhoods and the resulting convergence rate of search algorithms. Both of these aspects have to be taken into consideration. The search neighborhoods of algorithms are defined on the search space, whereas constraint evaluation is determined by the structure of the constraint space. These two influence each other profoundly; the type of constraints influences the optimal size and type of the search neighborhood by defining the rate at which the search can be executed, and the search neighborhood defines the number and type of constraint evaluations that is required during the search. One could argue that this interaction between the traversal in the search space and the traversal in the constraint space is central to the design of suitable encodings.

2.4.3 Graph Encodings for Routing Problems

From a solution methodology point of view, perhaps the most suitable encodings for the VRP are based on graphs. This is due to the fact that one of the most used solution method components, local search, often operates on neighborhoods defined on nodes or arcs of a graph (see Chapter 3 for more details). A graph encoding can be constructed as follows. Let a simple graph $G = (V, E)$, where V is the set of nodes, and $E = V \times V$ is a set of arcs connecting the nodes. A path $P = \langle p_1, \ldots, p_q \rangle$, where $p_1, p_q \in V$, is a finite sequence of nodes in G, and $e_{ij} \in E$ an arc connecting the nodes p_i, $p_j \in V$. For convenience, we denote here $e \in P$ and $p \in P$ for both arcs and nodes on the path, respectively. Moreover, let $\mathcal{C} \colon E \to \mathbb{R}$ be the function describing the cost of traversing edge $e \in E$. Note that that we consider a *directed* graph G, thus in general $\mathcal{C}(e_{ij}) = \mathcal{C}(e_{ji})$ does not hold.

These constructs can be used for two distinct approaches in graph-based modeling: *partitioning to multiple tours* and *forming a giant tour*. In the former, each vehicle $k \in K$ is associated with a path P_k, whereas in the latter all paths are concatenated into a single path P on G.

For the sake of illustration, consider a simple VRPTW. Recall that the objective is to visit a scattered set of customers from a central depot using a set of vehicles in a way that each customer is served, any vehicle does not exceed its capacity, the time windows are respected at each customer, and the total distance

traveled by all the vehicles is as short as possible. An example of such a vehicle routing instance solved by using four vehicles is depicted in Figure 4.

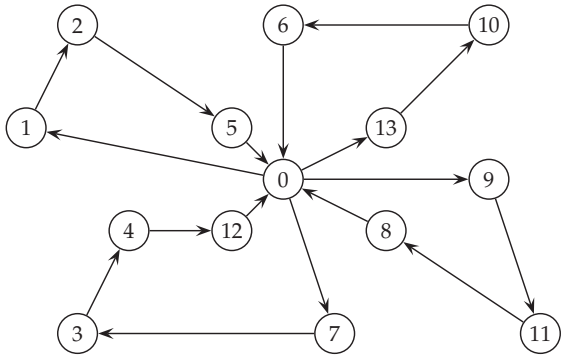


FIGURE 4 A vehicle routing problem instance with 13 customers and four vehicles.

The **partitioning-based encoding** relies on the notion of Hamiltonian cycles. The nodes of the graph are partitioned by forming multiple Hamiltonian cycles, each corresponding to a route of a vehicle. Each node must be part of some cycle, and the depot node is the only one allowed to occur in more than one cycle. In fact, each path must contain the depot node as its first node. For clarity, we may also define that a valid route is a path starting from the depot node and ending at it. Thus in the illustration in Figure 4, the paths are $\langle 0, 7, 3, 4, 12, 0\rangle$, $\langle 0, 9, 11, 8, 0\rangle$, $\langle 0, 13, 10, 6, 0\rangle$, and $\langle 0, 1, 2, 5, 0\rangle$.

Let p_i be the i^{th} node in a path. Following the notation used in mathematical programming formulations, z_{p_i} is then the load at p_i, and similarly, w_{p_i} the time when the vehicle starts its service at that node. Furthermore, let $z_{p_1} = 0$ and $w_{p_1} = 0$ for each path. Now $z_{p_{i+1}} = z_{p_i} + \delta_{p_{i+1}}$ and $w_{p_{i+1}} = \max(\alpha^{\tau}_{p_{i+1}}, \tau_{p_i,p_{i+1}} + w_{p_i})$, $\forall\, i \in \{1, \ldots, q-1\}$ where q is the length of the path. Thus a path is feasible when $\alpha^{\tau}_p \leq w_p \leq \beta^{\tau}_p \;\forall\, p \in P_k, \forall\, k \in K$, and $z_p \leq \phi \;\forall\, p \in P_k, \forall\, k \in K$. Note that similarly to (12), due to the monotonicity of capacity in the VRPTW, it is enough to check $z_{p_q} \leq \phi$ on every path $P_k = \langle \ldots, p_{q-1}, p_q\rangle, k \in K$. Finally, the objective is to minimize the total sum of costs on each path, i.e., $\min \sum_{k\in K} \sum_{e\in P_k} \mathcal{C}(e)$.

In contrast, the so-called **giant tour encoding** represents the whole solution as a single Hamiltonian cycle [80]. Each vehicle is associated to distinct origin and destination nodes, similarly to the GPDP. Let 0^+_k denote the start node and 0^-_k the end node of vehicle k. The solution to the example is given by path $\langle 0^+_1, 7, 3, 4, 12, 0^-_1, 0^+_2, 9, 11, 8, 0^-_2, 0^+_3, 13, 10, 6, 0^-_3, 0^+_4, 1, 2, 5, 0^-_4, 0^+_1\rangle$. Note that the end nodes of vehicles are connected to the start nodes of the subsequent vehicle. Traversing these special arcs reset the capacity and time collected along the path, but not the total costs, thus resulting in similar conditions on feasibility, i.e., $\alpha^{\tau}_p \leq w_p \leq \beta^{\tau}_p \;\forall\, p \in P$, and $z_p \leq \phi \;\forall\, p \in P$, with the objective function $\sum_{e\in P} \mathcal{C}(e)$.

Now, the question is how to encode the variants and extensions formulated

in Sections 2.2–2.3. We notice that there are similarities in the formulation of consistency constraints (28) and (38). This is natural due to the nature of the constraints — for instance, (29) and (39) — imposed on the decision variables: they state upper and lower bounds on each node *accumulated* that far, and in fact, are not that much decision variables themselves. They are instead highly dependent on the main decision variables x and y. Moreover, observing the flow constraints (13)–(15) has a straightforward interpretation on networks: *traversing a path*. It is, therefore, quite natural to model the whole problem as accumulated values over the path traveled by the vehicle and check each node for the feasibility and objective value.

A suitable construct for modeling constrained traversal on a path does, in fact, exist: resource-constrained paths (RCPs). The RCPs have been examined in detail in the context of routing by, e.g., Irnich [110]. Finding optimal RCPs is related to solving another problem, the so-called shortest weight-constrained path problem (SWCPP). In the SWCPP, in addition to length, each edge has a weight associated to it, and the task is to find the shortest path between two nodes so that the sum of the weights does not exceed a predefined value. By generalizing the SWCPP to multiple resources and both a lower and an upper bound on accumulated values, we obtain the shortest path problem with resource constraints (SPPRC); and furthermore, if the paths in the SPPRC must be simple[6], we obtain the elementary shortest path problem with resource constraints (ESPPRC). Solving the ESPPRC occurs in column generation approaches as a subproblem (see, e.g., Ceselli *et al.* [36]). In fact, the usage of resource constrained paths also in heuristic methods may bridge the gap between exact and heuristic approaches in solving routing problems under constraints.

Resource-constrained paths can be formulated as follows. Let resource vector $\Gamma = (\Gamma_1, \ldots, \Gamma_{\bar{\gamma}}) \in \mathbb{R}^{\bar{\gamma}}$, where $\bar{\gamma}$ is the number of resources. We define that

$$\Gamma \leq \hat{\Gamma} \Leftrightarrow \Gamma_\gamma \leq \hat{\Gamma}_\gamma, \quad \forall\, \gamma \in \{1, \ldots, \bar{\gamma}\}. \tag{52}$$

For two resource vectors α and β, the interval $[\alpha, \beta]$ is defined as the set $\{\Gamma \in \mathbb{R}^{\bar{\gamma}} \colon \alpha \leq \Gamma \leq \beta\}$. Resource windows associated with a node i are denoted by $[\alpha_i, \beta_i]$ with $[\alpha_i, \beta_i] \subset \mathbb{R}^{\bar{\gamma}}, \alpha \leq \beta$.

The changes in resource values at each edge $(i, j) \in E$ are given by resource extension functions (REFs), which are defined as

$$\mathcal{F}_{ij} = (\mathcal{F}_{ij\gamma})_{\gamma \in \{1,\ldots,\bar{\gamma}\}} \colon \mathbb{R}^{\bar{\gamma}} \to \mathbb{R}. \tag{53}$$

A resource constrained path is a path which is constrained by a set of resource windows. A path is resource feasible when all the accumulated values fall within the given resource windows, that is, path P is resource feasible when for each consecutive vertex p_i and p_j on $P = \langle \ldots, p_i, p_j, \ldots \rangle$

$$\exists\, \Gamma^{p_i} \in [\alpha_{p_i}, \beta_{p_i}], \tag{54}$$

[6] A path is simple if no node appears more than once in the path.

s.t.

$$\mathcal{F}_{p_i,p_j}(\Gamma^{p_i}) \leq \Gamma^{p_j}. \tag{55}$$

Resource extension functions have properties that affect the efficiency of the local search operations within a problem defined by them. The most important properties are those that enable a constant time feasibility and objective evaluation. For example, the notion of availability of slack on a route has been successfully exploited in local search schemes in the VRP since first introduced by Savelsbergh [181]. In the context of time windows, the idea is to compute the maximum amount of time by which service at a customer can be postponed without any time window violations along the route. This enables a constant time local feasibility check of an edge or a node-exchange operation. This can be generalized to all resources (not just capacity and time) collected along the route. Many classical constraints can be modeled with resource extension functions that meet these properties, and each of those resources can therefore be checked for feasibility in constant time. Furthermore, classical REFs include the objective function of most vehicle routing variants, and this enables constant time objective function evaluation in local search.

Both of the encodings presented here can employ RCPs, and especially the giant tour encoding has gotten attention recently, due to its natural way of describing some of the inter-route dependencies. RCPs and REFs have been applied to the VRP setting quite recently by Irnich *et al.* in [113] where the concept of sequential search to vehicle-routing problems using resource extension functions was applied, and in [109] where a unified modeling and solution framework for vehicle routing and local search-based metaheuristics was formulated. Resource extension functions can also be used to model a number of VRP variants and extension. This has been examined in [110], where a formulation for (multiple and soft) time windows, multiple capacities, load-dependent costs, pickups and deliveries, limited waiting times, time dependent travel times, and complex cost functions are provided. More recently, Irnich *et al.* applied resource constrained paths to eliminate arcs from routing problems in [112]. RCPs have, however, limitations: they cannot express, for instance, compartment loading decisions without additional constructs. Modeling constructs are discussed when we introduce the modeling approach developed in this work — similarly based on RCPs — later in Chapter 5, and the different techniques for local search are examined in more detail in Chapter 6.

2.4.4 Constraint Satisfaction Models for Routing Problems

A somewhat different strategy for modeling routing problems is to build so-called *constraint satisfaction models*. The constraint satisfaction models are optimization models based on constraint satisfaction and propagation. These models are constructed and solved using constraint programming (CP), which has its roots in the research of artificial intelligence. Constraint programming has been applied into more constrained vehicle routing problems as metaheuristics seem to work better on loosely constrained problems. This may be due to the fact that

they traverse according to the objective function and their movement is restricted by the feasible region. Constraint programming, however, has the opposite, complementary, strategy. The search is guided by constraints and reasoning about them. CP also has quite a rich and natural way of declaratively describing constraints, which may be tedious in integer programming models. Also, mathematical programming models rarely communicate constraints in a form where they can be used to direct the search. Examples of constraints in CP models include not only those of classical inequalities but also more verbose ones, such as `all_different`($x_1, x_2, \ldots, x_n$), which requires that all variables $x_1, x_2, \ldots, x_n$ have distinct values; and `sort`($x_1, \ldots, x_n$; $y_1, \ldots, y_n$), which expresses that the n-tuple ($y_1, \ldots, y_n$) is obtained from the n-tuple ($x_1, \ldots, x_n$) by sorting the elements in nondecreasing order [20]. Due to scope limitations, we cannot describe in detail the constraint satisfaction models for routing problems, but refer to a VRPTW model described by Kilby *et al.* [124].

Constraint programming has been used to solve combinatorial optimization problems, and broadly speaking, CP may be well suited for scheduling problems, especially resource constrained ones, or other highly combinatorial problems, e.g., problems involving disjunctions, for which the integer programming model tends to be too large or have a weak continuous relaxation [20]. Also, interestingly, resource-constrained paths are not unlike, e.g, the path constraints used in the CP approach by Backer *et al.* [8]. Vehicle scheduling and routing problems are, therefore, one interesting candidate for constraint programming techniques.

As also noted by other authors working on the VRP and CP, Gendreau [86] highlights two major advantages of constraint programming: *expressiveness* and *flexibility*. CP allows introduction of complex side constraints with less effort than mathematical programming models. We will discuss the results obtained from optimizing constraint satisfaction models in Chapter 3 when examining the usage of exact methods in vehicle routing.

For a more comprehensive review and an introduction of applying constraint programming to vehicle routing, we refer to the work of Kilby and Shaw [123], and to constraint programming in general, that of Bockmayr and Hooker [20].

2.5 Summary of Routing Problem Models

The vehicle routing problem is a decades old combinatorial optimization problem that considers a fleet of vehicles and a set of geographically distributed customers, each of which have to be visited. During the years, the problem has been studied intensively and gradually refined into a more accurate description of problems occurring in real-life design of operations.

Mathematical programming techniques were used in this chapter to communicate the different problem variants and their subsequent extensions developed and formulated in the literature. These variants included those involving

multiple depots, backhaul operations, mixing pickups and deliveries. The extensions considered, for example, time windows, heterogeneous fleet, and other more complex constraints and cost structures. The downside of more accurate models is their increased computational complexity, and this has to be considered when formulating a routing problem.

The problem formulation is, however, just a statement of the problem, and does not necessarily provide a suitable foundation for solving the problem. Thus, we discussed in this chapter the design of efficient encodings along with a number of algorithmically operable encodings for routing problems.

Based on the discussion, we observe tension in two respects. First of all, we note that the models in routing problems are often described as linear integer programming formulations, but solved within nonlinear encodings. Secondly, the encoding that is suitable for algorithmic manipulation, may not be a very fitting representation of the *domain* in which the routing occurs. That is, even though we raised the abstraction from the mathematical statements into the problem encodings, we are still missing a connection to the entities that we typically operate on when designing the logistic operations.

There is a need for conceptual models, or in software engineering terms, *domain models*, that describe the problem at a suitable level of abstraction from the logistic designer perspective. However, before we move from encodings into domain models, we need to review the algorithms operating on the encoding level. This is the subject of the next chapter, where we provide an overview on the approaches used in solving the vehicle routing problem and its variants. This provides a solid base for developing a suitable modeling approach also from the solution methodology perspective.

3 VEHICLE ROUTING SOLUTION METHODOLOGY

"The significant problems we face cannot be solved at the same level of thinking we were at when we created them."

— ALBERT EINSTEIN

In this chapter, we provide an overview of both exact and heuristic solution methodology in solving vehicle routing problems. The literature on these methods is far too extensive for an in-depth analysis in this work, but we aim to highlight the general trends and to provide an overview on different algorithmic approaches in combinatorial optimization as well as to point to recent results from applying them into vehicle routing. This chapter is structured as follows. In Section 3.1, we briefly talk about computational complexity and algorithm evaluation. In Sections 3.2 and 3.3 we review some of the recent work on applying exact and heuristic solution methods to vehicle routing, respectively. Section 3.4 discusses metaheuristic search, and reviews different metaheuristics applied into routing problems in scientific literature. Finally, Section 3.5 concludes by attempting to form a more generic view on current solution methodology for providing a base for discussion on future directions.

3.1 Problem Complexity and Method Evaluation

Computational complexity refers to the inherent difficulty of solving a problem algorithmically [83]. The computational burden of solving a given problem increases along with the problem size, and the function describing the number of algorithmic operations needed for a given size is referred to as a *time complexity function*. The time complexity function is used to denote the efficiency of an algorithm, and when the increase in number operations needed is too rapid, the computation of anything but small problem instances becomes intractable. In many cases, the most difficult problems encountered in practical applications are

much larger than can be solved by an arbitrary complex algorithm. Thus, we are interested in finding the best possible algorithm for a given problem.

Computational problems can be classified according to the complexity of the algorithms that can be used to solve them exactly[1]. An algorithm is *polynomial-time-bounded* if, in the worst case, the number of algorithmic operations increases only polynomially with the problem input size. The set of problems on which such an algorithm exists (is known to exist) is called $\mathcal{P}$. In contrast, an algorithm is *nondeterministic polynomial-time-bounded* if, in the worst case, the number of algorithmic operations increases only polynomially with the problem input size *with a machine capable of computing any number of steps in parallel*. The set of problems on which such an algorithm exists is called $\mathcal{NP}$. The problems that are at least as hard as the hardest problem in $\mathcal{NP}$, are called $\mathcal{NP}$-hard, and any $\mathcal{NP}$-hard problem that is also in $\mathcal{NP}$, is called $\mathcal{NP}$-complete [83]. As there is no physical computer that can perform an arbitrary number of operations in parallel, problems that are $\mathcal{NP}$-hard or $\mathcal{NP}$-complete take an exceedingly long time to solve exactly.

The traveling salesperson problem is known to be $\mathcal{NP}$-hard [83]. The decision problem of the TSP on the other hand is $\mathcal{NP}$-complete. Since the vehicle routing problem generalizes the traveling salesperson problem, it is, along with all its variants, also $\mathcal{NP}$-hard. Finding the optimal RCPs is known to be $\mathcal{NP}$-complete, and solving the ESPPRC, as proven by Dror [62], $\mathcal{NP}$-hard. No polynomially bounded exact algorithm for the VRP is, therefore, known to exist. This makes solving the vehicle routing problem computationally difficult. Typically, if the size of a routing problem increases beyond 100 customer points, one has to rely on approximate methodology. In the subsequent sections, we briefly review different solution approaches, both exact and approximate.

Since we are interested in finding the best algorithm for a given problem[2], an appropriate evaluation criterion must be established. The evaluation of solution methods is typically done according to a number of criteria, including efficiency (in terms of both time and space versus the quality of solutions), robustness, and simplicity. While theoretical complexity analysis gives an estimate on the real performance of an algorithm in terms of the time and space used, empirical analysis is required to evaluate the method in terms of solution quality. This includes statistical analysis on different types of problems and on different problem characteristics and comparison of the results with solutions obtained by different methods on the same instances. A commonly used practice in the VRP research is to compare solutions within a well-established set of public test problems. The relationship between different points of evaluation is depicted in Figure 5. When assessing the solution quality, a theoretical estimate of a lower bound may be used to estimate the distance to the optimal solution, which, in general, is not known. The best known solution is not usually the optimal solution, but the gap between the best known and obtained solution is often used in vehicle routing when assessing the quality of the new solution methodology.

1 That is, by ensuring that the best solution is found.

2 In general, not specifically in this thesis.

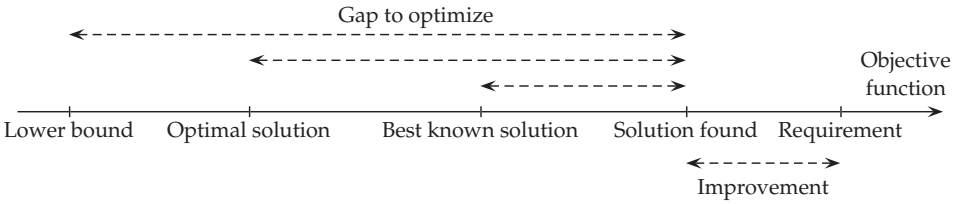


FIGURE 5 Different options for a comparative evaluation of an obtained solution [195].

The found solution, although not optimal, may still provide some improvement to the current situation or the required result.

Robustness of a method is defined as its ability to tolerate small changes in problem data and structure without losing its efficiency [195]. Method simplicity can be measured, for example, by the ease of the implementation, and flexibility (when contrasted to robustness) by the effort needed to adapt the methodology to different situations. In general, three families of solution approaches exist for the VRP, all differing in their qualities. These are *exact methods*, *heuristic methods*, and *metaheuristic methods*.

3.2 Exact Methods

By exact methods we refer to a class of optimization methods that, given an unlimited time, are guaranteed to find the optimal solution to a given problem if one exists. Two classes of exact methods have been applied into vehicle routing: *mathematical programming* and *constraint programming*.

Mathematical programming approaches can be classified into three categories: *enumerative*, *relaxation and decomposition*, and *cutting plane and pricing* [195]. Enumerative algorithms include those of the branch-and-bound (BB) family, which are essentially tree-search approaches based on the idea of dividing the problem into subproblems and solving each individually. Relaxation and decomposition techniques include, for instance, Lagrangian relaxation which removes some constraints and penalizes them in the objective function, and Bender's decomposition which fixes a set of variables and solves the resulting problem iteratively. Cutting plane techniques attempt to prune the search space by introducing additional constraints. These methods are often combined to strengthen the branch and bound approach.

Solving the vehicle routing problem with exact methods has been widely studied, and many of the approaches are based on linear programming (LP) solvers. LP solvers utilize, for instance, the well-known simplex-based branch-and-bound methods [200]. It has been noted, however, that in most VRP cases these methods are able to solve only small problem instances due to the fact that

the LP relaxation of the problem provides a weak lower bound [48]. This may be explained by the nature of vehicle routing problems; the linear parts of the problem, namely travel costs, are highly sensitive to the binary decision variables which are responsible for the combinatorial nature of the problem. These make linear programming solvers inefficient for solving the VRP directly, and this has increased interest in more powerful relaxation and mathematical composition techniques.

The two common variants of the VRP, the capacitated vehicle routing problem and vehicle routing problem with time windows, have been studied intensively [202], and in general, we observe that the following exact solution techniques have been the most successful. For the CVRP, the branch-and-bound technique has been relaxed by dropping for example the vehicle load constraints, and by forming different spanning tree based relaxations. In addition, the branch-and-cut method — a combination of the branch-and-bound and cutting plane — has been shown to outperform the branch-and-bound [48]. For the VRPTW, Lagrangian relaxation techniques, column generation — such as Dantzig-Wolfe decomposition, and the branch-and-cut have been used successfully [48]. In general, it must be pointed out that many of these methods typically rely on some assumptions on the structure of the given problem, which undermines their robustness and makes them less useful in practice.

Constraint programming models are solved, like LPs, by specific solvers which traverse the program search tree. In contrast to search-based methods, CP traverses the search space using constraints and is therefore able to exploit the structure of the problem in a way the mathematical programming approach cannot. CP, however, does not benefit from the continuous relaxation often available to mathematical programming algorithms [20]. A constraint program is traversed by fixing a subset of variables at each step so that constraints are not violated. The search tree can also be pruned using consistency checking and constraint propagation. They attempt to detect inconsistencies as soon as possible to avoid running into a dead end in the search tree. Optimization of a constraint program involves typically a branch-and-bound scheme, where the objective function is evaluated at each step and its value is used to prune the search if suboptimal branches are detected [176].

Recent reported results from exact methods indicate that mathematical programming algorithms can solve CVRP instances with up to 135 customers. See, for example, Fukasawa *et al.* [79], Baldacci *et al.* [12], Augerat *et al.* [5], and Ralphs *et al.* (up to 100 customers) [167]. Similar numbers are reported about the VRPTW; for instance, Bard *et al.* [13] reported solving all the 50 customer and a subset of the 100 customer benchmark instances by Solomon [190] to optimality. Interestingly, the TSP with pickups and deliveries has been relative difficult to solve exactly: an instance of 35 pickups has been solved to optimality only recently [65]. Constraint programming has not been applied to vehicle routing as widely as mathematical programming, but some effort has been devoted to the subject. Backer *et al.* [8] combined local search operators and constraint programming to solve the vehicle routing problem, and used path constraints in modeling

the capacity constraints. They noted that modeling of other constraints, such as time, weight, volume, and distance would be straightforward. They reported solving 100–200 customer instances to within 10% of the best known solution in a few seconds. A similar approach was taken by Shaw [188] who in contrast used larger neighborhoods. Constraint programming was used as a component within a broader solution methodology, e.g., by Backer *et al.* [9], Rousseau *et al.* [177], and Kilby *et al.* [124]. Kilby *et al.* report that adding additional side constraints decreases the performance of the conventional techniques, but does not affect the efficiency of the CP-based methods. More recently, Berbeglia *et al.* [17] addressed a dynamic dial-a-ride problem using a solution approach with a constraint programming component.

Given that the size of VRPs solvable in a reasonable time by exact methods is rather limited, in practice, we have to resort to inexact solution methodology. Thus, for the remainder of this thesis, we concentrate on the two main classes of approximate methods: *heuristics* and *metaheuristics*.

3.3 Heuristic Methods

Heuristic methods are a subclass of approximate methods, and do not guarantee optimality, or, in fact, any qualitative property of the solution, but instead tend to produce, rather fast, solutions that satisfy the given requirements for the solution quality. Heuristic methods are also typically reasonably easy to implement.

Heuristic methods employ problem-specific knowledge for generating solutions from other solutions. In vehicle routing, heuristic methods can be classified into two main categories: route construction and two-phase heuristics. Route construction heuristics typically start from an empty solution and build the routes step by step by inserting customers until all the customers have been assigned on a route. One of the classical route construction heuristics is the Clarke and Wright algorithm [39], where each customer is initially served by a separate vehicle and these routes are then combined according to savings obtained by the combination. Another generic construction approach is to gradually assign customers to routes according to an insertion cost (the so-called cheapest-insertion approach) [131]. In contrast, in the two-phase approach the customers are first partitioned into clusters and each cluster is then sequenced individually. The first two-phase approach was the *sweep* algorithm of Wren and Holliday [215]. The algorithm performed the insertion of the customers according to their polar angle from a central depot. Whenever a customer could not be inserted on a route, a new route was instantiated. Finally, each route was sequenced using a TSP algorithm. For a more thorough view on classical and modern heuristics for the vehicle routing problem, we refer to an article by Laporte *et al.* [131].

In general, these approaches are too simplistic for achieving high-quality solutions by today's standards, and are rarely used alone. Heuristic methods are instead employed as a part in more sophisticated solution approaches, such as

metaheuristics.

3.4 Metaheuristic Methods

By far the most successful approach to solving vehicle routing problems has been the use of metaheuristics. Metaheuristics are also approximate methods, thus yielding good solutions in an acceptable time, but without the guarantee of optimality. Metaheuristic methods include local search (LS), other single solution metaheuristics often based on LS, and population metaheuristics, which employ more than one solution at a time. We start, however, by briefly discussing the general principles of metaheuristic search.

3.4.1 Principles of Metaheuristic Search

The main principle of metaheuristic search is that it guides the problem-specific search operators at a generic level providing a template for traversing the search space. A common concept in the search space traversal is that of *search neighborhood*. A neighborhood is defined by a neighborhood function, which is a mapping $\mathcal{N}\colon \mathbb{S} \rightarrow 2^{\mathbb{S}}$ that assigns to each solution $s \in \mathbb{S}$ a set of solutions $\mathcal{N}(s) = \mathbb{S}_s \subset \mathbb{S}$ [195]. A solution in the neighborhood is called a neighbor, and this neighbor is generated from the solution by a *move operator* that alters the solution slightly.

Since the search space is too large for exhaustive traversal, only some regions should be subject to inspection. The question of how intensively to study a certain area for better solution versus how diverse areas to explore is a central one in metaheuristic search. This is often referred to as the diversification versus intensification problem, and much of the design around metaheuristics involves balancing between these two conflicting criteria. The diversification versus intensification is depicted in Figure 6. There are also other aspects differentiating metaheuristics, including how they recognize promising areas and how fast they can inspect a given region.

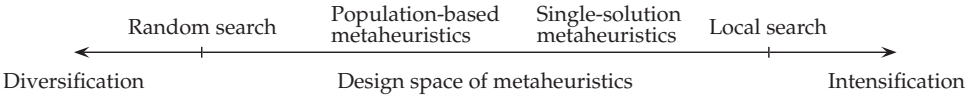


FIGURE 6 Two conflicting criteria in designing metaheuristic search: intensification and diversification [195].

On one end of the spectrum is local search which concentrates the search on a very narrow region, and on the other, a random traversal of the search space. Single solution metaheuristics attempt to diversify the local search by providing additional guidance, and population metaheuristics try to cover more of the

search space by introducing a number of solutions that are considered simultaneously. A general scheme for single solution metaheuristics is given in Algorithm 1 [195].

Algorithm 1 Template of single solution metaheuristic.

Require: s_0
  $i \leftarrow 0$
  $s \leftarrow s_0$
  repeat
    $\mathbb{S}_{s_i} \leftarrow$ GenerateCandidates(s_i)
    $s_{i+1} \leftarrow$ Select($\mathbb{S}_{s_i}$)
    if $\mathcal{C}(s) > \mathcal{C}(s_{i+1})$ **then**
      $s \leftarrow s_{i+1}$
    end if
    $i \leftarrow i + 1$
  until stopping criteria satisfied
  return s

In single solution metaheuristics, an initial solution is first generated. Then a set of candidates is generated from the current solution iteratively. In each iteration, a single candidate is selected as the next solution, and if it is better than the best found, it is stored. This process is repeated until the stopping criteria is satisfied, for instance, a certain number of iterations has passed.

In contrast, population metaheuristics generate a population of solutions and based on that, select a new one by combining the current and previous population using different techniques. A general scheme for population metaheuristics is outlined in Algorithm 2 [195].

Algorithm 2 Template of population metaheuristic.

  $i \leftarrow 0$
  $\mathbb{S}_i \leftarrow$ GenerateInitialPopulation()
  $s \leftarrow \mathit{SelectBest}(\mathbb{S}_i)$
  repeat
    $\mathbb{S}'_i \leftarrow$ GeneratePopulation()
    $\mathbb{S}_{i+1} \leftarrow$ SelectPopulation($\mathbb{S}_i \cup \mathbb{S}'_i$)
    if $\mathcal{C}(s) > \mathcal{C}(\mathit{SelectBest}(\mathbb{S}_{i+1}))$ **then**
      $s \leftarrow \mathit{SelectBest}(\mathbb{S}_{i+1})$
    end if
    $i \leftarrow i + 1$
  until stopping criteria satisfied
  return s

For a brief but recent overview on TSP and VRP solution methodology, we refer to a paper by Laporte [129]. For more general overview on metaheuristics we refer to an article by Gendreau and Potvin [89], and for an in-depth view, a book by Talbi [195].

3.4.2 Local Search

LS[3] is probably the oldest and simplest metaheuristic. Dantzig was perhaps the first to apply local search to optimization with the introduction of the simplex algorithm which can be seen as a local search for linear programming problems. We take here a closer view on LS due to its wide application to vehicle routing, and its usage as a base for many subsequent metaheuristics.

An initial solution is first generated by using a heuristic method, and is then replaced by LS iteratively by one in its neighborhood if that solution improves the objective function. The process is continued iteratively until no improving move is found and the method has found a local optimum. Since enumerating all the candidate solutions takes far too long, LS speeds up the search by concentrating on a more narrow, local neighborhood thus restricting the search to reasonable areas of the search space. The local search scheme is illustrated in Algorithm 3. s is the current solution to the problem and $\mathbf{S}_s$ the search neighborhood defined by the search operator used.

Algorithm 3 Local search.

```
s ← GenerateInitialSolution()
while not termination criterion holds do
  S_s ← GenerateCandidateNeighbors( s )
  if no better neighbor exists then
    return s
  end if
  s ← SelectBest( S_s )
end while
return s
```

The type of neighborhood greatly impacts the performance and quality of local search. The larger the neighborhood, the typically better the quality, and the greater the search effort. Much of the work in applying local search involves designing good search operators and from that, neighborhoods. The two most used neighborhood types in vehicle routing have been the so-called *node neighborhood* and *edge neighborhood*. In node neighborhoods, a set of nodes of the graph is moved, added, or removed, and the possible moves form the neighborhood. An example of an operator defining a node neighborhood is the relocate operator, which removes a single node from its place and inserts it at another location on a route. An intra-route variant of relocate is depicted in Figure 7.

An edge neighborhood is defined by removing and adding a set of edges within a solution. An example of an operator defining an edge neighborhood is the classical 2-opt operator, which removes two edges and adds two different edges to reconnect the parts as depicted in Figure 8.

The local search operators that has been proposed for the VRP are numerous. These include those of k-opt family, see, e.g. Croes [51], most typically 2-opt

[3] Also known as, for example, hill climbing.

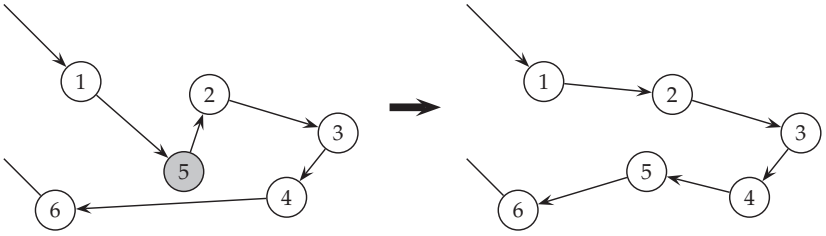


FIGURE 7 A relocate operator moving node 5.

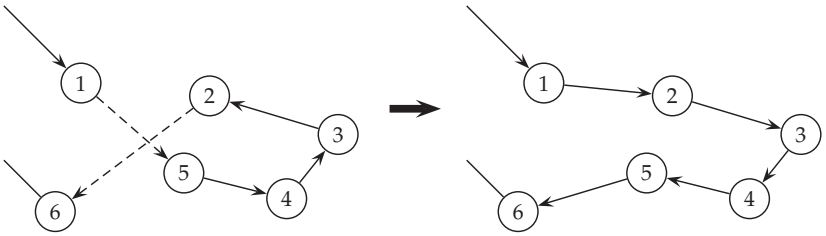


FIGURE 8 A 2-opt operator replacing edges (1,5) and (2,6) with (1,2) and (5,6).

and 3-opt; k-opt* by Potvin [166] (according to [80]); Or-opt by Or [159]; CROSS-exchange by Taillard *et al.* [194]; GENI-exchange by Gendreau *et al.*, see, e.g., [88]; Lin-Kernighan by Lin and Kernighan [136]; and λ-interchange by Osman [160]. Also, a large neighborhood search (LNS) involving larger numbers of routes simultaneously have been proposed, including ejection chains, cyclic transfers, and different partial destruction and construction neighborhoods (see, e.g, [80] for details). Still, the main weakness of local search is that it stops immediately when it cannot find a better move, which makes it highly vulnerable to converging to a local optima. For overcoming this problem, a number of approaches have been proposed. We review the most important ones in the subsequent section where we discuss more sophisticated single solution metaheuristics.

Due to its importance to the VRP, a substantial amount of research has been devoted to LS, and in addition to designing search operators, theoretical study has also been conducted. A conceptual integration of LS was outlined by Funke *et al.* [80] who analyzed the structures of different neighborhoods. Irnich *et al.* [113] applied sequential search to speed up local search in a vehicle routing setting. Sequential search explores a local neighborhood more efficiently by decomposing the local search operators and pruning the search based on a partial evaluation. More recently, Zachariadis and Kiranoudis [218] introduced a strategy for reducing the computational complexity of local search by storing static descriptors of all the possible local search moves and updating the non-static data (costs) every time the solution is modified. This method trades memory for execution speed by keeping the tentative data precalculated and sorted according to the costs. The speedup is gained by fewer evaluations required in exploring the search space.

For a more detailed view on construction and LS in the context of routing with time windows, we refer to a survey by Bräysy and Gendreau [30].

3.4.3 Other Single Solution Metaheuristics

Four different strategies for overcoming the main weakness of local search have emerged. These are *acceptance of non-improving moves*, such as in tabu search (TS) and simulated annealing (SA); *usage of different neighborhoods*, such as in variable neighborhood search (VNS); *changing of objective function*, such as in guided local search (GLS); and *iteration with different solution*, such as in iterative local search (ILS).

Simulated annealing is based on principles of statistical mechanics, more particularly the formation of crystalline structure in metal substances achieved by a careful annealing process involving attaining a thermal equilibrium at each temperature. The process was developed in the 1950s and the main idea was first applied to optimization by simulation in 1983 and 1985 by Kirkpatrick *et al.* [126] and Černý [205], respectively. Simulated annealing is a stochastic algorithm which allows degrading of the solution under certain conditions, which enables the search to escape local optima. SA proceeds iteratively by selecting a random neighbor at each iteration. A non-improving move is accepted depending on the temperature φ and the change in solution quality ΔE. When an equilibrium is achieved, that is, a fixed or an adaptive number of iterations has passed, the temperature is decreased. The gradual decrease in temperature ensures convergence of the method. A description of simulated annealing is given in Algorithm 4.

Algorithm 4 Simulated annealing.

Require: Cooling schedule $\mathcal{S}$
  $s \leftarrow$ GenerateInitialSolution()
  $\varphi \leftarrow \varphi_{max}$
  repeat
    repeat
      $s' \leftarrow$ GenerateRandomNeighbor()
      $\Delta E = \mathcal{C}(s) - \mathcal{C}(s')$
      if $\Delta E \leq 0$ **then**
        $s \leftarrow s'$
      else if Random() $< e^{-\frac{\Delta E}{\varphi}}$ **then**
        $s \leftarrow s'$
      end if
    until equilibrium condition
    $\varphi = \mathcal{S}(\varphi)$
  until $\varphi < \varphi_{min}$
  return s

A **Tabu search** algorithm for combinatorial optimization was suggested by Glover [91]. TS explores the search space in a similar manner to LS, but escapes local optima by allowing non-improving moves when improving moves are not found. To prevent cycling between two adjacent solutions, a search memory is employed which prohibits the search from visiting a set of recently explored so-

lutions. This short-term memory is referred to as the tabu list. Moreover, additional advanced mechanisms have been used in both diversification and intensification of the tabu search scheme. Intensification, or medium-term memory, stores the best solutions found during the search and guides the search according to the attributes of these solutions. Diversification, or long-term memory, on the other hand uses knowledge of the visited solutions to direct the search towards unexplored areas of the search space. An overview of tabu search is given in Algorithm 5.

Algorithm 5 Tabu search.

```
s ← GenerateInitialSolution()
InitializeMemories()
repeat
  S_s ← GenerateCandidateNeighbors( s )
  s ← SelectBestAdmissible( S_s )
  UpdateMemories()
  if intensification criterion holds then
    PerformIntensification()
  end if
  if diversification criterion holds then
    PerformDiversification()
  end if
until stopping criteria satisfied
return s
```

To diversify the search, **iterative local search** applies the local search scheme iteratively by restarting the search from a solution related to the current local optimum. This strategy was first proposed by Martin *et al.* [139]. When a local optimum is reached, a perturbation operation is carried out, in which parts of the solution are discarded. An overview of ILS is given in Algorithm 6.

Algorithm 6 Iterated local search.

```
s ← GenerateInitialSolution()
s ← PerformLocalSearch( s )
repeat
  s' ← Perturb( s, search history )
  s'' ← PerformLocalSearch( s' )
  s ← Accept( s, s'', search memory )
until stopping criteria satisfied
```

Guided local search, proposed first by Voudouris and Tsang [209], escapes local optima by changing the objective function. Each solution has a structure, and features of that structure, for example, the existence of especially short routes, can be analyzed. Whenever a local optimum is reached, the objective function is altered by penalizing the features that occur in the generated local optima. This is

done by defining a set of features with an associated cost and a penalty indicating its importance. Furthermore, a utility is defined for each feature, and the feature with the highest utility will be penalized. This will intensify the search in promising regions and diversify the search in the vicinity of a local optimum. One of the most successful approaches for the capacitated vehicle routing problem and the vehicle routing problem with time windows is the guided search described by Mester and Bräysy in [142] and [141], respectively. An overview of guided local search is given in Algorithm 7.

Algorithm 7 Guided local search.

```
s ← GenerateInitialSolution()
InitializePenalties();
repeat
  s' ← PerformLocalSearch( s )
  for all feature i of s' do
    ComputePenalty( i )
  end for
  SelectHighestUtility()
  AdjustObjective()
  s ← s'
until stopping criteria satisfied
return s
```

Variable neighborhood search was proposed by Mladenovic and Hansen [144]. Instead of a single neighborhood, VNS explores different neighborhoods to escape from the local optima. This is possible due to the fact that different neighborhoods may generate different local optima. VNS has several variants of which the variable neighborhood descent (VND) is a deterministic version of a more general variable neighborhood search scheme. The VND uses successive neighborhoods in descent to a local optimum. Whenever a local optimum is reached, the neighborhood structure is changed to the next in succession, and if an improvement to the solution is found, the first neighborhood is reselected. An overview of the variable neighborhood descent is given in Algorithm 8.

The illustrated metaheuristics have been applied to different vehicle routing problems numerous times, and the obtained results generally indicate that problem sizes up to a few thousand customers can be solved in reasonable computational times. More advanced techniques are still being developed. An example of this is the inclusion of learning methodologies, such as in the work by Ropke and Pisinger [174], who introduced an adaptive large neighborhood search for the pickup and delivery problem with time windows. For a more detailed view on metaheuristics in the context of the vehicle routing problem with time windows, we refer to the second part of the survey by Bräysy and Gendreau [31]. For a categorized bibliography on recent application of metaheuristics to vehicle routing we refer to work of Gendreau *et al.* [90].

As of today, single solution metaheuristics outperform exact methods by an

Algorithm 8 Variable neighborhood descent.

Require: $\mathbb{S}^l, l = \{1, \ldots, l_{max}\}$
  $s \leftarrow$ GenerateInitialSolution()
  $l = 1$
  while $l \leq l_{max}$ **do**
    $\mathbb{S}_s^l \leftarrow$ GenerateCandidateNeighbors($\mathbb{S}^l$, s)
    $s' \leftarrow$ SelectBest($\mathbb{S}_s^l$)
    if $\mathcal{C}(s') < \mathcal{C}(s)$ **then**
      $s \leftarrow s'$
      $l \leftarrow 1$
    else
      $l \leftarrow l + 1$
    end if
  end while
  return s

order of magnitude. Exact methods, however, are able to provide more robust approach to small instances, since they guarantee an optimal solution provided they are able to solve the problem. On the other hand, for most practical purposes, the solution quality obtained by single solution metaheuristics is sufficient, and they are currently the primary tool in real-world routing.

3.4.4 Population Metaheuristics

Population-based metaheuristics operate on multiple solution candidates rather than one. The search procedure is carried out by generating a new population of solutions and integrating this to the existing one. During the integration phase, a selection process is executed, which guides the search. Three main categories of population metaheuristics are evolutionary algorithms (EAs), scatter search (SS), and swarm intelligence methods [195]. Evolutionary algorithms were outlined by Holland [105] in 1962 and the scatter search approach was first used by Glover [92] in 1977. Ant colony optimization (ACO), the most prominent of the swarm intelligence methods, was introduced by Dorigo in 1992 [61].

Genetic algorithms (GAs) are perhaps the most used of EAs, and they mimic the natural selection process witnessed in living organisms. In GAs, solutions are encoded into a chromosome representation and operated on by different mutation and crossover operators in the generation of a new population. A selection is then done according to the fitness of each individual solution, defined by the fitness function. Another EA, scatter search, uses a reference set, that is, a collection of good representative solutions of the population. Solutions in the reference set are combined to provide starting solutions for single solution metaheuristics, and after applying the search, both the reference set and solution population are updated according to the given intensification and diversification criteria. In a way, SS combines single solution and population metaheuristics. Ant colony optimiza-

tion, on the other hand, is based on an idea of collective memory and independent cooperating agents. In ACO, each agent (or ant) moves towards attractive areas in the search space and leaves trails of itself, which again attract other agents to promising regions. The collective behavior thus converges towards an optimum.

Population-based metaheuristics have not yet been applied to vehicle routing as widely as single solution metaheuristics, but they have, quite recently, emerged as a viable option. However, some shortcuts may have to be taken due to the complexities in the problems. In particular, the encoding of solutions into chromosomes is either completely ignored by working directly on the solutions or specifically designed for specialized operators [89]. This is primarily due to the complexity of combining solution instances into a feasible offspring instance as is often needed in population metaheuristics. The constraints that are present in routing problems are perhaps less suitable for these operations than in other types of combinatorial optimization problems.

In EA methodology, one promising approach to the VRP has been a GA-type method by Nagata [150], in which the author applied an edge assembly crossover operator for the capacitated vehicle routing problem, and further, based on this, introduced strategies for limiting local search in [151]. Moreover, a memetic approach, combining EAs and LS, was introduced in [152] and [153]. Swarm intelligence has been applied to the VRP in, for instance, Reimann *et al.* who applied an ACO-based approach in [168]. Their D-Ants strategy decomposes the problems into smaller instances and uses the information gained from the solution process of these smaller problems to update the global memory. This is then used to aid in the search for solutions to the original problem instance. Tarantilis applied an adaptive memory methodology [196] where the algorithm keeps track of the best components of the solutions visited during the search and combines these components into a new solution. The solution is further intensified by local search. More recently, Gajpal and Abad [81] used a multi-ant colony system for solving the vehicle routing problem with backhauls. They employ two different types of ants and the search is based on the two-phase heuristic. Many of the early applications of population metaheuristics report solutions of good quality in reasonable running time.

3.5 Unifying Elements in Solving Routing Problems

During the fifty years of research on routing, we have witnessed a development of an impressive number of techniques and approaches for solving the problems. Exact methods, mathematical programming and constraint programming, have provided the theoretical foundation on these problems, and first the heuristic methods and then metaheuristics have resulted in practical applications. In this section, we discuss some of the recent developments in solving the vehicle routing problems from a more generic perspective.

The general trend seems to be from simple to more sophisticated in a sense

that new elements are being added to the existing approaches, and composite techniques of different kinds begin to emerge. Many of the metaheuristics presented in the previous sections have existed for quite a while now, and their central ideas are being employed over and over again in different combinations. Some also suggest [195] that hybridization of different approaches, for instance metaheuristics and mathematical programming, may result in better methods. One of the most interesting lines of development is, as noted before, the research on metaheuristics, due to their wide applicability and performance. Moreover, the key criterion for applying recent advances into a practical setting seems to be the *flexibility* in modeling, solution methods, and system design. Thus, even though a single method outperforms others in one area, in general-purpose tools needed in the industry, a robust and modifiable approach is selected instead.

To provide a modifiable and robust approach from metaheuristics, a unified view on the ideas behind the different metaheuristics needs to be formed. And although metaheuristics illustrated in Sections 3.4.2–3.4.4 seem to differ in their approach, Gendreau and Potvin [89] have, indeed, identified a common set of elements within these methods. They concluded that the research on metaheuristics for solving combinatorial optimization problems converges towards a general framework and identified several central elements from the methodology. These elements are the building blocks of metaheuristic solution methods and correspond to different phases of the solution process. The common phases identified are construction, recombination, random modification, improvement, memory update, and parameter and neighborhood update.

In the *construction* phase, the metaheuristic algorithm creates an initial solution, or initial solutions in case of population metaheuristics. In the *recombination* phase new solutions are generated from current ones through a recombination process. The *random modification* phase is used to modify the current solution through perturbation. The *improvement* phase is used to improve the current solution explicitly, for instance, by selecting the best solution in the neighborhood, by applying a local descent, or by projecting the solution to a feasible region. The *memory update* phase is needed by the methods storing search history or subsets of solutions. Finally, the *parameter/neighborhood update* phase adjusts the parameter values or modifies neighborhood structures in a reactive manner.

All metaheuristics adhere to this general scheme, albeit some omit certain phases. For instance, scatter search can be described as *construction* $\rightarrow$ *recombination* $\rightarrow$ *improvement* $\rightarrow$ *memory update* $\rightarrow$ *recombination* $\rightarrow$, variable neighborhood search as *construction* $\rightarrow$ *random modification* $\rightarrow$ *improvement* $\rightarrow$ *neighborhood update* $\rightarrow$ *random modification* $\rightarrow$, and ant colony optimization as *construction* $\rightarrow$ *memory update* $\rightarrow$ *construction* $\rightarrow$. This means that only a few general elements are needed for representing the concepts employed in metaheuristic search. This may open up a possibility for implementing a general metaheuristic search framework that would offer the balance between intensification and diversification by analyzing the problem structure *a priori*, or by adjusting the search by selecting the appropriate elements reactively.

In addition to the intensification versus diversification problem discussed

earlier, Gendreau and Potvin identified a number of common issues and trends. These include *reactivity*, that is, automatic and dynamic adjustment of parameters before or during the search to ease the manual tuning of parameters often required due to the heterogeneity of the cases; implementation of *memories* which guide the search according to the previously seen solutions; and *traversal of large-scale neighborhoods* but in a way that the methods concentrate on promising candidates.

As we observe from the literature, much of the research on vehicle routing is concentrated on one or few vehicle routing variants and solution methods at a time. This is reasonable, since we need to establish knowledge on the solution methods appropriate to different classes of problems. At times, however, a synthesis is formed from the obtained results. Currently, the interest is shifting towards solving a more heterogeneous set of problems with a generic solution methodology. In fact, these common trends can be seen as *a way to adapt to the situation*, which is indeed a desired property. And, as we discuss later when examining the realization of a practical system, the ability to adapt may turn out to be a necessity for the future methodology.

4 IMPLEMENTABILITY IN ROUTING SYSTEMS

"All problems in computer science can be solved by another level of indirection."

— DAVID WHEELER

In this chapter, we bring the routing problems and solution methodology towards the context of implementation. While academic research has been successful in defining, analyzing, and solving these problems, far less attention has been paid to the realization of systems capable of handling the particularities of the domain. There are two major challenges in implementing a routing system: how to manage the complexity of the domain, and how to balance between different conflicting requirements in building these systems. This chapter discusses these challenges in implementation and examines some promising approaches for overcoming these difficulties from both vehicle routing and software engineering point of view. Furthermore, this analysis provides a rationale for our approach described in the subsequent chapters.

This chapter is divided into five sections: first in Section 4.1, we form a synthesis of the observations made thus far, and based on this, in Section 4.2, we address the relevant quality attributes of an optimization system capable of addressing the current issues in the routing domain. In Section 4.3 we discuss one of the primary techniques used in overcoming the challenges in technical issues and the implementation effort in large systems: software reuse. Section 4.4 then further examines large-scale reuse techniques relevant to the VRP domain. Finally in Section 4.5, we summarize the implementability aspects discussed in this chapter.

4.1 Synthesis on Vehicle Routing Research

As our aim is to construct a system which would be applicable to a wide range of routing problems, we need to begin by forming an overview of the topics exam-

ined so far. Chapter 2 discussed the different routing problems and their modeling, and in Chapter 3 we examined the means for solving these problems. The recent developments can be characterized along the two sides of the VRP research: modeling and solution methodology. Perhaps the single most prevailing trend in the vehicle routing research is the progress towards rich VRP models. New real-life aspects are constantly being incorporated into the models and an impressive suite of solution methods has been developed to solve the resulting cases. Developments in solution methodology include moving from heuristic methods to metaheuristics and, especially recently, population metaheuristics. Thus, these two chapters may perhaps be summarized by the general trends (with the emphasis on recent developments) illustrated below.

Models: simple $\rightarrow$ rich $\rightarrow$ unified $\rightarrow$ generic $\rightarrow$ composite
Methods: simple $\rightarrow$ complex $\rightarrow$ adjusting $\rightarrow$ adapting $\rightarrow$ learning

The **simple** models covered the capacitated and time constrained vehicle routing problems and several of their subsequent extensions, for instance, backhauls, and pickups and deliveries. The solution methods were **simple** heuristics and local search which formed the theoretical base for understanding search in routing problems. The application of metaheuristics moved the solution methodology towards more **complex** algorithms and especially the recent combination of the elements from multiple metaheuristics increased the overall complexity of the solution methods. Gradually, **rich** models were formulated, and the introduction of intricate side constraints — usually in an *ad hoc* manner — increased the expressiveness of routing models. Quite recently, as we noted, efforts to **unify** the models have taken place. **Self-adjusting** methods are beginning to emerge, attempting to reactively modify their parameters according to the behavior they observe. This is also highlighted in the scientific literature by the trend of reporting explicit avoidance of heavy adjustment of optimization parameters before running the benchmark tests. The current research seems to point towards **generic** optimization models and more **adaptive** algorithms where the methods employ mechanisms selecting an appropriate set of algorithms for the particular problem. This is a necessary development in the solution algorithms if the models become more general: an increase in applicability requires an increase in robustness.

The shift towards generic models and adaptive methods is visible, but its usefulness has yet to be critically examined. This work has merely begun and our contribution is a step towards that direction. The next step in optimization methods could be a **learning** algorithm that would analyze the problem structure, classify and identify it, measure its properties and select a suitable set of components of metaheuristics and applicable local search methods, and set all the parameters of the optimization accordingly, as well as apply this approach during optimization. Models could be **composed** from the relevant aspects as needed by the characteristics of the case, and perhaps models at multiple different fidelity levels could be generated, solved separately, and combined to guide the search. In this light, these techniques provide a plausible research topic, and

they will be brought up again in more detail in the subsequent chapters, especially when addressing the topics for further research.

There has, indeed, been advances in solution methodology performance and expressiveness in modeling. To conclude this synthesis, we would like to point to some specific instances within the larger developments in order to highlight the aspects we consider especially interesting.

First, we note that Pisinger and Ropke [164] introduced a general heuristic for several vehicle routing problem variants simultaneously by utilizing an adaptation mechanism that selects an appropriate set of algorithms for the problem at hand. Derigs *et al.* [58] developed a richer VRP model for compartment considerations, and conclude that adding additional constraints to existing VRP algorithms instead of developing whole new ones is a promising approach. This is, interestingly, in contrast with the conclusions from the open routing research.

Some have addressed the issue of implementability in vehicle routing systems, and software engineering topics have been discussed, more or less indirectly however, in the vehicle routing literature. A notable exception is the work of Hasle and Kloster [103], who examine some of the design decisions behind a commercial realization of a SPIDER VRP solver. The solver was based on a single, rich, generic VRP model and a unified algorithmic approach to reduce the development and maintenance efforts. They also employed a metaheuristic solution approach for achieving flexibility and robustness. In addition, Hasle [101] describes a GreenTrip Generic Toolkit developed in an EU project "GREENTRIP" (global reactive efficient and environment-friendly transportation logistics)[1] undertaken in 1996–1999. The toolkit employs an approach which achieves an easy configuration of a VRP solver using a specific configuration generation tool for developers and end users. Other recent developments include the work of Groër *et al.* [98] who developed an object-oriented library for local search operators and heuristics for solving vehicle routing problems. They published their work along its source for academic use. We note that the development of standardized elements indicates a certain maturity of the methodology, and these kinds of initiatives are required for addressing the challenges posed by the inherent heterogeneity of the domain. Furthermore, the object-oriented approach described in the framework indicates that separating algorithms from the solution structure is beneficial for achieving a looser coupling. This underlines the fact that general reusable frameworks have emerged and are becoming more widespread.

On the theoretical level, the emergence of frameworks can be seen in the work by Gendreau and Potvin [89], who proposed a general framework for metaheuristics, and Funke *et al.* [80] who introduced a conceptual integration of local search operators. The application of constraint programming to the VRP for achieving flexibility was studied by Backer *et al.* [9] and Kilby *et al.* [124], and on a similar motivation, Irnich *et al.* [109], [110], [104] proposed a unifying approach for modeling the VRP using a giant tour and resource-constrained paths. Despite the number of available algorithms, and as noted by Sörensen [191], the variable neighborhood search approach with tabu search or a simulated annealing scheme

[1] `http://cordis.europa.eu/esprit/src/20603.htm`

is widely used in commercial settings due to their simplicity and flexibility. Similar note was also made by Cordeau *et al.* [45]: there is need for powerful and robust but simple algorithms. Furthermore, an interesting analysis by Wolpert and Macready [214] illustrates that in optimization, it is impossible to construct a single algorithm that performs best in all problems. Also this suggests that a number of methods may be needed in a problem space as heterogeneous as vehicle routing.

As Chapter 2 discussed the **formulation** and **modeling** of routing problems, and Chapter 3 the state of the art in **solving** these problems, this Chapter progresses towards the next step, **applying** the solutions into practice. This step has, however, a number of obstacles including for instance the integration of the optimization system to the everyday process of the logistic operator. But in this work, we are especially concerned with the *general applicability* of these results. In other words, we need to consider what can be done to enable the utilization of the recent academic advances in the industry in general. Implementing a system that can be employed by a substantial subset of logistic operators is still a major technical challenge — even if we assumed that the current solution methodology and modeling techniques are sufficient for a relevant subset of routing problems.

There are several reasons for the challenges faced in implementation of routing systems. Firstly, as we observed, models and methods are becoming more complex to manage, and generic approaches are emerging. Complexity and generality pose a challenge in providing a system without losing performance and without requiring insurmountable efforts in implementation. Secondly, as Sörensen *et al.* [191] pointed out, real-life vehicle routing problems are not standalone problems, but have an impact on other decisions in a company's operations, resulting in more complex design problems that perhaps do not adhere to the boundaries of the theoretical routing problem at all. And thirdly, even the well-defined subset of the VRP domain is extremely difficult to address by a single system, yet it cannot be tackled with an introduction of a new system for each particular case. Thus, something "in between" these two extremes is needed. We argue that central factors in overcoming the challenges are *adaptation* and *reuse*. These aspects, in turn, are realized by building a system which is both flexible and reusable. Flexibility and reusability are, however, only some of the *quality attributes* [210] required from a general optimization system. In fact, as with any system, several key attributes define the overall quality of an optimization system. Thus, to understand and analyze the trade-offs involved in implementing such a system, we need to take a software quality viewpoint on routing software and discuss these attributes in more detail.

4.2 Routing Systems from Software Quality Viewpoint

As we discussed the performance of methods and the accuracy of models in the previous parts of this thesis, we deliberately overlooked a number of essential

attributes relevant to the implementors and users of automated routing systems. In this work, we consider not only the low-level implementation, especially solution methodology, that has gained attention in recent work on the VRP, but also the overall structure of the system. Examining a software system from this kind of top-down point of view requires us to consider — in addition to *functional* requirements — the *nonfunctional* requirements, or the quality attributes, of such a system. Whereas the functional requirements state *what* the system should do, the nonfunctional express the criteria that can be used to assess the operation of the system, i.e., *how* it behaves under different situations[2] [210].

The IEEE standard on software quality metrics methodology [108] defines the software quality as "the degree to which software possesses a desired combination of attributes". The standard then states that "this desired combination of attributes shall be clearly defined; otherwise, assessment of quality is left to intuition". As we have discussed different quality attributes in this work, we have not really clearly stated what is required of the system from the nonfunctional viewpoint. Furthermore, the set of required quality attributes varies from system to system [210], and we are not aware of any work addressing software quality attributes in the context of vehicle routing systems. Therefore, to evaluate the trade-offs involved in designing an architecture for such a system, we define the quality attributes referred to in this work. Note that although a number of other attributes are relevant to a routing system as a whole, our primary focus is on the attributes relevant from the *optimization process* point of view. The discussion here is concerned with the *quality criteria* of such a system, and leaves the question of the *evaluation of the system* according to these criteria untouched. In essence, this section attempts to characterize the nonfunctional requirements of routing systems and to provide a base for heuristic qualitative analysis of the *architectural structure* of these systems in general.

We can identify three types of quality attributes that are relevant from the optimization point of view: *computation related* attributes, *application related* attributes, and *implementation related* attributes. The computation related attributes refer to the attributes governing the costs of performing the optimization, the application related to the costs of applying the results in practice, and the implementation related attributes to governing the costs of implementing the system. The computation and application related attributes are especially relevant from the end-user viewpoint, whereas the implementation related attributes are mostly a concern for the producers of the software. These costs form the total cost of ownership (TCO) [67] of the optimization process, and the ability to balance between these costs is a key element in utilizing the system cost-efficiently in varying contexts.

The **computation related** attributes include the performance and scalability of the system. As discussed in Chapter 3, algorithms vary in their performance, and theoretical complexity analysis gives an estimate on the performance of an algorithm in terms of the time and space used, whereas empirical analysis is re-

[2] Quality attributes are not necessarily related to functional requirements — for instance, *maintainability* considers how good the structure of the system is for maintenance activities.

quired to evaluate the method in terms of the solution quality. In terms of the performance of *the system* itself, it can be seen as the performance of the algorithms utilized for each problem. If there are multiple alternative algorithms for solving the problem, we define the performance of the system as the best of those algorithms. The performance of an optimization system is, therefore, defined in terms of the amount of computational time and space needed by the system to produce high quality solutions using any algorithm available within the system. The notion of a high quality solution refers here to a solution meeting a case-by-case defined criterion. Scalability, on the other hand, is required for the system to be able to address the problems in the magnitude required in real-life routing, i.e., to maintain performance while the size of the computational tasks increases.

The **application related** attributes include the robustness, fidelity, and accuracy of the system. The robustness of an algorithm is often defined as its ability to tolerate small changes in problem data and structure without losing its performance. Likewise, the robustness of an optimization system is its ability to withstand changes from the algorithm and optimization model points of view: the ability of the system to tolerate changes in the problem definition, data, and model without losing its performance. From the modeling point of view, instead of discussing model accuracy alone, we describe here two different quality attributes: fidelity and accuracy. Fidelity concerns the modeling constructs available for describing the real-life situation, and the accuracy of data used within these constructs. The fidelity of an optimization system is defined in terms of the degree of how well the optimization model describes the real-life situation it models. That is, how much manual adjustments are needed in applying the obtained solutions into practice. While fidelity is concerned with the *ability* to describe a given situation, accuracy is the measure of the *exact usage* of that description. The accuracy of an optimization system is its ability to understand and store data required to describe the real-life situations. The accuracy of the system is decreased when, for example, the system cannot utilize all relevant data from a road network, or all dynamic travel time data available cannot be employed by the system, for instance, due to memory or other data processing limitations. This is not exactly the same as model fidelity as the model may have the capability to express the problem, but the system is unable to construct an exact instance of that model. It is not the same as quality of data either as although data may well be free of errors, the system is still unable to fully utilize it. Note, however, that accuracy is still in this context *limited* by the quality of data available.

The computation and application related attributes constitute a *usable* optimization system. The system should describe the problem in high fidelity and accuracy and be able to perform optimization to the problem at hand with the given time and space resources regardless of minor changes in the problem structure or attributes. In short, an optimization system is usable when it provides correct and high quality results to a given problem in a timely manner. Thus, the usability of an optimization system can be seen as a combination of performance, robustness, fidelity, and accuracy. Note that this discussion considers usability as a direct result of the optimization process of the system. In practice, other at-

tributes need to be considered as a part of usability. For instance, the learnability of the user interface is the main component of the usability of the system as a whole. But as mentioned, we restrict ourselves here to the attributes related to the optimization process.

The **implementation related** attributes are directly relevant to the scope of this work. Four major implementation related attributes are applicability, simplicity, flexibility, and reusability. The attributes are interconnected, as, for example, an increase in reusability often increases simplicity and vice versa, and an increase in flexibility tends to increase applicability.

The question of whether the system is applicable to a given situation is especially relevant when addressing a set of heterogeneous problems. The applicability of the system describes how well it is suited to solve the problems within a domain, in other words, how wide an array of problems can be addressed by it. Applicability differs from performance in that without applicability, performance is not relevant as the system is not able to provide solutions in the first place (due, e.g., to an inability to define the objective correctly or to search the whole search space). On the other hand, with applicability to a given problem but with a low performance, the system can find solutions but this would require an excessive time or space. In a sense, applicability expands the problem space covered. This in turn results in increased requirements for robustness for the system to be useful.

The simplicity of the optimization system includes that of the optimization methodology and describes the effortlessness of realizing such a system: it is the negative of the cost of the implementation effort needed in constructing the system. The simplicity (or the lack of it) is one of the key attributes behind the cost of implementation partly due to the fact that simplicity is required for achieving reusability: complex elements and systems are difficult or impossible to reuse.

The flexibility of a system describes its ability to respond to changes. This can be measured by the effort needed to change the system to accommodate new requirements. It is the negative of the average effort needed when adding a feature to problem definition or solution methods to address a new problem. Flexibility is similar to applicability, and these attributes often reinforce each other, but while applicability can be seen as an attribute defining the size of the set of addressable problems, flexibility defines the effort needed in adding new elements to that set.

As we mentioned in the beginning of this chapter, we argue that resource reuse is essential to overcoming the challenges in implementing a generic optimization system, and perhaps the most critical resource in this context is the system itself. Reusability defines the likelihood of a part of the system to be useful in other contexts, i.e., the probability of an element of a system to be usable in a different context with minor or no modifications.

Finally, using the terminology given, we may define another composite attribute to clarify the discussion and to provide meaning to the central theme of this work: implementability. While implementability is generally related to simplicity, in the context of routing systems we also accommodate the notion of ap-

plicability as we consider it as the ability to implement a suitable, or applicable, solution. In addition, as we deal with a heterogeneous domain, flexibility is a central element for adapting the system to different cases, and reusability for performing this with reasonable effort. A system is, therefore, *implementable* if it is a suitable solution to a given problem and simple to realize in an efficient manner in varying situations. We argue that in routing systems, achieving implementability in a generic setting is especially dependent on the reusability: one cannot address each case by an introduction of a new system: most elements of the existing system have to be used also in the new context. The research on software reuse and software reusability considers techniques and processes for achieving a higher reusability within the pieces of software, and in the subsequent sections, we examine the recent advances on this topic as approached in the software engineering literature.

4.3 Software Reuse

As long as there has been interest in the use of software, the practitioners of software development have also taken an interest in its reuse. Reuse, using the same artifact[3] in multiple places after its initial usage, is a direct consequence of striving for cost-efficiency. Software, unlike physical goods, can be multiplied to infinite instances, and once realized, a piece of software can be used any number of times. In theory that is. Possibilities for software *reuse* are bounded by the possibilities of its *use*, and a particular piece of software can be used if it is suitable to the given context. In general, pieces of software are tailored to a certain situation, but as reusability in parts of the system typically lowers the costs of the system as a whole, at some point it becomes beneficial to deliberately increase the possibilities for reuse up-front to achieve savings later.

Techniques for software reuse have been developed since the 1960s, and this has resulted in a transition from *ad hoc* reuse to systematic procedures [155]. Initial techniques included forming **subroutines** from commonly used sets of instructions, which eventually evolved into **modules** of common subroutines. The next major event was coupling of data and operations, which resulted in the concept of **objects**. Later, these objects were combined into **components**, and the component-based approach provided possibilities for larger-scale reuse. Similarly, frameworks provided a generic collection of components and objects, and were designed to be used in multiple places. Decoupling the functionality of the components and objects from their exact location resulted into reusable **services**, which provided a transparent way to systematically reuse existing functionality. Finally, most recently, **software product lines** evolved to provide possibilities for more fine-grained reuse on an architectural level by designing the overall structure of the systems around reusable components and services.

[3] A software artifact refers to any result of design or development during realization of a system; e.g., requirements, design documents, code, tests, and documentation.

Reuse is generally regarded desirable due to, e.g., cost savings, increased quality, and reduced maintaining efforts [14], but the difficulty of utilizing it increases as the scale of reuse increases. Reuse of subroutines and modules may be considered a solved problem, but reuse at a larger scale remains to be challenging. For instance, realizing a product line architecture is still a demanding and nontrivial process [14]. It has been observed that reuse at this scale is best achievable when addressing a family of related systems, and thus the attempts to tackle reuse have recently concentrated on a more narrow context. For example, systems built within a single domain are in many cases more or less slightly different variants of each other [76]. This insight has resulted in a shift in focus from frameworks and components to *architecture-level* reuse: services and software product lines.

In this work, we discuss the architecture-level reuse, especially techniques that are relevant to building a family of systems. These techniques include *software frameworks*, *software product line architectures*, and *model-driven architectures*. We also briefly examine some of the new developments in this area for a discussion on future research.

4.4 Large-scale Software Reuse Techniques

In general, architecture-level reuse is concerned not only with the pieces of software themselves, but also all the other artifacts resulting from the software engineering process. However, as we deal with a family of related systems, there are two types of artifacts that we are especially interested in: *the architectural design itself*, and *the implemented system elements*. Reusing the architecture makes the family of systems possible by providing a common structure to build on, and reusing the functional elements of the system reduces the effort needed to construct the individual systems within the family. Therefore, in this work, we concentrate on techniques that both impact the architectural design decisions and enable the reuse of the software elements in the implementation level. In other words, we examine the properties of the high-level design of the system and attempt to achieve reuse in the lower levels. Since the architectural structure of the system is in a major role in defining the quality attributes of the overall system, shifting focus to the architecture-level allows us to manage and balance between these different quality attributes and the TCO of the resulting system. Note that, although reuse is a primary goal in these techniques and useful in itself, employing systematic reuse often results in improvements in other quality attributes, such as flexibility and simplicity.

From the implementation viewpoint, the central elements in this work are software architecture and software product lines. For a general introduction to software architectures we refer to a book by Bass *et al.* [14], for an introduction to software product lines, a book by Clements and Northrop [41], and for product line engineering in general, a book by Pohl *et al.* [165].

4.4.1 Software Frameworks

A major element in the reuse landscape is the concept of *software framework*. An object-oriented framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than individual classes [118]. Although their usage has not been widely regarded as the solution to the reuse problem itself, software frameworks provide a useful theoretical and practical base for examining large scale reuse. We introduce the main concepts of object-oriented software frameworks in order to examine their usage within combinatorial optimization. A software framework is a specific type of software library; it provides abstractions of implementation wrapped into an interface. Also, as with libraries, frameworks are specific to a set of related functionality.

The features that separate software frameworks from libraries and other applications are typically the following: I) inversion of control, II) default behavior, III) extensibility, and IV) non-modifiability. Unlike in libraries, the control flow of the program is often controlled by the framework (property I), and this is achieved by the so-called inversion of control mechanisms (for an introduction, see, for instance, Fowler [74]). Inversion of control offers locations for extending the default behavior (property II) typically offered by a framework. This extensibility (property III) is the sole approach for customizing a framework as it does not allow itself to be modified (property IV). A framework addresses typically a single functional area, such as providing a graphical user interface (GUI) or distribution of system elements. As noted by Fayad and Schmidt [70], frameworks decouple the abstract functionality from the concrete behavior of the application, and this separation of responsibilities is the key to reuse: the abstract functionality forms the reusable part of the framework.

Naturally, frameworks and reuse are not a solution to all the problems in large scale software development, and also this approach comes with a cost: Fayad and Schmidt [70] list a number of weaknesses in framework-based development, and present the following main challenges. First of all, developing high quality, extensible, and reusable frameworks for complex application domains is difficult, and learning to use an application framework effectively requires considerable effort. Secondly, the application development will be increasingly based on the integration of multiple frameworks, but many earlier generation frameworks were not designed for this. Thirdly, as frameworks evolve, the applications that use them must evolve with them and a deep understanding of the framework components and their interrelationships is essential to perform framework maintenance successfully — in some cases, the application developers must rely entirely on framework developers. Finally, generic components are typically harder to validate and the inversion of control causes the control flow to oscillate between the application-independent framework infrastructure and the application-specific method callbacks, which decreases performance and makes defect removal and maintenance activities more tedious.

Despite the challenges in framework development, a number of optimiza-

tion frameworks have been implemented and are in use today (see, for instance, a book by Voß and Woodruff [208]), but they address a larger domain of optimization and metaheuristics. In combinatorial optimization, it is essential that the search operators are tailored to the problem and problem representation. This is in contrast, for instance, to linear programming solvers, where the search operators are well-defined. While it is possible to utilize a metaheuristics framework for vehicle routing, the framework itself considers only the metaheuristic search aspect of the solution process and leaves the question of the representation of the problem untouched. And as mentioned, the inherent heterogeneity of the vehicle routing domain makes the implementation tedious and heavy customization a necessity. It is, therefore, unlikely that any single framework solution will suffice: building an optimization system for each unique case, even by using a framework, renders itself infeasible. The sheer number of problem variants, modeled aspects, solution methods, and case-specific requirements calls for a systematic approach on a higher level of abstraction for managing this variability within the VRP domain. To achieve this, we need to shift our focus to the architecture-level.

4.4.2 Software Product Lines

While software frameworks increased the possibilities for reuse by providing a strong separation of the abstract and the specific behavior of the system, providing this separation on the architectural level requires building a "framework for architectures", that is, a generic architecture on which other architectures are built. This implies a need to build several systems with common elements. One well-known technique for building this kind of family of systems is the software product line (SPL)[4]. It is defined as a set of software-intensive systems sharing a common, managed set of features that satisfies the specific needs of a particular market segment or mission and that is developed from a common set of core assets in a prescribed way [14]. The SPL is essentially a technique for achieving reuse on the level of the whole system. It yields a larger reuse potential than frameworks by systematically identifying commonalities and explicitly defining the allowed variation within the family of systems. This results in a structure that has (depending on the domain) large parts of the common elements abstracted to a general platform. Software product lines also typically consider only a specific domain which eases the building of the common platform. Software product lines enable *strategic reuse* of *core assets*, the common platform and all adjoining artifacts that can be shared across the product line.

Software product lines have some properties in common with frameworks, but they are not exactly similar; SPLs tend to be larger scale systems than frameworks, and can, in fact, employ one or more frameworks as part of their core system or a variable subsystem. The most important similarity is the fact that

[4] The term *software product family* is also used interchangeably with product line, although some suggest that a product line may or may not consider a family of products, the former being more suitable for product line approach. In this work we use the term product family interchangeably with product line, but prefer the latter.

both contain so-called "cold spots" (property IV) and "hot spots" (property III). These are referred to as core assets and variability in SPLs. Cold spots are unmodifiable parts of the framework or the SPL, and hot spots are places where custom behavior can be inserted. Frameworks often capture the exact process according to which the system operates, and this requires inversion of control, that is, the framework controls the program flow (property I). The user of the framework adjusts the details or parts of the process as needed, but the overall flow cannot usually be modified. In software product lines, the control may be inverted in some parts as the SPL may contain modules that are similar to individual frameworks, but the overall process can also be controlled by the elements in variation points; this enables more control on the process in SPLs. SPLs also allow modifications to core assets if a commonality is identified within all systems. It is not unlikely that the SPL evolves heavily during its lifetime. This is related to property II, which in frameworks indicates that the framework should be self-sufficient and provide a useful default behavior, whereas in SPLs the core assets may not function by themselves. Finally, while frameworks are specific to a given set of functionality, SPLs are specific to a particular domain. From a broader viewpoint, the scale and the systematic strategic reuse differentiates SPLs from frameworks, and often requires utilization of two separate development processes: one for building the generic reusable parts, and another for realizing the application-specific implementations.

In software product line engineering, the two subprocesses of development are referred to as *domain engineering process* and *application engineering process*. These processes produce the domain and application layers of the software product line, respectively. The domain layer serves as the core platform for building specific application instances in the application layer. The division into two processes is illustrated in Figure 9. The processes result in a number of artifacts and employ the typical software engineering subprocesses, i.e, requirements engineering, design, implementation, and testing. Domain engineering results in a set of domain artifacts, including requirements, design documents, implementation, and tests. These domain artifacts, or phase products, are then utilized in one or more application engineering processes, which in turn produce individual applications. Moreover, a variability model is produced for the domain layer and for each application in the application layer. The variability model describes the designed variability on the domain level and the realized variability on the application level.

The rationale behind the separation of processes is the different nature of the activities in them. The aim of the domain engineering process is to define and realize the commonality and the variability of the product line, and of the application engineering, to derive specific applications by exploiting the variability of the product line [165]. Modifiability within the domain platform is, therefore, achieved by a systematic introduction of variability. Identifying this variability requires first identifying the commonalities between the systems. This is often challenging and requires deep domain knowledge. After the common elements have been identified, a plan for systematic exploitation of variation has to be de-

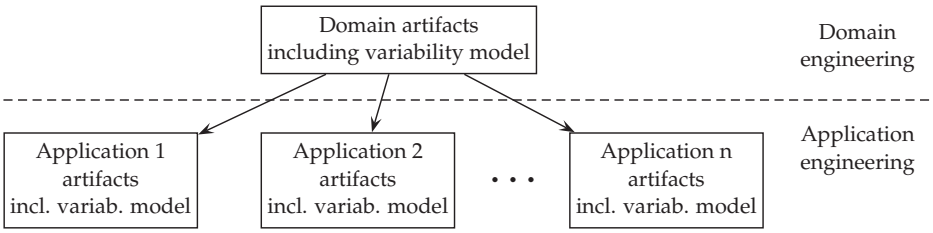


FIGURE 9 Processes and resulting products in software product line engineering.

signed. One approach for achieving this is the introduction of a product line architecture.

4.4.3 Product Line Architectures

Within the software product line, perhaps the most central asset is the *reference architecture*. It is the architectural structure resulting from the domain engineering process and provides a base for application-specific architectures. The reference architecture, with its variability model, forms the so-called product line architecture (PLA). In this work, we concentrate on defining commonalities and case-specific variations on the architectural structure, thus concentrating on PLAs, even though we note the fact that every artifact within the SPL can be exploited in similar fashion.

Broadly speaking, variability represents the system's capability to change or adapt, that is, its ability to facilitate modifications. As noted by Becker [15], such a change or adaptation can affect both the behavior of the system as well as its qualities. From the implementation point of view, variability allows delaying certain design decisions to a later point in the development [99]. Product line architectures attempt to capture this variation explicitly and exploit it systematically.

The systematic exploitation of variation is not a straightforward activity and requires effort in the management of design, implementation, documentation, and testing. In common language use, the term variability refers to the ability or the tendency to change. In the context of product line architectures, we are interested not in variability occurring by chance but in one that is brought about deliberately. To formulate this kind of variability, we need to define two concepts: *variability subject* and *variability object*. The former is an answer to the question on what varies, and the latter, how it varies. According to Pohl *et al.* [165], a variability subject is, therefore, a variable item of the real world or a variable property of such an item, and a variability object, a particular instance of a variability subject. Variability subjects and the corresponding variability objects are present in the context of the PLA, and to continue using the terminology by Pohl *et al.*, a variation point (VP) is a representation of a variability subject within domain artifacts enriched by contextual information. They define a *variant* as a representation of

a variability object within the domain artifacts, but as noted by Clements and Northrop [41], Jacobson *et al.* [115] define the different types of variation more broadly to include also variants residing on the application layer. These include, e.g., introduction of new subclasses into the application layer. A similar approach is also taken by Gomaa and Webber [96] in their variation point model, and thus also we refer to *both the domain and application layer artifacts representing a variability object* as variants. Variation points are at the heart of product line architectures. A variation point is, in a sense, a place in the architecture that one can point to capture a variation within the family of systems. Capturing variation has to be addressed at least from three different viewpoints: *designing*, *documenting*, and *managing* the variation within the PLA, and these activities have received scientific attention for two decades.

The **design of variability** includes both designing VPs along with their acceptable variants. These variants may also have complex dependencies; a variant may require another variant to be present and prohibit a number of others from being selected to the realized application. Another issue in designing VPs is to plan the exact point in time at which the delayed design decision has to be made, i.e., what is the *binding time* of the variation point. In the software engineering context, the binding time refers to the point in time in which a variable receives its type. Traditionally, two types of binding times are distinguished; early binding, that is, at compile time where a compiler assigns a type; and late binding where the type of a variable is inferred at runtime, as is the case with, e.g., polymorphism in object-oriented languages. In product line architectures, binding time refers to the point at which a specific variant within its variation point gets assigned — and one technique for achieving, e.g., runtime binding is, indeed, utilization of object oriented polymorphism. From this perspective, domain engineering can be seen as the process of designing variation points by identifying the commonalities between the systems and implementing shared artifacts while preserving the required variability by deferring the binding time as late as possible; and application engineering, the process of binding the selected variants to their variation points.

To effectively utilize product line architectures, one has to **document the variability** of the software development artifacts. Two basic strategies exist. One is to define variability as *an integral part of the artifacts*, and the other, to introduce *a separate variability model*. Researchers have suggested the integration of variability in traditional software development diagrams or models, such as use cases, feature models, and class diagrams. For instance, Kang *et al.* [121] use feature models, whereas Halmans *et al.* [100] capture variability in use case models. But as pointed out by Pohl *et al.* [165], modeling variability using the traditional software development models has significant shortcomings. Variability information spread across different models is almost impossible to keep consistent, tracking dependencies between variants is difficult, the already complex design documents get even more complex with the addition of variability, and getting an overview across different elements and representing variation unambiguously is tedious.

To address the issues in variability documentation, a separate model, often referred to as the *orthogonal variability model*, has been introduced [165]. This model describes variation points, variants, and their relationships. Variability modeling was studied, for example, by Bachmann *et al.* [7] and Jaring and Bosch [116] who presented a metamodel for variability in a product line context, and analyzed modeling dependencies between variable elements, respectively. A metamodel was also introduced by Becker [15] who outlined a modeling language for specifying variability in product line core assets. More recently, the issue of variability modeling was addressed by Niu *et al.* [154] who proposed a modeling technique based on lattice ordering. Their technique is capable of handling and resolving inconsistent variability requirements.

In **variability management**, the key question is what to design for reuse and what not. On one hand, if the emphasis is too much on reuse, product differentiation suffers and the time required to produce a product or new functionality is increased due to the additional effort required in making each element reusable. On the other hand, if the emphasis is too much on differentiation, the benefits of reuse start to decline and maintaining the product portfolio becomes cumbersome. Finding the balance between the commonality and the variability, reuse and differentiation, involves evaluation of both the technical and business aspects of the product line.

The variability management has been studied intensively as it is one of the most challenging aspects in employing a product line. For an overview on the issues in variability management, we refer to the work of Bosch *et al.* [22] who identified challenges in both the domain engineering and the application engineering processes. More recently, Jha and O'Brien [117] conducted a survey to investigate how software reuse is adopted in product lines and identified some prevailing issues and concerns. Savolainen *et al.* [182] proposed a three-level product line in which the application layer is split into a differentiation layer and a reuse layer to overcome the challenges in balancing between reuse and differentiation. For a recent review on variability management, we refer to the work of Chen *et al.* [37].

Despite the technical challenges and the required up-front investments, software product lines have been successfully used to reduce the total effort in building a set of related systems. This is best illustrated by Figure 10, where the accumulated effort of developing a single system versus the effort of developing a product line is depicted. The recent research indicates that the break-even point is around three distinct systems, but for especially technically challenging tasks, such as combinatorial optimization systems, there may be benefits in using a software product line approach in a two-system environment due to the effort needed in producing the specialized functionality of such a system.

Software product lines have been used to enable large-scale reuse successfully for two decades. Classical examples of applying software product lines include the case of Boeing described by Sharp [187], where an initiative of building a software product line for avionics systems led to a reusable architecture for a number of subsystems, and resulted in reduced costs of avionics software. An-

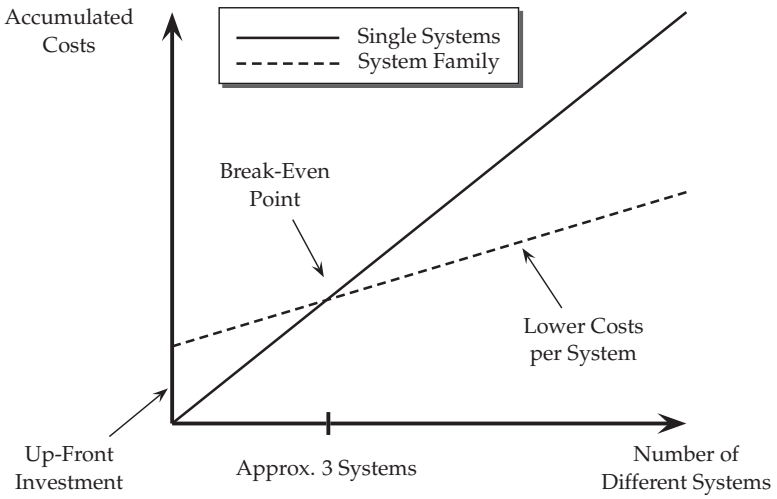


FIGURE 10 The effort for developing n systems compared to product line engineering [165].

other example is that of CelsiusTech Systems AB, a Swedish naval defense contractor, who adopted a product line approach to provide command-and-control systems for navy vessels. The results reported by Brownsword and Clements [33] state that code reuse on new systems averages almost 80%, resulting in CelsiusTech inverting its software to hardware cost ratio from 65:35 to 20:80. Nokia was reported to increase its production of new mobile phone models from 4 to 25–30 annually [14], and a number of other successful cases, including the development of a gateway system by Ericsson Software Technology, the implementation of the Mozilla web browser, and the development of the Symbian Operating System are described by Svahnberg *et al.* [193].

Despite the successes in other industries, we are not aware of any PLA approach for vehicle routing software, although some highly modular implementations do exist. Moreover, we are not aware of any scientific work addressing the software product line engineering in the context of vehicle routing. This may be due to the relatively recent advances in generalizing the vehicle routing problem and unifying the routing domain. Also, building a combinatorial optimization system for vehicle routing is itself a challenging task, which discourages software vendors from adopting techniques that pose additional technical risks.

Vehicle routing software is, however, a prime candidate for the software product line approach. Vehicle routing software has several areas that need to vary from case to case, including the optimization model, and from the system design and software architecture points of view, these areas are attracting contestants for variation points.

Since the optimization model is a VP, we need a system that supports varying the model from case to case. The issue is, however, that the optimization

methodology is directly dependent on the optimization model, which limits the reuse of optimization algorithms in differing cases. Furthermore, building an optimization model and the adjoining algorithms equals, effectively, building an optimization system for the case. To address the issue of varying the optimization model, we propose the construction of a metamodel on which the solution methodology can operate. By using a model transformation to transform a domain model into an optimization model conforming to the metamodel, we can remove the dependency. To provide a theoretical base for this kind of modeling approach, we need to discuss modeling and metamodeling in more detail.

4.4.4 Model-driven Engineering

Model-driven architecture (MDA) can be seen as a software development framework[5] which has been put forward by the object management group (OMG) in 2001. The framework builds on a series of specifications, including the MDA Guide [143], which introduces the concepts of the MDA. The MDA relies on two principles: firstly, it utilizes models and modeling techniques in developing software systems, and secondly it requires separation of system specification from the implementation of the system. These techniques have been used in computer science and other disciplines, but the emphasis on modeling in all phases of the software development process distinguishes the approach from traditional software engineering practices.

Model-driven architecture attempts to solve portability-related problems. The rationale is that if software needs a concrete platform to run on, and this platform is subject to continuous change due to emerging new technologies, we need to abstract away the platform-specific details and concentrate the development effort on abstract models from which the platform-specific implementation can be generated as needed. Kleppe *et al.* [127] list the problems addressed by the MDA as follows: productivity, portability, interoperability, and maintenance. Productivity in implementation and maintenance is increased by shifting activities from low-level implementation to high-level specification. The increase in portability and interoperability, that is, the ability to use the same artifacts in different platforms and to integrate elements produced using different technologies, are achieved by abstracting the implementation details from the development process.

To advance the quality attributes, the MDA distinguishes two types of models: platform independent models (PIMs) and platform-specific models (PSMs). In this context, according to the MDA Guide [143], a platform is any "set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented". This platform independence serves a similar purpose as the introduction of the abstract domain layer in software product line engineering: *reuse*. Thus, from this perspective, model-driven architec-

[5] Not to be confused with *software framework*.

tures and software product lines can be seen as complementary strategies for achieving possibilities for reuse.

The central concept in model-driven architecture is *model*. The term has numerous definitions ([128], [127], [143]) and some disagreement of the exact definition of a model in model-driven techniques still lingers, but in general, a model can be seen as a representation of a part of reality for gaining insight into some aspects of it by providing a useful abstraction. In software engineering, a model is often a symbolic representation and expressed using a *modeling language*. A modeling language is an artificial language used to express systems using a structure defined by a consistent set of rules.

While a model describes a system in the real world, a *metamodel*[6] describes a model. The prefix *meta-* indicates that the concept is an *abstraction* of the concept. In essence, a metamodel is a model of a model, or in other words, a model of the modeling language used to describe the original model. A model is, therefore, an instance that can be said to *conform to* a metamodel if the metamodel describes the language used to describe the model.

The process of metamodeling can be applied iteratively to metamodels. This results in a potentially infinite stack of metamodels describing other metamodels. Such a construction is referred to as a *metamodeling architecture*[7] It is, however, not necessarily meaningful to continue this process to infinity as is done in the so-called *open* metamodeling architectures. A *closed* metamodeling architecture can be introduced by applying the modeling capabilities at a certain level to that level itself. A well-known closed metamodeling architecture is the meta object facility (MOF) architecture described by the OMG. The MOF can be used to model modeling languages and it has been applied to describe, for instance, the unified modeling language (UML) [157]. An example of a MOF-based metamodeling architecture is illustrated in Figure 11.

In the depicted MOF-based architecture, the lowest level *M0* contains the runtime instances of the real system. The *M1* level describes the first modeling level on which the user-defined UML models are described. The third level, *M2*, is the familiar UML metamodel, which describes how UML models should be constructed. The fourth level is the meta object facility level *M3*, a *metametamodel* describing a model for constructing modeling languages. Since the MOF is a modeling language, it can be used to describe itself. The MOF is widely used by the OMG in order to unify the modeling effort [143]. Any new language introduced must be expressed by constructs in the MOF.

Models are useful in understanding the problem domain and analyzing candidate solutions, but model-driven engineering also attempts to achieve a higher productivity, to ensure compatibility between systems, and to yield simpler design processes by systematically manipulating the models [143]. This manipulation is done by applying a series of transformations to convert models into other

---

6 This should not be confused with meta-modeling, which is used, for example, in optimization involving expensive objective function evaluations [195]. In such cases, the metamodel is a simplified representation of the actual model, and is also referred to as a *surrogate model*.

7 Also referred to as *metamodeling stack* and *metamodeling framework*.

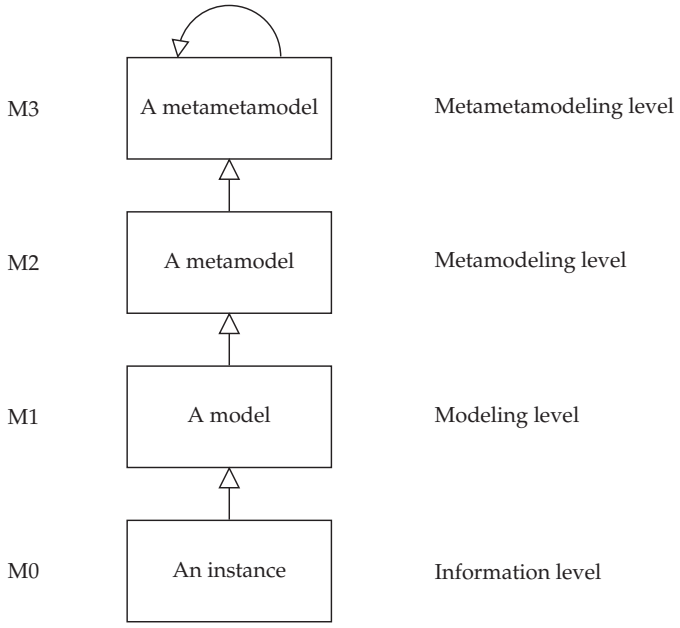


FIGURE 11 A closed metamodeling architecture.

models. This process can be applied until actual executable code is generated. This is made possible by introducing another essential concept of MDA: *model transformation*. The basic process of transforming a model is straightforward. A model transformation is performed by a *transformation engine*, which executes a *model transformation definition*. The definition refers to two metamodels, and effectively describes how the elements in the source metamodel are transformed into the elements of the target metamodel. The *source model*, conforming to the *source metamodel*, is read by the transformation engine and, according to the transformation rules, written as the *target model*. The basic process of model transformation is illustrated in Figure 12.

Model transformations can be classified according to their source and target metamodels and their approach for manipulating the models. Different taxonomies have been proposed, but we use a classification from Mens *et al.* [140] and Czarnecki and Helsen [53], and refer to their work for a detailed analysis.

If the source and target models refer to the same metamodel, the transformation is *endogenous*, and if they refer to different metamodels, the transformation is *exogenous*. If the level of abstraction changes in the transformation, the transformation is *vertical*, and if the level of abstraction remains the same, it is *horizontal*. A model-to-model transformation can be *direct-manipulation*, *relational*, *graph-based*, *structure-driven*, or *hybrid*. A direct manipulation transformation offers an internal model representation and an application programming interface (API) for manipulating the model. These transformations are usually implemented within an object-oriented framework, and the users of such frameworks imple-

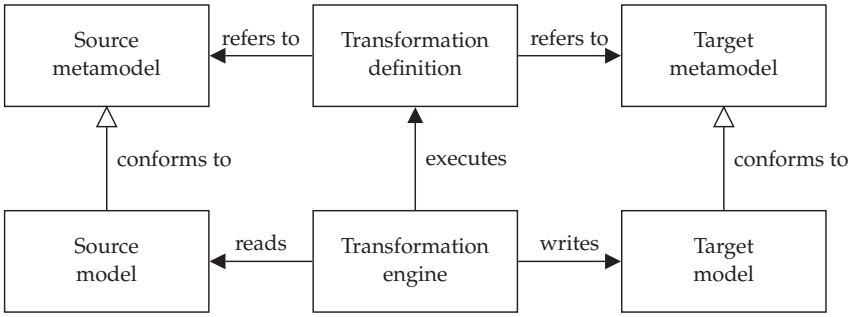


FIGURE 12 Basic concepts of model transformation.

ment transformation rules mostly from scratch using a general purpose programming language. Relational transformations operate on mathematical relations and use constraints to specify the type of the relations between different source and target elements. The definition is non-executable but can be given execution semantics using logic programming, or more specifically, a constraint programming approach. Graph-based transformations employ the theoretical work on graph transformations and operate on typed, attributed, labeled graphs specifically designed to represent UML-like models. Structure-driven transformations have two distinct phases: the first phase creates a hierarchical structure of the target model and the second sets the attributes and references in the target. The idea is to copy model elements from the source to the target, which then can be adapted for the desired transformation. This categorization can be divided into two basic approaches: direct-manipulation transformations are *imperative*, whereas relational, graph-transformation, and structure-driven are *declarative* in nature. Hybrid transformations mix the two types. Finally, transformations can be *bidirectional* or *unidirectional*, where they can be applied from one metamodel to another and vice versa, or only from one metamodel to another, respectively.

A model transformation is expressed in a model transformation language. The language is used to define the transformation rules, and the rules may be written in any executable language. Both general-purpose and domain-specific languages have been used for this. As noted by Bezivin *et al.* [19], transformations can also be seen as models, and in fact, the OMG also specified that transformation languages in the MDA must be defined as MOF metamodels, and thus any transformation definition written with such a language is a M1-level model.

Models, metamodels, and transformations are the basic building blocks of model-driven architectures, but the MDA itself does not concern the process of applying a model-driven approach. A more general concept, model-driven engineering (MDE) was first considered by Kent [122], who articulated requirements for both the processes and the adjoining tools for realizing the MDE in practice.

Kent argued that the development process should guide both the process of constructing the individual models and coordinating the efforts related to different models, e.g., which models should be built first. Kent also stated that tooling is essential to maximize the benefits of the MDE. However, Mohagheghi and Dehlen [146] reviewed experiences in applying model-driven engineering in the software industry. They found that in most cases the maturity of development tools is still perceived as unsatisfactory for large-scale industrial adoption. They listed reports of improvements in software quality and of both productivity gains and losses, but the results were primarily from small-scale studies, and concluded that there is too little evidence to allow generalization of the findings. However, despite the lack of available tooling, model-driven techniques can be applied to enhance other software engineering approaches as well.

The model-driven approach has a natural connection to software product lines, where the separation of abstract behavior and product-specific functionality is rather similar to the distinction between the platform-independent layer of the design (models) and the resulting platform-dependent realization (implementation). Note that the SPL refines the concept of model into two types, depending on the process that generated it. Therefore, in the context of software product line architectures, we distinguish two types of metamodeling architectures: *domain metamodeling architecture* and *application metamodeling architecture*, describing model constructs in the domain layer and the application layer of the software product line, respectively. This is especially relevant in the M1 layer; an *application model* is often refined from a more generic *domain model*. Note that, for instance, metamodels in M2 may also be refined in a similar fashion, but within the context of this work, we focus on model-level variation. Model-driven engineering can be utilized for managing and implementing variability in SPLs; for instance, model transformations can be applied to domain models to produce application models; models may represent VPs, which in turn result in different application level implementations; and transformations may be varied as needed. Model-driven SPLs, thus, employ one or more of the MDA concepts, but not necessarily to their full extent.

The recent attraction to model-driven techniques in software product lines is a direct result of the increasing amount of variability in SPLs. According to Bosch *et al.* [23], there are two reasons for this: firstly, there is a tendency to move variability from hardware to software, thus increasing the flexibility of the system configuration and decreasing the cost of variance; and secondly, design decisions are usually delayed as much as possible during the software development process, and often, variability is fully resolved at the moment the software system is installed. This would suggest that to maximize the flexibility of the overall system, binding time is being pushed towards runtime binding in all elements. This flexibility increases the applicability and decreases the costs of modifying the system. Moreover, model-driven development opens the possibility for automating parts of the software product line processes by using transformations to generate applications from the domain layer. This process is often referred to as *automatic product derivation*, and the adjoining SPL a *configurable* product line. Due

to these developments, a considerable effort has been made in developing a theoretical understanding of the techniques in model-driven SPLs and implementing the necessary tool support for utilizing the alleged benefits.

The model-driven software product line (MDSPL) is not an entirely new concept: for instance, Muthig and Atkinson [148], Deelstra *et al.* [56] and Czarnecki *et al.* [52] considered the prospect of combining the abstraction capability of the MDE and the variability management capability of SPLs. González-Baixauli *et al.* [97] applied model-driven techniques to variability management, and Santos *et al.* [178] and Oliveira *et al.* [158] explored automatic product derivation from the SPL using the MDE. More recently, Tawhid and Petriu [197] proposed a technique for integrating a performance analysis to the model-driven product line development and Schaefer [183] introduced a variability formalization based on Δ-models which can be employed on every level of modeling, and are preserved under model transformations. Schaefer separates the core model and adjoining Δ-models representing changes to core assets to incorporate product features. These models are transformed independently to obtain a more detailed view of the MDSPL.

Although the varying of model transformations is a well-established scenario in model-driven engineering, we are aware of only one work addressing a model transformation as a variability subject *in a product line architecture context*, namely that of Trujillo *et al.* [203]. They do not provide a concrete application of such an approach but contemplate with the idea that metamodels and model transformations should be considered as candidates for variability subjects. They conclude that a shift is needed in research attention, from the variability of models to variability of metamodels and model transformations. Another interesting approach to variability management is to view SPL features as transformations that take a program as an input and produce another program with that feature included. Since models in the MDA can be executable models, this is compatible with the concept of model transformation. An example of such a technique is given, for instance, by Freeman *et al.* [78].

4.4.5 Aspect and Context Oriented Techniques

Naturally, model-driven engineering is not the only approach used in managing variability in software product lines. One general concern which is not addressed by the MDE alone, is the so-called cross-cutting functionality. A cross-cutting functionality is a certain type of functionality that affects several, or all, parts of the systems. A typical example of this is logging, which is typically needed by almost all elements. Changing a logging mechanism scattered to all parts of the system can be tedious. To cope with these kinds of situations, the so-called aspect-oriented software development (AOSD) has been introduced [71]. The main idea is to separate the cross-cutting functionality, the so-called **aspects**, and to automatically generate them into the parts needed by marking the required elements *a priori* or by inferring the generation requirement from other rules. This kind of generation is often referred to as *weaving*. In software product lines, this

technique has been used to manage variability scattered to different parts of the architecture, especially by weaving variability into models and metamodels. This technique has been recently demonstrated, for instance, by Morin *et al.* [147] who apply aspect-oriented techniques to perform product line derivation by weaving variability into domain metamodels; and Voelter and Groher [207], who achieve modularization of variability on different levels by separating the features of different models and composing them by aspect-oriented techniques. Schaefer [183] also noted that their Δ-models can be implemented using aspect-weaving.

The exact variants of an application in a software product line are usually bound before the system is run. However, if the application is required to resolve a variant at runtime, that is, to be *runtime adaptable*, or especially if the system itself needs to be capable of adapting its runtime behavior according to the context it perceives, that is, to be *self-adaptive*, more sophisticated techniques may be required. Some initial work has been done on the so-called **context-oriented** and **self-adapting** software product lines, where the variability may be resolved by the system itself according to its environment at runtime. According to Costanza [49], context-oriented programming (COP) has been defined as an approach that focuses on programming constructs which enable grouping, referencing, and activating and deactivating layers of behavioral variation. This approach has its roots in the research of context acquisition and reasoning. Some researchers have addressed the issue of runtime variation. Costanza describes the current state of the COP and their implementation, which is arguably the most mature realization of the COP concepts. Classen *et al.* [40] discuss the issues in SPLs when addressing self-adaptive and dynamic systems. Alves *et al.* [2] give a review on variability management in software product lines and runtime adaptable systems, and Hubaux and Heymans [106] discuss configuration techniques in product lines, including those taking place at runtime.

Creation of hybrid methods, such as the model-driven aspect-oriented software product line, attempts to alleviate the weaknesses of any single approach. The challenge is the growing complexity of the methodology, and again as also noted by Kent [122], proper tool support is essential for achieving full-fledged wide-spread utilization of these techniques.

Nevertheless, the emergence of these sophisticated approaches would suggest the potential for further utilization of software engineering methodology in the implementation of decision support systems such as route optimization software. As we will observe, the possibility of automatic adaptation and product derivation is an interesting venue for further research also in the routing domain.

4.5 Summary of Implementability Aspects of Routing Systems

The VRP domain is complex and heterogeneous: optimization models are increasingly intricate and solution methods need to be efficient, robust, adaptive and, if possible, simple. Despite these challenges, utilizing automated routing is

an alluring option: creating an efficient automatic planning system results in effective logistic operations, which is the aim of many, if not all, operators performing these activities. Such a system has, however, a number of requirements to be useful. We consider some of them in this work, and achieving the two central ones, usability and implementability, is by no means straightforward. Managing the complexity of implementing the system is critical, and to mitigate the risks in implementation of these complex systems, research on software engineering has resulted in systematic techniques that aid in this task.

Software engineering techniques, such as software product line architectures, attempt to manage the complexity by providing a systematic approach for reuse and variability management. The key for complexity management is the development of *abstractions* [42]. Software architecture, software product lines, and model-driven engineering are, at their core, techniques for abstracting details, and in order to deal with the growing complexity of software, more heterogeneous domains, and the increasing need for customization, we argue that there is a need to identify and systematically utilize the general structures and their properties of the systems at hand. Moreover, proper architectural design enables us to break the dependencies between models, search operators, metaheuristics, and solution presentation by introducing generic interfaces through which the different elements can interact. In other words, such an approach promotes a separation of concerns in the varying elements and decreases the effort of building the system by simplifying the implementation and providing means for introducing flexibility into the structure.

The task of building a single system capable of addressing all the problems in the routing domain is likely to be unattainable, and as noted by other authors, building a system for each individual case is not an option from the practical point of view [191]. Building a software product line for a family of related systems, is, however, a viable option for the VRP domain. Generalizing the commonalities of the routing problems at the domain level yields reusable components and lowers the overall effort of building the system, and varying at the application level enables the system to reasonably adapt to the particularities of the case. Note that the SPL is, indeed, a proper extension of the two extreme alternatives: if a system for a particular case can be constructed entirely from existing reused components, we get one extreme, and if all components need to be built from ground up at the application level, we have the other extreme. A direct result is that this approach allows us to control and manage the costs and the quality attributes of the resulting system. For instance, if higher performance is required, more customization can be made at the expense of reusability. These considerations are relevant from the practical point of view of both the organization that constructs the system and the one utilizing it.

While the discussion in Chapter 2 concerned, from the software quality viewpoint, the fidelity of a VRP system, and in Chapter 3 performance and robustness, the focus in this chapter was in applicability, flexibility and simplicity. These aspects have not been widely discussed in the context of usability and implementability in the current literature, and especially, the needed com-

ponent, *reuse*, has been missing from the discourse. We would like to emphasize the importance of managing the complexity during implementation, and we argue that this can be achieved by constructing generic elements and employing a decoupling of concerns. As noted, applying research into practice is challenging and requires techniques for coping with the needed variability. Fortunately, software engineering practices for solving these problems have been proposed, and indeed, these recent efforts address the needed applicability, flexibility, and simplicity. This work attempts to bring these aspects into the domain of vehicle routing. Fidelity and accuracy are further increased by an introduction of a more generic model. Performance, that is, developing better methods, was not in the scope this study, and we address the issue of robustness when, again, discussing directions for future research.

We are at the verge of building generic and adapting routing systems, but we are not yet certain of the most suitable approach. However, we argue that utilizing knowledge from software engineering will be essential, and this work presents our proposals. Here they are.

5 MODELING FRAMEWORK

"There are only two hard problems in computer science: cache invalidation and naming things."
— PHIL KARLTON

In this chapter, we describe the modeling framework developed in our research. This framework forms the main contribution of this thesis and addresses the heterogeneity and the fragmented nature of the VRP research. It should be noted that this theoretical system has its roots in a practical need of a more general way of describing and solving vehicle routing problems in a software system. There are a number of properties that will be discussed using formal definitions, but many — although not all — features have also been implemented and tested in practice. We discuss the specifics of implementation in the subsequent chapters.

The chapter is structured as follows. In Section 5.1, we provide an overview of the framework and an insight into the application domain in which we attempt to operate. In Section 5.2, we present a simplified model of this domain, and in Section 5.3, a metamodel for routing problems. In Section 5.4, we then describe a model transformation from the domain model to the models described by this metamodel. Finally, we briefly summarize this chapter in Section 5.5.

5.1 Overview

The modeling framework consists of *a domain model of routing problems*, *a routing metamodel for algorithmic manipulation* and *a model transformation* with an adjoining transformation language. A central element of the developed framework is the routing metamodel which is used to describe the actual optimization models. The metamodel is described here by giving a set of invariants on the structure of the optimization models conforming to the metamodel. These invariants also give an unambiguous description of the process of solving the problems described

by the models. The model transformation is used to produce these case-specific optimization models from the domain model. In practice, the domain model is an object-oriented representation of the relevant parts of the domain of routing problems. As the domain model is conceptually relatively simple, it is presented here primarily to provide a common understanding of the problem domain.

An overview of the problem domain can be given with the following example. Consider a case where a logistic operator performs the design of operations involving a transportation of perishable goods (such as food supplies) from a number of warehouses to small stores in an urban area. Moreover, some stores produce goods (such as empty bottles) that need to be transported from the store to a collection center.

The resources the operator can employ consist of a fleet of vehicles and a set of drivers for these vehicles. The vehicles reside on two depots near the city center, and for flexible utilization, there are three types of vehicles with differing sizes and costs. Some of the transported goods need special equipment, such as a cold storage, and a subset of vehicles has this capability. Furthermore, a set of trailers can be assigned to some of the vehicles to provide extra capacity with an additional cost. Not all customers, however, can be served with large vehicles or with trailers as some of the shops are in small alleys.

Other requirements for the plans include that the drivers have differing abilities; some, for instance, do not have licenses for operating all the vehicles or driving with trailers. The legislation also requires that each driver must have breaks of 15 and 30 minutes during the nine hour day. Furthermore, the transported goods should not remain in the vehicle for exceedingly long periods, some goods should not be transported together in the same compartment, and the goods should arrive at the shops during designated time windows.

Given the problem definition, the task of the operator is to choose which drivers employ which vehicles and which vehicles visit which stores. In addition, the goods should be assigned to a suitable compartment. As the plan is being executed, new orders may be placed and the plan needs to be adjusted to accommodate these changes. The objective is to minimize the total costs to the operator.

In the subsequent sections, we go through modeling tools that are capable of expressing these elements of routing problems. The choice of assigning the deliveries to drivers and sequencing them to concrete drive plans requires mapping the tasks to the drivers. This forms the basis of these types of problems. The modeling elements for these decisions are examined in Section 5.3.2. As, e.g., the time used during the operation is central in many of the requirements stated for the operator, we examine accumulation of these types of resources in Section 5.3.3 under the name “resource projection”. In addition, the operator needs to consider different types of capabilities for performing the tasks; for instance, depending on the choice of vehicle, the driver may be able to employ a trailer or serve a specific store. The dynamics of capabilities are captured by “capability projection” examined in Section 5.3.4. These two concepts are combined under the element “partial resource projection” in Section 5.3.5. This combination al-

lows specification of more detailed requirements. For example, the orders can be tracked individually, which allows the expression of the maximum allowed time on a vehicle for each order. As the usage of a trailer increases the travel times and costs, a mechanism named "stack resource projection" is introduced to model these types of situations. Section 5.3.6 introduces such a concept for expressing the changes in the delta of resources or costs due to different capabilities. Finally, Section 5.3.7 presents a mechanism for expressing a set of mutually exclusive decisions. This can be used to model, for example, the loading decisions of the goods and the decisions for the breaks of the drivers. The different mechanisms are composed of different modeling elements, which are introduced throughout this chapter.

5.2 Domain Model

The domain model within the system is the representation of the problem domain in object-oriented terms. We present here the part of the domain model that is visible to the model transformation engine; thus, for instance, classes representing the domain data in a more human-readable form are omitted. Examples include entities such as schedules, routes, and depots. The core domain model is depicted in Figure 13 and illustrates the base structure of the model which can then be extended case by case within the product line architecture. The overall schema is somewhat simplified, but does communicate the relevant elements required in modeling rich routing problems in a coherent way. The figure illustrates the domain elements and their relationships. The domain elements given are enumerated below.

`Plan` is the root object of the structure and represents the collection of all the drivers, the equipment for the vehicles, and the tasks in the problem. It also keeps track of unassigned tasks — that is, tasks that have not been given to any vehicle.

`Driver` represents a driver of a vehicle. Each driver corresponds to a potential route on the problem as they are needed for performing the tasks. In cases where the drivers are not relevant to the problem, vehicles and their properties may be used instead when specifying the properties of the empty routes. In both cases, each driver is eventually given a vehicle whose list of tasks indicates the tasks that are performed by the driver. In addition, `Driver` inherits the `Locatable` entity as each driver may have a predefined start and end location and time windows indicating availability.

`Vehicle` represents either a potential route, or the start and end locations of the vehicle, depending on the specifics of the case. Each `Driver` may have one designated vehicle whose properties are given to the route, or, if fleet selection has to be made, the assignment of vehicles to drivers may be left for the optimizations. A vehicle has a list of tasks that have been assigned to be performed using the vehicle. `Vehicle` inherits the `Locatable` entity as each vehicle may have

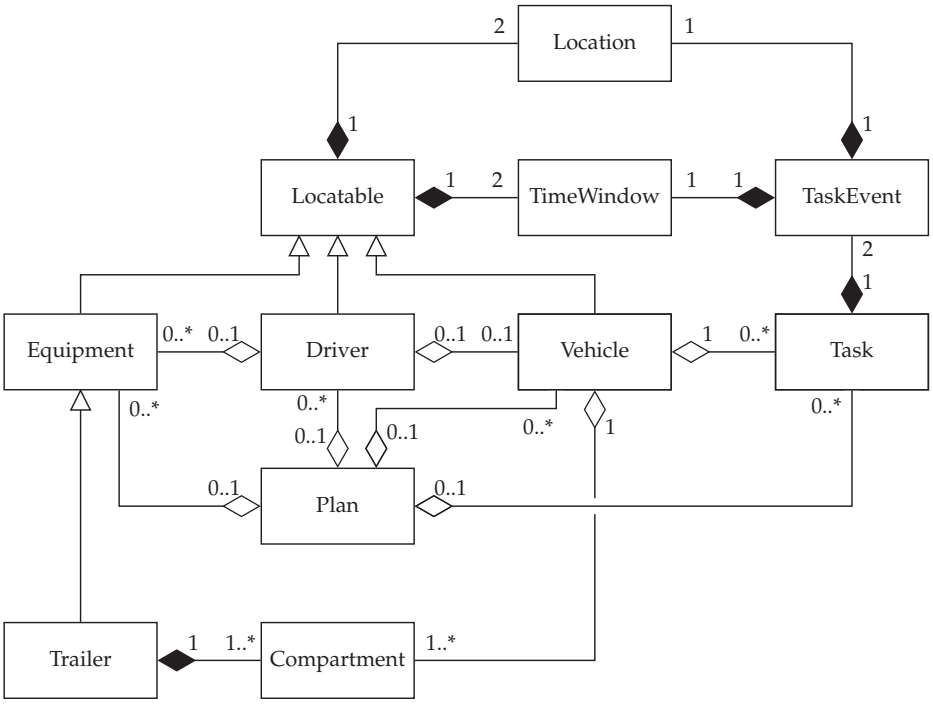


FIGURE 13 An overview of the elements of the domain model visible to the model transformation engine.

a predefined location and time windows indicating availability.

Task represents an abstract task on a route. As the domain model is based on pickup and delivery problems, a task typically consists of two parts: pickup and delivery, or the start and end `TaskEvent` of a task. Properties of tasks include, e.g., capacity and compatibility requirements.

TaskEvent represents a half of a task; pickup or delivery. Each task event has properties describing the task itself, such as location, time windows, compatibilities, requirements, and duration of the task.

Locatable is an abstract entity representing an entity with two locations, start and end, and adjoining time windows. This entity is extended by `Equipment`, `Vehicle`, and `Driver`.

Location represents an instance within, e.g., the distance matrix. Each vehicle, piece of equipment, driver, and task has two locations: start and end.

TimeWindow represents an interval in time and is used to represent, for instance, deadlines for tasks and availability of vehicles.

Compartment represents a capacitated container capable of providing multidimensional capacity for completing tasks of corresponding capacity. In multicompartment routing, each compartment corresponds to a decision by itself. In cases where the loading decisions are not part of the problem, the total capacity of compartments is used as the capacity of the vehicle.

Equipment represents a piece of equipment that may be needed for performing certain tasks. In some problems, these properties are generated to the vehicles, but if there is a need to include them as decisions, they may be generated as decisions on the route. Properties of these entities include compatibilities with drivers and vehicles and the capabilities they add for performing tasks.

Trailer represents a special piece of equipment that holds additional compartments. If trailers are needed as decisions within the problem, they can be included as decisions on the route; otherwise they may be generated within the properties of the vehicle or driver they are used by.

The schema here is not a complete representation of the domain model, and is intended to be more illustrative than definitive. This is the generic domain the optimization is applied on. We examine the properties of these entities in more detail when we discuss the transformations within different cases. The properties of these entities are not listed here, and they are discussed solely from the optimization model viewpoint. That is, we examine only the parts used in the problem generation process.

5.3 Routing Metamodel

In this section, we formally define the elements of the developed conceptual modeling framework[1]. First, in Section 5.3.1, we locate the elements of our modeling framework within the broader picture and provide an overview of their structure.

[1] As opposed to a software framework.

In Sections 5.3.2–5.3.8, we formulate the model elements and gradually build a complete modeling approach. The overview is discussed using generic algebraic formalizations, and the remaining sections — to provide bridge to implementation — in Z notation [114], a formal specification language. For an introduction and an informative description of the language, we refer to the Z notation reference manual by Spivey [192].

5.3.1 Classification of Constraints

Combinatorial optimization is defined as the act of choosing, partitioning, and ordering a finite set of elements. Now, solving routing problems, defined by choosing, ordering, and assigning a set of tasks to a set of vehicles, can be seen more generally as an act of mapping and ordering a (possibly sub-) set of given *activities* to a given set of *actors* performing these activities.

In this chapter, we design a graph encoding for routing problems. The encoding is partitioning-based and builds on resource constrained paths along with some additional constructs. As discussed in Chapter 2, we form paths on the graph of the problem, and each such path is a sequence of operations performed by the entity assigned to operate on that path. Each decision on which operations are performed by whom is a decision on *mapping*, and each decision on the exact sequence, that of *ordering*.

The decisions on mapping and ordering are subject to constraints. Generally, a constraint function is any function $\mathcal{R} : \mathbb{M} \times \mathbb{O} \rightarrow \mathbb{R}^n, n \in \mathbb{N}$, where $\mathbb{M}$ is the space defined by the values of the decision variables on mapping, and $\mathbb{O}$, the space defined by the values of the decision variables on ordering. The feasible region is defined by a set of inequalities for the values of the function in $\mathbb{R}^n$. As the structure of these constraint functions largely defines the computational complexity of the solution process, it is useful to identify classes of functions that are expressive enough for describing routing problems, but efficient enough for an optimization process. In particular, we are interested in the properties that affect the *computational complexity* of their evaluation. In this section, we identify a set of such key properties for analyzing the optimization in the developed framework later in Chapter 6.

We identified a class of constraint functions in Chapter 2 that is particularly suitable for routing. *Resource extension functions* are a special case of a class of constraints we denote here as *mapping-ordering* constraints. These type of constraints depend on both decisions on mapping and ordering. Other such classes are then constraints on *mapping* and constraints on *ordering*, which can be computed solely from the values of the decision variables on mapping, and those on ordering, respectively.

Constraints on mapping are functions of type $\mathcal{R} : \mathbb{M} \rightarrow \mathbb{R}^n, n \in \mathbb{N}$. These types of functions are useful for stating restrictions between the entities the mapping is done to. The computation of these functions does not require traversal of the sequences resulting from the decisions on ordering, and this makes the computation of these functions solely dependent on the number of changing vari-

ables involved. In practice, computing global mapping constraints can be tedious as they may affect structures in different parts of the solution (e.g., decisions on different routes). Constraints on mapping can be used, for example, to require a balance in the number of activities between routes, to define mutually exclusive activities, and to state incompatibilities between activities and actors. One type of mapping constraints will be examined within the developed framework.

In contrast, **constraints on ordering** are functions of type $\mathcal{R} : \mathbb{O} \to \mathbb{R}^n, n \in \mathbb{N}$. Constraints on ordering are less useful in routing problems as the decision on mapping is central to the design of the routes, but these types of constraints could be used to state that, for example, *the third node of each route* should adhere to some restriction regardless of the actual route and the nodes preceding it[2]. As all of the typical constraints in routing problems depend on some decisions on the mapping of activities, these types of constraints are not employed within the developed framework.

Constraint functions of mapping-ordering type, that is, $\mathcal{R} : \mathbb{M} \times \mathbb{O} \to \mathbb{R}^n, n \in \mathbb{N}$ can be divided into several subclasses according to their computational properties. The most interesting class of constraints in the context of routing problems is the constraints based on resource extension functions. In the broader class of constraint functions, the characteristic of these functions is that they can be computed incrementally along a path. These can be defined as follows. Let p_i be the i^{th} node on path P. Given a constraint function $\mathcal{R}$, constraints based on generic resource extension functions can be expressed by stating that

$$\mathcal{R} = \mathcal{F}_{p_{q-1},p_q}(\mathcal{F}_{p_{q-2},p_{q-1}}(\dots \mathcal{F}_{p_2,p_3}(\mathcal{F}_{p_1,p_2}(\emptyset))\dots)), \tag{56}$$

where $q \in \mathbb{N}$ is the length of the path, $\mathcal{F}_{ij} : \mathbb{R}^{\bar{\gamma}} \to \mathbb{R}^{\bar{\gamma}}$ is the resource function describing the change from $i \in P$ to $j \in P$, and $\bar{\gamma}$ is the number of resources in the problem.

The given generic definition does not yet communicate much structure on resource extension functions. There are four properties that largely define the structure of these types of constraints, that is, their

- monotonicity,
- sources of dependency,
- types of dependencies, and
- boundedness.

Monotonicity refers to the monotonicity of the constraint function that results from the composition of a set of resource extension functions. The constraint function can either describe a change that is non-negative (or non-positive) at each transition, i.e., be *monotonic*, or allow both types of values, i.e., be *non-monotonic*. This property affects some solution approaches, but is not directly

[2] We do not know the preceding node unless we know which route we are on.

relevant in the context of the developed framework as it stands now. This property is included mainly for completeness.

Sources of dependencies refer to those resources whose values are needed in computing the value of the resource in question. The trivial case occurs when *no dependency* is present, but this case is degenerated as the value cannot accumulate if there is no dependency on the previous value of the resource. The typical case is one where the resource is dependent on *itself*. A resource can also be dependent on *other* resources, or *both* its previous value and that of other resources. These dependencies affect the order of computation that must occur in the evaluation of the constraint functions, and if a resource is dependent on another resource which is computationally expensive to evaluate, also the evaluation of the resource in question becomes expensive in this sense.

Types of dependencies refer to the types of the functions needed in computing the change in a resource while traversing the path. There are two base types of dependencies: *rule dependency* and *value dependency*. A resource is rule-dependent on another resource if its value not only depends on the value of the other resource but also *becomes irrelevant* with respect to the rules on the allowed values. In a sense, these types of resources can be seen as subresources of a given resource. Value dependency, on the other hand, can further be divided into three distinct classes. The generic case is given in Equation (56), and named here as *undefined* dependency. The name refers to the fact that we cannot determine the exact change in resource values unless we have evaluated the situation at the previous node on the path. These types of functions are, therefore, here referred to as *situation dependent*. A special case of the generic case is *transitional* dependency which allows us to evaluate each resource extension function as a component of the sum along the traversed path. These types of functions can be written in the form

$$\mathcal{R} = \mathcal{F}_{p_1,p_2} + \mathcal{F}_{p_2,p_3} + \cdots + \mathcal{F}_{p_{q-1},p_q}. \tag{57}$$

A further special case of transitional functions are the so-called *fixed* functions whose value depends only on the nodes on the path, not the traversed edges. These types of functions can be written in the form of

$$\mathcal{R} = \mathcal{F}_{p_1} + \mathcal{F}_{p_2} + \cdots + \mathcal{F}_{p_n}. \tag{58}$$

These two types of constraint functions are much easier to evaluate when performing small changes to the decision variables as they can be decomposed into independent components. This will be examined in more detail in the subsequent chapter.

Boundedness refers to the property which defines whether the function can be bounded after each resource extension evaluation. A function is *free* when it has not been bounded, and *bounded*, if it is, for example in a *transitional* case, of the form

$$\mathcal{F}_{p_i,p_j\gamma}(x) = \max\{\alpha^{\gamma}_{p_i,p_j}, x + \Delta^{\gamma}_{p_i,p_j}\}, \tag{59}$$

where $\alpha^{\gamma}_{p_i,p_j}$ is the lower bound and $\Delta^{\gamma}_{p_i,p_j}$ is the change in value for resource γ when traversing from p_i to p_j.

Within these classes of constraint functions, we may identify, for example, the generic resource extension function. They can be classified as non-monotonic, bounded, and value-dependent (undefined) on both their values and those of other resources. The classical resource extension function [110] can be characterized as monotonic, bounded, and value-dependent (transitional) on itself.

In the following sections, we construct a modeling approach using the different classes of constraints, and introduce, for example, *a set of different types of projections on different types of resources*. These mechanisms introduce a set of computation rules for these resources. The sections are named according to the primary element computed by the projection in the given section. As a whole, this **mechanism** for computing with the modeling elements in question defines a set of constraint functions. These functions consist of a set of resources, their types, dependencies between the resources, a definition of the accumulation of these resources, and the adjoining rules determining the allowed range of the resulting values. For convenience, we list all the schema types in this specification in an index in Appendix 1.

We follow a set of conventions in introducing the elements of the framework. Whenever applicable, the following outline is used. First, we introduce the mechanism and provide an illustrative example. Then the input to the mechanism is discussed. This is followed by an introduction to the basic types and a definition of a binding schema that binds the new variables introduced to the existing ones. To differentiate between constraints and rules, we refer to the schemas computing the values of the constraint functions as **projections** as they can be thought to "project" the values of the decisions into the constraint space. The schema specifies the declarations needed in the mechanism and the predicates used to describe their relation to the existing declarations. Subsequently, we introduce a more detailed schema, which is used to describe the predicates needed in computing with the mechanism. The names of these two types of schemas are suffixed with *-Binding* and *-Restriction*, respectively. These schemas are then combined into a complete mechanism, and a comment on its location in the given taxonomy is given. This is followed by an introduction to the necessary **rules**, also in two parts. Finally, any possible additional related schemas are introduced, and the section is completed by a summary in which we combine the newly introduced schema into the larger schema type being developed.

When we discuss resources within the projections of different types, we need to distinguish two distinct types: those that are *resources in a classical sense*, and those used here as an abstract type. While resource projection introduces a set of resources that corresponds one to one with classical resources, the subsequent mechanisms introduce more complex cases. Capabilities, for instance, can be seen as *a pair of classical resources*, and thus when we discuss the classification of the elements of these mechanisms, we explicitly refer to classical resources where needed. Otherwise, the resources refer to the types defined in the given schemas.

Before we move to examine the different constraint-based constructs of routing problems, we begin the definition of the modeling framework by formalizing the needed decisions.

5.3.2 Decision Variables

Mapping and ordering decisions are encoded into the structure of the routes by assigning the activities to actors and constructing sequences from these activities. This section specifies the formal structure used in this modeling framework for capturing the exact meaning of these two decisions. The modeling concepts are expressed in Z notation, and for a reader unfamiliar with Z, an introduction to the notation is given in Appendix 2.

We begin by formalizing the basic concepts of *activity* and *actor*.

$$[Activity]$$

$$\begin{array}{|l} \textit{Actor} \\ \hline start : Activity \\ end : Activity \\ \hline \end{array}$$

Actors have starting and ending activities, which must be found on the route of that activity. These activities equip actors with properties which any regular activity might have.

Next we define the schema of the *Ansatz* itself. First, however, we introduce a concept of *paths*, defining a (non-empty) sequence of elements related to an actor.

$$\text{paths}\, X == \{ f : Actor \leftrightarrow \text{seq}_1\, X \}$$

Now the *Ansatz* can be defined as follows.

$$\begin{array}{|l} \textit{Ansatz} \\ \hline actors : \mathbb{F}\, Actor \\ activities : \mathbb{F}\, Activity \\ routes : \text{paths}\, Activity \\ assigned : \mathbb{F}\, Activity \\ unassigned : \mathbb{F}\, Activity \\ \hline \text{dom}\, routes = actors \\ assigned = \bigcup \{ route \in \text{ran}\, routes \bullet \text{ran}\, route \} \\ assigned \cup unassigned = activities \\ assigned \cap unassigned = \emptyset \\ \forall\, actor \in actors \bullet \forall\, route \in routes(\!|\{actor\}|\!) \bullet \\ \quad actor.start = head\, route \land \\ \quad actor.end = last\, route \\ \hline \end{array}$$

In the described schema, *actors* is the set of actors associated with the problem the *Ansatz* corresponds to, *activities* the set of activities defined within the problem, and *routes* a relation mapping a set of activities to actors and ordering them into sequences. Furthermore, we denote the set of activities that have been mapped to

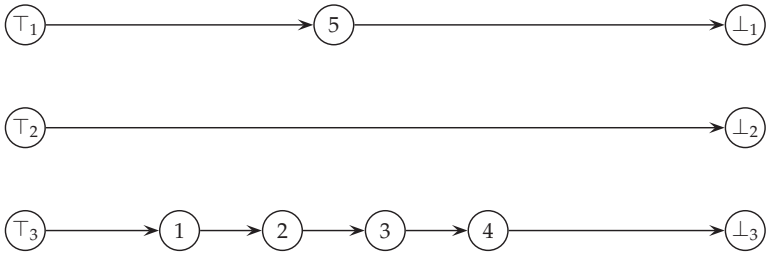


FIGURE 14 An example of an acceptable mapping forming three routes within an *Ansatz* of a mapping and ordering problem.

an actor by a collection of activities named *assigned*, and those that have not been mapped to an actor, *unassigned*.

The defined properties of the *Ansatz* state the allowed values of each variable. First, we require that the domain of the relation mapping the sequences of activities to actors is the set of activities within the problem. Second, the set of assigned activities is exactly those activities mapped by relation *routes*. In addition, the activities within the problem are either on the set *assigned* or the set *unassigned*. Finally, we state that each sequence of activities must contain the start and end activities of its actor as its first and last element.

When the values of all the decision variables within the problem can be evaluated unambiguously from the mapping and ordering of activities, the problems described by the given schema correspond to those given in Chapter 2. However, for this we must assume that it is never optimal to wait at an activity if it is not specifically required, that is, leaving later will never, e.g., lower the costs or result in an earlier arrival. This issue is examined in more detail when discussing the usage of resource extension functions within this modeling framework.

An example of an *Ansatz* with five activities assigned to three actors forming three routes is depicted in Figure 14. The following convention is used in the figures of this chapter: the actor start for the actor number[3] i is denoted with the symbol $\top_i$, and the actor end, with $\bot_i$. Each activity is denoted with a plain number. In the example, actor 3 has been assigned activities 1, 2, 3, and 4, whereas actor 1 has only one activity. Actor 2 has an empty route. Note that an empty route is formed by a set containing only a starting activity and an ending activity. As such, this construct is analogous to the partitioning mapping in the graph-based encoding described in Chapter 2, but with each vehicle having a unique start and end.

As mapping and ordering the activities to actors results in sequences and in order to compute changes in the state of these paths as we traverse them, we will establish a set of computation rules for this purpose. To do this conveniently, we

[3] Strictly speaking, we do not identify actors with an id, but the reader may assume an injection from natural numbers to all the actors of the problem for this purpose.

define operators for expressing the traversal between the activities along these sequences.

$[X]$

$$next : (\operatorname{seq} X \times X) \pfun X$$
$$prev : (\operatorname{seq} X \times X) \pfun X$$

$$\forall s : \operatorname{seq}_1 X;\ x \in \operatorname{ran} front\, s \bullet next(s, x) = \mu\, n : \mathbb{N} \mid s(n) = x \bullet s(n+1)$$
$$\forall s : \operatorname{seq}_1 X;\ x \in \operatorname{ran} tail\, s \bullet prev(s, x) = \mu\, n : \mathbb{N} \mid s(n) = x \bullet s(n-1)$$

The *next* operator for an element of a given sequence identifies at most one other element from that sequence. That element is the immediate successor to the former. Similarly, the *prev* operator for an element of a route identifies at most one other element from that route. That element is the immediate predecessor to the former.

Note that both of these functions are well-defined for every activity that has been assigned to an actor as these activities are guaranteed to have at least the actor start activity occurring before it and the actor end activity occurring after it. The results of these functions are left unspecified if the next or previous element does not exist.

Our definition of mapping and ordering problems did not specify how the mapping and ordering should be done. In this context, we refer to mapping and ordering problems with three additional assumptions; that is,

- each actor has a unique route,
- the routes are disjoint, and
- each activity occurs at most once on a route,

and denote such problems as *proper* mapping and ordering problems.

Although we could include the assumptions into the schema of the *Ansatz*, we present each assumption individually as we would like to keep the base schema as generic as possible for further revision and a possible relaxation of some of these assumptions. It is also more straightforward to discuss each assumption individually, and this allows us to highlight the differences between the generic and the proper mapping and ordering problems as these assumptions are somewhat strong and central to our modeling approach.

To identify actors with routes, we define that

Uniqueness

$$routes : \operatorname{paths} Activity$$

$$routes \subseteq \{ f : Actor \pfun \operatorname{seq} Activity \}$$

which gives each actor in the *Ansatz* at most one route. This means that, for instance, periodic problems must be modeled as additional visits to a depot instead of actual multiple sequences of activities. When the activities are not assigned on multiple routes and the constraints refer to a single actor, the constraint evaluation is simplified: the changes in a route of one actor do not require evaluation of the feasibility and objective on other routes.

To define that activities must reside on at most one route at a time, we may state that the routes must be disjoint. Stating disjointness is a matter of enumerating all actors and their activities and using the actors as an index set for the operator *disjoint*.

Disjointness

$$routes : \operatorname{paths} Activity$$

$$\mathsf{disjoint}\ \{ actor : \operatorname{dom} routes \bullet actor \mapsto \operatorname{ran} routes(\!|\{actor\}|\!)\ \}$$

If we assume disjointness, depots and other shared locations have to be modeled by generating distinct activities for each potential visit to them. This has naturally the downside that it usually results in a larger problem.

As we discussed in the preceding section, resource extension functions are computed by traversing a path. Now, if we assume that each activity occurs at most once at a sequence, we are able to define a unique value of a resource extension function for each pair of activities. Thus, we define that

Injectivity

$$routes : \operatorname{paths} Activity$$

$$routes \subseteq \{ f : Actor \leftrightarrow \operatorname{iseq} Activity \}$$

which results in injectivity on the ordering function; and since each activity can now appear on a route at most once, this results in that each task, for instance, at a single customer, has to be modeled as an individual activity.

Using the schemas given above, we may now define the proper *Ansatz* for mapping and ordering problems as

$$\begin{aligned} ProperAnsatz \mathrel{\widehat=} & \ Ansatz \land \\ & Uniqueness \land \\ & Disjointness \land \\ & Injectivity. \end{aligned}$$

The resulting model is similar to the GPDP where each vehicle starts from and ends at an arbitrary point, and each task may have multiple parts. In the subsequent sections, we further expand the model for expressing a deterministic, discrete general pickup and delivery problem with complex constraint, decision, and objective extensions modeled by resource constrained paths, resource extension functions, mutually exclusive and optional points, and specific labeling mechanisms for modeling dependencies between different resource extension

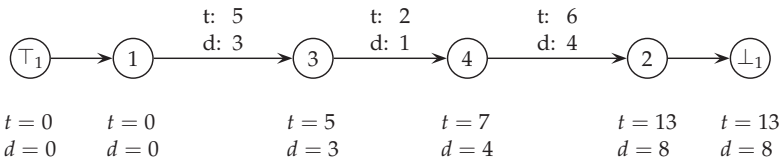


FIGURE 15 An example of the results of two resource delta functions, travel time and distance, for usage on transitional projections.

functions. However, within the framework, we do not assume that each problem is of the PDP type: as we will see, these constructs are able to express, for instance, a simple CVRP.

5.3.3 Resources

The resources accumulated along a route have certain limits which correspond to constraints for defining an acceptable path. In this section, we define a relatively simple resource projection mechanism for tracking accumulated numeric values during the traversal of the route and defining the acceptable ranges of the resulting values. The two basic elements of such a mechanism are *deltas* and *values*. The former are used to describe the change at a given point during the traversal, and the latter, the result of the accumulation.

The basic principle of resource projection can be illustrated as in Figure 15. The deltas of two functions are depicted above the paths, and the resulting values of the accumulated resources are listed underneath each activity. The set of all resources within the problem is denoted by the set *resources*. In the given example, $\mathit{resources} = \{t, d\}$. Now, traversing, e.g., from activity 1 to activity 3, increases the time traveled ("t") by 5 units, and the distance traveled ("d") by 3 units, and these are computed in a similar fashion between all the activities on the route. The accumulated values are stored for each activity and denoted here as *resourceValues*. Notice that, in this case, traveling from the route start activity and to the route end activity has zero effect on these two resources.

Resource projection is given a set of resources and, for each such resource, a delta function that describes its change whenever a traversal between two consecutive activities occurs on a path. As the deltas are functions from the Cartesian product of activities to real values, they can also be represented as matrices. The often needed time and distance matrices are, in fact, encoded into these transitional functions.

As we need to restrict the allowed values, the mechanism can be given functions defining both upper and lower bounds on the accumulated values. The upper bounds are used to state a maximum on the accumulated values, and the lower, a minimum. However, the lower bounds may be used in two ways: we may require that the accumulated value is strictly within the resource window, or

we may allow automatic accumulation — "waiting" — such that the lower bound is met. Bounds on resources with the possibility of waiting are later used to define, e.g., regular time windows on activities. Both of these bounds are encoded into the projection, and the distinction between these two modeling elements is done according to their effect on the computation. Any element affecting the result is defined within a *projection*, and any constraint on these values is introduced within a *rule*. Therefore, as the lower bounds may affect the result of the projection due to a possibility of *waiting*, they are defined within projections. The upper bounds — asymmetrically — appear only in rules.

We begin formalizing the resource projection mechanism by defining its basic building blocks. First, as we traverse the defined path, we compute the changes in the value of the resource and for this we employ a transitional projection as discussed in the model element taxonomy. A resource delta function is such a projection: a function from two activities to a numeric value.

$$ResourceDelta == (Activity \times Activity) \pfun \mathbb{Z}$$

As we have mentioned, there may be several resources in a problem, and we introduce an abstract type for identifying these different resources.

$$[Resource]$$

Applying these computations results in a set of resource-specific numeric values. The function mapping each resource to its value is used in this specification.

$$ResourceValue == Resource \pfun \mathbb{Z}$$

Using the building blocks given, we may now define the first projection in detail. For resource projection, we introduce the elements given in the schema below.

ResourceBinding

$$\begin{aligned}
&activities : \mathbb{F}\, Activity \\
&assigned : \mathbb{F}\, Activity \\
&resources : \mathbb{F}\, Resource \\
&resourceDeltas : Resource \pfun ResourceDelta \\
&resourceLowerBounds : (Activity \times Resource) \pfun \mathbb{Z} \\
&strictLowerBoundResources : \mathbb{F}\, Resource \\
&resourceValues : Activity \pfun ResourceValue \\
\hline
&\mathrm{dom}\, resourceDeltas = resources \\
&\forall f \in \mathrm{ran}\, resourceDeltas \bullet \mathrm{dom}\, f = activities \times activities \\
&\mathrm{dom}\, resourceLowerBounds = activities \times resources \\
&strictLowerBoundResources \subseteq resources \\
&\mathrm{dom}\, resourceValues = assigned \\
&\forall f \in \mathrm{ran}\, resourceValues \bullet \mathrm{dom}\, f = resources
\end{aligned}$$

First, a finite set of resources is defined for the problem. The resource delta functions are defined on each of these resources, describing the change of value for that particular resource while traversing the given path. As the path consists of activities defined in the problem, each such resource delta function is defined on the Cartesian product of these activities. The lower bounds on resources are defined using a function from a resource-activity pair to a numeric value. In addition, we define a set of resources whose accumulated values must remain strictly within the resource windows, that is, allow no waiting. This set is a subset of all resources on the problem. The resource value type stores a numeric accumulated value for each resource, and as we need an accumulated value for each activity on the path, the set of resource values is a function from all the assigned activities of the problem to such a resource value. These resource value functions describe values for the set of resources defined.

Computing with resources is defined in the following schema.

ResourceRestriction

$$
\begin{array}{l}
routes : \operatorname{paths} Activity \\
resources : \mathbb{F}\, Resource \\
resourceDeltas : Resource \pfun ResourceDelta \\
resourceLowerBounds : (Activity \times Resource) \pfun \mathbb{Z} \\
strictLowerBoundResources : \mathbb{F}\, Resource \\
resourceValues : Activity \pfun ResourceValue \\
waitingAt : (Activity \times Resource) \pfun \mathbb{Z} \\
\hline
\forall a, b : Activity;\ r : Resource \bullet resourceDeltas(r)(a, b) \geq 0 \\
\forall route \in \operatorname{ran} routes;\ r \in strictLowerBoundResources \bullet \\
\quad (\forall a \in \operatorname{ran} route \bullet waitingAt(a, r) = 0) \\
\forall route \in \operatorname{ran} routes;\ r \in resources \setminus strictLowerBoundResources \bullet \\
\quad (waitingAt(head\, route, r) = \\
\qquad resourceLowerBounds(head\, route, r)) \land \\
\quad (\forall a \in \operatorname{ran} tail\, route \bullet waitingAt(a, r) = max\{0, \\
\qquad resourceLowerBounds(a, r) - \\
\qquad\quad (resourceValues(prev(route, a))(r) + \\
\qquad\quad resourceDeltas(r)(prev(route, a), a))\}) \\
\forall route \in \operatorname{ran} routes;\ r \in resources \bullet \\
\quad (resourceValues(head\, route)(r) = waitingAt(head\, route, r)) \land \\
\quad (\forall a \in \operatorname{ran} tail\, route \bullet resourceValues(a)(r) = \\
\qquad resourceValues(prev(route, a))(r) + \\
\qquad resourceDeltas(r)(prev(route, a), a) + \\
\qquad waitingAt(a, r))
\end{array}
$$

First, we require that the result of resource delta functions is nonnegative. This assumption simplifies the computation with resources and makes the projection monotonic. To compute the projected values, we first need to define the computation of waiting. This is performed as follows. For each resource with a strict lower bound requirement, the length of waiting is always zero. For all other resources,

at the route start the length of waiting is the value of the corresponding lower bound, and at the subsequent activities, it is the difference between the value at the previous plus a delta and the lower bound. The values can now be computed by stating that the value at the route start is the length of waiting at that point, and that each subsequent value is equal to the value at the previous activity plus a delta plus the waiting at the current activity.

We combine the two schemas into a complete resource projection schema as follows.

$$ResourceProjection \mathrel{\widehat=} ResourceBinding \land ResourceRestriction$$

The mechanism presented in this section defines a set of mapping-ordering type of constraint functions which are monotonic, bounded, and have a transitional value-dependency on themselves. Thus we assume here that the resources defined in this mechanism are independent of each other. The upper bounds are defined on the corresponding rule as they do not alter the result of the projection.

The resource rule states that each accumulated value on an activity must fall on a certain range. Defining the rule introduces the following elements:

ResourceRuleBinding

$$activities : \mathbb{F}\, Activity$$
$$resources : \mathbb{F}\, Resource$$
$$resourceUpperBounds : (Activity \times Resource) \pfun \mathbb{Z}$$

$$\mathrm{dom}\, resourceUpperBounds = activities \times resources$$

The rule is given a set of upper bounds on each activity-resource pair. The upper bound is defined on the Cartesian product of the activities and resources of the problem.

The rule imposes a property described by the following schema:

ResourceRuleRestriction

$$routes : \mathrm{paths}\, Activity$$
$$resources : \mathbb{F}\, Resource$$
$$resourceValues : Activity \pfun ResourceValue$$
$$resourceLowerBounds : (Activity \times Resource) \pfun \mathbb{Z}$$
$$resourceUpperBounds : (Activity \times Resource) \pfun \mathbb{Z}$$

$$\forall\, route \in \mathrm{ran}\, routes;\ r \in resources \bullet (\forall\, a \in \mathrm{ran}\, route \bullet resourceValues(a)(r) \geq resourceLowerBounds(a, r) \land resourceValues(a)(r) \leq resourceUpperBounds(a, r))$$

For all resources, the resource values at each activity of each route must be greater than or equal to the corresponding lower bound and less than or equal to the upper bound on that activity. Note that if a resource does not require zero waiting, it conforms to the first part of this rule by definition.

We combine the two schemas into a complete resource rule as follows:

$$ResourceRule \mathrel{\widehat=} ResourceRuleBinding \land ResourceRuleRestriction$$

In this REF based approach, it is easy to see the reason for assuming that the decisions are encoded into the mapping and ordering of tasks. In the formulations given in Chapter 2, the arrival times, for instance, were considered decisions, but were in fact directly dependent on the decisions of travel. The formulations in that chapter exhibit a problem structure where unnecessary waiting always increases the costs or at least does not lower them. In the formulation given here, the accumulated values are by definition a direct and unambiguous result of the resulting paths; for instance, departure times can be computed from the departure time at the previous activity and any adjoining constraining time windows. This enables us to consider the decisions on activities as the sole decision variable.

As we noted, some constraint functions can be optional in a sense that they do not alter the feasible region in the search space, but change the structure of the constraint space so that evaluating feasibility and the objective can be performed with less computation. An example of such a function is the *resource slack function*. A slack function is similar to the corresponding regular constraint function, but it introduces additional dimensions to the constraint space. These dimensions denote the allowed change of value in different variables so that the *Ansatz* remains in the feasible region, and this information can be used to efficiently compute feasibility of a potential move as the optimization is performed. The slack functions have no corresponding rules as they are computed only in a feasible situation defined by the regular constraint functions.

The slack function for resources introduces the following elements.

ResourceSlackBinding

$$assigned : \mathbb{F}\, Activity$$
$$resources : \mathbb{F}\, Resource$$
$$resourceSlackValues : Activity \pfun ResourceValue$$

$$\mathrm{dom}\, resourceSlackValues = assigned$$
$$\forall f \in \mathrm{ran}\, resourceSlackValues \bullet \mathrm{dom}\, f = resources$$

Resource slack values is a partial function from activities to partial functions from resource to a numeric value. The function is defined on the currently assigned activities of the problem and the resulting functions on the resources of the problem.

This function can be computed without additional input data, except a sufficiently large number M, and it results in slack values on each activity on the problem.

ResourceSlackRestriction

$$
\begin{array}{l}
routes : \text{paths}\, Activity \\
resources : \mathbb{F}\, Resource \\
resourceValues : Activity \pfun ResourceValue \\
resourceUpperBounds : (Activity \times Resource) \pfun \mathbb{Z} \\
resourceSlackValues : Activity \pfun ResourceValue \\
M : \mathbb{Z} \\
\hline
\forall\, route \in \text{ran}\, routes;\ r \in resources \bullet \\
\quad (resourceSlackValues(last\, route)(r) = M) \land \\
\quad (\forall\, a \in \text{ran}\, front\, route \bullet \\
\qquad resourceSlackValues(a)(r) = \\
\qquad min\{resourceSlackValues(next(route, a))(r), \\
\qquad\quad resourceUpperBounds(a, r) - resourceValues(a)(r)\})
\end{array}
$$

A sufficiently large number M is given as a starting slack at the last activity of each sequence. At all other activities on the sequence, the slack value is the smaller of the slack value at the next activity and the upper bound on the activity minus the projected resource value.

We combine the two schemas into a complete resource slack function as follows:

$$
\begin{array}{l}
ResourceSlackProjection \mathrel{\widehat=} ResourceSlackBinding \land \\
\quad ResourceSlackRestriction
\end{array}
$$

Finally, the resource constrained routing problem can be defined by combining the given schemas with the proper *Ansatz* defined earlier:

$$
\begin{array}{l}
ResourceConstrainedProblem \mathrel{\widehat=} ProperAnsatz \land \\
\quad ResourceProjection \land \\
\quad ResourceRule \land \\
\quad ResourceSlackProjection
\end{array}
$$

The defined schema is, however, not used for defining constraints in the final formulation of the routing metamodel. Instead, we employ the mechanism as a component of other mechanisms, and more importantly, use it for defining the *objective function* of the optimization problem. This is discussed in more detail in Section 5.3.8.

5.3.4 Capabilities

The mechanism for computing capabilities is a central element in the developed routing metamodel. It is used to describe the dynamics of the *capability* of an actor to perform tasks. An active capability denotes an ability to perform activities, such as serving a customer or picking up a trailer at a given time. Some activities increase the capabilities of the actor, some require, or consume, them. At all times, we should hold enough of each capability to enter an activity that

consumes those capabilities. This section will formally define this mechanism and illustrate its usage in various situations in different routing and scheduling variants and extensions.

The underlying motivation to employ capabilities comes from two primary sources. Firstly, the product line architecture, for which the modeling framework has been constructed, relies on a unified metamodel of routing problems, and this mechanism can be used, with the aid of other mechanisms, to generate different variants in a unified manner. Secondly, the structure of complex routing variants is particularly suitable for resources modeled as capabilities: there is a broad class of constraints that are relevant only at a subset of the route of an actor. These constraints are, in this sense, local. This structure effectively defines a "state" at each activity, and, based on the currently *active* capabilities on the route, we can track that state while we traverse the route. This state captures the relevant constraints at a given time. In other words, we attempt to model the state of an actor at various points in time. Indeed, this approach bears some resemblance to computing with state machines. This accumulated state can then be used to define different aspects of different routing variants and extensions.

The accumulated states defined by the mechanism computing the capabilities tackle two distinctive classes of use cases: *consumed capabilities* and *non-consumed capabilities*. Consumed capabilities can be used to model non-monotonic resources such as capacity in PDP cases, tying together different parts of tasks, imposing ordering on activities, setting bounds on completing tasks that consist of multiple parts, and so on. Non-consumed capabilities may be used to express requirements for certain capabilities, such as special equipment which is not consumed when it is being used, and other activity-specific constraints, such as time windows. In this section, however, we do not yet consider bounds on other resources, such as time; this is addressed in the subsequent section where we combine the principle of resource projection with capabilities.

In practice, capabilities are defined in terms of identifiers and their changes in activities. Each capability at an activity is identified with an identifier, or label, and a value it changes at the activity. The figures present the capabilities of activities above the depicted routes and the corresponding accumulated values below them. In general, we employ the following convention in notation of capabilities. The semantics of "x $+n$" are that we acquire n units of capability "x", and similarly, "x $-n$", we consume n units of the same "x". In other words, the prefix denotes whether we require ("$-$") capability or acquire ("$+$") it, and the number, how much. Moreover, the value "x 0" is used to denote activation of "x", but no change in its value. In addition, we use "x $\exists$" ("x $\nexists$"), for which — we give the exact definition shortly, but for the time being — the semantics can be expressed as a requirement (prohibition) for capability "x" without altering its value.

For illustration, consider a pickup and delivery instance where a vehicle has to perform three pickups on a route ensuring that each pickup is delivered with the same vehicle. In addition, one pickup and delivery task (pair 3–4) can only be served by a certain vehicle, and two of the deliveries cannot be transported simultaneously (pairs 1–2 and 5–6). This instance is depicted in Figure 16. The il-

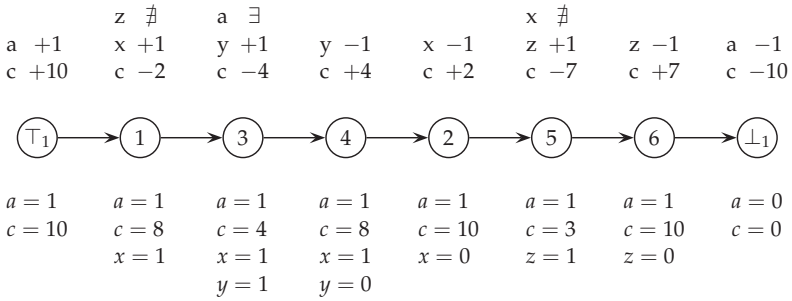


FIGURE 16 An example of a PDP instance with two pickups and deliveries on one route.

lustration is interpreted as follows. We begin the route at the start activity, where we obtain one unit of capability "a" and ten units of capability "c". The former denotes the used vehicle, and the latter, one-dimensional capacity. Upon entering activity 1 we make our first pickup losing two units of capacity, and obtain capability "x" denoting the ability to perform the corresponding delivery. We also ensure that the incompatible delivery ("z") is not in the vehicle. Entering activity 3 consumes four units of capacity and acquires, likewise, the ability to perform the corresponding delivery activity. As the current vehicle is the only capable of performing this task, we require capability "a" to be present. Subsequently, activity 4 is a delivery activity, and in it not only do we gain four units of free capacity, but also cede our cargo identified with "y". In activity 2, a similar operation is performed. Note how capability "y" is no longer relevant after it decreased to zero in the previous activity and was deactivated. The same operation occurs for "x" in the next activity, and this enables us to perform pickup for the last task as a prohibition for this capability is present at that activity. This task is completed at activity 6 and the route end activity removes the capability of the vehicle along its free capacity.

In practice, capability projection is given a set of capabilities (identifiers). Each activity is given a subset of these capabilities and an adjoining delta value which describes the changes in capability values at that activity. Using specific computation mechanics, the capabilities then specify the state of the actor during the traversal of the route of the actor.

Analogously to resource projection, we first introduce the basic building blocks of this projection to formally define it. The type describing the function used in computing the capability changes during the traversal of the path is a function from a single activity.

$$CapabilityDelta == Activity \pfun \mathbb{Z}$$

As with resources, we need to differentiate between different capabilities using a capability type.

$$[Capability]$$

The resulting values are numeric values for different capabilities.

$$CapabilityValue == Capability \pfun \mathbb{Z}$$

Using the types defined, we may give the basic declarations and predicates of capability projection. The projection introduces the following elements:

CapabilityBinding

$$activities : \mathbb{F}\, Activity$$
$$assigned : \mathbb{F}\, Activity$$
$$capabilities : \mathbb{F}\, Capability$$
$$capabilityDeltas : Capability \pfun CapabilityDelta$$
$$capabilityValues : Activity \pfun CapabilityValue$$

$$\mathrm{dom}\, capabilityDeltas = capabilities$$
$$\forall c \in capabilities \bullet \mathrm{dom}\, capabilityDeltas(c) \subseteq activities$$
$$\mathrm{dom}\, capabilityValues = assigned$$
$$\forall a \in activities \bullet \mathrm{dom}\, capabilityValues(a) \subseteq capabilities$$

First we introduce a finite set of capabilities to the problem. A capability delta function describes the capability change for each capability in the problem, and the domain of each function in the range of the delta function is a subset of the activities in the problem. This means that the activities of the problem have a subset of the capabilities defined on them. Each such capability is said to be active on that activity. Note that this is different from the activeness of the projected value; the capability is active on an activity on a path, if its capability value exists, and active on the activity, if its capability delta value exists. The projected values are given by a function which maps a capability value for each currently assigned activity on the problem. Each capability in the domain of the range of this function is a subset of all capabilities in the problem. This defines, as mentioned, that not all capabilities are active on the activities on the path.

Capabilities are projected into constraint space much like resources, but with the distinctive difference that all the capabilities are not relevant at every activity of the problem. We effectively define a sparse constraint matrix and keep track of the relevant parts of that matrix at each activity. The changes to the state are computed by a specific *capability arithmetic* within the mechanism. These rules for computing the capability values from activity to another define four different cases: simple accumulation, both activation and deactivation of capabilities, and requiring existence without consuming. Simple accumulation either increases ("$+$"), decreases ("$-$"), or keeps the value of a capability the same (zero or an inactive capability). Activation occurs whenever an inactive capability is first increased above zero. Deactivation, on the other hand, occurs whenever an inactive capability is added into a zero capability.

To define the capability arithmetic described, we need to specify *activeness* status of different capabilities formally. To do this, we employ the behavior of functions in Z: that is, *partial functions may be defined on a subset of their*

domain of declaration and left undefined in other parts. In this specification, we state that the value of capability c is active in activity a, if it is in the domain of $capabilityValues(a)$. To make the semantics of capability projection explicit, we introduce functions for defining the activeness and value of capability values, as well as obtaining the actual value or zero (in case of inactive capability) from a capability value function.

The functions used in the capability arithmetic and the subsequent schemas can now be given as in the following schema:

$$
\begin{array}{l}
isActive : \mathbb{P}(Activity \pfun CapabilityValue \times Activity \times Capability) \\
isZero : \mathbb{P}(Activity \pfun CapabilityValue \times Activity \times Capability) \\
valueOrZero : (Activity \pfun CapabilityValue \times Activity \times Capability) \pfun \mathbb{Z} \\
\hline
\forall f : Activity \pfun CapabilityValue; \; a : Activity; \; c : Capability \bullet \\
\quad (isActive(f, a, c) \Leftrightarrow (a \in \mathrm{dom} f \land c \in \mathrm{dom} f(a))) \land \\
\quad (isZero(f, a, c) \Leftrightarrow (isActive(f, a, c) \land f(a)(c) = 0)) \land \\
\forall f : Activity \pfun CapabilityValue; \; a : Activity; \; c : Capability \bullet \\
\quad (valueOrZero(f, a, c) = \\
\qquad ((isActive(f, a, c) \Rightarrow f(c)(a)) \land (\neg\, isActive(f, a, c) \Rightarrow 0)))
\end{array}
$$

Using these definitions[4], the capability computation can be expressed as follows:

$$
\begin{array}{l}
\textit{CapabilityRestriction} \\
\hline
routes : \mathrm{paths}\, Activity \\
capabilities : \mathbb{F}\, Capability \\
capabilityDeltas : Capability \pfun CapabilityDelta \\
capabilityValues : Activity \pfun CapabilityValue \\
\hline
\forall route \in \mathrm{ran}\, routes \bullet \\
\quad (\forall c \in capabilities \mid head\, route \in \mathrm{dom}\, capabilityDeltas(c) \bullet \\
\qquad capabilityValues(head\, route)(c) = \\
\qquad\quad capabilityDeltas(c)(head\, route)) \land \\
\quad (\forall a \in \mathrm{ran}\, tail\, route \bullet \\
\qquad capabilityValues(a) = (capabilityValues(prev(routes, a)) \oplus \\
\qquad \{ c \in capabilities \mid a \in \mathrm{dom}\, capabilityDeltas(c) \bullet \\
\qquad\quad c \mapsto valueOrZero(capabilityValues, prev(routes, a), c) + \\
\qquad\quad\quad capabilityDeltas(c)(a) \}) \setminus \\
\qquad \{ c \mapsto v \in \mathrm{dom}\, capabilityValues(prev(routes, a)) \mid v = 0 \}))
\end{array}
$$

The first activity on each sequence has all its capability values equal to the capability delta values of that activity. All the subsequent activities are computed

[4] Note that, although strictly speaking they are required, we have omitted the formal relation definitions of these axioms, e.g., "`relation`(*isActive* _)". Adding these formalizations is a straightforward task if there is need, for instance, for automated verification of the specification.

by adding the capability values to their predecessor on the sequence. This is performed by stating that the capability values of the current activity are those present at the previous activity, plus overriding values of those capabilities that have deltas on the current activity. The value is computed by adding the delta to the present value, or zero in case of inactive capability (activation of a capability). In addition, we remove those capabilities that have zero value at the previous activity (deactivation of capability).

We combine the two schemas into a complete capability projection schema as follows:

$$CapabilityProjection \mathrel{\widehat=} CapabilityBinding \land CapabilityRestriction$$

The defined mechanism introduces, in the classical sense, two resources for each capability: the value of the capability and its activeness status. The value resource is a fixed free non-monotonic resource with a rule-dependency on the activeness resource of that resource. The activeness resource is also a fixed free non-monotonic resource that has been limited to two values. The distinctive feature of these resources is the defined algebra for the value changes. It allows the activation and deactivation of the capabilities during the traversal.

Capability projection is used to govern an ability to perform tasks, and these values are, quite naturally, subject to rules. There are four rules governing the feasibility of capabilities on a route. As one might infer from the examples given so far, all *active* accumulated values must be non-negative (the non-negativity rule) and sum to zero at the end (the completeness rule). Furthermore, as we mentioned, capabilities are used to state requirements for existence and nonexistence of other capabilities, effectively either requiring that a capability is present, or preventing it. Thus we define the four rules of capability projection as in the following.

Completeness of capabilities is defined as follows:

CapabilityCompletenessRule

$$routes : \operatorname{paths} Activity$$
$$capabilities : \mathbb{F}\, Capability$$
$$capabilityValues : Activity \pfun CapabilityValue$$

$$\forall route \in \operatorname{ran} routes \bullet (\forall c \in \operatorname{dom} capabilityValues(last\ route) \bullet capabilityValues(last\ route)(c) = 0)$$

The completeness rule states that each route end activity must have numeric values of zero for every active capability.

The non-negativity of capabilities is defined as follows:

NonnegativeCapabilityRule

$$
\begin{array}{l}
routes : \text{paths}\, Activity \\
capabilities : \mathbb{F}\, Capability \\
capabilityValues : Activity \pfun CapabilityValue \\
\hline
\forall\, route \in \text{ran}\, routes \bullet (\forall\, a \in \text{ran}\, route \bullet \\
\quad (\forall\, c \in \text{dom}\, capabilityValues(a) \bullet capabilityValues(a)(c) \geq 0))
\end{array}
$$

The non-negativity rule states that all activities must have their active capability values nonnegative at every activity.

The existence rule introduces the following element:

CapabilityExistenceRuleBinding

$$
\begin{array}{l}
activities : \mathbb{F}\, Activity \\
capabilities : \mathbb{F}\, Capability \\
capabilityExistence : Activity \pfun \mathbb{F}\, Capability \\
\hline
\text{dom}\, capabilityExistence = activities \\
\forall\, p \in \text{ran}\, capabilityExistence \bullet p \subseteq capabilities
\end{array}
$$

The function defining the requirement for the existence of capabilities is defined on the activities of the problem, and each such set of capabilities is a subset of all the capabilities of the problem.

The existence rule is defined as follows:

CapabilityExistenceRuleRestriction

$$
\begin{array}{l}
routes : \text{paths}\, Activity \\
capabilityExistence : Activity \pfun \mathbb{F}\, Capability \\
\hline
\forall\, route \in \text{ran}\, routes \bullet (\forall\, a \in \text{ran}\, route \bullet \\
\quad \forall\, c \in capabilityExistence(a) \bullet c \in capabilityValues(a))
\end{array}
$$

The rule states simply that each capability for which existence is required, is found in the capability values of the activity in question.

The Non-existence rule introduces the following element:

CapabilityNonexistenceRuleBinding

$$
\begin{array}{l}
activities : \mathbb{F}\, Activity \\
capabilities : \mathbb{F}\, Capability \\
capabilityNonexistence : Activity \pfun \mathbb{F}\, Capability \\
\hline
\text{dom}\, capabilityNonexistence = activities \\
\forall\, p \in \text{ran}\, capabilityNonexistence \bullet p \subseteq capabilities
\end{array}
$$

The function defining the prohibition of the existence of capabilities is defined on the activities of the problem, and each such set of capabilities is a subset of all the capabilities of the problem.

The non-existence rule is defined as follows:

CapabilityNonexistenceRuleRestriction

$$routes : \text{paths}\, Activity$$
$$capabilityNonexistence : Activity \pfun \mathbb{F}\, Capability$$

$$\forall\, route \in \text{ran}\, routes \bullet (\forall\, a \in \text{ran}\, route \bullet \forall\, c \in capabilityNonexistence(a) \bullet c \notin capabilityValues(a))$$

The rule states that no capability for which nonexistence is required can be in the capability values of the given activity.

To conclude this section, we define, for completeness, the schema for capability problems as an extension to the proper *Ansatz* as follows:

$$CapabilityProblem \mathrel{\widehat=} ProperAnsatz \land CapabilityProjection \land CapabilityCompletenessRule \land NonnegativeCapabilityRule \land CapabilityExistenceRule \land CapabilityNonexistenceRule$$

Next, however, we examine the combination of the two constructs discussed so far — resource projection and capability projection — and finally approach a more complete set of modeling tools.

5.3.5 Partial Resources

In almost any practical routing problem we need both accumulated resources and capabilities. Partial resources attempt to combine these two in order to track the accumulation for all the values necessary for modeling complex routing problems. As with the capabilities, some of the resource values are *relevant only on a part of the route*. With this motivation, we introduce the combined mechanism. This mechanism allows us to compute, if necessary, the accumulated resource values for each individual capability, and these can be used to state dynamic intricate relationships between activities and to define more complicated rules on resource bounds. Similarly to resource projection, we begin by formalizing the basic computation and introducing the necessary rules. The system is then extended to utilize a parent system to be able to define both activity-specific and capability-specific bounds in a unified manner. This section also presents the first practical illustrations on how the developed modeling constructs can be used to formulate reasonably complete routing models.

In practice, partial resources are built on capabilities and use their activation and deactivation mechanism to keep track of the relevant values. When resource projection is associated with each capability, we obtain a system where each *active* capability has its own vector of resource values, which enables us, e.g., to track the resource values of each individual pickup and delivery task. This can be illustrated by Figure 17 where we compute the values of two capabilities and two resources. In this example, activity 1 has one active capability with resource values

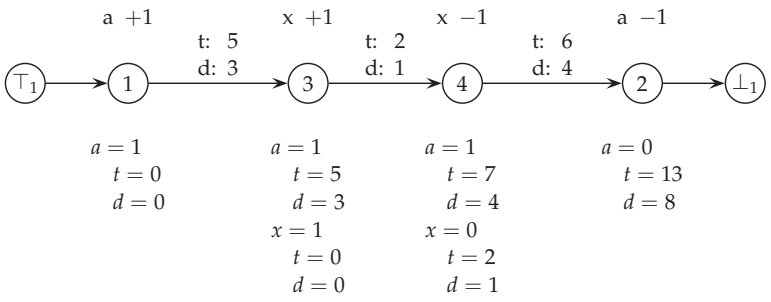


FIGURE 17 An example of computing values for two capabilities and two resources on a route.

of zero. Upon entering activity 3, the resource values of "a" have increased according to the transitional resource delta functions. Notice how any newly active capability obtains resource values of zero. Traversing to activity 4 increases the values of resources in both active capabilities, and this results in different accumulated values due to the differing number of activities on which the capabilities have been active. Notice that computing the values of resources is necessary as long as the capability is active, even though its value has decreased to zero (as in activity 2). As the capability is no longer active in the actor end activity, the values of its resources are also irrelevant at that point. Observe how there are multiple values of resources present at the activities; for instance, at activity 4, the value of accumulated time "t" is 7 for capability "a" and 2 for capability "x". If there are lower bounds on both values of "t", the computation of the most restricting lower bound has to be performed and its contribution to the travel time must be included in all the values of "t". This is in contrast with resource projection where each resource had a single value at each activity.

Partial resource projection computes, much like resource projection, the resource values at each activity on each sequence. Each resource value is, however, computed per capability, and only the resources for active capabilities are considered. Partial resource projection is thus given a set of lower and upper bounds on pairs of capabilities and resources. The partial resource deltas are effectively the same as in resource projection.

Analogously to resource and capability projections, we first introduce the basic building blocks of partial resource projection. Partial resource delta is the same type as the resource delta as the inner mechanism is exactly the same.

$$PartialResourceDelta == ResourceDelta$$

The resulting values are, in contrast, stored over the Cartesian product of resources and capabilities.

$$PartialResourceValue == (Capability \times Resource) \pfun \mathbb{Z}$$

Using the given definitions, we may now formulate the basic declarations

and predicates of the mechanism for computing the partial resources. The mechanism introduces the following elements:

PartialResourceBinding

$$
\begin{array}{l}
activities : \mathbb{F}\, Activity \\
assigned : \mathbb{F}\, Activity \\
resources : \mathbb{F}\, Resource \\
capabilities : \mathbb{F}\, Capability \\
partialResourceDeltas : Resource \pfun PartialResourceDelta \\
partialResourceLowerBounds : (Capability \times Resource) \pfun \mathbb{Z} \\
strictLowerBoundResources : \mathbb{F}\, Resource \\
partialResourceValues : Activity \pfun PartialResourceValue \\
partialValueAt : (Activity \times Capability \times Resource) \pfun \mathbb{Z} \\
activeWaitingAt : (Activity \times Resource) \pfun \mathbb{Z} \\
\hline
\mathrm{dom}\, partialResourceDeltas = resources \\
\forall f \in \mathrm{ran}\, partialResourceDeltas \bullet \mathrm{dom}\, f = activities \times activities \\
\mathrm{dom}\, partialResourceLowerBounds = capabilities \times resources \\
strictLowerBoundResources \subseteq resources \\
\mathrm{dom}\, partialResourceValues = assigned \\
\forall f \in \mathrm{ran}\, partialResourceValues \bullet \mathrm{dom}\, f = capabilities \times resources \\
\mathrm{dom}\, partialValueAt = assigned \times capabilities \times resources \\
\mathrm{dom}\, activeWaitingAt = assigned \times resources
\end{array}
$$

The partial resource mechanism is defined in terms of resources and capabilities. The functions describing the partial resource deltas are defined on the resources of the problem, and each such delta function is defined on the Cartesian product on the activities of the problem. The functions describing the lower bounds are defined on the Cartesian product of the capabilities and resources of the problem. This means that the bounds are specified per capability rather than per activity. In addition, the resources that allow no waiting are introduced, and this set is a subset of the resources defined for the problem. The results of the computation, the functions describing the partial resource values, are defined for each assigned activity of the problem, and each such function is defined on the Cartesian product of the capabilities and resources of the problem. The function *partialValueAt* is used to obtain the accumulated value at each activity, and is defined on the Cartesian product of the currently assigned activities, the capabilities of the problem, and the resources of the problem.

In the defined mechanism, multiple lower bounds on the same resource may be present at an activity. To accommodate this option, we need to obtain the most restricting resource from the lower bounds, the so-called *active waiting* value. Computing the active waiting is possible even when different lower bounds are imposed on the same resource. Computing with multiple lower bounds *on different resources* (with waiting allowed) is, however, not supported. In order to accommodate also this option, one would have to provide transformation functions for "the cost of waiting" on the start of the resource window for each other

resource. Fortunately, this is not a common scenario as time is usually the only resource with lower bounds and waiting allowed.

The computation of partial resources is defined in the following schema:

PartialResourceRestriction

$$
\begin{array}{l}
routes : \text{paths}\, Activity \\
resources : \mathbb{F}\, Resource \\
capabilities : \mathbb{F}\, Capability \\
capabilityValues : Activity \pfun CapabilityValue \\
partialResourceDeltas : Resource \pfun PartialResourceDelta \\
partialResourceLowerBounds : (Capability \times Resource) \pfun \mathbb{Z} \\
strictLowerBoundResources : \mathbb{F}\, Resource \\
partialResourceValues : Activity \pfun PartialResourceValue \\
partialValueAt : (Activity \times Capability \times Resource) \pfun \mathbb{Z} \\
activeWaitingAt : (Activity \times Resource) \pfun \mathbb{Z} \\
\hline
\forall\, a, b : Activity;\ r : Resource \bullet partialResourceDeltas(r)(a, b) \geq 0 \\
\forall\, route \in \text{ran}\, routes;\ r \in strictLowerBoundResources \bullet \\
\quad (\forall\, a \in \text{ran}\, route \bullet activeWaitingAt(a, r) = 0) \\
\forall\, route \in \text{ran}\, routes;\ r \in resources \setminus strictLowerBoundResources \bullet \\
\quad (activeWaitingAt(head\, route, r) = max\{0, \text{ran} \\
\qquad (\lambda\, c \in \text{dom}\, capabilityValues(head\, route) \mid \\
\qquad\quad isZero(capabilityValues(head\, route, c)) \bullet \\
\qquad\qquad partialResourceLowerBounds(c, r))\}) \land \\
\quad (\forall\, a \in \text{ran}\, tail\, route \bullet (activeWaitingAt(a, r) = \\
\qquad max\{\text{ran}(\lambda\, c \in \text{dom}\, capabilityValues(a) \mid \\
\qquad\quad isZero(capabilityValues(a, c)) \bullet \\
\qquad\qquad max\{0, partialResourceLowerBounds(c, r) - \\
\qquad\qquad\quad partialResourceValues(prev(route, a))(c, r) + \\
\qquad\qquad\quad partialResourceDeltas(r)(prev(route, a), a)\})\})) \\
\forall\, route \in \text{ran}\, routes;\ c \in capabilities;\ r \in resources \bullet \\
\quad (isActive(capabilityValues, head\, route, c) \Rightarrow \\
\qquad (partialResourceValues(head\, route)(c, r) = \\
\qquad\quad activeWaitingAt(head\, route, r))) \land \\
\quad (\forall\, a \in tail\, route \bullet \\
\qquad (isActive(capabilityValues, a, c) \Rightarrow \\
\qquad\quad partialResourceValues(a)(c, r) = \\
\qquad\qquad partialValueAt(prev(route, a), c, r) + \\
\qquad\qquad partialResourceDeltas(r)(prev(route, a), a) + \\
\qquad\qquad activeWaitingAt(a, r)))
\end{array}
$$

The active waiting denotes the "waiting" needed to conform to the lower bounds on an activity. We have defined a function for denoting this waiting value at each assigned activity in the problem. As in resource projection, for each resource with a strict lower bound requirement, the length of waiting is always zero. For other resources, each route start activity is given an active waiting value that is the

maximum of the lower bounds for active zero capabilities at that activity. Each subsequent active waiting value is computed by finding the largest nonnegative difference between a lower bound that is being imposed by zero-valued capabilities, and the value of the bounded resource.

Apart from the values of waiting, the projection is computed similarly to resource projection. Each start of a route has each of its resource values initialized to the value of *active waiting*, and each subsequent activity has its resources computed as accumulated value from the previous value plus a result of a given delta function and the waiting at that activity.

We combine the schemas into that of partial resource projection as follows:

$$PartialResourceProjection \mathrel{\widehat=} PartialResourceBinding \land PartialResourceRestriction$$

Note that we left details of *partialValueAt* unspecified for possible further refinement. In the elementary case, we return the actual partial value computed for the previous activity.

$$\begin{array}{|l} \textit{IdentityPartialResourceValues} \\ \hline activities : \mathbb{F}\, Activity \\ resources : \mathbb{F}\, Resource \\ partialResourceValues : Activity \pfun PartialResourceValue \\ partialValueAt : (Activity \times Capability \times Resource) \pfun \mathbb{Z} \\ \hline \forall\, a \in activities;\ c \in capabilities;\ r \in resources \bullet \\ \quad partialValueAt(a, c, r) = partialResourceValues(a)(c, r) \\ \hline \end{array}$$

According to these definitions, the computed partial resource values must have their values greater than the corresponding lower bound on that particular resource and the given capability on those activities where the capability values go to zero. This is in contrast with resource projection where the values were restricted on an activity basis. This makes the constraints effectively a tuple of lower bound, upper bound, capability, and resource, or in terms of the schemas used, functions from a pair of capability and resource to a numeric value for both the lower and upper bound.

By combining the schema with that of the partial resource projection, we obtain the complete schema for computing partial resources:

$$IdentityPartialResourceProjection \mathrel{\widehat=} PartialResourceProjection \land IdentityPartialResourceValues$$

The specification given here defines the mechanism so that it introduces n new classical resources for each capability, where n is the number of resources in the problem. Each classical resource introduced by a capability has a rule dependency on the activeness of the corresponding capability. This highlights the reason for not needing to compute the values of the resources if the capability has

been deactivated. Otherwise the classical resources in the mechanism behave as in resource and capability projections.

Similarly to the previously discussed mechanics, we define rules for acceptable values for partial resource projections. As in resource projection, we define that the values must not fall outside the predefined bounds. The rule for expressing this introduces the following element:

PartialResourceRuleBinding

$$
\begin{array}{l}
resources : \mathbb{F}\, Resource \\
capabilities : \mathbb{F}\, Capability \\
partialResourceUpperBounds : (Capability \times Resource) \pfun \mathbb{Z} \\
\hline
\mathrm{dom}\, partialResourceUpperBounds = capabilities \times resources
\end{array}
$$

The function describing the upper bounds on partial resource values is defined on the Cartesian product of the capabilities and resources of the problem.

The partial resource rule is defined by the following schema:

PartialResourceRuleRestriction

$$
\begin{array}{l}
routes : \mathrm{paths}\, Activity \\
resources : \mathbb{F}\, Resource \\
capabilities : \mathbb{F}\, Capability \\
capabilityValues : Activity \pfun CapabilityValue \\
partialResourceLowerBounds : (Capability \times Resource) \pfun \mathbb{Z} \\
partialResourceUpperBounds : (Capability \times Resource) \pfun \mathbb{Z} \\
partialValueAt : (Activity \times Capability \times Resource) \pfun \mathbb{Z} \\
\hline
\forall\, route \in \mathrm{ran}\, routes;\ r \in resources;\ c \in capabilities \bullet \\
\quad (\forall\, a \in \mathrm{ran}\, route \bullet isZero(capabilityValues, a, c) \Rightarrow \\
\qquad partialValueAt(a, c, r) \geq partialResourceLowerBounds(c, r) \land \\
\qquad partialValueAt(a, c, r) \leq partialResourceUpperBounds(c, r))
\end{array}
$$

The resource values at each activity must be greater than or equal to the corresponding lower bound and less than or equal to the upper bound of each capability and resource if the capability value is deactivating at that activity.

By combining the schemas, we obtain the one for the partial resource rule:

$$
\begin{array}{l}
PartialResourceRule \mathrel{\widehat=} PartialResourceRuleBinding \land \\
\quad PartialResourceRuleRestriction
\end{array}
$$

Consider, for illustration, a DARP case where we need to restrict the length of a driver's shift and ensure that any customer does not spend more than a certain amount of kilometers on the vehicle, for instance, due to the maximum allowed deviation from the shortest available distance. If "a" is the label used in activities denoting the vehicle, and "x" for the customer, we set, for example,

$$
\begin{array}{l}
partialResourceUpperBounds(a, t) = 15 \\
partialResourceUpperBounds(x, d) = 3.
\end{array}
$$

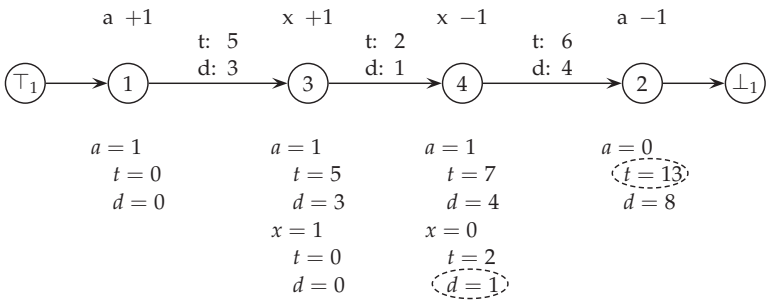


FIGURE 18 An example of checking constraints on the route length and time spent on a vehicle.

The resulting required checks on feasibility are illustrated in Figure 18. The constrained resource values are those whose corresponding capability value is deactivating; in this case, activities 4 and 2, for resources "d" and "t", respectively.

The illustrated constraints are defined on the capabilities of the problem and as such work well whenever we need to restrict, e.g., the length of a given operation. Often, however, there is a need to restrict the values *on a given activity*. This is the case with, for example, time windows. To model these two cases in a unified manner, we introduce a *mechanism for substituting the initial values of resources on a capability with those of another capability*. This enables us to specify, for example, activity-specific constraints by giving the activity a unique capability and stating the constraint in terms of this capability. This relationship is an elementary hierarchy of capabilities, and is here referred to as a capability parent system. A capability may, then, have another capability as its *parent capability* from which it obtains its resource values when activating.

In order to employ the parent capability values, we need to redefine the *partialValueAt*. This is done as defined in the following schema:

ParentPartialResourceValues

$$
\begin{array}{l}
activities : \mathbb{F}\, Activity \\
resources : \mathbb{F}\, Resource \\
capabilities : \mathbb{F}\, Capability \\
capabilityParents : Capability \pfun Capability \\
partialResourceValues : Activity \pfun PartialResourceValue \\
partialValueAt : (Activity \times Capability \times Resource) \pfun \mathbb{Z}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{dom}\, capabilityParents \subseteq capabilities \\
\mathrm{ran}\, capabilityParents \subseteq capabilities \\
\forall\, a \in activities;\ c \in capabilities;\ r \in resources \bullet \\
\quad (c \in \mathrm{dom}\, capabilityParents \Rightarrow partialValueAt(a, c, r) \\
\qquad = partialResourceValues(a)(capabilityParents(c), r)) \land \\
\quad (c \notin \mathrm{dom}\, capabilityParents \Rightarrow partialValueAt(a, c, r) \\
\qquad = partialResourceValues(a)(c, r))
\end{array}
$$

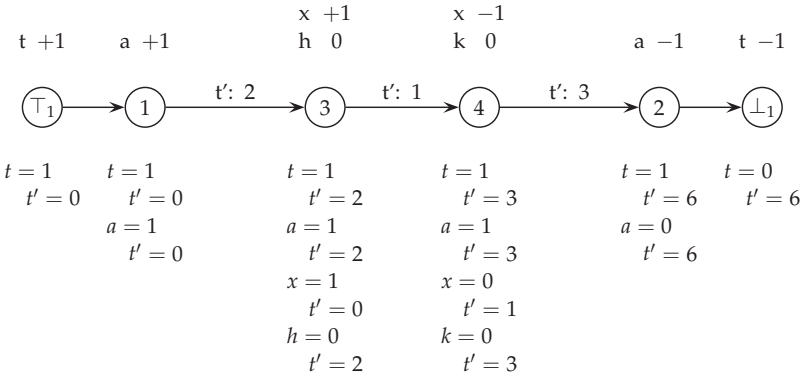


FIGURE 19 Modeling activity-specific time windows with zero capability and capability parents.

The function specifying the capability parents is defined on a subset of the capabilities of the problem, and each capability parent is also a capability in a subset of the capabilities. To compute the projection, we first find the active capabilities of the given activity. We then define the function *partialValueAt* such that if a capability parent is given, its resource values are used instead of the current capability. Otherwise, we proceed as defined earlier.

As we simply redefine the function describing the value at each activity, there is no need to introduce additional rules. We may thus combine parent value projection with partial resource projection to obtain the schema for computing partial resources with parents.

$$\begin{aligned} &\mathit{ParentPartialResourceProjection} \mathrel{\widehat{=}} \mathit{PartialResourceProjection} \wedge \\ &\quad \mathit{ParentPartialResourceValues} \end{aligned}$$

The usage of the parent system can be illustrated with an example of regular time windows. As we discussed, zero capability deactivates immediately, and if it has a parent, we use the projected resource values of that parent in evaluation of bounds. Consider an example where we set

$$\begin{aligned} &\mathit{partialResourceLowerBounds}(h, t') = 1 \\ &\mathit{partialResourceUpperBounds}(h, t') = 4 \\ &\mathit{partialResourceLowerBounds}(k, t') = 3 \\ &\mathit{partialResourceUpperBounds}(k, t') = 7, \end{aligned}$$

and

$$\mathit{capabilityParents} = \{k \mapsto t, h \mapsto t\}.$$

We may now model individual time windows as depicted in Figure 19. Note how capabilities "h" and "k" are zero, but their values with respect to "t'" reflect that of accumulated from the beginning of the route and equal the corresponding

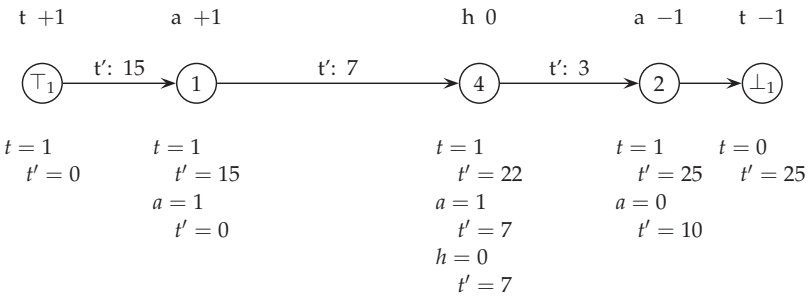


FIGURE 20 Modeling decision-dependent time windows with zero capability value and capability parents.

value of capability "t". The depicted case remains feasible as

$$partialResourceLowerBounds(h, t') \leq partialValueAt(3, h, t') \wedge$$
$$partialValueAt(3, h, t') \leq partialResourceUpperBounds(h, t') \wedge$$
$$partialResourceLowerBounds(k, t') \leq partialValueAt(4, k, t') \wedge$$
$$partialValueAt(4, k, t') \leq partialResourceUpperBounds(k, t').$$

A different variant of this case is the so-called decision dependent time window, where we do not use a capability of universal time for every actor (the "t" in the previous example) but use some other active label instead. Consider an example where we need to have a break after a given time from the beginning of a driver's shift, but such that the exact start time of the route is also a decision and cannot be determined *a priori*. In this example we set

$$partialResourceLowerBounds(h, t') = 5$$
$$partialResourceUpperBounds(h, t') = 10$$

and

$$capabilityParents = \{h \mapsto a\}.$$

This case may be modeled as illustrated in Figure 20. In this case, note how the resource "t'" for capability "h" obtains its value from capability "a", instead of "t" as in the previous example. Also this case remains feasible as

$$partialResourceLowerBounds(h, t') \leq partialValueAt(4, h, t') \wedge$$
$$partialValueAt(4, h, t') \leq partialResourceUpperBounds(h, t').$$

These examples illustrate the usage of the partial resource mechanism. Although we did introduce the notion of slack projections on resources and the approach is also useful in partial projections, we omit the actual schemas in this mechanism as the principle is basically the same. The main difference is that we may have several upper bounds on the same resource, and we must find the active upper bound, that is, the most constraining value on the slack of

that resource. The procedure is an enumeration of the possible bounds over the capability-resource pairs resulting from all the values further along the sequence.

To conclude, we combine the schema of partial resource projection with that of the capability problem to form the schema for the partial resource problem as follows:

$$\begin{aligned} &\mathit{PartialResourceProblem} \mathrel{\widehat{=}} \mathit{CapabilityProblem} \land \\ &\quad \mathit{ParentPartialResourceProjection} \land \\ &\quad \mathit{PartialResourceRule} \end{aligned}$$

The resulting schema is able to express all the necessary elements for representing a variety of constraints commonly defined in routing problems, but, for example, the notion of objective has not yet been introduced. Before discussing the projection for the objective, however, we introduce, in the next sections, two additional constructs. In the next section, we aim to increase the flexibility of the resource (and objective) computation by introducing yet one level of indirection. We define a mechanism for computing with capability-dependent resources.

5.3.6 Stack Resources

Stack resources are used in a mechanism attempting to address some of the inflexibility present in the other mechanisms of computing the accumulated resource values. Often, the exact delta function is dependent on dynamic factors, such as vehicle type, equipment, or current cargo; that is, the resource delta is dependent on *active capabilities*. This is due to the inherent dependencies between resources, and this section presents a mechanism for altering the resource accumulation accordingly. The formalization introduces an additional projection, but no new rules are needed as we only alter the delta functions.

In practice, the stack resources keep track of the currently active resource delta function by associating capabilities with functions that should be activated when the capability itself is activated. As the name suggests, we employ stack semantics in this mechanism. Note that there is no definition for a data structure with stack semantics in the Z notation. Thus we define one in Appendix 3.

Consider, for illustration, a case where the travel time between locations is given by function f. If, however, we perform a particular task, say collecting a large machine which needs to be towed by the vehicle, our speed slows down considerably. This affects the travel time and is given by function g during the task. This can be illustrated as in Figure 21. In the example, the function f is activated with capability "a" and is used from the beginning of the route until g is activated through capability "b". This is where we perform the pickup of the machine. The function g is used to compute the travel times until activity 2, at which the function f is again employed. Here the machine is delivered. Observe how the functions are placed on a stack and the currently topmost is used to compute the traversal.

While in the mechanism for computing the partial resources we provided the functions describing the change in resource values, the key idea in stack re-

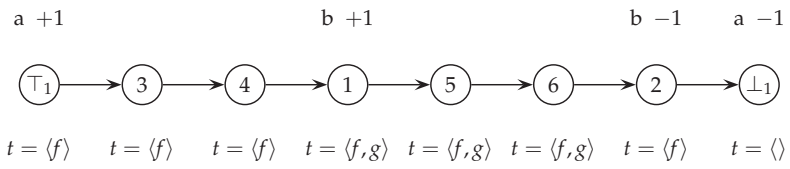


FIGURE 21 Modeling capability-dependent travel time.

sources is to replace these deltas with a stack mechanism. The input to this mechanism is a set of functions from capability-resource pairs to partial resource delta functions. The capability-resource pair is used to indicate that the corresponding partial resource delta function should be used to compute the change in the resource value from the point the given capability is activated, to the point it is deactivated. The set of functions from capability-resource pairs to the delta functions which activate from the given capability is denoted here as *activating functions*.

First, to track the stacks of partial resource delta functions at each activity, we define a partial function from resources to stacks of these functions. These sets of resource delta stacks are defined for each activity to track the state of the stacks during the traversal.

$$ResourceDeltaStacks == Resource \pfun Stack[PartialResourceDelta]$$

We may now introduce the elements used in stack resource projection.

StackResourceBinding

$$\begin{aligned}
&activities : \mathbb{F}\, Activity \\
&assigned : \mathbb{F}\, Activity \\
&resources : \mathbb{F}\, Resource \\
&capabilities : \mathbb{F}\, Capability \\
&activatingDeltas : (Capability \times Resource) \pfun PartialResourceDelta \\
&resourceDeltaStacks : Activity \pfun ResourceDeltaStacks \\
&activatingCapabilities : (\mathrm{seq}\, Activity \times Activity) \pfun \mathbb{F}\, Capability \\
&deactivatingCapabilities : Activity \pfun \mathbb{F}\, Capability \\
\hline
&\mathrm{dom}\, activatingDeltas \subseteq capabilities \times resources \\
&\mathrm{dom}\, resourceDeltaStacks = assigned \\
&\forall f \in \mathrm{ran}\, resourceDeltaStacks \bullet \mathrm{dom}\, f = resources \\
&\forall s \in \mathrm{ran}\, \mathrm{ran}\, resourceDeltaStacks \bullet \\
&\quad \mathrm{dom}\, \mathrm{ran}\, s.elements = activities \times activities \\
&\mathrm{dom}\, activatingCapabilities = \mathrm{seq}\, assigned \times assigned \\
&\mathrm{ran}\, activatingCapabilities = \mathbb{F}\, capabilities \\
&\mathrm{dom}\, deactivatingCapabilities = assigned \\
&\mathrm{ran}\, deactivatingCapabilities = \mathbb{F}\, capabilities
\end{aligned}$$

The function *activatingDeltas* is defined on a subset of the Cartesian product of the capabilities and resources of the problem. The stacks of resource delta functions are defined on the assigned activities of the problem as we need to track the accumulated stack at each activity on the defined paths. Each function mapping the resources to the stacks is defined on the resources of the problem, and the elements in the stack, the delta functions, are functions from the resources of the problem to functions from the Cartesian product of the activities of the problem to a numeric value. The function *activatingCapabilities* denotes the capabilities within the problem that are activating at the given activity, and is defined on the Cartesian product of the sequences of activities and the activities currently assigned in the problem. Likewise, the function *deactivatingCapabilities* denotes those capabilities within the problem that are deactivating on a given activity. These functions are also defined on the set of assigned activities of the problem.

Before specifying the detailed computation within stack resource projection, we introduce functions for conveniently checking activation and deactivation of capabilities as these define the change in the stack values during the traversal of the route. This is expressed formally in the following schema:

StackResourceProjectionUtilities

$$
\begin{array}{l}
routes : \operatorname{paths} Activity \\
resources : \mathbb{F}\, Resource \\
capabilityValues : Activity \pfun CapabilityValue \\
partialResourceDeltas : Resource \pfun PartialResourceDelta \\
activatingDeltas : (Capability \times Resource) \pfun PartialResourceDelta \\
activatingCapabilities : (\operatorname{seq} Activity \times Activity) \pfun \mathbb{F}\, Capability \\
deactivatingCapabilities : Activity \pfun \mathbb{F}\, Capability \\
\hline
\forall\, route \in \operatorname{ran} routes;\ a \in activities;\ r \in resources \bullet \\
\quad activatingCapabilities(route, a) = \{\, c \in \operatorname{dom} capabilityValues(a) \mid \\
\qquad \neg\, isActive(capabilityValues, prev(route, a), c) \land \\
\qquad isActive(capabilityValues, a, c) \land \\
\qquad (c, r) \in \operatorname{dom} activatingDeltas \,\} \\
\forall\, a \in activities;\ r \in resources \bullet \\
\quad deactivatingCapabilities(a) = \{\, c \in \operatorname{dom} capabilityValues(a) \mid \\
\qquad isZero(capabilityValues, a, c) \land \\
\qquad (c, r) \in \operatorname{dom} activatingDeltas \,\}
\end{array}
$$

If a capability, for which an activating function exists, is active at a given activity, but was not active at the preceding one, it is said to be in the set of activating capabilities for that activity. Similarly, capabilities which are zero at a given activity are said to be in the set of deactivating capabilities for that activity.

For notational convenience, we introduce a function for selecting the single value from a set of size one as follows:

$$
\begin{array}{|l}
\hline
[X] \\
single : \mathbb{F}\, X \pfun X \\
\hline
\operatorname{dom} single = \{ S : \mathbb{F}\, X \mid \#S = 1 \} \\
\forall S \in \operatorname{dom} single \bullet single(S) = (\mu\, x : X \mid x \in S) \\
\hline
\end{array}
$$

Using the given definitions, the computation within the structure is now captured by the following schema. The key idea is that, by treating *partialResourceDeltas* as a part of the problem definition, we introduce a different element as an input to the *partialResourceDeltas*: the *activatingDeltas*. This function describes the association of capabilities with the actual resource delta functions used in computation, and must now be defined as a part of the problem data. The stack resource mechanism keeps track of the currently active resource delta function by a stack structure, and the *partialResourceDeltas* is defined as the current top element of that stack at each activity on a route. If the stack is empty, a zero value is assumed.

$$
\begin{array}{|l}
\hline
StackResourceRestriction \\
activities : \mathbb{F}\, Activity \\
routes : \operatorname{paths} Activity \\
resources : \mathbb{F}\, Resource \\
partialResourceDeltas : Resource \pfun PartialResourceDelta \\
resourceDeltaStacks : Activity \pfun ResourceDeltaStacks \\
activatingCapabilities : (\operatorname{seq} Activity \times Activity) \pfun \mathbb{F}\, Capability \\
deactivatingCapabilities : Activity \pfun \mathbb{F}\, Capability \\
\hline
\forall r \in resources \bullet \\
\quad (\forall a, b \in activities \bullet (\textbf{let}\ s == resourceDeltaStacks(a)(r) \bullet \\
\qquad (\#s.elements > 0 \Rightarrow \\
\qquad\quad partialResourceDeltas(r)(a,b) = peek(s)(a,b)) \land \\
\qquad (\#s.elements = 0 \Rightarrow partialResourceDeltas(r)(a,b) = 0))) \land \\
\quad (\forall route \in \operatorname{ran} routes \bullet \\
\qquad (\textbf{let}\ activating == activatingCapabilities(route, head\, route) \bullet \\
\qquad\quad (\#activating = 1 \Rightarrow \\
\qquad\qquad resourceDeltaStacks(head\, route)(r).elements = \\
\qquad\qquad\quad \langle activatingDeltas(single(activating), r) \rangle) \land \\
\qquad (\forall a \in \operatorname{ran} tail\, route \bullet (\textbf{let}\ cs == resourceDeltaStacks(a)(r); \\
\qquad\quad ps == resourceDeltaStacks(prev(route,a))(r); \\
\qquad\quad activating == activatingCapabilities(route,a); \\
\qquad\quad deactivating == deactivatingCapabilities(a) \bullet \\
\qquad\quad (\#activating = 1 \Rightarrow cs = push(ps, single(activating))) \land \\
\qquad\quad (\#deactivating = 1 \Rightarrow cs = first\, pop(ps) \land \\
\qquad\qquad second\, pop(ps) = single(deactivating)) \land \\
\qquad\quad (\#activating \neq 1 \land \#deactivating \neq 1 \Rightarrow cs = ps))))) \\
\hline
\end{array}
$$

Each resource has its own resource delta function stack, and the stack is initialized with a resource delta function defined by a unique activating capability at

the start of the route. Each subsequent activity either adds an element to the stack or removes one from it if the set *activatingDeltas* contains a function for a capability-resource pair found at that activity. If the capability at the activity is deactivating, the corresponding resource delta function is removed from the top of the stack, and if the capability at the activity is activating, the corresponding resource delta function is added to the stack as the new top. Otherwise, the stack remains unaltered. Currently activating capabilities are defined as those whose activeness has changed from zero to one when traversing from the previous activity to the current, and that are in the set defined by *activatingDeltas*. Currently deactivating capabilities are those whose value is zero and which are, likewise, in the set given by the same function. The informal idea of the activation and deactivation of capabilities conforms to that presented here, without, however, the requirement of the capability residing in the set *activatingDeltas*. Note also that from the definition we can see that removing from the stack does require removal of the function associated with the deactivating capability. That is, on a feasible route, the activities removed from the stack must occur in the reverse of the order they were added.

Note that the stack resources can model dependencies from a single capability at a time. A more general case is the one where a *combination* — that is, a set — of active capabilities defines the function used to compute the resource accumulation. These types of projections are *situation dependent*, and we examine them in the subsequent sections.

We combine the three schemas into partial resource projection as follows:

$$\begin{aligned} &\mathit{StackResourceProjection} \mathrel{\widehat=} \mathit{StackResourceBinding} \land \\ &\quad \mathit{StackResourceProjectionUtilities} \land \\ &\quad \mathit{StackResourceRestriction} \end{aligned}$$

Stack resource projection is a mechanism that introduces one classical resource for each new delta function introduced. This resource tracks the location of the corresponding activation capability in the stack. In addition, it introduces new dependencies between the classical resources: the stack state resource has a fixed value dependency on the activeness of the activation capability; this is how the stack is controlled. The resources of the problem also become value-dependent of the stack state resource. These types of dependencies have the potential to make the computation excessively complex, but fortunately, the defined mechanism does not typically introduce a large number of different values on the stack state resource.

To illustrate the defined mechanism, assume that we have the option to attach trailers to vehicles. This results in additional available capacity, but affects the speed of the vehicle negatively. Design decisions of this kind may be present in real-life VRP cases involving complex fleet management. We may construct an example of this type by setting

$$\mathit{activatingDeltas} = \{(t,a) \mapsto f, (t,b) \mapsto g, (d,a) \mapsto h\},$$

where t is an identifier for time, d for distance, a for the vehicle, b, for the trailer, and f, g, and h are transformation functions for describing travel time without a

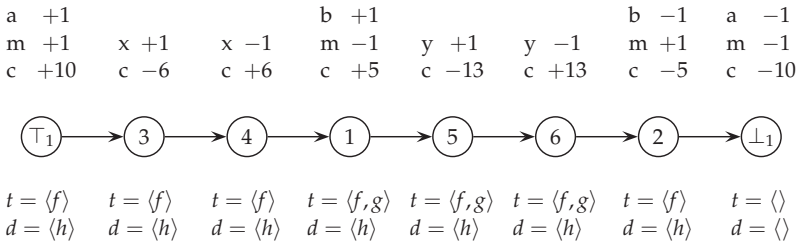


FIGURE 22 An example of capability dependent transformations: a trailer affects the traveling speed of a vehicle.

trailer, travel time with a trailer, and travel distance, respectively. The resulting computation is depicted in Figure 22. For brevity, we have omitted the values of the active function transformations between activities. In the example, we begin the route by acquiring the capability "a". This results in inclusion of transformation f in the transformation set of resource "time", and inclusion of transformation h in the transformation set of resource "distance". Note that we also obtain label "m", indicating that we are able to pick up a trailer. We then perform a pickup and a delivery of "x" traversing by using the resource delta functions f and h. In activity 1, we pick up a trailer and acquire five units of additional capacity. We also consume the ability to pick up a trailer "m" as well as activate capability "b", resulting in a change in the set of transformations of resource "time": the transformation g is added into the stack. Note how distance is not affected by the change, since we defined that the trailer only affects the speed[5] of the vehicle. As we traverse the path, the resources of all labels are accumulated according to the latest activated function (recall from the example in Figure 17 that each capability has their set of values). We then perform another task in activities 5 and 6 traversing by using functions g and h. At activity 2, we leave the trailer, which deactivates the capability "b" and results in removal of transformation g from the distance transformation set, effectively reseting the travel speed again to that of the vehicle. Notice how after dropping off the trailer at activity 2, the vehicle might pick up another trailer if needed. Leaving the vehicle at the end of the route deactivates the capability "a", which removes the remaining transformations from the stacks.

To conclude this section, we define the corresponding problem in terms of this projection as follows:

$$\mathit{StackResourceProblem} \mathrel{\widehat{=}} \mathit{PartialResourceProblem} \wedge \mathit{StackResourceProjection}$$

The defined problem is able to express the necessary mapping-ordering constraint functions needed within this modeling framework. The next section will

[5] Travel distance could also be modified if traveling with a trailer affects the available routes in the road network, e.g., due to restrictions on driving with a trailer in certain road segments.

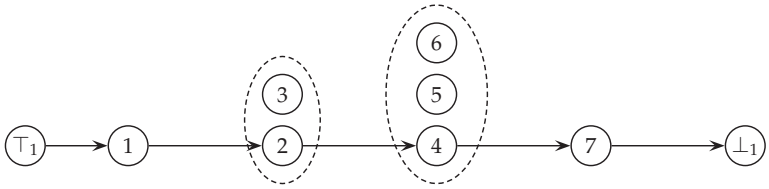


FIGURE 23 Examples of groups of activities within an *Ansatz*.

define a constraint function on mapping for addressing also that aspect of routing problems.

5.3.7 Activity Groups

As we discussed in the taxonomy, there are a number of possibilities for mapping constraint constructs. We chose to implement only the simplest yet sufficient ones for expressing structures arising in some of the most common variants. *Grouping* is a mapping constraint from decision-independent sets of activities. Using this projection, we can define a set of topological rules which can be used in both communicating the structure of the problem to algorithms and providing a mechanism for constructing additional logical dependencies between activities. More specifically, we can mimic an exclusive logical disjunction (XOR) operator, which is useful in, e.g., modeling compartments and breaks.

Consider, for illustration, an example where we need to choose between two alternative pickup locations, and three alternative corresponding delivery locations for a certain task. This is depicted in Figure 23. In the example, activities 2 and 3 are the alternative pickup locations, and activities 4–6 the alternative delivery locations[6]. The dashed lines denote the groups of activities for which we impose a rule that only one activity from that group can be in the set of assigned activities in a feasible *Ansatz*. The depicted route is feasible as we visit only the activities 2 and 4 of those groups.

To formally define grouping projection, we first define grouping as a finite set of finite sets of given elements, that is, a set of groups. This type is defined as follows:

$$\text{grouping}\, X == \mathbb{F}(\mathbb{F}\, X)$$

Using the grouping type, we may now give the basic declarations and predicates of the projection. The projection introduces the following elements:

[6] We have omitted the capabilities and resources from the figure for clarity.

GroupingBinding

$$
\begin{array}{l}
activities : \mathbb{F}\, Activity \\
groups : \text{grouping}\, Activity \\
groupValues : \text{grouping}\, Activity \\
\hline
\forall g \in groups \bullet g \subseteq activities \\
\forall g \in groupValues \bullet g \subseteq activities
\end{array}
$$

Groups are the sets of activities which define the groups within the problem. Group values contain the projected values of those groups. Both sets of groups contain groups that consist of a subset of the activities in the problem.

The details of grouping projection can now be specified as follows:

GroupingRestriction

$$
\begin{array}{l}
assigned : \mathbb{F}\, Activity \\
groups : \text{grouping}\, Activity \\
groupValues : \text{grouping}\, Activity \\
\hline
groupValues = \{ g \in groups \bullet g \cap assigned \}
\end{array}
$$

Each group in group values contains exactly the groups of the problem so that only the assigned activities are in the projected groups.

We combine the two schemas into a complete grouping projection schema as follows.

$$
GroupingProjection \mathrel{\widehat=} GroupingBinding \land \\
\quad GroupingRestriction
$$

As we can see, *GroupingProjection* is a constraint function on mapping: we require that each group is a subset of the activities in the problem, and *groupValues* are computed from those groups by including the activities that have been assigned to an actor.

Next, we define rules for feasible groupings. To simplify notation, we introduce concepts of *interval*, which may be used to define inclusive or exclusive intervals, and *bounds*, which can be used to limit the number of elements in a set.

Interval

$$
\begin{array}{l}
lower : \mathbb{N} \\
upper : \mathbb{N} \\
\hline
lower \leq upper
\end{array}
$$

$$
\text{bounds}\, X == \mathbb{F}\, X \pfun Interval
$$

Using the definitions, we may now specify the basic declarations and predicates of the grouping rule. The rule introduces the following elements:

GroupingRuleBinding

$$
\begin{array}{|l}
\hline
groupValues : \text{grouping}\, Activity \\
groupBounds : \text{bounds}\, Activity \\
\hline
\text{dom}\, groupBounds \in groupValues \\
\hline
\end{array}
$$

Group bounds is a set of functions specifying intervals for groups. The function is defined on the group values of the projection, that is, specifying a limit on the size of each projected group.

The grouping rule can be defined as follows:

GroupingRuleRestriction

$$
\begin{array}{|l}
\hline
groupValues : \text{grouping}\, Activity \\
groupBounds : \text{bounds}\, Activity \\
\hline
\forall g \in groupValues \bullet \\
\quad groupBounds(g).lower \leq \#g \land groupBounds(g).upper \geq \#g \\
\hline
\end{array}
$$

Each projected group must have the number of elements in the inclusive interval defined to the group.

Constraints on mapping may or may not span over multiple actors. However, if we want to ensure that grouping constraints affect only a single actor, we may set the upper bound of the corresponding rule to one, limiting the number of the elements in the sets in the constraint space. This can be expressed with the following schema:

SimpleGrouping

$$
\begin{array}{|l}
\hline
groupBounds : \text{bounds}\, Activity \\
\hline
\forall g \in groupValues \bullet \\
\quad groupBounds(g).lower = 0 \land groupBounds(g).upper = 1 \\
\hline
\end{array}
$$

SimpleGrouping prevents modeling inter-route dependencies. If there are no constraints that propagate from a route of an actor to another, it is possible to partition the problem along the routes. A multi-threaded implementation, for example, could benefit from such a partition: modifications in more than one route can be made simultaneously. Both simple and generic grouping constraints are illustrated in Figure 24. In the example, each route has one activity assigned on it: route 1, activity 4; route 2, activity 3; and route 3, activity 1. In addition, two constraints defining groups of three activities have been introduced. Firstly, we restrict the number of activities simultaneously assignable to any of the routes to 0 or 1 for the group of activities 4–6, and secondly, similarly, restrict the number of simultaneously assignable activities to 2 or less for the group of activities 1–3. The depicted example illustrates a feasible mapping according to these constraints. This also illustrates the difference between simple grouping and generic grouping: in the latter, feasibility checks can propagate from one route to another.

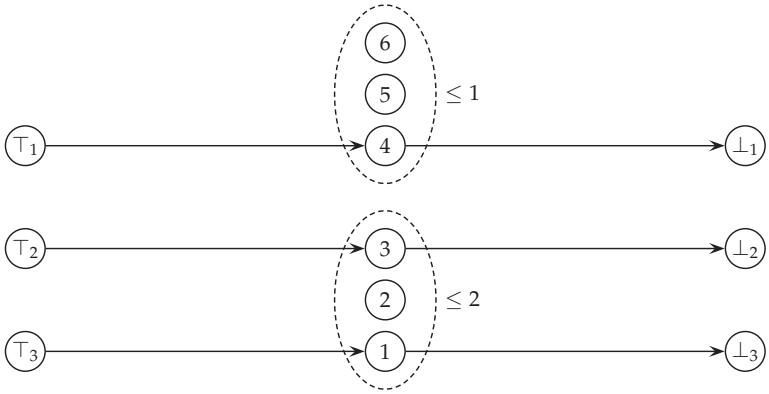


FIGURE 24 Examples of different types of grouping constraints within a proper *Ansatz*.

To summarize, we define a routing problem with grouping constraints as follows:

$$\begin{aligned} &\mathit{GroupingProblem} \mathrel{\widehat{=}} \mathit{ProperAnsatz} \land \\ &\quad \mathit{GroupingProjection} \land \\ &\quad \mathit{GroupingRule} \land \\ &\quad \mathit{SimpleGrouping} \end{aligned}$$

In the next section, we conclude the definition of the developed metamodel by adding a few missing elements, such as the objective function, and by examining the developed model as a whole.

5.3.8 Implemented Model

Using the definitions given in the preceding sections, we are now able to construct the full, formal definition of feasibility and objective values within the model. As we have seen, many of the constructs are heavily based on resource extension functions, but we have introduced them in a more fine-grained manner. This approach is justified when we introduce the modeling language employing the constructs defined here. The modeling language then provides a systematic way of expressing different routing problem variants and their extensions.

We have largely concentrated on constraints in this discourse, and the one element lacking a formal definition is the objective of the problem. As mentioned, profit can be seen as a special resource accumulated along the route, and the value of the objective of the optimization problem is then the sum of all profits collected on all routes. In a typical case, the profit delta function is identical to the resource delta, that is,

$$\mathit{ProfitDelta} == (\mathit{Activity} \times \mathit{Activity}) \pfun \mathbb{Z}.$$

The profit is considered separately and assumed to be present in every problem. It differs from other resources in that it is *capability invariant*. That is, for

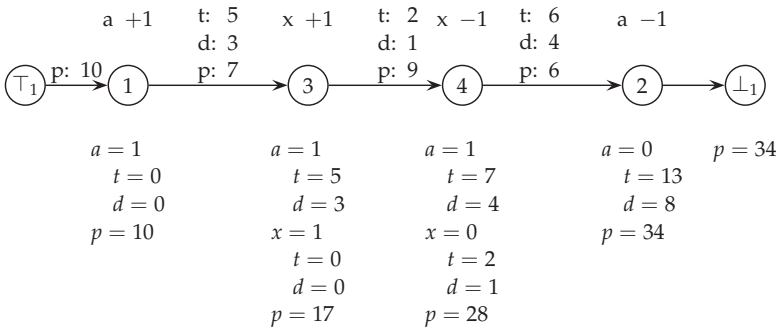


FIGURE 25 An example of two resources, travel time and distance, and a profit value on a route.

each activity of the route, there is a single value of profit collected that far, and *no capability-specific profits are computed*. In other words, profit employs resource projection instead of partial resource projection within the stack resource mechanism. The difference between computing a capability invariant resource — profit — and capability variant resources — time and distance — is illustrated in Figure 25. As we traverse to the activity 1, we obtain 10 units of profit, and likewise 7 when entering activity 3, and so on. In this example, the profit collected is $10 - d$, that is, the so-called *base profit* minus the expenses. This example demonstrates the usage of profit maximization in cases such as the classical VRP where the distance is minimized.

To summarize, we revisit the schemas defined in the preceding sections. The generic routing problem modeled with mutually exclusive and optional activities, resources, capabilities, and additional dependency mechanisms can be described as in the following schema:

$$
\begin{aligned}
&\mathit{GenericRoutingProblem} \mathrel{\widehat{=}} \mathit{ProperAnsatz} \wedge \\
&\quad \mathit{CapabilityProjection} \wedge \\
&\quad \mathit{CapabilityCompletenessRule} \wedge \\
&\quad \mathit{NonnegativeCapabilityRule} \wedge \\
&\quad \mathit{CapabilityExistenceRule} \wedge \\
&\quad \mathit{CapabilityNonexistenceRule} \wedge \\
&\quad \mathit{ParentPartialResourceProjection} \wedge \\
&\quad \mathit{PartialResourceRule} \wedge \\
&\quad \mathit{PartialResourceSlackProjection}^{7} \wedge \\
&\quad \mathit{PartialResourceSlackRule}^{7} \wedge \\
&\quad \mathit{StackResourceProjection} \wedge \\
&\quad \mathit{StackProfitProjection}^{7} \wedge \\
&\quad \mathit{GroupingProjection} \wedge \\
&\quad \mathit{GroupingRule} \wedge \\
&\quad \mathit{SimpleGrouping}
\end{aligned}
$$

7 Schema omitted.

We begin with the *Ansatz*, and add the properties of uniqueness, disjointness and injectivity. Capability projection is included, as is partial resource projection with the parent system. Stack projection is included for resources and profit. Finally, we include simple grouping. Note that we have omitted some schema definitions. The structure of the omitted schemas is similar to those already defined, and were thus not defined explicitly. For instance, slack projections for partial resources are quite the same as for resources. Profit projection is based on stack projection used in conjunction with resource projection, and it employs a profit function of type *ProfitDelta*.

In the context of this schema, we would like to note that these definitions do not incorporate the possibility to perform search on the infeasible region, even though this is sometimes desirable for, e.g., allowing local search to escape local optima. We have defined here the feasibility as a property of the problem schema, and as such it cannot be violated. Thus, to introduce feasibility violations, one would have to add additional schemas in where the feasible region is *redefined*, using proper rules and disjunctions, along with the modifications on objective projection to penalize these violations. This is a potential area for further research.

A close examination of the schemas also reveals an order in which the projections need to be performed. Capabilities on a given activity must be computed first, and, based on their activeness, the corresponding resources are computed. Finally, the profit projection is computed for the activity based on the accumulated values of capabilities and resources.

The taxonomy of modeling elements identified three types of value dependencies: undefined, transitional, and fixed. The capability mechanism is fixed and the partial resource mechanism, transitional. In addition, we introduced the stack resources to accommodate some special cases of the dependencies between resources. Nevertheless, we are not able to model all the types of dynamic situations; for instance, the resources that are freely dependent on other resources cannot be expressed by transitional projections. *Situation dependent* functions, however, can express these types of situations. These situations include dependencies between resources (e.g., time dependent profit, as in soft time windows), dependencies within resources (e.g. time dependent time, as in modeling of rush hours), and capability dependent resources (e.g., capacity dependent time, as in load dependent travel times). The computation with these types of functions is not as efficient as with transitional, but as they are indeed more generic, they can express a wider variety of cases.

To express cases with complex dependencies between resources we may introduce new types of delta functions that are otherwise similar to those already defined, but include an additional parameter that describes the accumulated situation on the path at that point. The accumulated situation is a set of partial resource values, and thus the type definition of such function is the following:

$$DependentResourceDelta == (\mathbb{F}\, PartialResourceValue \times Activity \times Activity) \pfun \mathbb{Z}.$$

Similarly, the generic case of profit function can be defined as the following type:

$$DependentProfitDelta == (\mathbb{F}\, PartialResourceValue \times Activity \times Activity) \nrightarrow \mathbb{Z}.$$

The schemas employing these types of functions have been omitted as they are similar to those defined for transitional delta functions. When we introduce the language for describing the optimization models, we assume that both types of projections are available.

We should also note that we have omitted some minor details from the overall schema to make the basic principles clearer. One interesting case for capabilities in this schema is the non-monotonic capacities in PDP cases where the planning continues over the planning horizon. In these cases, there is a need to be able to specify that certain capabilities may not need to be zero at the end, relaxing the *CapabilityCompletenessRule*. This is due to the fact that in continuous planning, the state of the actor, in terms of capability, may be different in the beginning of the route than in the end. Thus, in the subsequent section, we assume that we may relax the completeness rule on a given subset of capabilities. Formally this could be defined as a disjunction of the completeness rule and an additional rule stating that the capability belongs to this subset. However, we do not present the schema for this as it is a straightforward extension.

To conclude this discussion on the developed model, we point out that while defining the schemas, we left some parts of some of the functions unconstrained by the schema predicates. These unconstrained elements from the individual schemas define the inputs required for defining a complete routing problem. If we go through the schemas, we see that a routing problem, as defined here, consists of the following data:

- *actors*, a set of actors;
- *activities*, a set of activities;
- *resources*, a set of resources;
- *capabilities*, a set of capabilities;
- *strictLowerBoundResources*, a set of resources that allow no waiting;
- domain and range of *capabilityDeltas*, a set of capabilities and their values for given activities;
- range of *capabilityExistence*, a set of capabilities which are required to be on the situation of the previous activity;
- range of *capabilityNonexistence*, a set of capabilities which are not allowed on the situation of the previous activity;
- range of *partialResourceLowerBounds*, numeric values limiting the accumulated values from below;

- range of *partialResourceUpperBounds*, numeric values limiting the accumulated values from above;
- domain and range of *capabilityParents*, a set of capabilities for given capabilities;
- domain and range of *activatingDeltas*, resource delta functions for given activation resource-capability pairs;
- domain and range of *activatingProfitDeltas*[8], profit functions for a given activation capability; and
- *groups*, a set of sets of activities that are not allowed to be assigned simultaneously.

As we have illustrated, for example, time windows can be defined using specific *capabilities*, *resources*, *activatingDeltas*, *partialResourceLowerBounds*, and *partialResourceUpperBounds*. Now, defining these elements and their dependencies forms the base for generating the different model variants and their extensions. This is the subject of the next section, where we tie together the domain and optimization models using a model transformation.

5.4 Model Transformation

As we discussed in Chapter 4, our approach for addressing a heterogeneous set of problems and managing the resulting complexity is to introduce variability into the product line architecture in a systematic way. Model-driven approaches employ both models and especially *model transformations*, and in this section we construct and illustrate a modeling language for transforming the UML-based concepts described in Section 5.2 into optimization model constructs described in Section 5.3.

5.4.1 Introduction

The constructs of the modeling framework can be divided into four layers. The elements of the domain model form the first layer, and we may locate the model transformation on the second layer. The third layer is used through the transformation and consists of the elements defining the data in the metamodel. These in turn reside in the elements of the fourth layer. This division is illustrated in Figure 26, where a subset of the elements of the framework is depicted. The overarching idea is to map different elements in the domain model into those in the metamodel in order to transform the data into a form that is suitable for optimization. The arrows between the elements denote the flow of data from one layer to another.

[8] Schema omitted.

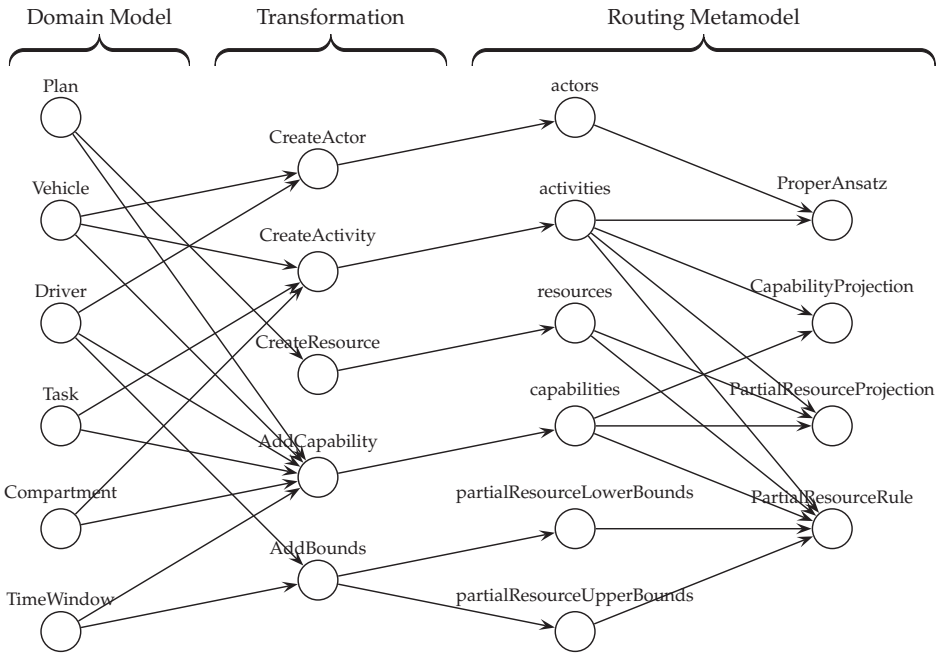


FIGURE 26 A subset of modeling elements and their relationships in the modeling framework.

Now, the metamodel layers address solely the modeling aspects within the system. The different modeling elements are not yet associated with any element of the implementation, more specifically, with any variation point or architectural element of a product line. The variation on the optimization model is introduced through the choice of mapping the elements from the first layer to those on the second. That is, the case-specific usage of the model transformation defines the optimization model variant. We give some illustrative examples in Section 5.4.3, but a more detailed list of model variants is given in Chapter 7 where we examine the usage of the transformation for expressing model variation within a product line. The list also provides an explanation on how to use the transformation in different cases.

In practice, the transformation used within the modeling framework is a horizontal exogenous transformation, since we transform between models conforming to two different metamodels and the abstraction level of the model increases during the transformation. This is clear due to the fact that several different constructs in the domain model are transformed into the same constructs in the optimization model. The implementation of the transformation is of a direct-manipulation type, and we have defined an API for this purpose. Finally, we note that the transformation itself is unidirectional, and applying the changes made to the optimization model instance during optimization are transferred back to the source model instance by mapping and storing the corresponding elements on each model instance during the transformation process and then using this mapping to resolve the original entities on the source model when needed.

Conceptually, we may locate the model generation on the M1-level of the metamodeling architecture. The layer on which the model generation process takes places in the metamodeling stack as well as the relations between different modeling levels within this modeling framework are depicted in Figure 27. As illustrated earlier in Chapter 4, the transformation definition refers to two metamodels, here, UML and routing metamodels, and the transformation engine reads one model conforming to the source metamodel, here the domain model, and writes another model conforming to the target metamodel, here the optimization model. The optimization model is then generated from the domain model. The domain instance is used to populate the solution instance with the appropriate starting solution depending on the state of the domain instance. After the optimization has been performed, the changes in the solution instance are resolved into the domain instance based on the mapping of different model elements done in the generation phase.

5.4.2 Transformation Interface

The model transformation is executed by reading the domain model element by element and generating the corresponding elements in the optimization model. The model generation takes place within the model transformation engine, and the domain model elements are used to generate the appropriate optimization model elements. The transformation is case-specific and employs an API through

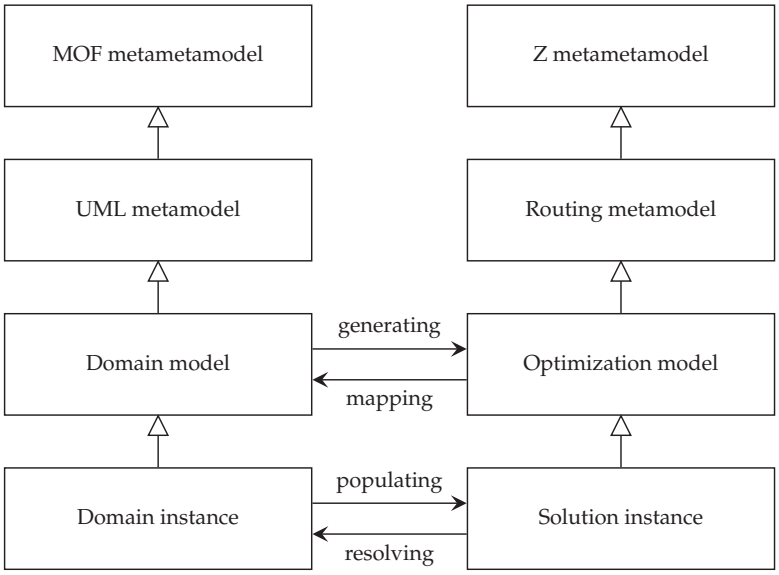


FIGURE 27 The modeling process and the two different metamodeling stacks within the framework.

which the optimization model elements are created. The operations are described here using Z schemas.

`CreateActor` — Creates a new actor with given start and end activities and adds the actor to the problem.

CreateActor

$\Delta GenericRoutingProblem$
$start? : Activity$
$end? : Activity$
$actor : Actor$

$actors' = actors \cup \{actor\}$
$actor.start = start?$
$actor.end = end?$

`CreateActivity` — Creates a new activity and adds it to the problem and assigns a unique activity identifier to it.

CreateActivity

$\Delta GenericRoutingProblem$
$act : Activity$
$a! : Activity$

$activities' = activities \cup \{act\}$
$a! = act$

CreateResource — Creates a new resource and adds it to the problem and assigns a unique resource identifier to it.

CreateResource

$$\Delta GenericRoutingProblem$$
$$res : Resource$$
$$r! : Resource$$

$$resources' = resources \cup \{res\}$$
$$r! = res$$

CreateCapability — Creates a new capability and adds it to the problem and assigns a unique capability identifier to it.

CreateCapability

$$\Delta GenericRoutingProblem$$
$$cap : Capability$$
$$c! : Capability$$

$$capabilities' = capabilities \cup \{cap\}$$
$$c! = cap$$

SetStrictLowerBound — Sets a requirement for a strict lower bound for a given resource.

SetStrictLowerBound

$$\Delta GenericRoutingProblem$$
$$r? : Resource$$

$$strictLowerBoundResouces' = strictLowerBoundResouces \cup \{r?\}$$

SetParent — Sets a given capability as a parent to another given capability.

SetParent

$$\Delta GenericRoutingProblem$$
$$c? : Capability$$
$$parent? : Capability$$

$$capabilityParents' = capabilityParents \oplus \{c? \mapsto parent?\}$$

SetAllowNonempty — Relaxes the capability completeness rule for a given capability.

SetAllowNonempty

$$\Delta GenericRoutingProblem$$
$$c? : Capability$$

This schema has been omitted.

AddCapability — Adds a capability with a given value to an activity with a given identifier.

AddCapability

$$\Delta GenericRoutingProblem$$
$$a? : Activity$$
$$c? : Capability$$
$$value? : \mathbb{Z}$$

$$capabilityDeltas' = capabilityDeltas \oplus \{c? \mapsto a? \mapsto value?\}$$

AddCapabilityRequirement — Adds a capability requirement for a given capability to an activity with a given identifier.

AddCapabilityRequirement

$$\Delta GenericRoutingProblem$$
$$a? : Activity$$
$$c? : Capability$$

$$capabilityExistence' = capabilityExistence \oplus \{a? \mapsto capabilityExistence(a?) \cup \{c?\}\}$$

AddCapabilityProhibition — Adds a capability prohibition for a given capability to an activity with a given identifier.

AddCapabilityProhibition

$$\Delta GenericRoutingProblem$$
$$a? : Activity$$
$$c? : Capability$$

$$capabilityNonexistence' = capabilityNonexistence \oplus \{a? \mapsto capabilityNonexistence(a?) \cup \{c?\}\}$$

AddActivityGroup — Adds a new simple group of activities from a given set of activity identifiers.

AddActivityGroup

$$\Delta GenericRoutingProblem$$
$$g? : \mathbb{F}\, Activity$$

$$groups' = groups \cup \{g?\}$$

AddBounds — Adds both a lower and upper bound on a given capability-resource pair.

AddBounds

$\Delta GenericRoutingProblem$
$c? : Capability$
$r? : Resource$
$lower? : \mathbb{Z}$
$upper? : \mathbb{Z}$

$partialResourceLowerBounds' = partialResourceLowerBounds \oplus$
$\quad \{(c?, r?) \mapsto lower?\}$
$partialResourceUpperBounds' = partialResourceUpperBounds \oplus$
$\quad \{(c?, r?) \mapsto upper?\}$

`AddTransitionalResourceFunction` — Adds the specified transitional resource delta function to the problem for a given resource and an activation capability.

AddTransitionalResourceFunction

$\Delta GenericRoutingProblem$
$c? : Capability$
$r? : Resource$
$f? : PartialResourceDelta$

$activatingDeltas' = activatingDeltas \oplus$
$\quad \{activatingDeltas(c?, r?) \mapsto f?\}$

`AddSituationDependentResourceFunction` — Adds the specified situation dependent resource delta function to the problem for a given resource and an activation capability.

AddSituationDependentResourceFunction

$\Delta GenericRoutingProblem$
$c? : Capability$
$r? : Resource$
$f? : DependentResourceDelta$

This schema has been omitted.

`AddTransitionalProfitFunction` — Adds the specified transitional profit function to the problem for a given activation capability.

AddTransitionalProfitFunction

$\Delta GenericRoutingProblem$
$c? : Capability$
$r? : Resource$
$f? : ProfitDelta$

This schema has been omitted.

`AddSituationDependentProfitFunction` — Adds the specified situation dependent profit function to the problem for a given activation capability.

AddSituationDependentProfitFunction

$\Delta GenericRoutingProblem$
$c? : Capability$
$r? : Resource$
$f? : DependentProfitDelta$

This schema has been omitted.

Using this relatively simple interface we are able to generate the problem input for the routing metamodel described. To illustrate the usage of the model transformation, we examine here the definition of some variants of the routing problem. The expressiveness of the framework as a whole is analyzed when we address the issues of variability in the subsequent chapters.

For completeness, we conclude by defining the initialization of a new instance of a routing problem. This schema illustrates the state in which the generation of metamodel elements is started.

InitRoutingProblem

$GenericRoutingProblem$

$actors = \emptyset$
$activities = \emptyset$
$resources = \emptyset$
$capabilities = \emptyset$
$strictLowerBoundResources = \emptyset$
$capabilityDeltas = \emptyset$
$capabilityParents = \emptyset$
$activatingDeltas = \emptyset$
$activatingProfitDeltas = \emptyset$
$groups = \emptyset$
$\mathrm{ran}\, capabilityExistence = \emptyset$
$\mathrm{ran}\, capabilityNonexistence = \emptyset$
$\mathrm{ran}\, partialResourceLowerBounds = \emptyset$
$\mathrm{ran}\, partialResourceUpperBounds = \emptyset$

All the input sets of the problem are initially empty.

5.4.3 Usage

As we have examined the structure of the system, we have referred to a case-specific *transformation definition* that controls the generation of the optimization models. In practice, these definitions are simple algorithms that employ the model generation API presented in the previous section.

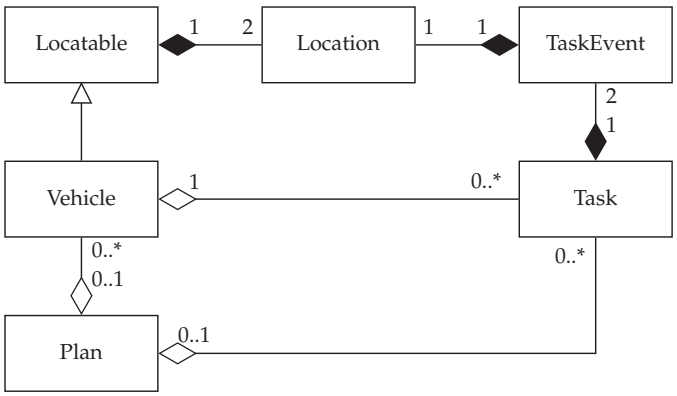


FIGURE 28 A simplified domain model for CVRPs, PDPs, and VRPBs.

The given API provides the means for generating the optimization model, but to provide a mapping from the optimization model elements to the domain model, that is, to enable the *decoding* of the solution attempt, we need to specify each domain object from which each activity was generated. This process is not explicit in the subsequent transformation definitions. Still, we assume that the mapped elements are available through a set of decoding methods. For example, method *GetTaskEvent* returns the *TaskEvent* object mapped to an activity, and *GetVehicle*, the corresponding vehicle.

As we go through the example transformations in this and the subsequent section, we follow the following conventions. The variable names of the capabilities and resources correspond to those in the adjoining illustrations where possible, that is, they are usually abbreviated to single characters. The domain objects are referred to with their full names. In addition, the properties of the objects are referred to as in *vehicle.Capacity*. Arrays and dictionary types are denoted with square brackets, *distances[a,b]*; and methods, with the typical parentheses, *AddResource()*. In all the model transformations, the domain model is accessed through a plan instance supplied as a parameter.

To illustrate the usage of the transformation, we employ different types of variants. There are basically two archetypes of problems in routing: the vehicle routing problem, and the pickup and delivery problem. These two differ profoundly in structure, and other variants and extensions can largely be discussed within the context of these two base structures. We also discuss a more complicated example of vehicle routing with backhauls as an illustration of a more complex problem structure. The domain model used in the three cases is illustrated in Figure 28. The illustration is a subset of the larger domain model given earlier and serves as a basis for the model transformations presented here.

The capacitated vehicle routing problem model consists of a single capability (as we need at least one), and a resource for capacity (in addition to the profit resource which is assumed to exist at every instance). We denote the capability

here as "v" for "vehicle", and the resource "c" for "capacity used". Each vehicle in the plan corresponds to an actor, and each task to an activity. The generation of VRP instances is defined in Algorithm 9. We begin the model generation by gen-

Algorithm 9 Generating capacitated VRP model instance.

```
v ← CreateCapability()
c ← CreateResource()
for all Vehicle vehicle in plan do
  start ← CreateActivity()
  end ← CreateActivity()
  CreateActor( start, end )
  AddCapability( start, v, +1 )
  AddCapability( end, v, −1 )
end for
for all Task task in plan do
  CreateActivity()
end for
AddBounds( v, c, 0, plan.Vehicles.Any().Capacity )
AddTransitionalResourceFunction( v, c, ( a, b ) → capacity[b] )
AddTransitionalProfitFunction( v, ( a, b ) → M−distance[a,b] )
```

erating the capability for the vehicle and the resource for capacity. Each vehicle within the plan on the domain model is then transformed into a pair of activities which are in turn used to define an actor. These activities are given the capability "v" as this ensures that the capability is present during the whole route. A single activity is then generated from each task. In addition, a constraint on the value of the capacity is set as the vehicle capacity and assumed to be identical between vehicles.

Finally, we generate the necessary functions. A transitional resource delta function is bound to the capability "v" and the resource "c". The function returns the capacity of the given activity. In practice, the capacities of different activities are assumed to be generated into an array for efficient retrieval. This approach is followed also in the examples to simplify the transformation definitions. The definitions assume implicitly that the computation of the capacities is performed after all the activities have been created. This process is depicted in Algorithm 10.

Algorithm 10 Generating capacity array.

```
for all Activity a in this do
  capacity[a] ← GetTaskEvent( a ).Capacity
end for
```

Similarly to the capacity function, the profit is defined as a function from a pair of activities into $M-$distance, where M is a sufficiently large value and distance is a matrix which is computed using the pseudocode given in Algorithm

Algorithm 11 Generating distance matrix.

```
for all Activity a in this do
  for all Activity b in this do
    distance[a,b] ← GetTaskEvent( a ).Location.GetDistanceTo(
        GetTaskEvent( b ).Location )
  end for
end for
```

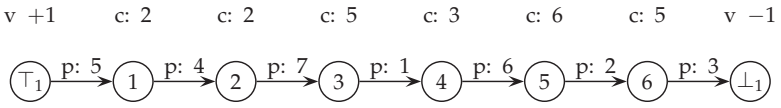


FIGURE 29 An example of a VRP model instance.

11. We assume here that the distances between locations have been computed by whichever approach needed by the case. The different approaches for obtaining these values in different cases are discussed in more detail in Chapter 7.

An example of a part of the resulting VRP instance is given in Figure 29. In general, we illustrate the results of the transformations with examples of the resulting routes of one or a few actors. The figures are not complete in a sense that they contain only those elements relevant to the transformation in question. Each activity accumulates both the profit (denoted here as "p") and the used vehicle capacity ("v") along the route. The capacity constraint is checked at the end of the route as the vehicle capability ("v") deactivates.

The pickup and delivery problem differs from the VRP within this framework in that we do not employ a resource for expressing the vehicle capacity. Instead, a capability is defined for this purpose as the vehicle capacity function is non-monotonic in the pickup and delivery problem. Moreover, each task consists of two parts, both of which are generated into an activity. The generation of PDP instances is illustrated in Algorithm 12. We begin by generating the capability denoting the vehicle capacity "c". Each actor start is then given the amount of vehicle capacity available, and each actor end, the same as negative. Each task is both given the vehicle capacity as negative in the pickup part and an additional precedence capability as positive, and vice versa for each delivery activity. Finally, the profit function is defined as in the VRP case. An example of the resulting instance is depicted in Figure 30. In contrast to the VRP, each activity consumes the capability of the actor and is linked to another activity by a unique capability, which could be interpreted as "the capability to deliver the goods picked up". As we observed, a PDP model is capable of expressing VRP problems by locating all the pickups (or all the deliveries) on the central depot. This approach can also be used in the model transformation, and while this is not typically an efficient representation of the problem, it may sometimes be useful to be able to employ algorithms for PDP problems in a VRP setting.

Algorithm 12 Generating PDP model instance.

```
c ← CreateCapability()
for all Vehicle vehicle in plan do
  start ← CreateActivity()
  end ← CreateActivity()
  CreateActor( start, end )
  AddCapability( start, c, +vehicle.Capacity )
  AddCapability( end, c, −vehicle.Capacity )
end for
for all Task task in plan do
  pdp ← CreateCapability()
  pickup ← CreateActivity()
  AddCapability( pickup, c, −task.Pickup.Capacity )
  AddCapability( pickup, pdp, +1 )
  delivery ← CreateActivity()
  AddCapability( delivery, c, +task.Delivery.Capacity )
  AddCapability( delivery, pdp, −1 )
end for
AddTransitionalProfitFunction( c, ( a, b ) → M− distances[a,b] )
```

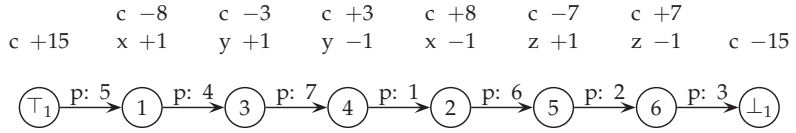


FIGURE 30 An example of a PDP model instance.

The vehicle routing problem with backhauls is an interesting intermediate of the two base types. In this case, we need to track the state of the actor on a route: whether linehauls or backhauls are being collected at each point. This state defines the tasks that can be served at that point on the route. Recall that there are two restrictions in the VRPB: each linehaul must occur before any backhaul, and a route cannot consist only of backhauls.

The routing problem with backhauls can be modeled similarly to the VRP by employing resources for capacity, but by keeping track of two different capacities: one for linehauls and another for backhauls. As the generation of these constraints is somewhat complex, we do not present this approach here. Instead we model the case using the PDP approach given in Algorithm 12 and illustrate only the generation of the additional constraints. The generation of these constraints in a VRPB instance is illustrated in Algorithm 13. To state the restrictions

Algorithm 13 Generating VRPB model instance.

```
lhd ← CreateCapability()
bhp ← CreateCapability()
for all Task task in plan do
  if task.IsLinehaul then
    AddCapability( delivery, lhd, +1 )
    AddCapabilityProhibition( delivery, bhp )
  else
    AddCapability( pickup, bhp, +1 )
    AddCapabilityRequirement( pickup, lhd )
  end if
end for
SetAllowNonempy( lhd )
SetAllowNonempy( bhp )
```

to the problem structure, we introduce two capabilities that serve as status flags to indicate which types of tasks have been performed by the vehicle. Each linehaul delivery has capability to indicate that linehauls have been served on this route ("linehaul delivered" or "lhd") and a prohibition for backhauls as they all must appear after all linehaul operations. Each backhaul pickup, in contrast, has a capability to indicate that backhauls have been picked during the route ("backhaul picked" or "bhp") and a requirement for linehaul delivery as the route cannot consist entirely of backhaul customers. Finally, we state that both of the flag capabilities can have a non-empty value at the end of the route as the routes may contain any number of these activities. An example of the resulting instance is depicted in Figure 31. As we can see, the two types of tasks differ in their capabilities as described in the algorithm.

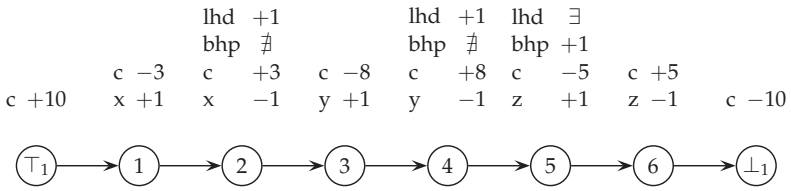


FIGURE 31 An example of a VRPB model instance modeled as a PDP with additional constraints.

5.5 Summary of Approach

There is a need for a unified modeling approach to rich problems. Such an approach would enable a single system to address a wider variety of cases of the heterogeneous domain of vehicle routing. Resource constrained paths provide one mechanism for modeling rich interactions and more complex resource calculations than previously possible. Using this as a starting point, we attempted to increase the expressiveness of resource accumulation mechanisms. We aimed to do this by introducing, in a structured way, new constructs which could, together with different mechanisms for computing resources, be used in expressing much of the real-life inspired constraints now beginning to appear in a variety of VRP cases.

We introduced a domain model of routing problems, a routing metamodel, and a model transformation from the domain model into routing models. The routing metamodel was used to provide building blocks for modeling different variants and extensions. The set of tools should provide means for describing a variety of routing problems within a software product line.

It is notable that while the modeling framework does make it easier to define a routing variant, considerable knowledge of both models and solution methods is still needed to utilize the system. The proposed approach does not remove the operations researcher from the process, although, arguably, makes her life a bit easier by offering an expressive toolkit.

A hint of the expressiveness of the system was provided by the examples of different cases throughout this chapter. As we went through the illustrations, we encountered increasingly complex problem instances. It is evident that the combinations of these cases are even more complex, and combining all the problem extensions mentioned in the preceding chapters would probably render the problem too complex to be solved currently, even if some of the modeling elements overlap. Fortunately, as we will see in the next chapter, most of the routing variants and extensions do not introduce an exponential number of resources (including capabilities) as the problem size increases, and this helps to control the dimensionality of the problem. It is, nevertheless, an open question how to solve the most complex combinations of these constructs. With this in mind, we now move back to the solution part of the process. In the next chapter, we examine the solution process within the developed framework.

6 OPTIMIZATION IN THE FRAMEWORK

As we now have the knowledge of the structure of the routing metamodel and are able to express different problem variants, in this chapter, we examine the optimization process within the developed framework. This examination focuses on local search as it is the central component in most of the solution methods applied to routing.

The structure of this chapter is the following. In Section 6.1, we describe the structure of the metamodel implementation and discuss the usage of solution methodology within the metamodel. Furthermore, an analysis of different modeling constructs and resulting search neighborhoods is given by examining the locality of feasibility checks and objective evaluation. In Section 6.3, we discuss the elements found in VRP models and analyze their effect on local search as well as analyze the local search operators necessary for traversing the search space defined by the routing metamodel. Finally, in Section 6.2, we provide some preliminary results from applying the system into practice in both benchmark and real-life cases.

6.1 Optimization Process

This section has two purposes: firstly, we aim to express the basic building blocks of the currently used local search operators in a formal way to ensure that we may, indeed, employ most of the existing solution methodology also within our modeling framework. Secondly, we address the issue of computational complexity within the metamodel to analyze the implications of the modeling framework to the performance of an optimization system.

6.1.1 Expressing Local Search

To express the basic building blocks of local search operators in the context of the developed metamodel, we define schemas for altering the state of the *Ansatz*.

Z notation offers tools for these types of definitions and as the *Ansatz* has been defined formally in the preceding chapter, we employ this notation here as well.

Local search consists mainly of manipulation of the sequences of activities mapped to actors. There are two ways of expressing the basic building blocks of these manipulation operations: per activity and per sequence. Per activity manipulation would describe the modification one activity at a time, but as a single activity is a special case of a sequence, we utilize here a more general approach, and express modifying the *Ansätze* one segment at a time. The basis of this manipulation is the ability to *add* and *remove* sequences. A third primitive is possible, but not mandatory; we employ here also a *reverse* operation for a clearer description of the search operators.

If we are able to express adding and removing sequences (and for convenience, reversing), we can build local search operators by combining these primitives. Next, we express these three operations formally. For completeness, however, we begin by stating the state of an *Ansatz* before the optimization start. The state of an *Ansatz* can be reset using the following schema:

ResetRoutingProblem

$$\Delta GenericRoutingProblem$$

$$\forall\, actor \in actors \bullet routes'(actor) = \langle actor.start, actor.end \rangle$$

After the operation, each route consists only of the corresponding actor start and end activities.

Adding sequences to routes is done by specifying an activity after which the new sequence is to be added. To express this operation without additional parameters, we first introduce a function for determining the current actor of a given activity. This is expressed as follows:

$$actorOf : \operatorname{paths} Activity \times Activity \pfun Actor$$

$$\forall\, paths : \operatorname{paths} Activity;\ a : Activity \bullet$$
$$actorOf(paths, a) = (\mu\, actor \in \operatorname{dom} paths \mid a \in \operatorname{ran} paths(actor))$$

As we manipulate the sequences, it would be convenient to be able to extract a subsequence from a sequence based on its start and end elements. For this, we introduce the following operator:

$[X]$

$$\_ \,\mathsf{to}\, \_ \sim \_ : X \times X \times \operatorname{seq} X \to \operatorname{seq} X$$

$$\forall\, x, y : X;\ s : \operatorname{seq} X \bullet$$
$$x \,\mathsf{to}\, y \sim s = s^{\sim}(x) \mathinner{..} s^{\sim}(y) \upharpoonright s$$

The operator, given two elements and a sequence these two elements are on, extracts the subsequence defined by the elements in between these two (including the elements themselves).

Using the given definitions, expressing insertion of a sequence into a given location is straightforward. The schema for performing this operation is given below.

Add

$$
\begin{array}{l}
\Delta GenericRoutingProblem \\
prev? : Activity \\
toAdd? : \mathrm{iseq}_1\, Activity \\
\hline
prev? \in assigned \setminus \{\, actor \in actors \bullet actor.end \,\} \\
\forall\, a \in \mathrm{ran}\, toAdd? \bullet a \in unassigned \\
(\mathbf{let}\ actor == actorOf(routes, prev?) \bullet \\
\quad (\mathbf{let}\ route == routes(actor) \bullet \\
\quad\quad routes' = routes \oplus \{\, actor \mapsto \\
\quad\quad\quad (head\, route\ \mathsf{to}\ prev? \sim route)^\frown \\
\quad\quad\quad toAdd?^\frown \\
\quad\quad\quad (next(routes, prev?)\ \mathsf{to}\ last\, route \sim route)\,\}))
\end{array}
$$

Firstly, we require that the activity after which we insert the sequence, *prev?*, is assigned to a route and is not any of the actor end activities. This is due to the restriction that algorithms should not move actor start and end activities. Secondly, we require that each activity on the inserted sequence is currently unassigned. The insertion is performed by overriding the route of the corresponding actor by concatenation of sequences; first we include the route from start to the insertion point defined by *prev?*, then we include the sequence to add, and finally, the remaining sequence to the end of the route.

However, manipulation of the sequences is not enough. We also need to ensure that the projected values are consistent with the changes made. We use resources here as an illustration of this operation. The following schema describes the update operation, given the changed sequence, for example, the one added.

UpdateResources

$$
\begin{array}{l}
\Delta GenericRoutingProblem \\
toUpdate : \mathrm{iseq}_1\, Activity \\
\hline
(\mathbf{let}\ route == routes(actorOf(routes, head\, toUpdate)); \\
\quad first == head\, toUpdate \bullet (\forall\, r \in resources \bullet \\
\quad\quad resourceValues'(first)(r) = max\{ \\
\quad\quad\quad resourceLowerBounds(first, r), \\
\quad\quad\quad resourceValues(prev(routes, first))(r) + \\
\quad\quad\quad resourceDeltas(r)(prev(routes, first), first)\} \wedge \\
\quad\quad \forall\, a \in (next(routes, start)\ \mathsf{to}\ last\, route \sim route) \bullet \\
\quad\quad\quad resourceValues'(a)(r) = max\{ \\
\quad\quad\quad\quad resourceLowerBounds(a, r), \\
\quad\quad\quad\quad resourceValues'(prev(routes, a))(r) + \\
\quad\quad\quad\quad resourceDeltas(r)(prev(routes, a), a)\}))
\end{array}
$$

The update is performed on the affected route and over all resource of the problem. The first activity, the head of the sequence *toUpdate*, is given its resource values from the original values of the preceding activity. The subsequent activities *on the whole route* have their values computed from the new values (denoted here as $resourceValues'$) at the preceding activity exactly as in the corresponding projection.

As updating the values of different projections repeats much of the computation already defined in the previous chapter, we omit the detailed schemas for updating the rest of the projections. The full update operation for a given primitive operation is given by the following schema:

$$
\begin{aligned}
&Update \mathrel{\widehat=} UpdateCapabilities^1 \land \\
&\quad UpdatePartialResources^1 \land \\
&\quad UpdateStackResources^1 \land \\
&\quad UpdateGroups^1
\end{aligned}
$$

Each element of the schema corresponds to a projection given in the preceding chapter.

The **removal** of sequences of activities is similar to the insertion. The following schema defines the operation for removing a given sequence from a route.

Remove

$$
\begin{aligned}
&\Delta GenericRoutingProblem \\
&toRemove? : \mathsf{iseq}_1\, Activity \\
&removed! : \mathsf{iseq}_1\, Activity \\
&\hline
&\exists\, actor \in actors \bullet toRemove? \mathsf{\ in\ } routes(actor) \\
&(\mathbf{let}\ actor == actorOf(routes, prev?) \bullet \\
&\quad (\mathbf{let}\ route == routes(actor) \bullet \\
&\qquad routes' = routes \oplus \{ actor \mapsto \\
&\qquad\quad (head\ route \mathsf{\ to\ } prev(routes, head\ toRemove?) \sim route)^\frown \\
&\qquad\quad (next(routes, last\ toRemove?) \mathsf{\ to\ } last\ route \sim route) \})) \\
&removed! = toRemove?
\end{aligned}
$$

First, we require that the sequence we are removing is a proper sequence on some route. The actual removal is performed simply by concatenating the sequences around the removed sequence. In addition, we return the removed sequence for further use using *removed!*; this is useful in composing the operations, and we will give examples of this shortly.

Reversing subsequences could be achieved by per activity manipulation, but as this is a frequent component of local search, we introduce it here as a primitive operation. The reversal of a subsequence in a route is defined by the following schema:

[1] Schema omitted.

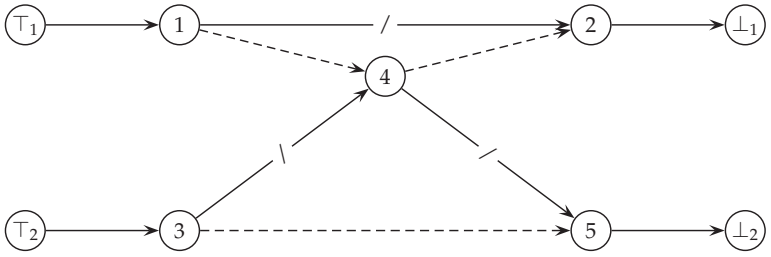


FIGURE 32 An example of a relocate operation.

Reverse

$\Delta GenericRoutingProblem$
$toReverse? : \mathsf{iseq}_1\, Activity$

$\exists\, actor \in actors \bullet toReverse?\ \mathsf{in}\ routes(actor)$
$(\mathbf{let}\ actor == actorOf(routes, prev?) \bullet$
$\quad (\mathbf{let}\ route == routes(actor) \bullet$
$\quad\quad routes' = routes \oplus \{ actor \mapsto$
$\quad\quad\quad (head\ route\ \mathsf{to}\ prev(routes, head\ toReverse?) \sim route)^\frown$
$\quad\quad\quad rev\ toReverse?^\frown$
$\quad\quad\quad (next(routes, last\ toReverse?)\ \mathsf{to}\ last\ route \sim route)\ \}))$

Similarly to removal, we require that the sequence to reverse is, indeed, a proper subsequence of some route. The reversal is done by concatenation of the beginning of the route into the reversed sequence, and the reversed sequence, to the end of the route.

These three basic primitives can now be used to express the local search operator in the defined routing metamodel. We illustrate this by giving **examples** of such operations. Firstly, we define a single relocation operation from a route to another, and secondly, a 3-opt move within a route. The formal descriptions of search operator building blocks can be used to describe the exact operation of different algorithms in a formal way. The algorithm designers may utilize this formalization as a way to define and examine search algorithms.

The relocation operator moves a single activity, or a sequence of one, from a route to another. A single step within the relocate algorithm is depicted in Figure 32. In the example, the activity 4 is moved from route 2 to route 1. This is done by replacing the arcs (1,2), (3,4), and (4,5) with (1,4), (4,2), and (3,5), and can be expressed formally as

$$(Remove \fatsemi Update[toRemove? / toUpdate?]) \gg$$
$$\quad (Add[removed? / toAdd?] \fatsemi Update[removed? / toUpdate?])$$

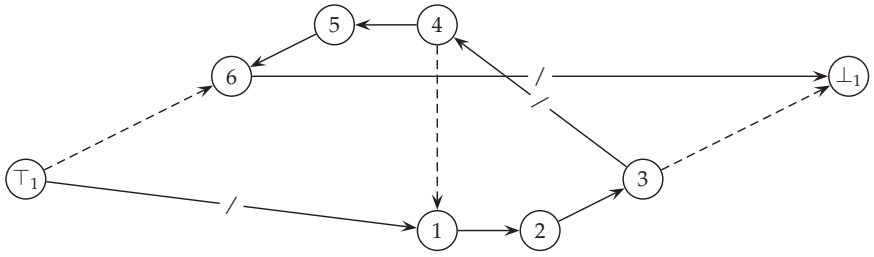


FIGURE 33 An example of a 3-opt operation.

where

$$toRemove? = \langle 4 \rangle,$$
$$prev? = 1.$$

In other words, we remove the activity 4 (within "toRemove?" as a parameter of *Remove*) and update its previous route without the activity. The result of this operation is then given to the operation which adds the given sequence after the activity 1 (in "prev?" as a parameter of *Add*) and updates the new route.

A more complex example is one defined in an arc neighborhood. The 3-opt operator removes three arcs and replaces them with three different ones. A single step in this algorithm can take the following form: remove a subsequence from the route, reverse it, and reinsert it to some other location. Such an operation is depicted in Figure 33. In the example, the original route is $\langle \top_1, 1, 2, 3, 4, 5, 6, \bot_1 \rangle$, and we remove the subsequence $\langle 4, 5, 6 \rangle$, reverse it, and reinsert it after the actor start. This results in replacement of the arcs ($\top_1$,1), (3,4), and (6,$\bot_1$) with ($\top_1$,6), (4,1), and (3,$\bot_1$). This can be expressed formally as

$$(Remove \fatsemi Update[toRemove? / toUpdate?]) \gg$$
$$(Add[removed? / toAdd?] \fatsemi Reverse[removed? / toReverse?] \fatsemi$$
$$Update[removed? / toUpdate?]),$$

where

$$toRemove? = \langle 4, 5, 6 \rangle,$$
$$prev? = \top_1.$$

In other words, we remove the given sequence and update the route, then reinsert the removed sequence and reverse it, and update the route again. In this formalization, the route is updated twice as we defined that only the subsequence from the first modification to the end of the route is updated, and we do not know which of the two updates is redundant. In practice, there is no need to update the changed sequence twice.

These examples illustrate the usage of the basic building blocks of local search. The *Add*, *Remove* and *Reverse* operations capture the operations needed

in traversing arc neighborhoods typically used in routing. These operations are the primary tools for algorithms to traverse the search space. As we are not aware of any widely used local search operator for the VRP that cannot be expressed using sequence manipulation, we are now able to perform search in the context of the developed model. Next, we address the issue of computational complexity of this search, and evaluate the applicability of the developed modeling approach in practice.

6.1.2 Complexity of Evaluation

The previous section demonstrated that we are able to perform the necessary computations with the metamodel. However, more interestingly, given a possible update operation, the key question is how easily we can determine whether we result on a feasible or infeasible region in the constraint space, that is, whether the predicates of the schemas would remain true after an update. Answering this question determines the computational complexity of the search and needs to be evaluated in order to be able to determine the suitability of the developed approach. Fortunately, there are some strong results and general frameworks against which we may analyze our approach. This analysis is done for the building blocks of the developed metamodel, thus removing the need to analyze different optimization models separately. The modeling elements used in each case determine the computational complexity of operating with the corresponding model.

Three generic approaches in employing local search have been fairly recently introduced in the literature: *lexicographic search*, *segment concatenation*, and *sequential search*. Lexicographic search was first introduced by Kindervater and Savelsbergh [125] (according to [109]), and it employs slack variables and maintains a set of global variables for achieving constant time complexity in feasibility and objective evaluation in search operators based on both node and arc neighborhoods. This approach is in use in the current implementation of the developed system, and will be described formally below. Segment concatenation with resource extension functions was introduced by Irnich in [110] as a strategy for achieving constant time evaluation of the result of concatenation of two arbitrary segments. This approach should be applicable in the context of the developed metamodel, but due to scope limitations we only briefly illustrate the approach and present a strategy for employing this approach in the future. Finally, sequential search was introduced by Irnich *et al.* [113]. It is an exact[2] search space reduction method that was shown to speed up typical local search operations considerably by discarding non-improving moves as soon as they can be detected. This approach is briefly discussed, but its applicability to the developed metamodel is left for future research.

Lexicographic search is largely based on slack projections. These projections allow constant time feasibility and objective evaluation for neighborhoods

[2] By exact, we mean one that guarantees that the (local) optimum can be reached even if the search space is reduced.

that consider *a single activity at a time*. As an example of this approach, we formulate the evaluation of feasibility in constant time in the resource case (as we defined slack projections in terms of this mechanism). The following schema can be used to determine whether insertion of an activity at a given location results in a feasible *Ansatz*:

TryInsertActivity

$$
\begin{array}{l}
\Xi GenericRoutingProblem \\
prev? : Activity \\
activity? : Activity \\
feasibility! : \mathbb{N} \\
\hline
feasibility = 1 \Leftrightarrow \\
\quad \forall r \in resources \bullet \\
\qquad max\{resourceLowerBounds(activity?, r), \\
\qquad\quad resourceValues(prev?)(r) + \\
\qquad\quad resourceDeltas(r)(prev?, activity?)\} + \\
\qquad resourceDeltas(r)(activity?, next(routes, prev?)) - \\
\qquad resourceValues(prev?)(r) \leq \\
\qquad resourceSlackValues(next(routes, prev?))(r)
\end{array}
$$

To compute whether an activity can be inserted on a route, we need to compute the increase in the resource values caused by the insertion. This is computed similarly to the resource projection; a lower bound or the value at the previous activity plus the delta between the previous and the current is selected as the value at the current activity. A delta to the next activity is then added to the value and from this the effective increase is computed by subtracting the value at the preceding activity. If the increase in the resource does not exceed the slack value, the situation remains feasible after the insertion. As we can see, the feasibility is computed in $\mathcal{O}(\#resources)$, which is constant and small in most VRP variants.

As mentioned, the technique of Kindervater and Savelsbergh can also consider arc neighborhoods. The search neighborhood of an algorithm that splits the route into k segments (by removing k arcs) and concatenates them into a different sequence (by adding k arcs) is of size n^k, where n is the number of activities on the route. As the segments considered in this neighborhood have arbitrary lengths, naively checking the feasibility of these segments in an arbitrary order requires traversing the segment at each step. However, if the computation is performed so that the different segments are checked in an order that progresses the route gradually and the intermediate values are stored, all the required computations can be performed in n^k steps. This can be achieved by introducing a set of gradually computed global variables: by adding one activity at a time to a sequence whose state is kept in the global variables, at each step the number of operations is independent of the segment length.

The lexicographic search technique is illustrated by the following example. The illustration is given in the context of partial resources. The target route consists of two activities, 1 and 2, and the route start and end activities have the

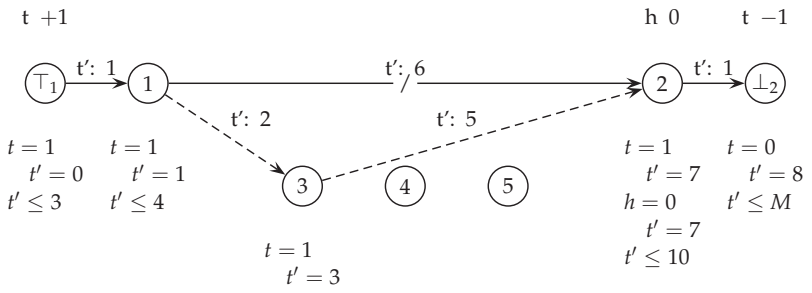


FIGURE 34 Lexicographic search — the first step.

capability "t" denoting time. In addition, activity 2 has a time window associated using capability h. The time window is imposed by setting

$$partialResourceUpperBounds(h, t') = 10$$

and

$$capabilityParents = \{h \mapsto t\}.$$

We attempt to check whether sequences can be inserted between activities 1 and 2.

The insertion of sequence of one activity is depicted in Figure 34. First we note that the activities have their maximum allowed values for resources[3] illustrated in the figure. As the route end activity has no time window defined, its maximum allowed value for the resource "t'" is given here a sufficiently large M. In contrast, as the activity 2 has an upper bound of 10 imposed for the resource "t'", its maximum value is set to 10, and depicted in the figure as $t' \leq 10$. Consequently, the maximum value at activity 1 equals 4 as the delta between activities 1 and 2 is 6 for "t'".

At the first step, we insert activity 3 between activities 1 and 2. The value at that activity for (t, t') now equals 3, and as the delta between 3 and 2 is 5 and the maximum value at 2 is 10, the insertion can be made. This can be verified in $\mathcal{O}(\#resources)$.

In the second step, we take advantage of the already computed values, which are stored as global variables similarly to the approach by Kindervater and Savelsbergh. The next activity, 4, has a capability "k", and a time window defined by

$$partialResourceLowerBounds(h, t') = 6$$

and

$$capabilityParents = \{k \mapsto t\}.$$

[3] For clarity, we depict the maximum values instead of slack. The corresponding slack can be computed from the maximum value and the current value at that activity.

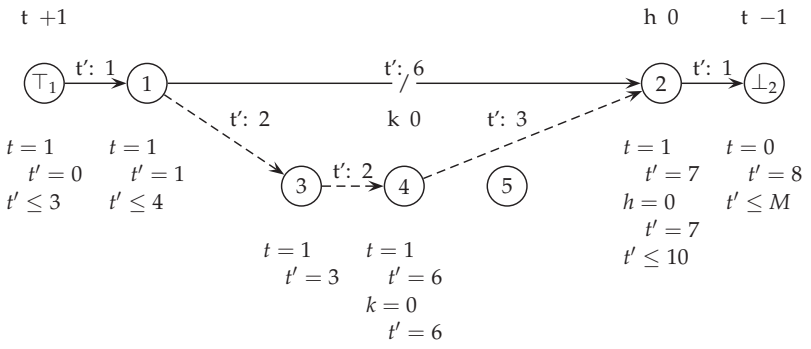


FIGURE 35 Lexicographic search — the second step.

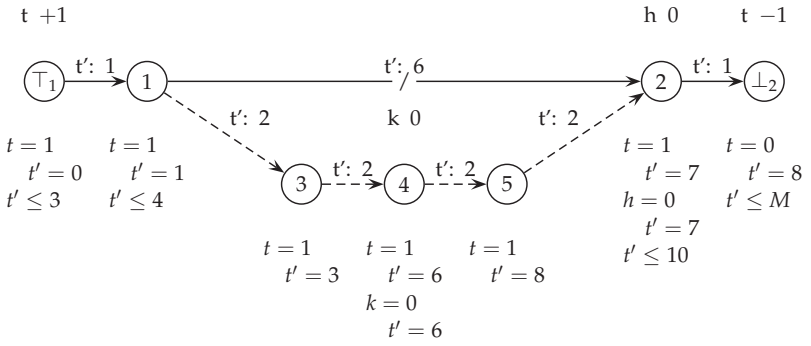


FIGURE 36 Lexicographic search — the third step.

Using these values, the second step of the search can be illustrated as in Figure 35. We now attempt to insert the sequence $\langle 3, 4 \rangle$ between the activities 1 and 2. The key here is that the values at activity 4 can be computed based on those already computed for activity 3. Due to the lower bound on "t'", the resulting value equals 6 (5 plus 1 for the waiting), and as the delta between 4 and 2 equals 3, the insertion can be made. This can be verified in $\mathcal{O}(\#resources)$, which is independent of the sequence length.

Similarly to step two, the third step inserts the sequence $\langle 3, 4, 5 \rangle$ and computes the values at activity 5 based on those already computed. This is illustrated in Figure 36. It can be seen that also this insertion can be checked for feasibility in constant time. This example illustrates the usage in a simplified scenario. In practice there is need to keep track of the changes on the *source* route as well (in case of segment relocation, for instance). The process is similar and thus not illustrated here.

Note also that in this context, the reverse operation is not exactly meaningful as sequences are constructed one activity at a time, but reversal can effectively be performed simply by inserting the activities one by one in a reverse order.

Moreover, interestingly, lexicographic search can also be used in insertion at multiple subsequent places if changes are made at multiple locations, that is, if we, for example, insert a pickup activity somewhere in the route and search for the location of the delivery, we may keep track of the already computed sequence between the two activities to avoid computing the same values repeatedly.

As we illustrated, the feasibility of *resources* can be computed in constant time. Moreover, *capabilities* can also be regarded as resources and their minimum required and maximum allowed values can also be computed. This would, in essence, provide us with information of the required capabilities on an activity such that the situation is feasible from that activity onward. Adding capabilities is a matter of computing these bounds and enumerating their *active* values in the feasibility check. It was shown by Irnich and Desaulniers [111] that different compatibility, incompatibility and precedence constraints can be modeled so that their resource feasibility can be checked in constant time.

Partial resources can, consequently, also be computed in constant time, even though computing with only one capability was illustrated in the example. In the case of partial resources, some care has to be taken in the computation of the lower bounds as we need to find the active value at each activity.

Stack resources are relatively simple to consider in lexicographic search — with one pitfall. In the typical case, one needs to compute with the current active delta function at the preceding activity to be able to evaluate the actual effects of a removal or insertion. However, if an activation or deactivation occurs in the newly added segment, one has to compute the effects of this change until the deactivation of that delta function to measure the exact effect. Now, it must be noted that perhaps the most usual use case is different cost structures in a heterogeneous fleet. In this case, the activation and deactivation occur in the actor start and end activities, which cannot be moved by the search operators. This makes the usage of stack resources in the usual cases a constant time operation: at each step we simply evaluate the change in values using the active delta function at each route.

Grouping constraints increase the computational burden, but not in the same sense as resources. While resource-based mechanisms introduce resources to evaluate, grouping constraints increase the number of potential local search operations. The number of potential local search operations is multiplied by the number of elements in the group under consideration. This is, however, more a matter of the actual search operations (neighborhoods) implemented rather than the implementation of feasibility checks. The design and evaluation of these neighborhoods is left largely for future research.

We have thus far limited our discussion to feasibility checks, but as profit can be regarded as a resource, the same computational complexity applies to its evaluation. In practice, the objective evaluation can be accomplished by considering the removed and newly added arcs and their contribution to the profit.

To summarize the feasibility and objective evaluation using lexicographic search, given a single search operation, the following computational complexity classes for different mechanisms in the metamodel can be stated:

- resources: $\mathcal{O}(\#resources)$,
- capabilities[4]: $\mathcal{O}(\#active\ capabilities)$,
- partial resources: $\mathcal{O}(\#resources \cdot \#active\ capabilities)$, and
- stack resources[5]: $\mathcal{O}(1)$.

This indicates a reasonable computational burden for local search, one that increases according to the properties of the case model and, primarily, not its size. However, as mentioned, the mechanism operating with stack resources is not fully compatible with lexicographic search. Moreover, we must note that, currently, constant time feasibility checks are not fully implemented for capabilities in the mechanism of partial resources in the lexicographic search scheme. This is due to the fact that we plan to employ a more powerful approach, *concatenation of segments in constant time*, in the next implementation of the metamodel. We will present a preliminary evaluation of this approach.

Segment — or sequence — concatenation with resource extension functions was applied to routing by Irnich [110]. In the paper, they introduced a technique for achieving a constant time evaluation of concatenating any two segments. This is made possible by computing additional data for each possible sequence currently in the *Ansatz*. More specifically, we need to store an increase of the value of each resource for each segment. Moreover, as there are bounds on the resources, the increase is dependent of the values at the preceding activity and thus is not directly computable. Despite of this, the evaluation of the increase can be achieved in constant time under some conditions. We do not use the conditions stated in [110] here to prove that these requirements are met in the developed metamodel, but leave this for further research.

The example given in the context of lexicographic search can be reformulated into that of segment concatenation as illustrated in Figure 37. In the example, we again insert the sequence $\langle 3, 4, 5 \rangle$ after activity 1. We have predetermined the length of the sequence with respect to "t'" (the combined arcs have a length of 4), and a *cut point* before which additional waiting occurs (at time 4). The cut point is computed from the lower bounds on the activities on the sequence and conceding, at each activity, the value after which waiting occurs. This is computed progressively backwards on the sequence by subtracting the delta from the lower bound. In this case, activity 4 has a lower bound of 6, which results in a cut point of 4 when the delta from 4 to 3 is subtracted. Given a start time *start* at which the sequence is started (here 3, the value at activity 1 plus the delta from activity 1 to 3), the increase in the resource "t'" can now be computed by $length(t') + max\{0, cut(t') - start\}$, which in this example, results in 5. When added to the value at activity 1 and the adjoining delta, this results in the value of 8 at activity 5. This is exactly the result obtained in the case with lexicographic search. Thus, we have computed the effective increase in resources when

[4] Not implemented.

[5] Assuming no activation and deactivation on the moved sequence.

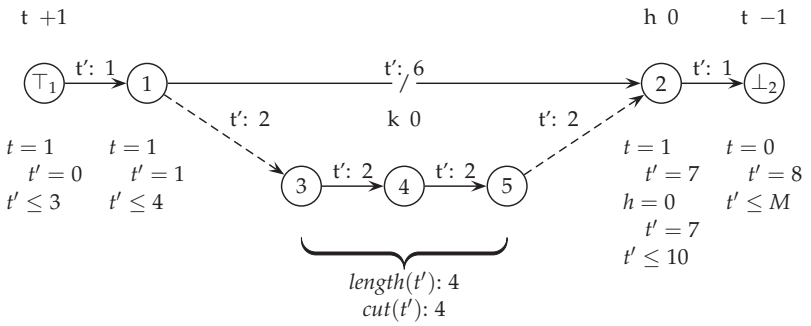


FIGURE 37 Concatenation of segments in constant time.

inserting the sequence after activity 1, making the constant time feasibility check possible also when using partial resources. It is worth pointing out that the option for reversing the sequences requires the data to be computed twice for each sequence — once for both directions.

Similarly, the mechanism using stack resources can be employed by computing the increase in each resource using each delta function defined to the problem. When inserting a sequence, the increase in resources is computed using the currently active resource. If deactivation of the delta function occurs within the sequence, the precomputed value can take this into account by calculating the traversal of each part of the sequences by a different delta function. Similarly, if activation occurs, the values at the sequence following the inserted can be obtained so that they have been computed with the newly activated delta function. This is possible as also these values have been precomputed. This indicates a constant time feasibility checks for the stack resources and the overhead for computing this additional data should remain reasonable in most routing cases.

The utilization of constant time segment concatenation is, indeed, promising for the developed metamodel, and the following generic strategy can be used to include the mechanism to the developed metamodel. In practice, the metamodel needs to be expanded by introducing additional slack projections that include the necessary data for checking feasibility and objective evaluation in constant time for concatenation of two arbitrary sequences. The utilization of this data is done by introducing schemas for the search operators using segment concatenation. The core schemas are somewhat independent of the search strategy, but the slack projections are highly strategy-specific. It is conceivable that the metamodel will be refined in terms of these additional projections in the future. The relative stability of the core schemas against changes in the search strategy would suggest that the description is somewhat stable.

Sequential search — a complementary strategy for segment concatenation — has recently gained attention in local search based methodology. This technique restricts the search space by evaluating *partial moves* and discarding those moves that cannot produce improving moves. The approach was discovered in

the 1970s and was applied to the TSP and the graph partitioning problem, but has only recently been applied to constrained problems, such as the VRP [109].

The technique is based on the decomposition of moves and their partial evaluation. The requirement is that each partial move is cost-independent and that the sum of partial moves is the value of the whole move. If the partial moves are ordered according to their partial profits, it has been shown that one only needs to consider the $p \in \{1, \ldots, n\}$ first partial moves where the partial profit is greater than $p(P/n)$, where P is the lower bound for the overall profit and n is the number of partial moves the move is composed of [113]. Note that the concatenation of arbitrary sequences is a requirement for the sequential search as the partial moves cannot be considered in the order of their profit in the lexicographic search strategy. The sequential search strategy should be applicable to a subset of problems described by the metamodel. The requirement for cost-independence restricts the applicability of the technique in some cases. For instance in the context of stack resources, the partial moves must be considered as long as there is at least one delta function yielding a potentially improving move. A detailed analysis of this technique is, however, left for further research.

The presented analysis of computational complexity assumed *transitional* resource and profit functions. But as discussed in the previous chapter, transitional projections are not able to model all types of dynamic situations. **Situation-dependent projections** are able to express, e.g., soft time windows, rush hours, capacity-dependent travel time, and overtime fees. Unfortunately, we also noted that the computation with these types of projections is not as efficient as with transitional projections. Most notably, the sequential search strategy cannot be applied in situations where the effect of each partial move cannot be evaluated independently. This is clearly the case with situation-dependent projections.

As we still need to express soft time windows, rush hours, and overtime fees, we currently employ a simple mechanism for computing with these types of problems. In these cases, we evaluate the whole changed route from the point of the change onwards. It is an open question whether there exists some other relevant intermediate approaches which are not transitional, but can be computed with less burden than traversing the whole route.

As the situation-dependent projections increase the computation beyond constant time, it is plausible that a *multi-staged approach* to the objective and feasibility evaluation would be beneficial. Such an approach would check, in constant time, the resources that can be evaluated this way, and subsequently, only if necessary, check the remaining resources (including profit). Furthermore, if the resource evaluation is especially complicated, an approximation could be used to discard those moves that are not likely to result in a feasible improving move. Ideally, the system would automatically recognize from the properties of the model which approach should be used. These topics are left for future research.

From the analysis given in this section, we conclude that reasonably efficient search techniques are applicable in the context of the developed metamodel. It is evident that the developed approach does induce some overhead through the corresponding implementation, but the brief theoretical analysis of complexity

indicates a constant overhead that should be reasonable in most VRP cases.

6.2 Preliminary Results

In this section, we examine the operation of the developed framework in practice. We first discuss the current state of the implementation, and subsequently aim to demonstrate that the implementation of the metamodel is, indeed, possible by solving both a set of commonly used standard benchmark instances and more complex real-life problems. As the evaluation of real-life instances is problematic, we limit the (brief) analysis to the benchmark instances.

6.2.1 State of Implementation

The implementation of the metamodel and the core parts of the product line was done using C#, and as it stands now, the total source code line count is in the order of 25 000. The total count which accounts for the optional modules of the product line and the currently implemented application-layer realizations, is in the order of 100 000 lines[6].

As the development of optimization methods was not the aim of this work, we do not present exhaustive testing in the context of the metamodel. However, some preliminary results from standard benchmarks will be presented as a way to verify the implementability of the approach. That said, we note that the first implementation of the metamodel is, indeed, partial. More specifically, the capabilities have not yet been included into the constant time feasibility check, and objective evaluation is performed as if the objective functions are situation-dependent in all cases. The feasibility checks are thus performed first on the local environment and only feasible moves are evaluated for the objective. Fortunately, the effects of this expensive objective evaluation are smaller in PDP instances as there are less feasible moves than in simple VRPs. Nevertheless, these shortcomings in the current implementation affect negatively the performance of the system.

Note that in the search techniques used within the context of the developed metamodel, there is, in fact, *no need to store resource values for each capability label* defined in the problem. Only those capabilities that have bounds defined on them can be violated in terms of resources, and, although we have illustrated a full Cartesian product on these values, not all values have to be stored and computed in practice. Effectively, this leads to a sparse data structure in the implementation. Now, the sparse data structure is often less efficient as a simple array lookup, which is possible in most VRP cases, and the currently implemented structure uses no sophisticated memory management for allocation and reallocation of these data structures. This causes a slight (constant) overhead.

[6] Including both production and test code

6.2.2 Benchmark Instances

We first tested the system with a set of benchmark instances from the literature. The instances were pickup and delivery problems with time windows.

As we go through the different models used in testing the system, we observe common elements. One such element is the time window, and to express the cases conveniently, we define the steps needed for generating them. Time windows, as mentioned, are generated by creating a capability for the window with a value of zero at the given activity and setting a parent for that capability. The capability is then constrained according to a resource and bounds. The algorithm is defined in Algorithm 14. The full model transformation used for

Algorithm 14 Generating time window.

```
Require: activity, parent, resource, window
  tw ← CreateCapability()
  AddCapability( activity, tw, 0 )
  SetParent( tw, parent )
  AddBounds( tw, resource, window.Start, window.End )
```

generating the problem instances is given in Algorithm 15. We begin the model transformation by generating a resource for time as we deal with a time windowed problem. Vehicle capacity and clock are needed as capabilities and both of these are given for all the vehicles in the problem. The capacity of the corresponding vehicle is used as the value of the capability for capacity. Each task is generated in two parts and they are given both capacities and time windows. A resource function is needed for accumulating the time used during the traversal, and a profit function employing the distance matrix is finally added.

The benchmark problems were solved with the developed system, and despite the apparent deficiencies in the current implementation, we obtained somewhat reasonable results using a fairly simple metaheuristic search procedure. The solution methodology used was a variable neighborhood descent with a multi-start cheapest-insertion type construction heuristic discussed in Chapter 3. The construction heuristic varied its parameter which controls the size of the neighborhood the method considers when building routes. The construction heuristic attempted to build routes by first considering activities near each other. The best of the initial solutions were used in the subsequent intensification and diversification phases iteratively. The intensification components of the VND used were

- 2-opt,
- intra-route relocate modified for PDP type of problems,
- inter-route relocate modified for PDP type of problems,
- Or-opt with additional option for segment reversion included, and
- 2-opt*.

Algorithm 15 Generating PDP benchmark instance.

```
time ← CreateResource()
capacity ← CreateCapability()
clock ← CreateCapability()
for all Vehicle vehicle in plan do
  start ← CreateActivity()
  end ← CreateActivity()
  AddCapability( start, clock, +1 )
  AddCapability( end, clock, −1 )
  AddCapability( start, capacity, +vehicle.Capacity )
  AddCapability( end, capacity, −vehicle.Capacity )
end for
for all Task task in plan do
  pdp ← CreateCapability()
  pickup ← CreateActivity()
  AddCapability( pickup, capacity, −task.Pickup.Capacity )
  AddCapability( pickup, pdp, +1 )
  CreateTimeWindow( pickup, clock, time, task.Pickup.TimeWindow )
  delivery ← CreateActivity()
  AddCapability( delivery, capacity, +task.Delivery.Capacity )
  AddCapability( delivery, pdp, −1 )
  CreateTimeWindow( delivery, clock, time, task.Delivery.TimeWindow )
end for
AddTransitionalResourceFunction( clock, time, ( a, b ) → travelTime[a,b] )
AddTransitionalProfitFunction( capacity, ( a, b ) → M−distance[a,b] )
```

The diversification components were three types of route destruction operations that considered different types of routes for partial or complete destruction and merging into other routes.

We used a commonly used PDP benchmark set of 400 customers by Li and Lim [135], which was available at the Sintef web-page[7] along with the best known results reported. The set contains 60 instances of varying parameters in both the strictness of constraints (time windows, capacities) and the physical distribution of customers. The instances prefixed with "lc" have a clustered distribution of customers, "lr" a uniform random distribution, and "lrc" mixed distribution. The subsequent number indicates whether the constraints are tight ("1") or loose ("2"). The second number indicates the size of the problem (here 4), and the last the number of the instance from 1 to 10.

The described solution methodology was employed and allowed to run up to 600 seconds. The instances were solved on a desktop PC with Intel Core2 Duo CPU E8400 @ 3.00GHz, and 3.21 GB of 2.99 GHz RAM, on Windows XP Professional SP3. The implementation was compiled on .NET 3.5 SP1 with Microsoft Visual Studio 2008 Express SP1. Two instances of the solver were run in parallel, that is, one on both cores. The optimization was executed to the time limit, after which the best found solution was returned along with the elapsed time at the finding of the best found solution.

The best reported solutions are evaluated using a lexicographic objective, that is, first, the number of vehicles is considered, and second, the total traveled distance. This can be modeled by setting a large enough travel cost from all route start activities, except for the direct traversal to the route end. The results were that when measured using the sum over all the 60 instances and compared to the best solutions reported, we obtained solutions approximately 13.04% worse in the number of vehicles and 17.80% in the traveled distance. Individually, on average, each case was 25.42% worse in the number of vehicles than the best known, and 15.00% in the traveled distance. From this we can see that if the orders can be served with a small number of vehicles in a given instance, the relative deviation from the best known is larger. This could be explained by the difficulty of removing long routes with the current set of algorithms. The standard deviation was 19.5 percentage units for the number of vehicles and 12.0 for the distance traveled. The best found solutions were obtained, on average, in approximately 6 minutes. Detailed results are given in Appendix 4. In addition, we examined briefly, in a separate measurement, the stability of the methodology when randomization elements were used in the diversification components. In this case, we solved all the 60 instances ten times and examined the standard deviation on both the number of vehicles and the length of the routes. Both values were, on average, in the order of 1.2% between different runs on the same instance, which would suggest that the method does perform consistently.

When compared to the simplicity of the metaheuristic used, the results are reasonable in quality, but perhaps surprisingly weak in robustness. It is difficult to evaluate the effect of the metamodel on the running time of the algorithms, but

[7] http://www.sintef.no/projectweb/top

the main contribution to the relative slowness of the execution (when comparing the running time to the obtained results) is most likely due to the somewhat partial implementation of potential slack projections for capabilities and simplistic memory management in the implementation. The inability to find better solutions than the given results (notice that there were only few instances that reported best known solutions close to the time limit) is most likely due to the simplicity of the solution algorithm: e.g., no non-improving moves were allowed in the intensification phase, which affects the effectiveness of the local searches negatively when compared to the state-of-the-art metaheuristics typically employed.

6.2.3 Real-life Instances

As we have argued, the modeling framework supports expressing different types of problems relatively flexibly. In this section, we examine a few real-life routing problems that were modeled and, to some degree, solved with the developed system. The problems originate from practical needs of logistic operators.

The first task was to **distribute free papers with different types of trucks** with a different capacity, costs, and speed. The case was modeled as a routing problem with a heterogeneous fleet and time windows. The model transformation for this case is given in Algorithm 16. The base case is similar to the VRP

Algorithm 16 Generating HVRPTW instance.

```
capacity ← CreateResource()
time ← CreateResource()
clock ← CreateCapability()
for all Vehicle vehicle in plan do
  v ← CreateCapability()
  start ← CreateActivity()
  end ← CreateActivity()
  AddCapability( start, v, +1 )
  AddCapability( end, v, −1 )
  AddCapability( start, clock, +1 )
  AddCapability( end, clock, −1 )
  AddBounds( v, capacity, 0, vehicle.Capacity )
  CreateTimeWindow( start, clock, time, vehicle.StartTimeWindow )
  CreateTimeWindow( end, clock, time, vehicle.EndTimeWindow )
  AddTransitionalResourceFunction( v, time,( a, b ) → travelTime[v,a,b] )
  AddTransitionalProfitFunction( v, ( a, b ) → M−cost[v,a,b] )
end for
for all Task task in plan do
  delivery ← CreateActivity()
  CreateTimeWindow( delivery, clock, time, task.Delivery.TimeWindow )
end for
AddTransitionalResourceFunction( clock, capacity, ( a, b ) → capacity[b] )
```

transformation described in the preceding chapter. We begin the model transformation by generating resources for time and capacity, and a capability for clock. Each actor is given a unique capability, bounds on the accumulated load, time windows, and functions for time and profit which are bound to the unique capability of the vehicle. Moreover, each task is given a time window, and a resource function describing the load accumulation is added to the model.

The second task was to design **newspaper delivery in bundles** for subsequent distribution by individual deliverers. The case was not constrained by the capacity of individual vehicles but the schedule limits on the print and on the routes of the deliverers. The bundles were obtained from the print at a certain point in time, and each deliverer should have completed her route by a given point in time. The distribution was divided into two phases where the bundles were first transported to deliverers, and as they began their route, the second phase was started in the bundle delivery. Each individual distributor had a second location to where the second bundle was delivered, and the delay between these two deliveries were not supposed to exceed the length of the route of the individual deliverer. The case was modeled as a multi-trip routing problem with time windows and dependency between the tasks on the two trips. The model transformation for this case is given in Algorithm 17. In contrast to the cases discussed so far, in this case we create three activities for each actor: start, end, and a middle task for dividing the route into two trips. The activities of the actors are constrained using two capabilities dictating the order in which they have to be performed. Each task is then given two activities which are required to be performed in a correct relation to the middle activity using capability requirements. The dependency between the two phases of the task is established using a capability "phaseLink" which is constrained in time by the maximum delay between the two phases of the task. Other elements of the transformation are similar to those illustrated earlier.

The third instance was a case of **oil product transportation with tankers**. The case involved pickup and delivery of products between harbors, and the operations are limited by both time constraints and multidimensional capacity of the tankers (both weight and volume). In addition, incompatibilities between ships and harbors may be present. The case also involved service events but these have been omitted for clarity. The objective function included travel costs, soft time windows, costs on contamination, costs for changing the plan near the start of the planning horizon, and relatively complex computation of costs and profits during harbor visits. The detailed structure of the objective function itself is not visible in the model transformation, but the transformation for generating the constraints of the case is defined in Algorithm 18. The distinct feature of this case is the situation-dependent profit function which is used to compute the relatively complex cost structure of the case. The transformation combines the base PDP model with a heterogeneous fleet. Each vehicle is given a unique capability which is then used in the situation-dependent profit function for identifying the vehicle in question. The capacities in this case are given in two dimensions, for both vehicles and tasks. A set of incompatibilities between the vehicles and the tasks

Algorithm 17 Generating two-phase distribution problem instance.

```
time ← CreateResource()
clock ← CreateCapability()
firstPhase ← CreateCapability()
secondPhase ← CreateCapability()
for all Vehicle vehicle in plan do
  start ← CreateActivity()
  middle ← CreateActivity()
  end ← CreateActivity()
  CreateActor( start, end )
  AddCapability( start, firstPhase, +1 )
  AddCapability( middle, firstPhase, −1 )
  AddCapability( middle, secondPhase, +1 )
  AddCapability( end, secondPhase, −1 )
  CreateTimeWindow( start, clock, time, vehicle.StartTimeWindow )
  CreateTimeWindow( middle, clock, time, vehicle.MiddleTimeWindow )
end for
for all Task firstTask in plan.FirstPhaseTasks do
  secondTask ← firstTask.SecondPhase
  first ← CreateActivity()
  second ← CreateActivity()
  phaseLink ← CreateCapability()
  AddCapability( first, phaseLink, +1 )
  AddCapability( second, phaseLink, −1 )
  AddCapabilityRequirement( first, firstPhase )
  AddCapabilityRequirement( second, secondPhase )
  CreateTimeWindow( first, clock, time, firstTask.TimeWindow )
  AddBounds( phaseLink, time, 0, secondTask.MaxDelay )
end for
AddTransitionalResourceFunction( clock, time, ( a, b ) → travelTime[a,b] )
AddTransitionalProfitFunction( clock, ( a, b ) → M−distance[a,b] )
```

Algorithm 18 Generating ship routing instance.

```
time ← CreateResource()
weight ← CreateCapability()
volume ← CreateCapability()
clock ← CreateCapability()
for all Vehicle vehicle in plan do
  v ← CreateCapability()
  capabilities[vehicle] ← v
  start ← CreateActivity()
  end ← CreateActivity()
  AddCapability( start, v, +1 )
  AddCapability( start, clock, +1 )
  AddCapability( start, weight, +vehicle.MaxWeight )
  AddCapability( start, volume, +vehicle.Volume )
  AddCapability( end, v, −1 )
  AddCapability( end, clock, −1 )
  AddCapability( end, weight, −vehicle.MaxWeight )
  AddCapability( end, volume, −vehicle.Volume )
  CreateTimeWindow( start, clock, time, vehicle.StartTimeWindow )
  CreateTimeWindow( end, clock, time, vehicle.EndTimeWindow )
end for
for all Task task in plan do
  pdp ← CreateCapability()
  pickup ← CreateActivity()
  AddCapability( pickup, weight, −task.Pickup.Weight )
  AddCapability( pickup, volume, −task.Pickup.Volume )
  AddCapability( pickup, pdp, +1 )
  CreateTimeWindow( pickup, clock, time, task.Pickup.TimeWindow )
  delivery ← CreateActivity()
  AddCapability( delivery, weight, +task.Delivery.Weight )
  AddCapability( delivery, volume, +task.Delivery.Volume )
  AddCapability( delivery, pdp, −1 )
  CreateTimeWindow( delivery, clock, time, task.Delivery.TimeWindow )
  for all Vehicle vehicle in task.IncompatibleVehicles do
    AddCapabilityProhibition( task.Pickup, capabilities[vehicle] )
  end for
end for
AddTransitionalResourceFunction( clock, time, ( a, b ) → travelTime[a,b] )
AddSituationDependentProfitFunction( clock, ( s, a, b ) →GetProfit( s, a, b ) )
```

is also defined using capability prohibition.

The first case illustrates generation of a somewhat typical routing problem, whereas the second case presents a slightly more unconventional structure. The third case illustrates the simplicity of employing multiple capacities and compatibility rules, but as noted, the objective function structures are quite implicit in the metamodel. The usage of the stacked resource extension instead of the partial resource extension with situation-dependent functions, if this can be done in the given case, mitigates this somewhat. Overall, we might conclude from the presented cases that fairly large differences in cases can be implemented on the metamodel level with relatively small changes.

These cases illustrate the heterogeneity of the problem domain, and as mentioned, the different characteristics of the problems most likely affect the suitable solution methodology. It would be interesting to analyze the modeling constructs, e.g., the effect on the constraints such as the link between the phases in the second example, but this work is left for future research due to scope limitations.

6.3 Implications of Model Characteristics

In the preceding sections, we examined in detail the algorithmic manipulation of the *Ansatz*. However, we did not answer the question of *selecting* the set of appropriate algorithms. Now, as there is considerable variation within the models described by the routing metamodel, it is conceivable that any single solution approach cannot address the heterogeneity of the domain sufficiently. In this section, we discuss the apparent implications of the characteristics of the models to the solution methodology. We abstain from studying the computational effects of model characteristics, but note that this is an interesting topic for further research.

6.3.1 Effects of Problem Structure

The structure of the optimization model affects the solution algorithms in three major ways. The base problem variant changes the nature of the search neighborhoods; capability and resource constraints alter especially the effectiveness of the algorithms by modifying the structure of the feasible region; and grouping constraints introduce an additional dimension to the search space, which complicates the search.

From the algorithm perspective, the **problem variant** refers to the natural base operation required from the algorithm. We can identify a hierarchy of decision variables according to the structure of the search neighborhoods defined. There are three major structures we observe from the defined metamodel. These are

- single-activity structures (e.g., the VRP),
- two-activity structures (e.g., the PDP), and

- multi-activity structures (e.g., the GPDP).

Single-activity structures correspond to search neighborhoods where the primary operation concerns moving a single activity at a time. We examined an example of such an optimization model in the preceding chapter when we generated a VRP model. Two-activity structures, in contrast, correspond to neighborhoods where the primary operation is to move two activities simultaneously. A natural example is the PDP type, where each pickup and delivery activity has to move, for example, from a route to another at the same time. Finally, multi-activity structures define more than two of these activities, which complicates the search further. These structures arise when the problem is of the GPDP type where, e.g., a set of pickups has to be performed for each delivery.

Note that while these structure types clearly affect node-neighborhoods, they also have implications on search operators defined on arc-neighborhoods. If, for instance, a segment is moved from one route to another, to make a valid move, one has to ensure that in two-activity structured problem each pair of activities resides either on the moved segment or on the remaining route.

Perhaps the most straightforward elements to address are the **capability and resource constraints**. These constructs do not alter the search space itself, only the feasible region. As we will observe in the subsequent chapter, the definition of the feasible region is transparent to the solution methodology. These constraints may also form a hierarchy, but its structure is less clear, and we do not analyze these structures in detail.

There is a hierarchy of **grouping constraints**, however, but as we restricted the current metamodel to *simple groupings*, this is the sole type of grouping constraints we examine for now. Other grouping constraints would introduce less strict bounds on the size of the group. In a sense, grouping constraints have the potential to introduce new decisions to the problem. These decision extensions alter the structure of the model perhaps most radically of the constraints. As we have discussed, decision extensions include decisions on loading compartments, fleet selection, driver legislation modeling, and equipment selection. The modeling of decision extensions is based in many cases — in addition to activity groups — on capabilities. These capabilities are, in fact, responsible for the *dynamic* nature of these mapping constraints: one break defines the need to take another, loading to one compartment prevents another pickup from being performed and so on.

Implementation of search operators has to take into consideration especially the structure type of the problem and the possible mapping constraints introduced. This remains true, despite the fact that the specifics of the optimization model have been abstracted behind the metamodel.

The problem with different structure types is the fact that algorithms need to perform different types of operations depending on the problem structure. There are essentially three strategies for addressing this issue. We may

- let the algorithms query their assumptions (or more specifically, form their assumption by querying) and adjust their operation accordingly,

- let the coordinating algorithms (such as metaheuristics) utilize their knowledge of the problem and select the appropriate algorithms (such as local search operators), or
- utilize some combination of the two.

Fortunately, the system structure as a product line is flexible in the sense that we may employ the third option as needed. Where there are requirements for reuse, we may let the algorithms adjust their operation as necessary, and where we emphasize the case-specific properties, we may provide customized algorithms. Thus, whichever approach is more beneficial at each case can be used. In this work, we do not provide a definitive answer for *when* each alternative should be used, but we advise erring on the side of reusability first, and customizing later if needed.

The grouping constraints pose a problem similar to the structure types. On one hand, if the algorithms are aware of the groups, they may exploit their existence in modifying the search space as needed, but they need to query their presence, which complicates the implementation and decreases their performance. On the other hand, if algorithms are unaware of the grouping structures, they are provided in a transparent and reusable way, but the algorithms may fail to perform the search operations needed for exploring the feasible region. For instance, inserting an activity to a route may require simultaneous substitution of one activity on the same route with its alternative to provide more space on the compartment. To the best of our knowledge, this operation is not yet part of any established local search algorithm. We also note that the structure of the problem can be deduced from the capabilities, groups, and resources. Deduction can be performed, e.g, from the fact that some capabilities do not sum to zero over all activities, which indicates that the problem must contain mutually exclusive activities with respect to that capability. However, in order to "understand" the structure (the dynamics) of the grouping, for example, it may be necessary to traverse the dependencies between the different sets of alternative activities through the capabilities. This is a subject for further research and is part of the effort needed in constructing generic local search operators in rich problems. We address this issue in the section where we discuss topics for further research and mention problem structure analysis.

Based on this discussion, we next inspect the suitability of different local search operators in complex routing problem variants and attempt to identify different levels of adequacy for these operators.

6.3.2 Adequacy of Local Search Operators

We are interested in whether a given algorithm is applicable to a given situation, that is, a given optimization model, the case instance, and its characteristics. To analyze the cases, we provide here a simple taxonomy based on the suitability of the local search operators.

The suitability of a search operator is directly dependent on the problem

structure, and, from the discussion in the preceding section, we see perhaps two types of structures emerging:

- those typically easy to include transparently, such as compatibilities and multiple dimensions on resources; and
- those requiring more sophistication in solution methods: for example, additional decisions, such as loading and equipment selection, and complex dependencies, such as tasks of multiple parts.

These differences are in some sense *syntactic* as they require the solution methods to understand the "syntax" in which the problem is described. In addition, although the structural elements that, for example, define breaks, optional drop-off points, and multiple compartments are the same (simple grouping constraints and capabilities), it cannot be assumed that the methods that solve one of them solves them all equally efficiently. This is due to the fact that the constraints generated on different situations differ *semantically*. This is a natural consequence of the different kinds of decisions that have to be made in reality. Furthermore, even if the real-life decisions are the same, the specific overall *configuration* (the number of vehicles and tasks, for instance) of the given problem instance places different kinds of requirements on the solution methods. This difference is, naturally, also a feature of the existing methodology in vehicle routing research.

To summarize, we may distinguish *four types of algorithms*: those which are

- inapplicable,
- syntactically applicable,
- semantically applicable, and
- configurationally applicable

to the given problem.

Inapplicable algorithms do not understand the structure of the model or some part of it, and because of this fail to traverse the search space in a reasonable manner. In essence, activation of a model element may thus exclude a subset of local search operators from use. These algorithms may, for example, fail to provide a feasible initial solution as they are unable to deduce the dependencies between different model constructs. For example, consider a case where compartment loading decisions are essential for constructing feasible routes. In this framework, this type of problem cannot be solved by local search operators that do not consider alternative activities a part of their core search operation. The operator would most likely need to perform moves and consider alternative activities simultaneously to be able to find feasible routes.

Syntactically applicable algorithms, in contrast, understand the problem structure but may not utilize the semantic structures behind the modeling elements. The applicability from the syntactic viewpoint can be analyzed from the problem structure: the set of modeling elements used dictates the algorithms that

can be applied. Examples of syntactically applicable algorithms to the VRP include the typical relocation of a single activity, and to the PDP, the relocation of two activities. In fact, many standard search operators belong to this class of algorithms in the context of most routing variants.

Semantically applicable algorithms employ the knowledge of the meaning of the decisions, and these algorithms are tailored to solve some specific problem type. The specifics of the problem can be inferred from domain knowledge, and this information may result in hints to the solution process. The usage of additional information is potentially useful but not necessary for the solution process. Examples of semantically applicable algorithms include those, for instance, restricting their search according to pickup-delivery sequence patterns known to be beneficial; in, e.g., some ship scheduling cases, the special structure of the preferred visit sequence can be utilized in this way. Another example is an algorithm specifically designed to order tasks in compartments to reduce the overall load required during the route.

Configurationally applicable algorithms are tailored to a certain problem instance type, and can employ instance-specific knowledge, which is often numeric data computed from the characteristics of the problem instance. These algorithms are typically applied to address very specific problems. Examples of configurationally applicable algorithms include construction heuristics that employ knowledge of, e.g., the degree of clusterization in the instance.

Observe that the possibilities for reuse decrease — or the requirements for *adaptation capabilities* increase — as we move from the former types to the latter. Customization typically increases the need for semantic knowledge in the algorithms. It is an open question how to model and transfer this knowledge to the solution process. In addition, we note that the distinction between configurationally and semantically applicable algorithms is not clear; it is perhaps the degree of inference that is required to employ the characteristics of the case which separates these two classes. The meaningfulness of this distinction should be critically examined in the future.

This discussion underlines an essential feature of the system: one cannot assume that after modeling the problem within the framework and developing a necessary set of operators for finding a feasible solution, the problem is yet solvable to an acceptable degree. This is of course a property of any optimization system, but may be present here more acutely. The requirements for robustness in a generic system are perhaps more difficult to satisfy than in a case-specific system. The basic level of robustness of an optimization system is achieved by algorithms that can exploit the features of the problem. For this reason, in combinatorial optimization in general, the operators are tailored to the problem at hand. However, as the potential problem set expands, the case-specific algorithms fail to provide solutions in all cases. To address this, the system should be able to adapt to the problem instance under consideration at a given moment. This might suggest that more automation in the areas of analysis of the problem and selection of methodology could be beneficial. We have identified this as a major topic for further research, and it will be addressed in more detail in Chapter 8.

7 PRODUCT LINE OF ROUTING SYSTEMS

"What I cannot create, I do not understand."
— RICHARD FEYNMAN

In this chapter, we apply the modeling framework described in Chapter 5 into practice. A model-driven software product line incorporating the domain model, routing metamodel and model transformation engine will be described. Moreover, we demonstrate the utilization of the variation mechanisms for describing different optimization models, varying the solution methodology, altering the domain processes, and changing interface presentations. When discussing the domain model, we also address the derivation of application models and their relation to the model transformation process.

This chapter is structured as follows. In Section 7.1, we take a look at the overall structure of the software product line, describe the reference architecture of this SPL, and present the variation mechanisms used within the developed product line. In Section 7.2, we demonstrate the utilization of the central variation point within the system: we give a set of transformation examples to describe different routing variants and their extensions. Finally, we perform a heuristic qualitative analysis of architectural structure of the developed system in Section 7.3.

7.1 System Design

This section discusses the structure of the developed optimization system as a whole and addresses the design philosophy behind its structure. We will briefly restate the objectives set and the approach used, present the overall structure of the system, and discuss its architectural quality attributes in the context laid out in Chapter 4. We then proceed to discuss the detailed structure of the system and the phases required from the solution process. Within this context, we also examine the variation mechanisms available within the developed architecture.

7.1.1 Objectives and Approach

In the context of this work, the objective is to provide an optimization system which is reusable across different types of routing and scheduling cases and still provides the needed solution quality and robustness without sacrificing the performance of the system. This could lower the amortized amount of resources needed in the building of a system for each case as large parts of the system would already exist. This approach has been used in the last decade or two in the field of general software engineering, and we would like to translate this approach to optimization systems. Now, while we analyze the technical aspects of realizing a product line architecture for a vehicle routing system and provide a general rationale for software product lines in this context, consideration of building such a system from a business point of view — e.g., portfolio management, market analysis, product-life cycle management, organizational issues — is largely outside the scope of this work. That said, the technical discussion presented here is not limited to a commercial setting; in fact, the approach is most suitable to research use in many respects. This point is emphasized especially when we contemplate the applicability of the framework and the system in general.

The architecture of a software system is, among other things, an articulation of **quality goals**, or more specifically, a statement of the desired balance between conflicting attributes [14]. In Chapter 4, we defined several attributes which may have not been addressed sufficiently in the context of optimization systems. These were fidelity, accuracy, performance, scalability, and robustness of the system, which together, as we defined, constituted for a *usable* system; and applicability, simplicity, flexibility, and reusability, which largely define the *implementability* of the system. Usability and implementability then determine the cost-efficiency of providing a system that is suitable for a given problem.

Chapter 5 addressed the applicability, fidelity, and accuracy aspects of the system by introducing a more detailed, generic description of routing problems; performance, scalability, and robustness are discussed in Chapter 8 when we examine the topics for further research. In this chapter, therefore, we concentrate on the simplicity, flexibility, and reusability of the developed system.

As the implemented **model-driven approach** was analyzed, it became apparent that the tool support is not adequate for a full-fledged model-driven engineering process in this particular problem domain. Much of the modeling and model transformation framework has been built from scratch, and this cannot be described as a full realization of the model-driven engineering paradigm. More specifically, we do not utilize all the automation techniques developed for generating models and executable artifacts as described in the literature (see, e.g., [138]). Instead, we concentrate especially on how model-driven techniques can be used to assist in managing variability.

The employment of a model-driven product line architecture described in this chapter provides the research contributions to the field of software engineering. To the best of our knowledge, this is the first application of model-driven architecture to the vehicle routing domain for building a software product line.

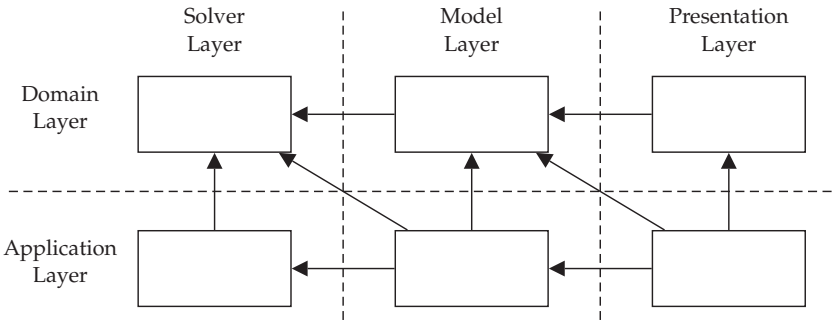


FIGURE 38 Layers within the product line architecture[1].

Moreover, this is the first work, as far as we are aware of, presenting a concrete study on a model transformation as a variation point in a software product line.

7.1.2 Overall Structure

A software product line consists of two separate layers. In this section, we describe the overall structure of both the *domain layer* and the *application layer* of the product line. This provides the basis for analyzing the introduction of variation as well as serves as an introduction to the functional modules required from optimization systems in general.

As we developed the system, we identified three distinct *functional layers* in the system, corresponding to presentation, model, and solver functionality. Now, as the product line approach divides the structure into two *structural layers*, the overall structure can be seen as a six- layer structure. This is illustrated in Figure 38. The responsibilities of each layer can be characterized as follows. The solver layer handles everything needed for solving the optimization problem: problem encoding functionality, routing metamodel, and the necessary algorithms and data structures. The model layer both controls the overall process of the system and contains the object-oriented domain representation of the problem. The domain representation offers a clean interface to the UI for manipulating the problem data by allowing adding, removing, and editing the data. The layer also includes, for instance, persistence and data connectivity functionality as well as functions for defining, tracking, and storing alternative scenarios of the given case instance. The presentation layer, on the other hand, raises the abstraction level of the domain representation and provides commonly needed services and components for realizing the user interface implementations.

The domain layer of the solver layer contains the optimization system itself, whereas on the application layer, for instance, a set of application-specific algorithms can be introduced. The domain layer of the model layer contains the

[1] The arrows in this and the subsequent figures denote dependencies between different elements. An arrow from element A to element B denotes the fact that A depends on (uses) B.

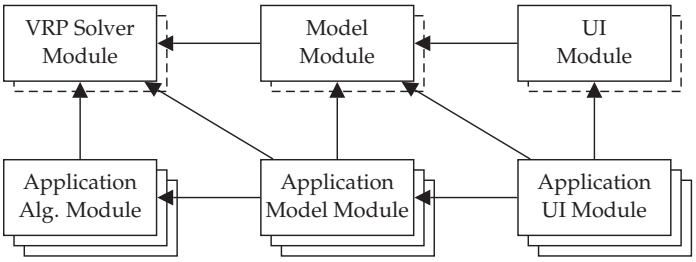


FIGURE 39 The module structure of the system.

abstract representation of the problems, such as the domain model described in Chapter 5, whereas the application layer of this layer may expand the functionality of the abstract domain model as needed by the cases. Finally, the domain layer of the presentation layer contains, for example, basic UI components developed for visualizing the data on the model layer, whereas the application layer of this layer usually contains the concrete, case-specific, user interface implementation. The variants and their combinations within the system have the potential to create relatively complex structures, and the structure on this level has been kept conceptually simple to avoid introducing unnecessary complexity.

Within this layer structure we can divide different layers into one or more *modules*. First, on the application layer, each module belongs to exactly one application. If a module is needed by more than one application derived from the product line, it is included into the domain layer as an optional core asset. For instance, the `VRP Solver Module` may be accompanied by a `SPP Solver Module` if needed by the case. We examine the optional modules in the subsequent sections when discussing variability in more detail. Note that the dependencies between modules follow those defined by the layers, that is, the modules on the presentation layer depend on those on the model layer which then depend on those on the solver layer. Furthermore, the modules on the application layer depend on those on the domain layer. The module structure is illustrated in Figure 39. In the example, there are three application level realizations. One of the applications, for example, extends the solution methodology by introducing additional case-specific algorithms on the solver layer. The domain layer contains optional modules which are not shared by all application level realizations. These modules are marked with a dashed line. These optional modules provide one variation mechanism, but in order to examine more fine-grained variation, we need to examine the structure of the system on the submodule level.

7.1.3 Submodule Structure

In order to understand the modeling and optimization process within the system, we need to examine a number of major components within each module. These

elements are here referred to as *submodules*. Each submodule is responsible for one or more major aspect of the module. For example, a transformation submodule is responsible for the model transformation in the model module, and a core algorithm submodule, the set of basic algorithms in the VRP solver module.

When we examine the three necessary modules on the domain layer, the `VRP Solver Module`, `Model Module`, and `UI Module`, along with an example realization of an application with corresponding three modules, we obtain a situation such as the one illustrated in Figure 40. In the example, the boundaries of the six modules are marked, and the figure presents the major subcomponents of those modules.

The six modules and their submodules govern the overall process of the optimization, including actions performed before and after the actual traversal of the search space. The process is first initiated from the `UI Submodule` by importing data from a data source using the `Data Connection Submodule`. The obtained data is translated into the object-oriented domain model presented in Chapter 5. This is performed in the `Model Submodule` and the `Domain Model Submodule`. Then, the transformation process is initiated through the `Control Submodule`. The `Transformation Submodule` then utilizes the `Transformation Definition Submodule` which is case-specific and holds the model transformation definition. This definition uses the model transformation presented in Chapter 5 for generating the case-specific optimization model which conforms to the developed routing metamodel. This optimization model is then fed into the `Optimization Submodule`, which, using the `Core Algorithm Submodule` and the `Algorithm Submodule`, solves the optimization problem. The solution is decoded by adjusting the instance of the domain model in the `Domain Model Submodule` according to the changes made in the optimization model instance. The problem and the solution are then visualized with the aid of the `Presentation Model Submodule` and `UI Component Submodule`.

The described process is illustrated in Figure 41. In the illustration, we first read the problem data using the application level model module, and generate the domain model from this data. In addition, we set the proper model transformation definition in place as given in the problem. In the second phase, we create the optimization model by using the transformation, and store the resulting model into the routing metamodel implementation in the solver layer module. The third phase performs the optimization where we may also employ case-specific algorithms from the application layer. The result is then translated back to the model layer. We discuss this translation process in more detail in Section 7.1.4. Finally, the results are presented to the user through UI modules.

The central elements of the system — the routing metamodel and the model transformation — can be seen in both figures. In the submodule structure, the `Transformation Definition Submodule` sends the `Transformation Submodule` data over the domain–application boundary of the product line. The model transformation definitions reside in the former, and the API defined in Chapter 5 is provided by the latter. This is visible also in the sequence diagram. In the first phase, the given transformation definition is given to the transformation engine

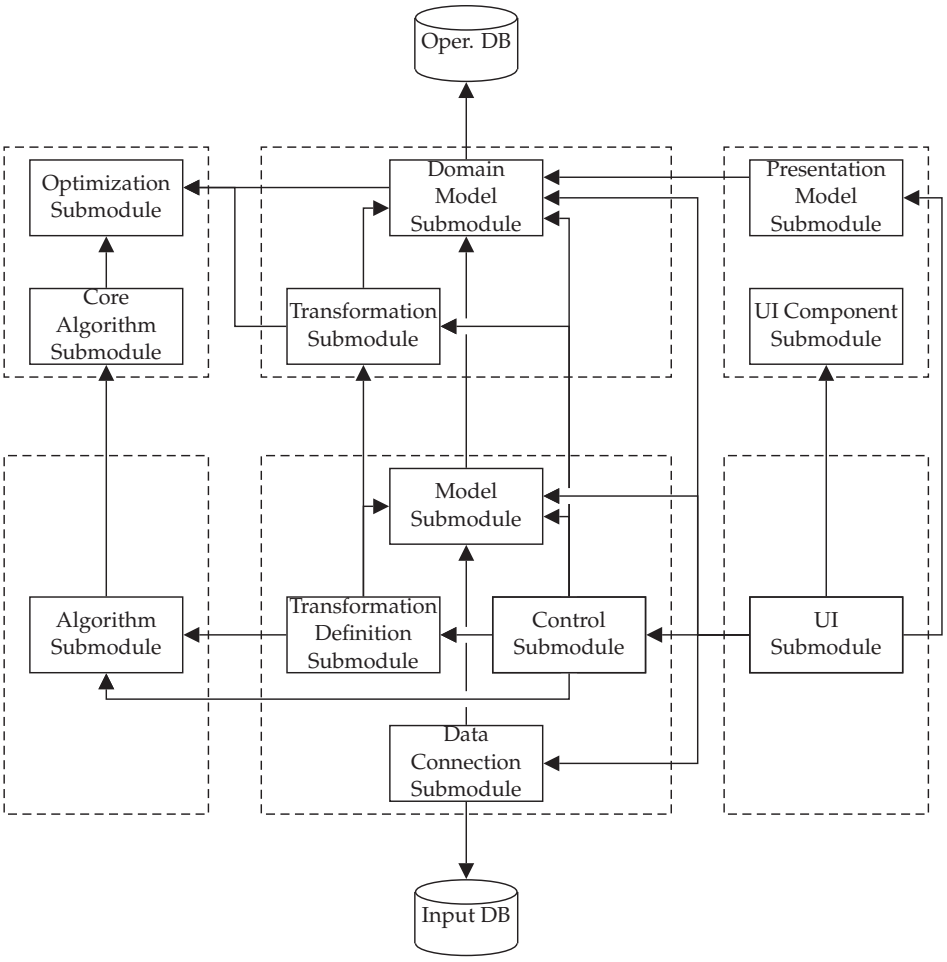


FIGURE 40 Submodule level structures of the domain layer and an example application layer realization.

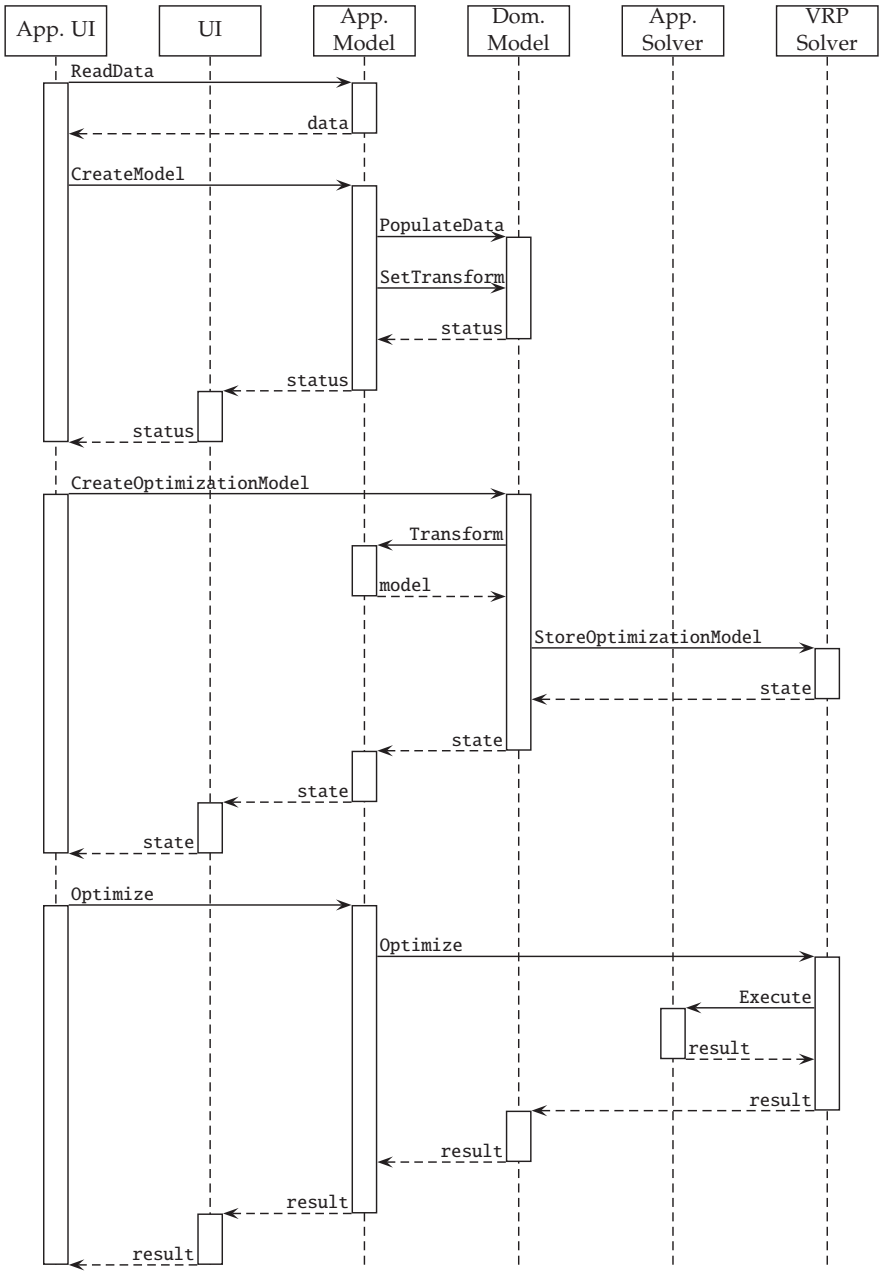


FIGURE 41 A simplified sequence of operations within the module structure.

in the Transformation Submodule by the SetTransform operation; and in the second phase, the operation Transform is performed using this definition. The transformation definition on the application layer of the model layer acts as a variation point controlling the actual optimization model used, enabling changing the model case by case. The routing metamodel is implemented in the domain layer of the solver layer, where the case-specific optimization model is stored using the StoreOptimizationModel operation in the second phase of the sequence diagram. The algorithms then operate on the metamodel in the third phase.

7.1.4 VRP Solver Module

One of the main modules of the system is the solver module. The VRP Solver Module has three major roles. Firstly, it attempts to remove the algorithms' dependency on the optimization models but in a way that the existing VRP algorithms can be used. Secondly, it attempts to provide a solution methodology that is comprehensive enough for a variety of problems. Thirdly, it attempts to act as the target for the model generation process with a transformation definition as a variation point for the needed modeling flexibility.

While the removal of **the algorithms' dependency** from the optimization models is conceptually simple, we quickly run into trouble with reuse as there is a many-to-many relationship between VRP cases and suitable solution algorithms, that is, one algorithm may be required in solving several cases, and one case is solved by a method which is a combination of several algorithms. To make the system flexibly adjustable (not to mention self-adjusting), we would have to be able to change both the underlying model and the solution methods independently of each other.

To remove the dependency, we structured the solver module as illustrated in Figure 42. The Transformation Engine from Transformation Submodule generates the actual model instance conforming to the routing metamodel defined within the Routing Metamodel element. The model generation is directed by the Transformation Definition Submodule in which the case-specific model transformation algorithm is defined. The metamodel is then accessed by the solution algorithms through a solution space explorer (SSE), which provides the route manipulation and feasibility check operations defined in Chapter 6.

Typically, the algorithm operates on the case, e.g., the VRPTW. In the developed structure, *both the individual models and the solution methods are instead dependent on the metamodel*. This enables the algorithms to operate without having to concern themselves with the particularities of the case. Our approach is in contrast to the most straightforward implementation of combinatorial optimization algorithms. In the simplest approach, the algorithms operate directly on the data structures of the optimization model. Although there is overhead from our approach, we argue, based on the analysis in the subsequent sections, that this trade-off is justifiable.

In addition to the removal of the dependency, we are able to separate the algorithms, metaheuristics, search space and its feasible region, and problem

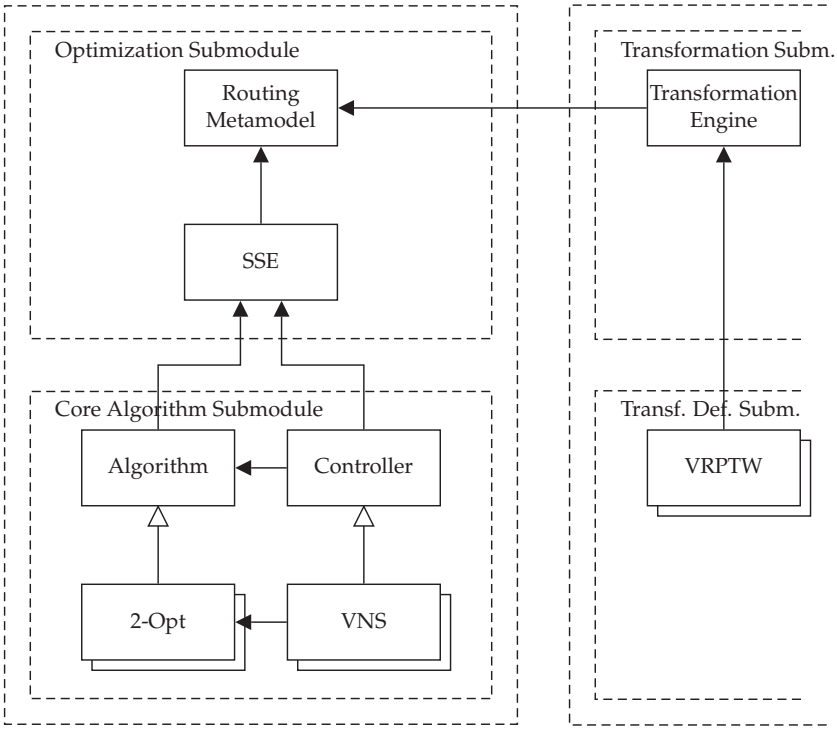


FIGURE 42 The major elements of the VRP solver module.

structure from each other. The introduction of the solution space explorer decouples the search space from the metamodel, which allows, for instance, adjusting the search space independently from the optimization model and solution algorithms.

The **solution methodology** within the module is divided into two hierarchies: algorithms and controllers. The former represent individual search operators and the latter metaheuristics or higher-level constructs. This allows both types of methods to be reused individually. The solution process is controlled from the application layer of the model layer of the system, which, as mentioned, also enables the utilization of case-specific information. After the algorithm objects have been created, they can be initialized using additional information gathered from the generated model (such as specific labels of different types, e.g., capacity) which can then be used within the algorithms during execution as needed.

The **model generation process** follows the one discussed in Chapter 5: the `Transformation Engine` generates the activities, resources, bounds, and so on. The `Transformation Submodule` also performs the decoding of the solution and offers mapping functionality from the generated model elements of the routing metamodel to the elements of the domain model. After the completion of the optimization, each activity, for example, is traced back into its corresponding domain model instance and the properties of this instance are adjusted according to the status of that activity.

7.1.5 Variation

Using the given system structure and process description, we may now discuss variability within the system. In this section, we list the relevant variation points, discuss some of the variants, and, for illustration, present a concrete example of varying one of the processes in the system.

As discussed in Chapter 4, a variation point is a representation of a variability subject — what varies — within domain artifacts enriched by contextual information, and a *variant* is used to denote a representation of a variability object — how it varies — within the domain and application artifacts.

One can identify at least seven major areas in which variation occurs within a routing system. These are

- the optimization model,
- the solution methodology,
- the domain model,
- the presentation model,
- data connections,
- dispatching, reporting, and tracking functionality, and
- the task generation process.

A number of minor areas of variation also exist within the system, especially on the presentation layer. We concentrate here on the model and solver layers as they are more relevant to the contributions of this work.

One of the major variation points is the **optimization model** realized in the routing metamodel within the `Optimization Submodule`. This variation is controlled indirectly through the model transformation definition residing in the `Transformation Submodule`, and variants of this variation point are the concrete transformation definitions given in the application-specific `Transformation Definition Submodule`. The variation is realized through object-oriented techniques, more specifically, a combination of inheritance and delegation.

Solution methodology refers to the variation point defining the algorithms and metaheuristics utilized within the system. The variation point resides in the `Optimization Submodule` which employs this methodology. There are two complementary techniques through which the variants of this VP can be defined: the `Control Submodule` may select and adjust the existing algorithms and their parameters, and the `Algorithm Submodule` may extend the available methodology in the `Core Algorithm Submodule` by introducing a case-specific algorithm or redefining functionality on the existing algorithms.

As we will observe in Section 7.2, the **domain model** presented in Chapter 5 does not contain all the case-specific classes and properties employed when expressing different model variants. We can consider the generic domain model in the `Domain Model Submodule` as a variation point, and we may indeed introduce application-specific variants of the domain model by selecting or introducing completely new child classes in the `Model Submodule`. Note that this refined domain model is visible to the case-specific model transformation, as seen in Section 7.1.3, and thus the transformation is able to employ the refined objects as needed. For example, the abstract type of vehicle can be extended into a concrete type of ship or a truck, depending on the case.

As with the domain model, the **presentation model** in the `Presentation Model Submodule` can be subject to variation case by case. The mechanism is also similar: the variants are selected and defined by inheriting the base classes and selecting existing child components in the `UI Submodule`.

Data connections for import and export are dependent on the external data definitions and the type of the storage. The abstract definition of the needed data connections resides in the `Domain Model Submodule` as it defines the data sources needed for the system: resources, tasks, geographical information, etc. There are optional variants for handling map data, for example, but some of the data sources are needed; the source of tasks and their properties, for instance, cannot be left undefined. The variants are defined primarily in the application-specific `Data Connection Submodule` as sources of data are often heterogeneous.

The `Domain Model Submodule` defines a set of optional **dispatching, reporting, and tracking functions**, and their variants such as the real-time tracking of vehicles, sending orders directly to portable devices on the vehicles, printing driving instructions, and providing or updating order or route data while the vehicle is on a route.

The **task generation process** is a composite of several variation points: e.g., creating locations, geocoding, and computing shortest paths. These VPs are defined in the `Domain Model Submodule`. Basic variants reside in the optional modules of the domain layer, but new variants can also be defined in parts of the process. We conclude this section by providing an illustrative description of the process in two application-level realizations. The example also illustrates the usage of different variants in the context of routing systems, and perhaps highlights the suitability of the product line approach to routing systems in general.

The abstract task generation process can be described as follows. After the data has been read and the domain instances created, we need to *create the location objects* representing the locations of each task, vehicle, driver, and piece of equipment. The locations are typically given as a string of characters, which we need to *parse* and perhaps reference against an existing address database as there may be errors in the data. The system then performs a *geocoding* process, in which the addresses are given relation to each other, either by identifying a location on a set of locations or connecting them as new nodes on a network. Using this data, we need to *compute the shortest distances* between the locations, and finally *hook up a resource function* for this matrix of distances to the transformation definition of the problem. The abstract process is the same for every case, and is therefore provided by the domain layer of the product line. The variations of this process can be illustrated by examining and comparing two canonical application layer realizations: one for ship scheduling and one for routing on a road network.

In a ship scheduling case, we do not always route on a complex network structure. The distances between all pairs of relevant harbors can often be stored into a database, and in the following example, we illustrate the usage of the application layer on the task generation process in this type of situation. This case is depicted in Figure 43. In this example, the task generation process is completed as follows. We first create the location objects, but the parsing of the harbor does not need to contain an actual operation. The geocoding is done by mapping the input data into domain objects by a simple identification, and solving the shortest path problem is a matter of table lookup for each pair of harbors. This table is provided by the application layer as a part of the input to the system. Finally, we construct a distance matrix based on the results and create an appropriate resource function to be set to the optimization model.

When routing on a road network, the problem definition is often impartial. For instance, routes between all possible locations can seldom be given as a part of the problem definition. Tasks and depots have addresses, but optimizing the routes requires knowledge of the distances between these locations. Some system has to offer services for constructing the data required by the optimization. In the following example, two optional modules are employed by the application layer of the product line for this purpose. The `SPP Solver Module` contains services for computing the shortest paths in the networks, and the `Road Model Module`, for instance, services for address correction, geocoding on the road network, and modeling the road traversal by extending the domain objects with road network specific properties. The abstract process is realized in this situation as illustrated

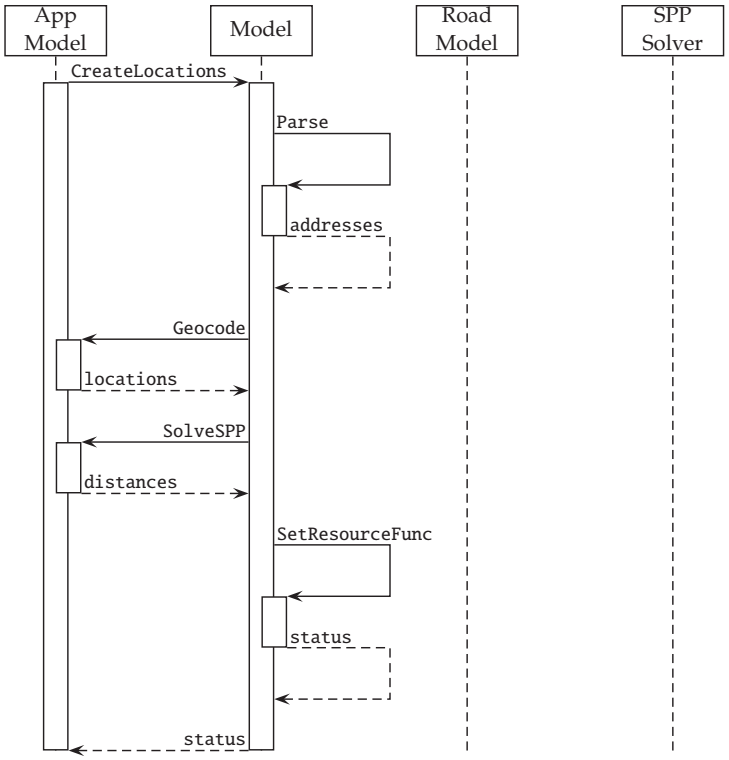


FIGURE 43 A simplified sequence of operations within the `PopulateData` operation while providing geocoding and the shortest paths on the application layer.

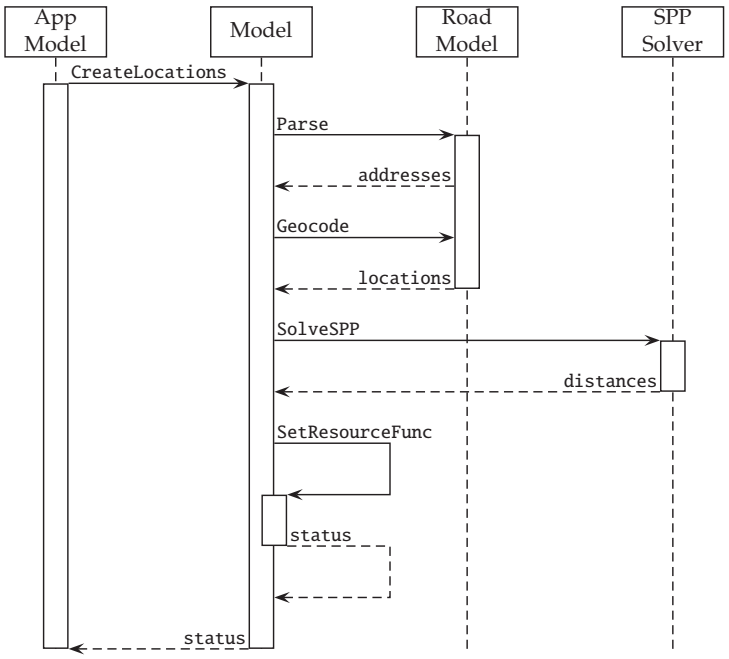


FIGURE 44 A simplified sequence of operations within the `PopulateData` operation while employing optional road modules.

in Figure 44. The task generation process is controlled by the `Application Model` module, which hooks the services of the `Road Model` and `SPP Solver` to the abstract process before the process is initiated. The task generation process is now completed as follows. First, we create the location objects and then parse the addresses of the locations and reference them against the address database. Subsequently, a geocoding process is performed by connecting the addresses as new nodes on the road network. Using this data, we then find the shortest paths between each pair of locations by utilizing the shortest path solver provided by the `SPP Solver`. The rest of the process is completed as in the ship scheduling case. The function may also be more complex than a simple distance matrix. For instance, another optional module might be used to provide data for time-dependent travel, that is, such that models the effects of congestion. Also these variants can relatively easily be implemented within the illustrated product line.

The previous example covers the practical routing in a real-life road network, but in scientific benchmarks, this aspect is often omitted. In benchmark cases, the process is considerably simpler as the distances are typically computed by calculating the Euclidean distance between each location. This alternative can also be defined in terms of the abstract process described, but we omit this variant as its process is relatively simple.

This section discussed the alternatives to variability definition including, e.g., optional modules. These variation points provide flexibility to the system

functionality. Two of the most relevant variation points within the system are the optimization model and solution methodology. The next section examines in detail the definition of variants for the variation point of the optimization model.

7.2 Model Variation

This section demonstrates in detail the usage of the API given in Chapter 5 by presenting transformation definitions for a relevant subset of problem variants and their extensions. It will provide insight into how the model elements can be combined into structures within real-life routing problems, and it serves as a base for evaluating the expressiveness of the framework.

As we give the transformation definitions, *we assume a set of properties from the domain objects* as needed by the case. These have not been formally defined as a part of the domain model, but as the extensions to the domain model are also variants of the system, we may vary their details as needed. Listing the case-specific properties explicitly would be tedious, and we discuss them here only as a part of the transformation definitions. In addition, the given transformation algorithms *do not present full transformations*; instead, they assume as their base the PDP model given in Chapter 5. A subset of these definitions can also be adapted into a simple VRP setting with minor changes, but these are not discussed in detail.

7.2.1 Elementary Constraints

Perhaps the simplest and most common constraints, some of which have already been discussed within the base examples given, are those of capabilities. These include simple precedence constraints and vehicle capacities. Among the mapping constraints, however, maybe the simplest constraint is one defining two or more alternative activities. We begin this overview to modeling of routing problem extensions by examining some of these constructs.

Capacity constraints have been discussed throughout the preceding chapters, and as demonstrated in the real-life case examples, we may utilize multiple dimensions in a relatively straightforward manner: we may freely add additional resources and capabilities. In the PDP case, we can create the required capabilities as illustrated in Algorithm 19. The case illustrates the usage of both weight and volume of the vehicle.

Precedence constraints can be used to dictate ordering which often occurs in feasible solutions. We have already seen precedence in PDP cases where the pickups must occur before the corresponding deliveries, but more elaborate constructs are possible. If, for example, we wish to impose a constraint stating that the task of activities 1 and 2 must be completed before the task of activities 5 and 6 as in Figure 45, we may add an additional capability for stating this. Here the capability “a” now imposes an ordering for the delivery of the first task and the

Algorithm 19 Generating multiple capacities on PDP model instance.

```
weightCapacity ← CreateCapability()
volumeCapacity ← CreateCapability()
for all Vehicle vehicle in plan do
  AddCapability( start, weightCapacity, +vehicle.WeightCapacity )
  AddCapability( start, volumeCapacity, +vehicle.VolumeCapacity )
  AddCapability( end, weightCapacity, −vehicle.WeightCapacity )
  AddCapability( end, volumeCapacity, −vehicle.VolumeCapacity )
end for
for all Task task in plan do
  AddCapability( pickup, weightCapacity, −task.Pickup.Weight )
  AddCapability( pickup, volumeCapacity, −task.Pickup.Volume )
  AddCapability( delivery, weightCapacity, +task.Delivery.Weight )
  AddCapability( delivery, volumeCapacity, +task.Delivery.Volume )
end for
```

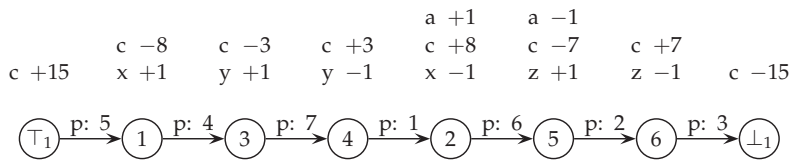


FIGURE 45 Modeling precedence constraints within a PDP instance.

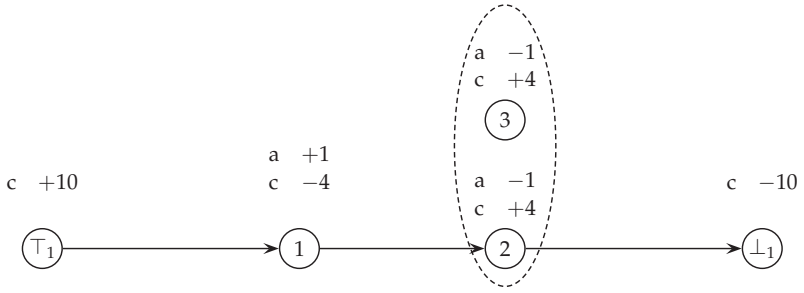


FIGURE 46 Modeling two alternative deliveries within a PDP instance.

pickup of the second. In conjunction with the other capabilities, this ensures that the tasks are completed in a proper order. Note that the same technique can be applied to multiple activities, that is, any number of activities may be required to be completed before a given activity. Furthermore, it is possible that tasks consist of more than two distinct activities, for example, two pickups and one delivery. In this case, we can generate the capabilities as "a $+1$", "a $+1$", "a -2" for the pickups and the delivery, respectively.

Alternative activities may occur when we need to decide, for example, between two alternative delivery locations. Sometimes, especially in school bus routing and scheduling, the pickup location may be fixed but the delivery location must be chosen between the target school and an intermediate bus stop where the child continues with a regular bus for the rest of the journey. Such cases can be modeled with grouping constraints, which effectively prevent feasible solutions from containing both of the activities at the same time. Generation of alternative delivery locations is illustrated in Algorithm 20. Here we assume

Algorithm 20 Generating alternative deliveries on PDP model instance.

```
for all Task task in plan do
  for all TaskEvent event in task.PossibleDeliveries do
    delivery ← CreateActivity()
    AddCapability( delivery, capacity, +event.Capacity )
    AddCapability( delivery, pdp, −1 )
    activities.Add( delivery )
  end for
  AddActivityGroup( activities )
end for
```

that the task contains a list of possible delivery events that can take place, and that the variable "activities" is a list of activities. We generate an activity for each alternative delivery location and add the task-specific "pdp" capability along with the necessary capacity to the activity. The resulting model is depicted in Figure 46. As we can see from the figure, strictly speaking, the grouping constraint is not necessary in this particular case for feasibility as the precedence capability is

unique to each task. However, the grouping constraints offer additional structure to the problem and their presence may be used to aid the algorithmic search.

7.2.2 Time Windows and Quality of Service

While capacities and capabilities were central to the base models of different problem variants, the most common extension to the problem is time windows. In addition to time window constraints, we have already introduced other constraints related to time and the quality of service, such as restrictions on time on a vehicle and the total route length. We now examine these constraints from a model generation viewpoint.

Time window constraints can be imposed on drivers, fleet, equipment, or facilities, but most often they constraint the arrival on a pickup or a delivery of a task. The presence of time windows adds a necessary resource to the model, and introduces a new data matrix: travel times. At this point, we cover the simple hard time window on pickups and deliveries. This is illustrated in Algorithm 21. We first introduce a capability for tracking the clock[2], and a resource "time" for

Algorithm 21 Generating time windows on PDP model instance.

```
clock ← CreateCapability()
time ← CreateResource()
for all Vehicle vehicle in plan do
  AddCapability( start, clock, +1 )
  AddCapability( end, clock, −1 )
end for
for all Task task in plan do
  ptw ← CreateCapability()
  AddCapability( pickup, ptw, 0 )
  SetParent( ptw, clock )
  window ← task.Pickup.TimeWindow
  AddBounds( ptw, time, window.Start, window.End )
  dtw ← CreateCapability()
  AddCapability( delivery, dtw, 0 )
  SetParent( dtw, clock )
  window ← task.Delivery.TimeWindow
  AddBounds( dtw, time, window.Start, window.End )
end for
AddTransitionalResourceFunction( time, clock, ( a, b ) → travelTime[a,b] )
```

accumulating elapsed time. Each route start and end is given the clock so that each route shares the same capability. Each pickup and delivery is then given a constraint as follows: first, a unique capability is created for the activity, and then a zero value is set to the activity for that capability, which is subsequently given a

[2] Although in the elementary PDP model the same can be achieved with the capability "v", we use "clock" here for brevity and compatibility with the subsequent models.

parent "clock", and a constraint is then generated from the time window values. Finally, a transitional resource delta function describing increase in "time" for "clock" is added to the model.

The illustrated model covers the case of a single hard time window, but other variants are also possible. *Multiple time windows* can be expressed using alternative activities with capabilities with different lower and upper bounds, similarly to the example on alternative deliveries depicted earlier. *Disallowing waiting* on activities can be enforced by stating that the resource denoting time has strict lower bounds. *Soft time windows*, on the other hand, can be modeled using profit functions. A situation-dependent profit function is needed for this purpose: the arrival time can be extracted from the accumulated resource "time", which can then be used to compute the deviation from the desired arrival time and to adjust the yielded profit accordingly.

Restrictions on time on a vehicle apply especially to dial-a-ride problems, but such constraints may also be present in other types of routing problems. In PDP problems, each task is identified with a unique capability, and this capability can be used to impose task-specific restrictions on the time between the parts of that task. This is illustrated in Algorithm 22. To restrict the time a passenger or

Algorithm 22 Generating restriction on time on vehicle.

```
for all Task task in plan do
  AddBounds( pdp, time, 0, task.MaxTimeOnVehicle )
end for
```

cargo stays on the vehicle, we simply need to constraint the resource "time" on the "pdp" of the task of transporting that particular passenger or cargo.

Restricting the length or the duration of a route can be done similarly to the restriction on time on a vehicle. In this case, the constraint is defined on the vehicle-specific capability (or in the elementary case, the common capability "v"). The elementary case is depicted in Algorithm 23. In order to impose the

Algorithm 23 Generating route length constraint.

```
for all Vehicle vehicle in plan do
  AddBounds( v, distance, 0, plan.MaxRouteLength )
end for
```

route length constraint, we simply set a suitable upper bound on the resource "distance" on the capability "v".

7.2.3 Situation-dependent Travel and Costs

A situation, as we have referred to it, is an informal term for the set of resource and capability values at a given activity, accumulated that far along the route before that activity. Situation dependency on resources means that the accumulated values define more subtle changes to the resource value than simple addition.

In fact, situation dependent functions are in most cases non-monotonic and usually difficult to approximate during local search. Despite these problems, they are every so often needed in modeling of resource accumulation. Sometimes the resources depend on the active capabilities, such as equipment or current cargo, and at other times, on other resources, such as time. We address capability dependent resources in the subsequent sections as we discuss, e.g., the fleet, drivers, and equipment, but cover the resource-dependent resources (including profit) here. Resource-dependent resources arise, for example, when modeling overtimes (profit depending on time), traffic (time and distance depending on time), effects of cargo on travel (time depending on capacity), changes in service times due lighting conditions, i.e., slower at night (time dependent on time), or cost of waiting (profit depending on time).

Overtime costs can be modeled by a situation-dependent profit function using, for example, a piecewise linear function from travel time to profit, where, after a certain time value, e.g., 8 hours, the coefficient of the costs changes to an overtime salary.

Waiting time costs can be modeled by a situation-dependent profit function that considers the time of arrival and the beginning of the corresponding time window and includes an additional penalty if the former is less than the latter.

Effects of traffic can be modeled using a situation-dependent resource delta function using the accumulated time as the input on function mapping start time, source, and destination to expected travel time. The same can be applied to distance, although this resource changes perhaps less in practice. The effects of traffic include modeling the fact that traveling is usually faster at night when there is less traffic in general.

Effects of cargo are captured by using a situation-dependent resource delta function which employs the accumulated capacity value as an additional input to the function mapping the capacity and source and destination to travel time (or distance, if after a certain weight, the truck needs to avoid certain road segments, such as bridges).

7.2.4 Fleet, Crew, and Equipment Selection

Fleet management extends the routing and scheduling problems into a new dimension. The decisions in fleet management include choice of vehicles with different capacities, availability, costs, speed, and compatibility; various equipment with compatibilities with tasks and vehicles; and management of trailers with differing capacity, availability, costs, speed and compatibility with tasks and vehicles, and which may need to be dropped off before, during, or after completing a route.

Vehicle selection involves also deciding which kinds of vehicles perform the designated tasks. This can be done by introducing additional activities. In these cases, capacities are not generated to the route start and end, but instead to the vehicle start and end activities. These activities are then used as part of the routes as any other activity, and thus become decisions themselves, effectively

resulting in a routing problem with a heterogeneous fleet. These activities may have differing capabilities depending on the vehicle they correspond to. This enables changes, e.g., in capacity and speed between the vehicles. Generation of a fleet of heterogeneous vehicles is depicted in Algorithm 24. Instead of generat-

Algorithm 24 Generating heterogeneous fleet of vehicles.

```
for all Vehicle vehicle in plan do
  v ← CreateCapability()
  vstart ← CreateActivity()
  stw ← CreateCapability()
  AddCapability( vstart, d, −1 )
  AddCapability( vstart, v, +1 )
  AddCapability( vstart, c, +vehicle.Capacity )
  AddCapability( vstart, stw, 0 )
  SetParent( stw, clock )
  window ← vehicle.Start.TimeWindow
  AddBounds( stw, time, window.Start, window.End )
  vend ← CreateActivity()
  etw ← CreateCapability()
  AddCapability( vend, d, +1 )
  AddCapability( vend, v, −1 )
  AddCapability( vend, c, −vehicle.Capacity )
  AddCapability( vend, etw, 0 )
  SetParent( etw, clock )
  window ← vehicle.End.TimeWindow
  AddBounds( etw, time, window.Start, window.End )
  AddTransitionalResourceFunction( time, v, ( a, b ) → travelTime[v,a,b] )
  AddTransitionalProfitFunction( v, ( a, b ) → M−travelTime[v,a,b] )
end for
```

ing a single capability "v", we generate an individual capability for each vehicle. We then proceed, much like in the example of time windows, to generate the start and end activities with time windows, capacities, and a precedence requirement. In addition, we need to generate a requirement for "d" ("driver"), which ensures that no driver employs two or more vehicles simultaneously. A similar procedure is performed to the vehicle end activity, but with opposite capability values. Finally, we add the vehicle-specific speed and cost matrices to the problem. Note that these are bound to activate on a capability unique to each vehicle. This enables individual distance and cost matrices potentially for each vehicle in the problem. The generation of vehicle activities is illustrated in Figure 47. In the example, capability "d" denotes the driver, "c" capacity, "v" vehicle, and "stw" and "etw" start and end time window constraints, respectively.

Compatibilities between crew, vehicles, equipment, tasks, and combinations of these may be present at a problem. These can be modeled with capabilities such that each requirement of a vehicle, or equipment, is present as negative

FIGURE 47 Modeling vehicle start and end activities.

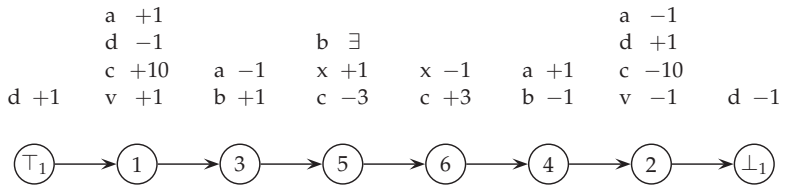


FIGURE 48 Modeling driver, vehicle, equipment, and task compatibilities.

capability in the pickup, and positive in the delivery; and each capability offered in the reverse order. This is depicted in Algorithm 25, where a driver has a capability to use a vehicle, and a vehicle enables usage of special equipment which is in turn required to perform a task. We assume here that we have created the start and end activities and the corresponding actors for each driver. First, we generate all the capabilities offered by the drivers, store them into table "capabilities", and add them to the corresponding actor start and end activities. Then, after generating the vehicles, we generate any requirements they might impose on the drivers, and any additional capabilities the vehicles themselves offer. Exactly the same process is done to the available equipment after a pair of activities has been generated for its start and end locations. Finally, each task is given its requirements. Note that capabilities from drivers, vehicles, or equipment may contribute to all the elements further down the route sequences; e.g., a special capability of a driver can be required for obtaining a piece of equipment or performing a given task. It is also possible to generate capabilities offered by tasks to perform other tasks, but we omit this for clarity as the action of picking up and returning a piece of equipment can be seen as such a task. An example of driver, vehicle, equipment, and task compatibilities is illustrated in Figure 48. In the example, "d" denotes the driver, "v" the vehicle, and "c" its capacity. The vehicle in question has a capability "a", for example, "is able to tow machinery". This capability is then consumed in activity 3 as the machine in question prevents other machines from being attached to the vehicle simultaneously. The capability of this machine "b" is then required at activity 5.

Incompatibilities between different types of activities can be modeled using the capability prohibition in the way discussed earlier when illustrating the capability extension in general. For instance, an incompatibility between a vehicle and an activity, say a ship and a harbor, can simply be stated by adding

Algorithm 25 Generating compatibilities within heterogeneous fleet.

```
for all Driver driver in plan do
  for all Capability capability in driver.Capabilities do
    capabilities[capability] ← GenerateCapability()
    AddCapability( start, capabilities[capability], +1 )
    AddCapability( end, capabilities[capability], −1 )
  end for
end for
for all Vehicle vehicle in plan do
  for all Capability capability in vehicle.Requirements do
    AddCapability( vstart, capabilities[capability], −1 )
    AddCapability( vend, capabilities[capability], +1 )
  end for
  for all Capability capability in v.Capabilities do
    capabilities[capability] ← GenerateCapability()
    AddCapability( vstart, capabilities[capability], +1 )
    AddCapability( vend, capabilities[capability], −1 )
  end for
end for
for all Equipment equipment in plan do
  for all Capability capability in equipment.Requirements do
    AddCapability( estart, capabilities[capability], −1 )
    AddCapability( eend, capabilities[capability], +1 )
  end for
  for all Capability capability in equipment.Capabilities do
    capabilities[capability] ← GenerateCapability()
    AddCapability( estart, capabilities[capability], +1 )
    AddCapability( eend, capabilities[capability], −1 )
  end for
end for
for all Task task in plan do
  for all Capability capability in task.Requirements do
    AddCapabilityRequirement( pickup, capabilities[capability] )
  end for
end for
```

a capability prohibition for the capability of the ship to the activity in question. Dynamic incompatibilities, that is, *contamination*, can be modeled with suitable resource and lower bounds. Contamination is an incompatibility caused by a task. This incompatibility is in effect for a given duration and during this period it prevents some tasks from being performed. Generation of contamination rules and the adjoining restrictions is illustrated in Algorithm 26. Each task has

Algorithm 26 Generating contamination restrictions.

```
d ← CreateResource()
SetStrictLowerBound( d )
for all Task task in plan do
  pickup ← CreateActivity()
  delivery ← CreateActivity()
  for all Contamination contamination in task.Contaminations do
    c ← CreateCapability()
    contaminations[contamination] = c
    pickups[task] = pickup
    AddCapability( delivery, c, +1 )
    SetAllowNonempty( c )
  end for
end for
for all Task task in plan do
  for all Restriction restriction in task.Restrictions do
    r ← CreateCapability()
    AddCapability( pickups[task], r, 0 )
    SetParent( r, contaminations[contamination] )
    AddBounds( r, d, restriction.Length, M )
  end for
end for
```

a list of unique contaminations it causes and these are generated as capabilities activating at the activity of that task. The delivery of the task activates the contamination caused by the task. Each task has also a list of restrictions for the contaminations, that is, which contaminations in fact prohibit the loading of the task and for how long. In this example, the restriction is a window to the resource "d" ("deliveries") and the lower bound equals to the length of the contamination. The restrictions are generated for the pickups of each task to ensure that there is indeed a sufficient number of task deliveries between the delivery of the contaminating task and pickup of the sensitive task. Note that to prevent simultaneous loading of incompatible tasks, one needs to impose also capability prohibition for the capabilities of the tasks. Furthermore, to enable the sensitive tasks to be serviced before any contaminating, one needs to add the restrictions to route start activities (both of these details have been omitted from the algorithm).

Figure 49 presents an example of modeling contamination. In the example, the task "x" contaminates the vehicle so that the task "z" cannot be performed

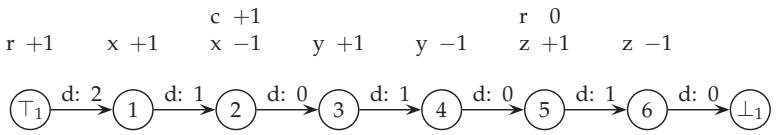


FIGURE 49 Modeling contamination rules.

unless one other task is performed completely in between. The resource "d" is accumulated by one every time a delivery is performed. Note that the number of deliveries must be sufficiently large when starting the route to ensure that the lower bounds are met even if no contaminating task has been performed. In the example, this is ensured by increasing the number of deliveries by 2 when leaving the start activity. The restriction "r" has a strict lower bound of 1 on the resource "d" and uses "x" as its parent. This makes it infeasible to visit the activity 5 immediately after the activity 2 as the value of "d" would be zero at that point. We must point out, however, that modeling contaminations using this approach is not suitable for problems in which a large fraction of the tasks *cause* contamination. Each contaminating task introduces two new capabilities and this can result in excessively large models in such cases.

Trailers can be seen as special equipment with capacity. Trailers — as well as other pieces of equipment — can alter also the costs and speed of traversal, and they may have individual time windows. A single trailer is typically attached to a vehicle and it adds to the capacity of that vehicle. Generating this type of case is illustrated in Algorithm 27, where the trailer increases the capacity of the vehicle but affects its cost and speed. The changes in resource and profit functions are introduced as in the nominal case, but the stacked resource extension mechanism is employed internally as soon as the capability "trailer" is activated. It is

Algorithm 27 Generating trailers affecting costs and travel times.

```
for all Trailer trailer in plan do
  t ← CreateCapability()
  tstart ← CreateActivity()
  AddCapability( tstart, h, −1 )
  AddCapability( tstart, t, +1 )
  AddCapability( tstart, c, +trailer.Capacity )
  tend ← CreateActivity()
  AddCapability( tend, h, +1 )
  AddCapability( tend, t, −1 )
  AddCapability( tend, c, −trailer.Capacity )
  AddTransitionalResourceFunction( time, t, ( a, b ) → travelTime[t,a,b] )
  AddTransitionalProfitFunction( t, ( a, b ) → M−travelTime[t,a,b] )
end for
```

assumed that vehicles capable of picking up trailers have capability "h" ("hitch")

FIGURE 50 Modeling trailer pickup and drop-off activities.

which is consumed at trailer pickup. In addition, each trailer adds additional capacity to the capacity of the vehicle. Otherwise the trailer generation follows that of equipment. Equipment, especially trailers, may also have alternative drop-off locations, and these are modeled similarly to the approach discussed earlier. An example of modeling trailer pickup and drop-off activities is illustrated in Figure 50. In the example, vehicles have a single hitch "h", which enables usage of a single trailer at a time. The trailer then consumes this capability, and increases the capacity "c" of the vehicle. Note also that it is, naturally, possible to generate trailer pickup and drop-off activities without generating the vehicle start and end activities, if the fleet and the set of drivers are otherwise homogeneous.

Service events refer to maintenance tasks or other activities during which the vehicle cannot be on active use, that is, no cargo can be on the vehicle. These types of activities can take place, however, anywhere during the route. The emptiness of the vehicle can be ensured by generating a task which consumes all the capacity available within the vehicle.

7.2.5 Drivers and Legislation

Several real-life restrictions manifest themselves in constraints on drivers. An individual driver may be incompatible with a given task, a given vehicle, or given equipment, such as a trailer. Drivers may also be allowed to complete multiple routes during the planning horizon, provided that certain criteria are met. Moreover, break legislation may introduce several different types of constraints on route structure.

Driver compatibilities may restrict the ability of the driver to perform tasks or use a vehicle or equipment. We covered much of the compatibility issues in the previous section. Many of these compatibilities are due to some regulatory measures, such as licenses to drive different classes of vehicles and permissions to operate different types of machinery. In some cases, the drivers may also be assigned to a certain set of tasks, for example, by previously used routines; each task may be located in some area which is familiar to some drivers and unfamiliar to others. In these cases, the operation times are lower when an accustomed driver visits a task. Sometimes the design of routes has to conform to these customs of zoning the operations, and these too can be modeled with compatibilities. An example of a driver-task compatibility is given in Figure 51. The driver is simply given a capability "familiarity" at the route start activity, and this capability is subsequently required by a subset of task activities.

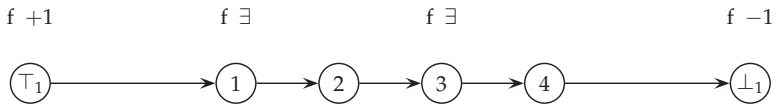


FIGURE 51 Modeling driver-task compatibilities.

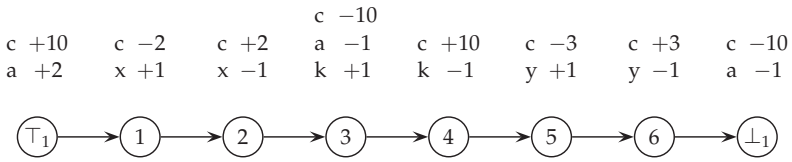


FIGURE 52 Modeling multiple uses of vehicles.

Multiple uses of vehicles refer to property of a problem that allows refilling (or emptying) vehicles at the depot during the planning horizon to serve more customers than would otherwise be possible due to capacity restrictions. These cases can be modeled exactly as the service events described earlier. These service events require the vehicle to be empty while visiting the depot. This leads to a situation where several independent routes are constructed between the service events. In fact, using this technique we may model a planning horizon of several days by introducing service events of, e.g., 16 hours to separate the workdays. Modeling multiple uses of vehicles within a PDP problem is depicted in Figure 52. In the example, each service event is modeled as two activities, first, activity 3, which consumes all the capacity of the corresponding vehicle, and second, activity 4, which restores it. The pickups performed thus have to be delivered before the next stop at the depot. Notice how we introduce "a +2" in the route start activity, and "a −1" at each stop at the depot (including the last). This means that in this case we need to visit the depot for an additional time during the complete route, effectively creating two routes — both of which may of course remain empty.

Breaks add additional tasks for the drivers: lunches and coffee breaks for instance. These activities, however, differ from the other typical tasks. Breaks have more complex rules on time windows: their exact location may not be defined and there may be several available options for taking breaks.

Breaks with defined locations are relatively simple to model by introducing additional tasks acting as breaks. In a case where two compulsory breaks have to be taken, we may use the technique discussed earlier: decision-dependent time windows. If we assume that the first break, "b", has to be taken after 4 hours of driving, and the second, "b'", after no more than 4 hours has passed from the first break, we may define that the lower bound on time to the latter "b'" is less than 4 hours. This is illustrated in Figure 53. In these types of cases, at the route start, we require that we take the first break, and when the first break is taken, we subsequently require the second. The time windows are set to these

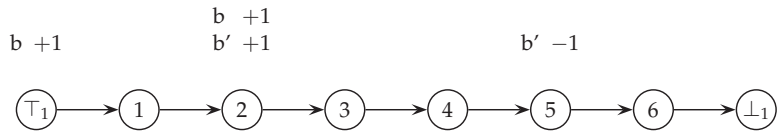


FIGURE 53 Modeling breaks with predefined locations.

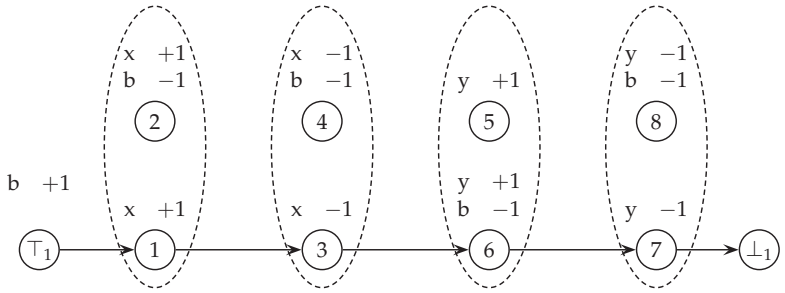


FIGURE 54 Modeling breaks with undefined locations.

two capabilities accordingly.

In contrast to breaks with defined locations, ones with *undefined locations* are not represented by additional tasks by themselves. As we do not know where the break is taken, the decision becomes "during which task is the break taken". Alternative activities can be generated to model this situation. Each task during which a break can be taken is duplicated and the duplicate is given almost the same properties along with the same location as the original task. The break-specific capabilities are different as well as the duration of the task. This represents the time required to complete the task and perform the break. This is illustrated in Figure 54. In the example, each pickup and delivery (the odd numbers) has an adjoining activity (the even numbers), and we require that exactly one of the even numbered is visited during the route by the capability "b". The service times at activities 2, 4, 6, and 8 are equal to the duration of the break plus the service time in activities 1, 3, 5, and 7, respectively.

We now revisit the two examples considered in the literature and discussed in Chapter 2. The cases were defined as follows. In the UK, "a driver cannot drive for more than 4.5 hours consecutively without having a break of 45 min. This break can be taken in a single period, two periods (one of 30 min and another of 15 min) or three periods of 15 min", and in Switzerland, "after 5.5 hours of uninterrupted work, or 4 hours of uninterrupted driving time, the driver must take at least a one-hour break". For illustration, we assume that in the former case, the locations of the breaks are not defined, and in the latter, they are given.

The Switzerland case with defined locations can be modeled as in Figure 53 given earlier, but with two resources: "travel time" and "work time". These two

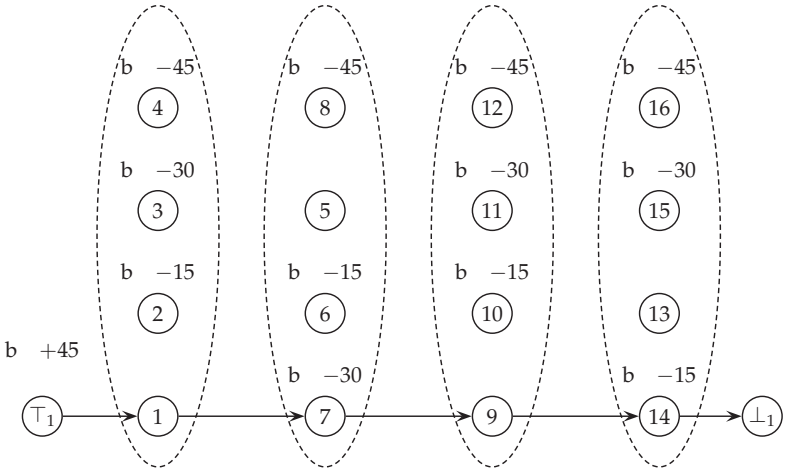


FIGURE 55 Modeling breaks according to the UK legislation.

are accumulated so that driving between locations increases both according to the time required to travel between the locations, but service times at each location increase only the resource "work time". The breaks are then given upper bounds of 5.5 hours for the "work time" and 4 hours for the "travel time". Whichever bound is first met dictates the time when the break has to be taken.

The UK case can be modeled similarly to the generic case with undefined locations. In this case, we have several alternative ways of taking the breaks and thus we introduce an additional activity for each of these alternatives. This is depicted in Figure 55. We begin the route by stating that 45 minutes of break has to be taken during the route, and introduce capability "b" with an appropriate value at the route start. Each activity corresponding to a task with no adjoining break, that is, activities 1, 5, 9, and 13 is then given three alternative activities. Activities 2, 6, 10, and 14 denote task and an adjoining 15 minute break; activities 3, 7, 11, and 15, that of 30; and activities 4, 8, 12, and 16; of 45 minutes. The route consists of two activities with no adjoining break, 1, and 9; an activity with a 30-minute break, 7; and an activity with a 15-minute break, 14. This combination consumes the required amount of break time, and the decisions involved are semantically the same as in the problem definition. Note that, although capabilities are here used to track the break time taken, the service times still have to be generated as usual: the original service times plus the duration of the break for each activity with an adjoining break.

Finally, we note that there is an inconsistency in the feasibility of empty routes in the figures of this section. Not all examples are feasible when the route is empty. The illustrated cases do, then, require an additional optional activity for allowing the empty route case. This activity has, however, been omitted from the figures for brevity. Ensuring that empty routes are allowed can be done by intro-

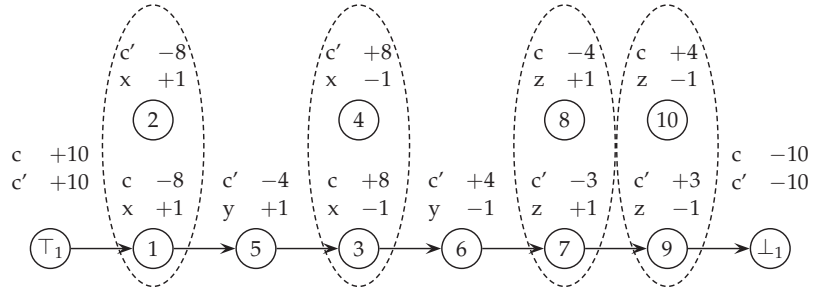


FIGURE 56 Modeling compartments with compatibilities within a PDP instance.

ducing an activity with appropriate capabilities, and a time window of zero. This effectively prevents the activity to be on the route if other activities are present, but assigning this activity to the empty route makes also the empty case feasible. The additional activity can then be interpreted as the task of completing the route without the breaks as they were not needed.

7.2.6 Compartment Loading Decisions

Some routing and scheduling problems include decisions on how to load the vehicles. Some problems even address the physical arrangement within the cargo space, but in this context we consider only the decisions on which compartment each object should be loaded. The compartments may vary with their compatibility and size, and some compartments may be included as optional, such as those in trailers. Moreover, each vehicle may contain a different set of compartments, that is, the fleet may be heterogeneous in this regard.

Compartments can be modeled using alternative activities denoting the alternative compartments to which the cargo can be loaded. Multiple types of compartments may exist as each compartment may be compatible with a subset of tasks, their size may vary, and also their capacity constrains may be defined in multiple dimensions. When modeling compartments, each compartment is introduced by giving an individual capacity to the actor. Each of the alternative activities then consumes one of these capacities. Additional dimensions on the capacities would introduce a new capability for each compartment, and each alternative activity would consume the capacity on each dimension. Introducing compatibilities between compartments and activities is in the simplest case a matter of dropping the alternative of loading the task in the incompatible compartment. A case with two compartments of equal size in a vehicle, and an incompatibility between a compartment and a task is illustrated in Figure 56. In the depicted case, the vehicle is given two compartments, “c” and “c′”. The first task consists of transporting cargo denoted with capability “x”, which can be loaded on either of the compartments. Both alternatives consume 8 units of the selected capacity. In the example, the compartment “c” is selected. The task denoted with

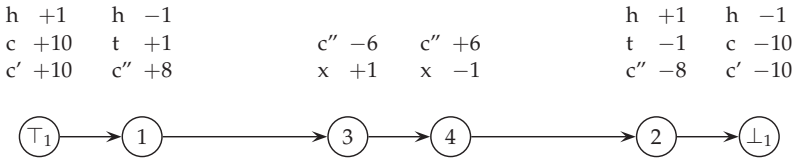


FIGURE 57 Modeling compartments in trailers within a PDP instance.

"y", in contrast, is compatible only with the compartment "c′", and is, therefore, modeled as a single pair of activities, 5 and 6. Finally, the task of transporting "z" illustrates the fact that the alternative activities do not have to be exactly similar. Perhaps some compartments are designed to hold certain cargo and contain appropriate methods for storing these objects, while other compartments do not. In these cases, the actual amount of capacity consumed differs from one compartment to another. In the example, we choose the compartment "c′" as storing the cargo there consumes less capacity.

Compartments within trailers can be modeled either as simple compartments, if we assume that each vehicle has one; or as optional trailers introducing an additional capacity instead of increasing the overall capacity of the vehicle. In fact, the nominal case described earlier can also be interpreted as a loading decision between the vehicle and its trailer. A case where the optional trailer can be introduced, however, is depicted in Figure 57. In the example, the vehicle has two compartments, "c" and "c′", and the trailer introduces a third compartment, "c″". A task compatible with only the newly introduced compartment within the trailer is then picked up to the trailer and delivered before the trailer is released in activity 2. After activity 2, the corresponding compartment is no longer available.

Combining compartments with fleet selection can introduce vehicle compartments in a way that each vehicle contains a subset of the compartment types within the problem. The alternative activities must therefore be introduced for each compatible compartment. Moreover, the compartments may be of different size between different vehicles. A case with three different types of compartments in two vehicles is depicted in Figure 58. In the example, vehicle 1 has compartments of type "c" and "c′", and vehicle 2 of types "c′" and "c″". The task of transporting "x" is generated as three distinct alternative activities as there are a total of three different types of compartments in the problem. As usual, the tasks compatible only with a certain type of compartment are generated using only the activities using the capability.

7.2.7 Notes on Interactive Optimization

In many cases, the planning of operations is an interactive task. This means that there can be changes occurring in the problem data as the planning takes place and some parts of the problem may need to be fixed as they may have already occurred. Moreover, the operator herself might want to restrict the structure manually. In addition to the constraints already discussed, there are also constructs

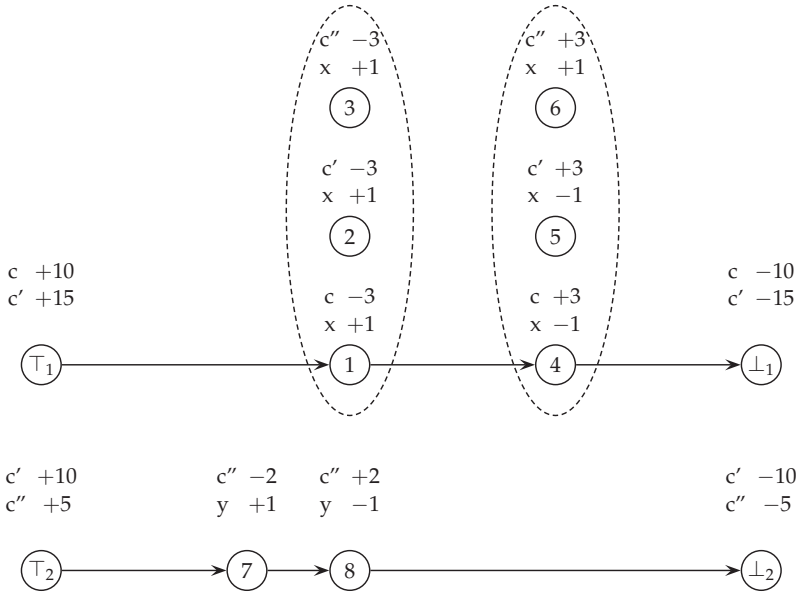


FIGURE 58 Modeling compartments within a heterogeneous fleet.

especially helpful in aiding in interactive planning. Restricting the route structure by locking tasks to given routes is useful when the operator wishes to aid, guide, or restrict the algorithmic search. Similarly, constructing routes from predefined sequences of activities, the operator may employ information which is not available to the algorithms.

Restricting the route structure can be done by employing capability constraints similarly to the techniques discussed in the preceding sections. Activities can be locked to a specific route simply by introducing a zero value for the unique capability of the corresponding actor. Forming predefined sequences of activities, however, requires a more complex construct. First, we need to introduce a resource, the so-called discrete resource which measures the activities visited by considering all the distances between different activities to be exactly one. We then introduce capability values of $+1$ and -1 at the ends of the locked sequence, add a child capability of that capability of a zero value at every activity in between, and require existence of the capability in each intermediate activity. In addition, we restrict each activity on the sequence by an upper bound on the discrete resource so that the first child activity is given an upper bound of 1, the second 2, and so on. Finally, we set the upper bound of the parent capability as $n - 1$, where n is the length of the sequence. These bounds prevent activities from being inserted within the sequence. The capabilities of this mechanism are depicted in Figure 59. In this case, we lock the sequence from activity 2 to activity 5. The capability "a'" has an upper bound of 1 on the discrete resource, "a"", upper bound of 2, and as the length of the sequence is four activities, "a" has an

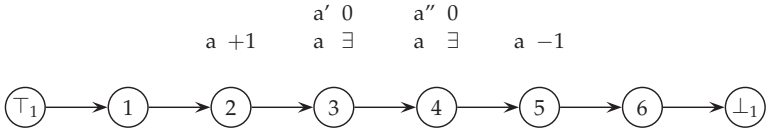


FIGURE 59 Modeling a locked sequence in the middle of a route.

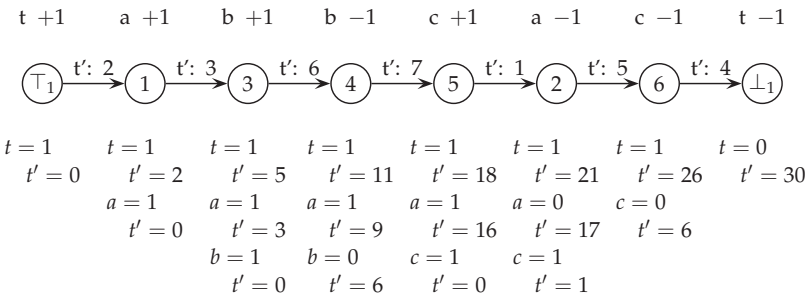


FIGURE 60 An example of situations during continuous planning.

upper bound of 3.

Continuous planning introduces a requirement to lock the start of the route similarly to the locking of an arbitrary sequence as there are some activities on the sequence that cannot be moved or removed (they have already taken place). The activities cannot simply be removed as the situation on the vehicle, for instance, has to be taken into consideration. The partially completed tasks have to be delivered and so on. However, we may avoid introduction of additional constraints as we did with the arbitrary sequences[3]. We may compress the route start into a single activity by storing the exact accumulated values to the route start activity. Strictly speaking, compressing the route start is not defined as a part of the model transformation; the operation instead skips the generation of confirmed activities and alters the data matrices and internal value fields accordingly as a preprocessing step; but for completeness, the technique is discussed here as it is available as a part of the process at runtime.

Figure 60 presents an example of a regular route and its accumulated values. In the example, three PDP tasks have been defined and their projected values have been computed. Now, a message arrives to the system indicating that the vehicle in question has completed activity 3. If there is a need to perform optimization after the fact, we may relax the problem by removing the activities preceding activity 3 and change the route start activity accordingly. This is illustrated in Figure 61. The resulting problem contains activity 3 as a starting activity. Note that the activity cannot be moved as it is now the first activity on the route. This approach captures the situation exactly and is transparent to the optimiza-

[3] Arbitrary sequences cannot be freely compressed into a single activity unless we assume some properties of the resource and objective functions; for instance, situation dependency prevents the computation of the total deltas of these sequences *a priori*.

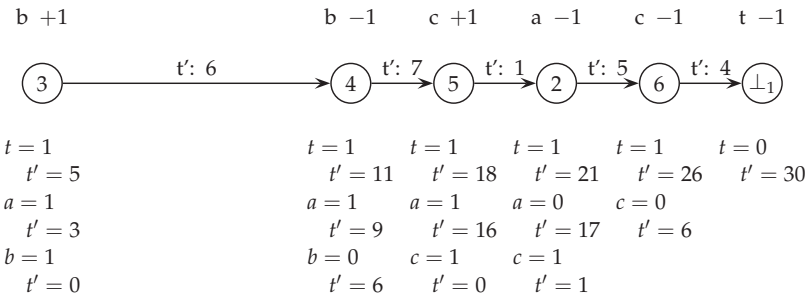


FIGURE 61 A result of compressing the route start sequence after activity 3 has been confirmed.

tion system. Moreover, note how tasks "a" and "b" have been completed only partially. They are stored into the values of activity 3 even though the pickup activities are no longer on the route. The corresponding deliveries can, therefore, no longer be removed from the route. In contrast, the task "c" can be moved freely. Observe also how the values of time, "t'", remain unaltered at the tail of the route.

7.2.8 Contrast to Existing Modeling Approaches

The examples presented in this section demonstrate the expressiveness of the developed modeling approach. The approach is only one among the recent attempts to unify the modeling and solving problems in routing and scheduling. Some of the prominent steps in unifying these problems have been taken in mathematical and constraint programming. Unified mixed-integer programming formulations for routing and scheduling with time windows were given by Desrosiers *et al.* [60] and later by Desaulniers *et al.* [59]. These frameworks are primarily intended to be used in exact solution procedures such as column generation and Lagrangian relaxation [109].

Perhaps the most comprehensive modeling and solution framework for *local search* has been provided by Irnich in [109] where an impressive number of variants and extensions are modeled using resource extension functions. They presented the low-level mechanisms needed in employing efficient local search in rich routing problems and formulated the necessary conditions for achieving constant time feasibility checks and objective evaluation in different situations. We examined explicitly the dynamics of different resources (resources and capabilities), accommodated the dimension of implementation, and refined the modeling approach to suit these aspects.

The framework proposed in [109] is based on giant tour encoding and is more naturally suitable for modeling *inter-route constraints*. They also examine *periodic routing*, which we omitted from this work, although we note that it should be compatible with our approach. In our classification, the work of Irnich is fo-

cused on mapping-ordering constraints, whereas we included also mapping in the form of activity groups. In many respects, the achieved expressiveness in our model is similar to that of Irnich, but we captured also some of the properties not directly compatible with their proposed framework: compartment loading decisions are not part of the framework, and soft time windows and time-dependent travel, for example, are not compatible with the sequential search used. The deliberate generality of our approach in this respect is reflected in the modeling language where the situation-dependent projections are considered "first class citizens" and their structure is explicitly defined. In fact, we have made the structure of the problem more explicit in general, which allows us to analyze it separately. Overall, we argue that the connection to the conceptual models is clearer in our approach. However, to be fair, as we did not intend to provide comprehensive solution methodology, our attempt is still limited in this sense.

From the implementation viewpoint, our approach, in fact, has introduced a simple *domain-specific language* [75] for the routing domain. The simplicity is perhaps due to the role of the language: it defines flexibly the different routing variants through a compact model transformation. The developed language supports the usage of the software product line in this domain by decoupling the solver from the domain representation. The simplicity keeps the coupling relatively loose, which is, indeed, a desirable property.

7.3 Quality Attributes

In this chapter, we began to analyze the *simplicity*, *flexibility*, and *reusability* of an optimization system built on a model-driven software product line and employing a generic optimization framework and a metamodeling approach. The examples presented in this chapter illustrate the flexibility of the system, albeit not simplicity *per se*. Reusability has been achieved in multiple locations on the overall architecture, as expected from the discussion earlier in the context of software product line architectures in general. In this section, we examine these three major quality attributes critically, and discuss their implications on the *performance* of the system.

Simplicity, as we defined in Chapter 4, refers to the negative of the effort needed in constructing the system. The heavy machinery of the product line approach and the design effort that has to be performed up-front certainly does not stand out as simple. But when examining each individual case, we require a considerably smaller amount of effort than building a complete system for each case, even from existing modules. The difference to a typical modular structure is the granularity at which reuse and systematic customization can be achieved. There are several aspects that illustrate the simplicity of the system. The major ones are listed here. First, we observed in Section 7.1.4 the simplicity of introducing new algorithms to the system, either as a core asset or a customization to an application. The algorithms, e.g., do not have to consider case-specific proper-

ties, unless specifically required. In Section 7.2, we described a number of model definitions, which demonstrates the simplicity of altering an existing or providing a new optimization model into the system. The transformation definitions communicate only the relevant aspects of the optimization problem using the constructs defined in the routing metamodel. Overall, the structure described in Section 7.1.2 simplifies the separation of different cases. The common elements can be included on the domain layer and in optional modules, keeping the case-specific implementations separate. This also highlights a common property of software product lines: the customization of each case is relatively simple. Moreover, we note that the object-oriented approach to domain modeling presented in Chapter 5 keeps the manipulation and presentation of the problem data relatively straightforward.

Flexibility refers to the amount of effort needed compared to the degree of change required when altering the system to different situations. The existence of a variation point does not automatically yield flexibility, although having explicit variation points is certainly helpful. The key to flexibility is in the simplicity of changing a variant or providing a new one. As we demonstrated in Section 7.2, we can flexibly alter the optimization model without altering other parts of the system. The metamodel hides the specifics of the optimization model by proving a layer of abstraction. Now, as we discussed in Chapter 6, the system may not perform equally well in every case, thus effectively requiring additional changes on the solution methodology parts of the system. This is true with every optimization system. However, in Sections 7.1.3–7.1.4, we examined the structure of the system and its variation. The structure shows that we can change and alter the used metaheuristics and algorithms flexibly. This can also be done on runtime. Thus, the system should be flexible enough to be adapted to different problems. From a broader perspective, the product line approach should be suitable to routing systems in general. For example, varying the task generation process as presented in Section 7.1.5 is a case where the product line approach offers flexibility with a relatively minor increase in complexity; employing optional or custom services from both domain and application layers can be performed as needed case by case. The product line approach also allows extending the problem domain and providing the necessary data connections, and dispatching, reporting, and tracking functionality. In general, the arguments in favor of product lines apply as the routing domain is, indeed, heterogeneous also from this viewpoint.

Reusability was defined as the probability of an element of a system to be usable in different contexts with minor or no modifications. The removal of algorithms' dependency from the optimization models discussed in Section 7.1.4 is perhaps the clearest example of reuse in this context. This allows the algorithms to be used without modifications in differing cases. As noted, they might need to be adapted, but this is not necessarily a major modification. The overall structure described in Section 7.1.2 provides a clear separation of concerns between the presentation functionality, the domain modeling functionality, and the optimization functionality. This structure enables reuse on several layers; e.g., generic reusable UI components can be developed and tailored for the VRP domain. The example

of varying the task generation process as presented in Section 7.1.5, the necessary data connections, and dispatching, reporting, and tracking functions can also be considered from the reuse viewpoint: also these variability points enable reuse of their variants in several application level realizations. Finally, we would like to highlight the fact that the *reference architecture* described in this chapter is regarded as a core asset, and thus a reusable artifact.

As we observed in Chapter 6, the **performance** of the system, especially in our prototype implementation, suffers from the included flexibility and generality. Moreover, we cannot yet be certain of the scalability of the system due to a lack of exhaustive testing with the best known solution methods and large-scale problem instances. However, it must be noted that it is difficult to compare methodology and models to existing implementations as they are not capable of expressing all the aspects presented in this work. Moreover, we demonstrated that existing solution methodology can be used within the framework developed, and theoretical analysis indicates only a constant computational overhead. If we manage to keep the overhead independent of the problem size, it is (at least in theory) possible to construct the system in a way that that overhead is covered by the increases in availability of computational power (e.g., by distributing and parallelizising computation). It remains to be seen whether the performance overhead prevents the large scale use of this approach. We note, again, that some practical cases were, indeed, solved successfully by this approach during the course of the research, and overall, the initial findings are promising. This is arguably not the one and only answer to the flexibility versus performance question in optimization systems, but can be considered as one possible approach worth pursuing.

From this discussion and the examples given in Chapters 5 and 6, it should be plausible that the system meets the quality requirements defined within the context of this work. We stated requirements for implementability, more specifically, simplicity, flexibility, and reusability, and these requirements have been assessed against the implementation. The main objectives were to be able to reuse several key elements in the structure, alter optimization models and solution methodologies flexibly, and keep the system relatively simple for defining individual problem variants. We constrained the design with the performance requirement that the system should be able to utilize existing solution methodology, and with the fidelity requirement that the system should be able to express the most common complex routing variants. We hope to have sufficiently demonstrated also these properties.

8 CONCLUSION AND DISCUSSION

In this chapter, we establish conclusions on the proposed approach, provide a broader view on the research conducted, and contemplate its possibilities and effects, as well as suggest new research topics for the future. While the discussion in Chapter 7 was concerned of the developed system in its currents state, this chapter will broaden the view also into future. The chapter is structured as follows. We begin by drawing conclusions on the overall work in Section 8.1, and further analyze the applicability of the developed approach into practice in Section 8.2. We continue by discussing the implications of a unifying modeling framework in Section 8.3 where we consider the possibilities of this approach, and provide some comments on the state of the art of the current VRP research. In Section 8.4, we suggest concrete directions for further studies in the areas of combinatorial optimization and software engineering.

8.1 Conclusions

As we have argued, we do not yet fully know how to make software that can be applied to solve today's heterogeneous set of routing problems. This dissertation presented one approach for addressing the complexities of implementing routing systems. We suggested the construction of a metamodel of vehicle routing problems and the application of model-driven software engineering practices for achieving an effective and systematic engineering approach for implementing these systems.

Our overall objective was to provide means for solving, cost efficiently, the design problems in routing and scheduling. We argued that this can be achieved by constructing an optimization system that is capable of handling a wide variety of different types of routing problems and can easily be adjusted according to the particular problem instance under consideration. We asserted that the key elements in this approach were a way to describe the different problems in a generic way and machinery that is able to utilize this general description.

To achieve our objectives, we provided a metamodel and a modeling language for describing rich vehicle routing problems, and a model-driven software product line architecture for vehicle routing systems. In addition we provided a formal specification of optimization models and processes in vehicle routing.

We described several rich VRP models motivated by real-life examples to demonstrate that it is possible to define a single modeling system capable of expressing a wide variety of VRP variants and extension without modifying the underlying neighborhood exploration system or algorithm implementations.

We expressed the atomic building blocks of local search algorithms within the defined framework and analyzed the computational complexity of the methods to show that such a system is able to utilize the existing knowledge from the current state-of-the-art algorithms and metaheuristics.

In addition, we presented a system with well-defined variation points in optimization models, solution methodology, and model representation, and expressed a number of routing problems by changing the variability object associated with the model variation point. This demonstrated that the developed framework can be embedded into a software product line in order to achieve reusability in both models and optimization methodology, and within this product line, a single variation point is able to express the modeled case.

From the analysis made, we conclude that the contributions of this dissertation enhance our abilities to manage the complexity of logistic planning and solving a variety of different problems arising in real-world routing. If the proposed approach can be applied into practice, it should decrease the effort needed in adapting routing systems to different situations within the heterogeneous domain of vehicle routing, which in turn should result in a faster application of the latest results in operations research to real-world operations and allow the logistic operators to benefit from the recent advances in automated decision making.

8.2 Applicability of Proposed Approach

As we have examined the developed system in Chapters 5–7, the implicit assumption has been that this approach is viable and applicable to practice. Indeed, we have argued that the problems identified and the requirements stated in Chapter 4 have been addressed by the system, but we have not yet critically examined the applicability of the presented approach into practice. In this section, we intend to analyze the applicability of the system from the implementation, modeling, and solving viewpoints.

The proposed framework and optimization system should be applicable to an academic setting from the **implementation** viewpoint especially when there is a need to produce a diverse set of experimental systems quickly. The general approach is applicable to a subset of current VRP research, and may be helpful when a unifying view on the implementation side of the solution methods and optimization models is needed. However, the approach may not be suitable in

situations where considerable up-front investments cannot be made, or when the software engineering knowledge required to build a product line is not available.

From the industry perspective, the approach can best be applied into a heterogeneous and possibly fragmented set of operators. The approach may be suitable for mid-sized operators who have not been able to afford custom made software packages, but require more customization than the current off-the-shelf software typically offers. The approach was tested in a set of real-life optimization cases, and the flexibility of the system allowed agile development of the case models and optimization algorithms, which is often necessary when tailoring a system into a new environment. Moreover, the approach may shorten the time required for methodology diffusion from academia to industry, making the approach applicable to situations where relatively new knowledge is required in solving the problem. Similarly to academia, the approach may not be suitable in situations where considerable up-front investments cannot be made, or the knowledge required to build a product line is not available.

In general, a product line approach has the potential of greatly reducing the effort needed in developing a solver for a new case through the systematic reuse of assets. Furthermore, as noted before, Cordeau *et al.* [45] conclude that there is a greater need for simplicity in solution methods. Now, if simplicity, effectiveness, and robustness are extremely difficult to achieve using a single method, perhaps multiple simple ones are better suited to the problem. A software product line should also make it less tedious to use a multitude of methods.

Modeling within a unified framework provides academics with a formal approach for understanding the domain on a higher level of abstraction. Furthermore, if fidelity, and the resulting relevance to real-life practice, is a requirement, the approach may provide more detailed optimization models than are currently widely employed. A unifying approach also provides opportunities for measuring and comparing effects of different aspects in optimization problems in a more or less commensurable way. Simultaneous applicability to a wider set of problems encourages research on robustness and adaptive methods, and enables more complex problems to be solved within a single system. In addition, the wide applicability may make the modeling framework, by itself, a useful tool for formal comparison of different VRP models. In contrast, the approach does not function as effectively if the optimization problem cannot be expressed using the modeling framework. In these situations, a considerable amount of work may be needed in modeling and implementation to expand the metamodel to accommodate the new requirements.

From the industry perspective, the unified modeling approach is probably best suited to a situation where the optimization problem is more complex than the current standard problem variants. For example, the developed framework was used to build an industry-strength optimization system with a relatively complex model, and was successfully deployed into operational use. This might suggest that the approach is applicable to practice. In contrast, it is notable that the approach may not be suitable in a situation where the modeling framework is unable to express all the necessary constraints. Furthermore, another situation

is perhaps one where the optimization problem is relatively large and needs extensive approximation. In these types of cases, the expressiveness of the model does not yield considerable advantages.

In general, the modeling approach requires more up-front investments than more straightforward approaches. Indeed, our model formulation is relatively complicated when compared with other routing models individually, but when considering the number of variants expressible, the trade-off may be justifiable. As we noted, some commercial applications use a flexible metaheuristic instead of the most powerful one and thereby trade performance for simplicity; and similarly, we sacrifice some performance for flexibility in order to efficiently study problem variants, and to apply methods and models into practice with less investments. We have argued that this trade-off is also justifiable. Nevertheless, it cannot be ignored that the resulting general model formulation is neither as elegant nor compact as the mathematical programming formulations presented in Chapter 2.

The main disadvantage of a unified modeling framework is, however, at its core: any framework, by definition, restricts itself to a subset of the problem domain. The optimization models are, then, restricted to the developed metamodel, and adding completely new constructs may be conceptually difficult and require modifications to the existing models. Furthermore, implementing functionality to the domain layer of the product line can require more technical expertise from the implementors of the system than in more conventional approaches. In addition, the framework approach has its limits. When incorporating new modeling constructs to a framework, at some point the complexity of the system increases more than the benefit from adding the model into a unified framework. As noted, the product line approach works best for a family of related systems, and if the relation becomes too weak, other methods may be needed.

The approach may be interesting from the **solution process** viewpoint in an academic context. The developed model may be viewed as a new, relevant optimization problem which needs to be solved robustly. This opens up possibilities for examining the robustness of existing methods in a heterogeneous problem setting and should be a suitable research platform for highly adaptive optimization systems. However, the performance overhead induced by, e.g., the flexibility of the system may prevent the framework from serving as a tool for developing and comparing case-specific methods in the highly contested "benchmark race" currently taking place within the VRP research.

Similarly to the modeling viewpoint, from the industry perspective, especially relatively simple cases may not benefit from the developed approach. In this case, the model fidelity is, again, not appropriate, and the solution methods operate on too much detail. However, in more complex cases the solution methodology may be flexibly adjusted to the case under consideration. For example, the ship scheduling case benefited considerably from case-specific construction heuristics and a set of custom-made search operators which considered larger neighborhoods than the standard operators. Adding these algorithms to the existing framework was a straightforward task.

To summarize, the unifying modeling framework and the software product line approach into modeling and solving optimization problems and implementing optimization systems, could be considered useful when

- one needs to address multiple different problems simultaneously due to research or practical reasons,
- the case is a complex combination of different variants and extensions and may need refinement in an agile manner during the development of the system, or
- high fidelity is a requirement and extensive approximations are not needed or cannot be made.

But then again, the approach may not be useful when

- the case is relatively simple and well-researched,
- performance of the system is the primary concern, or
- the case cannot be described by the modeling framework with enough fidelity.

From the discussion presented in this section, we conclude that the approach and methodology presented here should be applicable and relevant to a range of problems in both industry and academia.

8.3 Implications of Unifying Modeling Framework

In this section, we consider the implications of the developed modeling framework, and begin to contemplate its future developments. However, concrete future research topics are discussed in detail in the subsequent section. First of all, we address the adaptation capabilities of a system based on a unified modeling framework, and secondly, we discuss the broader possible future developments of an optimization system in the context of software product lines. In this light, we also comment on the current research within the VRP domain. Perhaps here we begin to see how this work may be a beginning of a broader research effort into understanding the complexities of implementing optimization systems.

The generic process in the decision making — that is, stating the problem, modeling the problem, solving the problem, and applying the solution into practice — is currently performed largely by an operations researcher. She captures the essentials of the problem and formulates a model from this. After this, she makes the appropriate assumptions on the problem and selects, perhaps iteratively, the solution methodology, and sets, again usually iteratively, the parameters of those methods. Finally, the obtained solution is analyzed together with domain experts and end users, and the process is repeated until a satisfying result is robustly obtained.

In the context of this process, we would like to emphasize a long-term development of expert systems, a category into which, clearly, a routing system also falls. If the system could replace the operations researcher at least in some phases of the process, not only would we decrease the overall costs of that process, but could also perform better at it. Of the phases in the process, we have already solved the problem solving part to some degree. The next candidate for automation could be the phase of adjusting the parameters, which, indeed, we have seen some promising results on. Automatically adjusting algorithms are beginning to perform this task. The next phase, however, is more complex: selection of suitable solution approach. We have witnessed only few steps into this direction. Recently, Garrido *et al.* [84] applied to the CVRP a *hyperheuristic* approach which chooses the suitable set of algorithmic components during the optimization. They subsequently used a similar approach in the VRP with continuous planning [85]. They note the adaptation capabilities of their approach and the sustained robustness over different problem configurations.

As we noted, Gendreau and Potvin [89] observed that the most effective solution methods combine elements from multiple simpler ones, and that many metaheuristics seem to converge towards a unifying framework made of few algorithmic components. They also noted that even though metaheuristics can quite easily handle the complicating constraints found in real-life applications, significant knowledge about the problem is required in developing a successful metaheuristic implementation. Now, it may turn out that this impressive number of solution methods, heuristics, metaheuristics, even hyperheuristics, and their parameters as well as the number of different cases of the VRP and their extensions begins to overwhelm the operations research experts. This means — no more than — that the computers may start to perform better in the task of selecting the appropriate solution methods and their parameters. This means, effectively, that the operations researcher may be replaced in some phases of the process by a *learning optimization system*.

The prospect of a learning system is interesting, but in order to gain enough advantage from adaptation, the system has to be able to represent the problems it should adapt to. The solution methods cannot adapt to different types of situations unless the different types can be described to them. In other words, the need for robustness is limited by the applicability of the system. It turns out that a generic framework is a suitable platform for an adaptive system. Moreover, the formal structure defined by the constructs in the metamodel can serve as a basis for techniques attempting to understand the problem instance and adapt accordingly. A number of *machine learning* and *artificial intelligence* techniques, such as statistical classification and prediction are potential approaches for accomplishing this task.

Adapting to the structure of the problem is, however, possibly difficult if there is little understanding of the dynamics of the formal structure of the optimization model. There has been a mismatch in the current modeling and solution methodology in the VRP research: the models are often described as mixed integer linear programming formulations, but solved using nonlinear graph en-

codings in a metaheuristic approach. This highlights an interesting feature: the corresponding problem structure has not been presented using a formalization that could be analyzed separately very well. Examples of ways to analyze and exploit the structure of the model include determining the type of local search operators needed in exploring the *alternative activities* in a given problem instance or analyzing dependencies between capabilities to find hubs of dependencies (e.g., vehicles and equipment), and using this information for diversification in the solution process.

We began providing understanding on the dynamics of the problem structure in this thesis by making the structure of the models explicit and by presenting a taxonomy of more abstract modeling constructs. These constructs capture the requirements for syntactically applicable algorithms. Now, we may hypothesize that much of the taxonomical work on *instance characteristics*, such as whether we are routing within a city, in an urban region, perishable goods or freight and so on, can be reduced into *feature descriptors* of different problem instances. These descriptors could then be employed by learning mechanisms which adjust themselves into the given situation, thus diminishing the need for a taxonomy for case characteristics from the algorithm point of view. This, in essence, is a method which composes semantically applicable algorithms from syntactically applicable algorithm components. Likewise, the same can be done to configurationally applicable algorithms.

As we noted earlier, one can measure and compare the applicability and robustness of different solution methods in multiple cases simultaneously within the system. This has not been done in a large scale, and it would be interesting to see whether there are patterns in applicability of different methods to different optimization problems. This work is, in fact, a prerequisite, for instance, for statistically learning algorithms, which would need to analyze data from the performance of different methods on problems with different characteristics. It is, therefore, an essential observation that the identified trend of unification does not mean that the case-specific research will be less important. On the contrary, systematic knowledge of which problems are best attacked by which methods is essential for the system to be robust in a generic setting.

The next phase we might attempt to remove the operations researcher from is the *modeling phase*. In this case, the system would not only select the solution approach, but also deduce the best way to describe the problem to the system. This would require determination of the criteria by which to choose the models, and touches the issue of *model reduction* in which one would search for the most suitable representation of sufficient fidelity. Formulating this problem in detail is outside the scope of this work, but in practical terms, in these situations the system would construct a model from available modeling elements and optimize its structure for efficient algorithmic manipulation. This approaches also the composite modeling contemplated in Chapter 4.

Finally, if we consider the metamodeling stack presented in Chapter 4, we observe that we have discussed instances, models, and metamodels in this work. There has been work on taxonomies on the instance level in the scientific litera-

ture, and here we concentrated on the model level by classifying and analyzing elements defined in the metamodel. Now, it is also possible to analyze languages for describing metamodels and to build a taxonomy of their structure. This opens up a possibility for building a system which selects the best metamodel for describing the optimization problems from a number of alternative elements. However, we note that the benefits of such an approach are yet unknown.

From the implementation perspective, we would like to emphasize the prevailing theme of this work: *implementability* and *analysis of the structure* of optimization systems. There are two lines of advances taking place in vehicle routing: academic and practical. Although many would argue this is the case in every field, we claim that in this field there is an additional gap due to the complexities involved in implementation. The academic research on how to implement industry-strength software for this particular domain is missing. This may suggest that the future research should provide insight into the implementation of optimization systems in general.

Our contributions included not only the — relatively obvious — idea of applying product line architectures into vehicle routing, but also the description of *how* to construct one. This approach is promising, and we may be able to achieve a system capable of automatically addressing a broad set of problems. This development implies that we might see a trend where research is pushed towards metamodeling of optimization problems. The product line approach has also a possibility to grow into a collection of models, algorithms, metaheuristics, and hyperheuristics; and as the product line evolves, its set of tools becomes larger, thus further decreasing the effort needed in modeling and solving new problem variants. We argue that it is also easier to construct a learning optimization system within a product line due to its systematic approach to variability modeling. In software engineering, these types of product lines are often referred to as "dynamic" or "self-adaptive" product lines, or described as "context-oriented". Such a systematic approach to software engineering may benefit the research also in the field of operations research.

As to the discussion on solution methods and their implementation in the context of a heterogeneous set of optimization models, we do not yet know which is the best approach. As mentioned, there are two primary strategies: *a single method* capable of adapting itself altogether into a given situation, and *a multitude of methods* from which we choose the most suitable ones. In the former approach, the algorithm itself must have enough intelligence to query the properties of the problem and this may increase the complexity of the method. On the other hand, selecting the best algorithm may not be trivial. Fortunately, this choice is not limited by our approach to implementation; either technique can be employed within a product line. We also note that if a hyperheuristic solution methodology based, e.g., on machine learning can be constructed, these two approaches begin to converge. Thus, we assume that the first step will be a multitude of methods, and the second step will later combine these under a smaller set of hyperheuristic approaches.

As the last point we observe that a major shift in the world of computation

has been taking place during the last couple of years, and this move towards increasingly parallel computing will change the landscape also in combinatorial optimization. The parallelization of algorithms will become a necessity if we wish to continue taking advantage of the increasing computational power. In this work, however, we settle for a brief comment on parallelization of solution methods within the developed modeling framework. The partition-based encoding provides possibilities for parallelizing also on the *Ansatz* level, which means that the possibility for running several algorithms simultaneously on a single *Ansatz* has been taken into consideration in the structure of the system. This allows parallelization of the solution process on several levels. Unfortunately, a more detailed analysis of parallelization of the optimization system is outside the scope of this work.

When contrasting our approach to previous work, especially the focus on reusability and easiness of configuration on a generic framework differentiates the prior approaches from that of ours. A clear exception is the mentioned work on the "GREENTRIP" project [101] in which a similar (albeit simpler) metamodel for routing problems was developed. The project resulted in a generic toolkit for configuring applications, but its commercialization was deemed to be too expensive [102] and its details have not been published. The generic toolkit has similarities to the product line approach, but product lines often take a more comprehensive view on the development. A configurable architecture is one (although important) asset in the product line. In this respect, our approach included, e.g., the definition of a domain-specific modeling language for additional flexibility in model definition. The novel approach of employing a model transformation as a variation point is a key difference to earlier approaches. Our approach allows the expression of complex dependencies between the model elements, including dependencies which often cannot easily be expressed using a tool. In addition, in SPLs the ability to implement and integrate completely new components within the application development process enhances the flexibility of our approach.

To conclude, we note that combining software engineering with operations research in this way has not been — to this extent at least — attempted before. There are at least two possible explanations. Either, for some reason, the combination of these topics has required advances in both fields and has just recently become realizable, or the approach is not well suited to this area, e.g., due to excessive overhead and strict performance requirements in solving the problems in practice. Either way, we are faced with a difficult task of implementing a performance sensitive expert system in a heterogeneous problem domain, and if we fail to provide definitive evidence of the suitability of this approach in the future in all cases, we hope to provide a relevant approach at least for a subset of the domain. Moreover, the prospect of a learning system capable of adapting to the problem at hand, almost by definition, requires a way to represent the different problems in a unified way, and we may even have opened a new problem area which may be worth examining. We hope that this research at least invites the possibility and pushes the boundary of routing and scheduling research in theory, if not in practice.

Having said all this, we are excited about the results thus far. In a sense, we have introduced a new problem, a complex and dynamic VRP variant. We may not be able to solve the problem completely yet, but there are no obvious reasons for not continuing the research on the subject. In fact, based on the discussion here and in this work as a whole, our prediction for the future of VRP research is that metaheuristic (and later hyperheuristic) solution methods continue to dominate pure exact methods[1] in both solution quality and the computational time needed, and the robustness will be achieved by applying machine learning and statistics on a common platform to a collection of algorithms, heuristic and metaheuristic, in a distributed and parallel environment. We will also continue to develop more complicated and integrated models and solve larger instances by splitting and distributing computing. In this light, the product line approach and a generic modeling framework provide, arguably, a reasonable base for future research.

8.4 Further Research

Although we attempted to sufficiently cover relevant subsets of two fields, there are, naturally, several exceedingly important areas we omitted from this work. Furthermore, during the course of this research, we raised a few new questions altogether. This section aims to address both of these topics by discussing concrete suggestions for further studies in the areas of combinatorial optimization and software engineering, and, especially, their intersection.

The topics for future research may be divided into five major areas. Firstly, we address the question of suitable solution methodology and its usage in the context of a generic modeling framework. Secondly, we examine some of the new possibilities for employing the modeling tools in practice. Thirdly, we list areas for future research on implementation, especially from the software engineering point of view. Fourthly, we comment on the possibility of further studying and developing the metamodel and metamodeling in general. And finally, we concretize the research topics for the adaptive and learning optimization systems discussed in the preceding section.

Perhaps the most pressing open question is the issue of solution methodology implementation in the context of *reuse* and — perhaps in the future — *adaptation*. The three strategies discussed in the context of problem structure were, in essence, points on a scale between reusability and performance. Some initial work has been done on the subject, and we hope to discuss experiences from implementing generic adaptive operators in the future. One of the potential approaches is a separation of the search neighborhood from the search operator. Although counter-intuitive, the approach may be able to differentiate between the *scope* of the neighborhood and its *logic of traversal*. The former would be dictated by the problem variant, and the latter by the search operator. This approach

[1] As opposed to metaheuristics with, e.g., exact methods as components.

may enable considerable reuse on the search operator level. Another relevant research topic from the solution methodology viewpoint is, as mentioned, the evaluation of existing advanced strategies for implementing local search: usage of, for example, *segment concatenation* and *sequential search* in the context of the developed metamodel. In addition, one interesting area is that of algorithm performance on a generic optimization model; we should evaluate different metaheuristics and local search operations in a common platform to study the effects of different problem characteristics (such as constraint types and tightness) on the solution method performance. Moreover, we should investigate how to incorporate allowing infeasibilities during the search within a modeling framework. In addition, in this context, one promising topic worth examining is the usage of *population metaheuristics*. Key issues here include how to combine two or more feasible *Ansätze* into a new feasible *Ansatz*. The performance of also these methods should be evaluated in a generic setting.

From the modeling perspective, we should first investigate the limits of the modeling framework and attempt to model *more complex variants* overlooked in this work, such as the periodic routing problem. Also, the prospect of including *job scheduling* might be worth investigating. The framework should be able to represent scheduling problems without modifications but this should be verified. To generalize further, we might be interested in the applicability of a generic modeling framework in combinatorial optimization in general. If the automated adaptation of the system can be driven far enough, there should be no theoretical limit for the applicability of the system in the domain of combinatorial optimization. At which point the implementation issues of such a system will prevent further developments in this area is an open question. In addition, a major area of research is the modeling of *stochasticity* within the framework. Modeling uncertainty has the potential of describing the problem instances more realistically, as the real life is, indeed, uncertain. Finally, the option for *multiple criteria optimization* will most likely be relevant also in the context of route optimization, and even though the topic was completely neglected in this work, a generic framework should provide a suitable platform also in this perspective.

From the software engineering point of view, there are two major areas which would benefit from further studies. Firstly, the *product line approach* could be refined using, e.g, the aspect-oriented approach, and studied further; and secondly, *further research on the solver layer*, e.g., in the form of parallelization should be investigated.

To begin with, the further research on the product line approach could make the *model transformation* more aspect-oriented. This may aid in achieving more flexibility and reuse at the model transformation level, making it easier to configure the model at runtime by enabling selection of a given set of relevant features. This, in turn, enables interactive adjustment of the model fidelity, which then can be used, for example, to obtain useful approximations during the optimization process. More control over the aspects of the model may also assist in evaluating and comparing different scenarios, and in performing sensitivity analysis over the features of the model. Another area on the product line approach are

the effects of *variation*. Variation can affect the qualities of the software, and this is an especially relevant concern in routing systems where many attributes are sensitive to changes in variants. There has been some work on systematic evaluation of the effects of variability on quality attributes, but none in the context of optimization systems.

On the solver layer, parallelization in the context of a generic model framework may be worth examining, addressing especially strategies for obtaining the needed robustness by, e.g., running different optimization algorithms against each other. There are also possibilities for improvement in the memory management within the sparse constraint matrices. In this work, we used a relatively simple approach, and the system could benefit from refinement in this area. Finally, we note that one unexplored area is the usage of so-called hybrid metaheuristics, which employ linear or constraint programming solvers within the metaheuristic search. As the context is a framework capable of representing problems with relatively complex combinations of constraints, combining the strengths of constraint programming and metaheuristic search may be worth investigating.

The metamodeling approach to routing could benefit from a detailed examination of different model representations from a theoretical perspective. It should be interesting to examine, for example, how to prove two or more models of the same metamodel equivalent. Altering the optimization model without changing its expressiveness is a requirement for the system to be able to construct and compare optimization models and search for the best problem representation for a given problem. In other words, the system would adapt its *encoding* to the problem automatically. From a more practical viewpoint, the developed metamodel should be subject to further refinement. *Inter-route constraints*, for instance, are prime candidates for future development. In addition, *increasing the expressiveness of the grouping constraints* by dropping the simple grouping rules may allow more complex relationships to be modeled. The taxonomy of the modeling elements discussed should be able to classify these new elements in a relatively straightforward way. A concrete topic for research could then be defining a *process* for expanding the metamodel, that is, constructing a framework for metamodel development. These enhancements are, however, likely to make solving of the problem more complex computationally. In this context, the main question in expanding the metamodel is perhaps the issue of necessity: at which point adding complexity to a model stops increasing the value of optimization result through model fidelity in the cases of practical routing occurring today. In other words, is there a point at which, for instance, modeling cost structures in more detail does not yield any more applicable optimization results and thus realized savings in practice? A system with the flexibility to adjust these variables easily will hopefully allow us to examine this issue in more detail than has previously been possible.

To conclude, we highlight some research topics from the discussion in the preceding section. Adaptation capabilities should continue to be investigated, and especially statistical classification and prediction techniques may provide the mathematical foundation for such methods. A concrete step in developing a

learning algorithm would be identifying the characteristics of the problems that affect the suitable algorithms and searching for patterns in the behavior of these methods. Moreover, investigation of automatic parameter selection may begin from simple methods such as construction heuristics and continue to selection of local search operators and later metaheuristics. These types of methods may be able to adapt to a wider variety of situations than has previously been possible. Indeed, the prospect of such a system has lead us to formulate a generic goal for a learning routing system: *to be able to feed the system any benchmark instance in routing (CVRP, VRPTW, VRPC, PDP, OVRP, TSPPD, etc.) defined in the scientific literature and get good results in a reasonable time without any adjustments needed in parameters, algorithms, or any other aspect of the system.* Although somewhat vaguely defined, the prospect of such a system is exciting, and the research on these areas should provide interesting challenges for both fields of combinatorial optimization and software engineering also in the future.

YHTEENVETO (FINNISH SUMMARY)

Tässä väitöskirjassa tarkastellaan menetelmiä joustavien reitinoptimointijärjestelmien toteuttamiseksi kustannustehokkaasti. Väitöskirjan suomenkielinen otsikko on "Metamallit ja metaheuristiikat — mallinnuskieli ja ohjelmistoarkkitehtuurituotantolinja reitinoptimointijärjestelmille".

Useat logistiset suunnitteluongelmat voidaan mallintaa reitinoptimointiongelmina ja ratkaista algoritmisesti. Esimerkiksi postinjakelun, koulukuljetusten, jätehuollon ja raaka-aineiden laivausten toiminnan tehokkuutta on pyritty parantamaan mallintamalla ongelma tietokoneelle ja hyödyntämällä optimointialgoritmeja.

Reitinoptimointiongelma on kombinatorinen optimointiongelma ja laskennallisesti vaativa erityisesti reaalimaailmassa esiintyvien ongelmien kokoluokassa. Tyypillisesti näitä ongelmia ei voida ratkoa tarkasti vaan on käytettävä niin sanottuja heuristisia menetelmiä.

Reitinoptimointiongelmia on useita eri tyyppejä ja ne eroavat toisistaan vaihtelevissa määrin. Perinteisesti eri ongelmavariantteja on ratkaistu kuhunkin tilanteeseen räätälöidyillä optimointisovelluksilla sekä heuristisilla ratkaisumenetelmillä, mikä nostaa ongelman ratkaisemisen kokonaiskustannuksia.

Tämän työn tavoitteena oli toteuttaa reitinoptimointijärjestelmä, joka soveltuu samanaikaisesti useammantyyppisten optimointiongelmien kuvaamiseen ja ratkaisemiseen mahdollisimman vähin muutoksin. Työssä esitellään metamalli erityyppisten reitinoptimointiongelmien mallien kuvaamiseen ja kuvataan kehitetty ohjelmistoarkkitehtuurituotantolinja, jonka avulla merkittävä osa toteutuksesta voidaan uudelleenkäyttää pienin muutoksin.

Kehitettyä metamallia arvioidaan sen ilmaisuvoiman näkökulmasta ja sen todetaan soveltuvan erityyppisten käytännössä esiintyvien monimutkaisten reitinoptimointiongelmien kuvaamiseen. Lisäksi metamallin implementaatio arvioitiin teoreettisesti laskennallisesti tehokkaaksi olemassa olevien ratkaisualgoritmien puitteissa. Toteutettua tuotantolinjaa tarkastellaan arkkitehtuurin laatuattribuuttien näkökulmasta ja sen voidaan perustellusti sanoa olevan rakenteeltaan joustava ja uudelleenkäytettävä. Alustavat numeeriset testit viittaavat siihen, että kehitetty lähestymistapa tuo laskentaan jonkin verran kiinteitä kustannuksia, mutta tätä pystyttäneen tulevaisuudessa vähentämään tehokkaammalla implementaatiolla.

Saavutettu järjestelmän kokonaisrakenne mahdollistaa menetelmien riippumattomuuden käytetystä mallista, luo mahdollisuuden aikaisempaa laajemmalle uudelleenkäytölle ja vähentää työtä, joka vaaditaan optimointijärjestelmän soveltamisessa uusiin ongelmiin. Kehitetty lähestymistapa on huomionarvoinen vaihtoehto, kun tarvitaan erityisesti joustavia optimointijärjestelmiä. Lähestymistapaa voitaneen tulevaisuudessa täydentää automaattisilla oppimismekanismeilla, jotka säätävät järjestelmää itsenäisesti kulloiseenkin tilanteeseen sopivaksi.

REFERENCES

[1] J.-R. Abrial, S. A. Schuman and B. Meyer, *A Specification Language*, in A. M. Macnaghten and R. M. McKeag (eds.), *On the Construction of Programs*, pp. 343–410, Cambridge University Press, Cambridge, UK, 1980, ISBN 0-521-23090-X

[2] V. Alves, D. Schneider, M. Becker and N. Bencomo, *Comparative Study of Variability Management in Software Product Lines and Runtime Adaptable Systems*, in *Proceedings of the 3rd International workshop on Variability Modelling of Software-Intensive Systems*, Online proceedings, 2009

[3] D. L. Applegate, R. E. Bixby, V. Chvatal and W. J. Cook, *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*, Princeton University Press, Princeton, NJ, USA, 2007, ISBN 0691129932, 9780691129938

[4] C. Archetti and M. G. Speranza, *The Split Delivery Vehicle Routing Problem: A Survey*, in *The Vehicle Routing Problem: Latest Advances and New Challenges*, *Operations Research/Computer Science Interfaces Series*, volume 43, pp. 103–122, Springer US, 2008, ISBN 978-0-387-77777-1

[5] P. Augerat, J. M. Belenguer, E. Benavent, A. Corberán and D. Naddef, *Separating Capacity Constraints in the CVRP Using Tabu Search*, in *European Journal of Operational Research*, volume 106(2–3), pp. 546–557, 1998, ISSN 0377-2217

[6] N. Azi, M. Gendreau and J.-Y. Potvin, *An Exact Algorithm for a Vehicle Routing Problem with Time Windows and Multiple Use of Vehicles*, in *European Journal of Operational Research*, volume 202(3), pp. 756–763, 2010, ISSN 0377-2217

[7] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh and A. Vilbig, *A Meta-model for Representing Variability in Product Family Development*, in *Software Product-Family Engineering*, *Lecture Notes in Computer Science*, volume 3014, pp. 66–80, Springer Berlin / Heidelberg, 2004, ISSN 0302-9743

[8] B. D. Backer, V. Furnon, P. Kilby, P. Prosser and P. Shaw, *Local Search in Constraint Programming: Application to the Vehicle Routing Problem*, in *Proceedings of CP-97 Workshop on Industrial Constraint-Directed Scheduling*, pp. 1–15, Online proceedings, Schloss Hagenberg, Austria, 1997

[9] B. D. Backer, V. Furnon, P. Shaw, P. Kilby and P. Prosser, *Solving Vehicle Routing Problems Using Constraint Programming and Metaheuristics*, in *Journal of Heuristics*, volume 6(4), pp. 501–523, 2000, ISSN 1381-1231

[10] E. Balas, M. Fischetti and W. R. Pulleyblank, *The Precedence-constrained Asymmetric Traveling Salesman Polytope*, in *Mathematical Programming*, volume 68(3), pp. 241–265, 1995, ISSN 0025-5610

[11] R. Baldacci, M. Battarra and D. Vigo, *Routing a Heterogeneous Fleet of Vehicles*, in *The Vehicle Routing Problem: Latest Advances and New Challenges*, *Computer Science Interfaces*, volume 43, pp. 3–27, Springer US, 2008, ISBN 978-0-387-77777-1

[12] R. Baldacci, N. Christofides and A. Mingozzi, *An Exact Algorithm for the Vehicle Routing Problem Based on the Set Partitioning Formulation with Additional Cuts*, in *Mathematical Programming*, volume 115(2), pp. 351–385, 2008, ISSN 0025-5610

[13] J. F. Bard, G. Kontoravdis and G. Yu, *A Branch-and-Cut Procedure for the Vehicle Routing Problem with Time Windows*, in *Transportation Science*, volume 36(2), pp. 250–269, 2002, ISSN 1526-5447

[14] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003, ISBN 0321154959

[15] M. Becker, *Towards a General Model of Variability in Product Families*, in *Proceedings of the First Workshop on Software Variability Management*, Online proceedings, Groningen, The Netherlands, 2003

[16] E. J. Beltrami and L. D. Bodin, *Networks and Vehicle Routing for Municipal Waste Collection*, in *Networks*, volume 4(1), pp. 65–94, 1974

[17] G. Berbeglia, J.-F. Cordeau and G. Laporte, *A Hybrid Tabu Search and Constraint Programming Algorithm for the Dynamic Dial-a-Ride Problem*, submitted to *INFORMS Journal on Computing*

[18] L. Bertazzi, M. Savelsbergh and M. G. Speranza, *Inventory Routing*, in *The Vehicle Routing Problem: Latest Advances and New Challenges*, *Operations Research/Computer Science Interfaces Series*, volume 43, pp. 49–72, Springer US, 2008, ISBN 978-0-387-77777-1

[19] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev and A. Lindow, *Model Transformations? Transformation Models!*, in *Model Driven Engineering Languages and Systems*, *Lecture Notes in Computer Science*, volume 4199, pp. 440–453, Springer Berlin / Heidelberg, 2006, ISBN 978-3-540-45772-5, ISSN 0302-9743

[20] A. Bockmayr and J. N. Hooker, *Constraint Programming*, in K. Aardal, G. Nemhauser and R. Weismantel (eds.), *Discrete Optimization*, *Handbooks in Operations Research and Management Science*, volume 12, pp. 559–600, Elsevier, 2005

[21] L. Bodin, B. Golden, A. Assad and M. Ball, *Routing and Scheduling of Vehicles and Crews — The State of the Art*, in *Computers & Operations Research*, volume 10(2), pp. 63–212, 6 1983

[22] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. H. Obbink and K. Pohl, *Variability Issues in Software Product Lines*, in *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pp. 13–21, Springer-Verlag, London, UK, 2002, ISBN 3-540-43659-6

[23] J. Bosch, H. Obbink and A. Maccari, *Research Topics and Future Trends*, in *Software Product-Family Engineering*, *Lecture Notes in Computer Science*, volume 3014, pp. 1–5, Springer Berlin / Heidelberg, 2004, ISBN 978-3-540-21941-5, ISSN 0302-9743

[24] J. Brandão and A. Mercer, *The Multi-Trip Vehicle Routing Problem*, in *The Journal of the Operational Research Society*, volume 49(8), pp. 799–805, 1998, ISSN 01605682

[25] J. Brandão, *A Tabu Search Algorithm for the Open Vehicle Routing Problem*, in *European Journal of Operational Research*, volume 157(3), pp. 552–564, 2004, ISSN 0377-2217

[26] J. Brandão, *A New Tabu Search Algorithm for the Vehicle Routing Problem with Backhauls*, in *European Journal of Operational Research*, volume 173(2), pp. 540–555, 2006, ISSN 0377-2217

[27] J. Brandão, *A Deterministic Tabu Search Algorithm for the Fleet Size and Mix Vehicle Routing Problem*, in *European Journal of Operational Research*, volume 195(3), pp. 716–728, 2009, ISSN 0377-2217

[28] J. Brandão and A. Mercer, *A Tabu Search Algorithm for the Multi-trip Vehicle Routing and Scheduling Problem*, in *European Journal of Operational Research*, volume 100(1), pp. 180–191, 1997, ISSN 0377-2217

[29] O. Bräysy, W. Dullaert, G. Hasle, D. Mester and M. Gendreau, *An Effective Multirestart Deterministic Annealing Metaheuristic for the Fleet Size and Mix Vehicle-Routing Problem with Time Windows*, in *Transportation Science*, volume 42(3), pp. 371–386, 2008, ISSN 1526-5447

[30] O. Bräysy and M. Gendreau, *Vehicle Routing Problem with Time Windows, Part I: Route Construction and Local Search Algorithms*, in *Transportation Science*, volume 39(1), pp. 104–118, 2005, ISSN 1526-5447

[31] O. Bräysy and M. Gendreau, *Vehicle Routing Problem with Time Windows, Part II: Metaheuristics*, in *Transportation Science*, volume 39(1), pp. 119–139, 2005, ISSN 1526-5447

[32] F. P. Brooks, *No Silver Bullet — Essence and Accident in Software Engineering*, in *Proceedings of IFIP Tenth World Computing Conference*, pp. 1069–1076, 1986

[33] L. Brownsword and P. Clements, *A Case Study in Successful Product Line Development*, Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1996

[34] A. M. Campbell and B. W. Thomas, *Challenges and Advances in A Priori Routing*, in *The Vehicle Routing Problem: Latest Advances and New Challenges*, *Operations Research/Computer Science Interfaces Series*, volume 43, pp. 123–142, Springer US, 2008, ISBN 978-0-387-77777-1

[35] P. J. Cassidy and H. S. Bennett, *TRAMP — A Multi-Depot Vehicle Scheduling System*, in *Operational Research Quarterly (1970-1977)*, volume 23(2), pp. 151–163, 1972, ISSN 00303623

[36] A. Ceselli, G. Righini and M. Salani, *A Column Generation Algorithm for a Rich Vehicle-Routing Problem*, in *Transportation Science*, volume 43(1), pp. 56–69, 2009

[37] L. Chen, M. Ali Babar and N. Ali, *Variability Management in Software Product Lines: A Systematic Review*, in *SPLC '09: Proceedings of the 13th International Software Product Line Conference*, pp. 81–90, Carnegie Mellon University, Pittsburgh, PA, USA, 2009

[38] M. Christiansen, K. Fagerholt, B. Nygreen and D. Ronen, *Chapter 4 Maritime Transportation*, in C. Barnhart and G. Laporte (eds.), *Transportation*, *Handbooks in Operations Research and Management Science*, volume 14, pp. 189–284, Elsevier, 2007

[39] G. Clarke and J. W. Wright, *Scheduling of Vehicles from a Central Depot to a Number of Delivery Points*, in *Operations Research*, volume 12(4), pp. 568–581, 1964, ISSN 0030364X

[40] A. Classen, A. Hubaux, F. Sanen, E. Truyen, J. Vallejos, P. Costanza, W. De Meuter, P. Heymans and W. Joosen, *Modelling Variability in Self-Adaptive Systems: Towards a Research Agenda*, in *Proceedings of International Workshop on Modularization, Composition and Generative Techniques for Product-Line Engineering*, pp. 19–26, Online proceedings, 2008

[41] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley Professional, 3rd edition, August 2001, ISBN 0201703327

[42] T. Colburn and G. Shute, *Abstraction in Computer Science*, in *Minds and Machines*, volume 17(2), pp. 169–184, 2007, ISSN 0924-6495

[43] J.-F. Cordeau, G. Desaulniers, J. Desrosiers, M. M. Solomon and F. Soumis, *VRP with Time Windows*, in P. Toth and D. Vigo (eds.), *The Vehicle Routing Problem*, pp. 157–193, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001, ISBN 0-89871-498-2

[44] J.-F. Cordeau, G. Laporte and A. Mercier, *A Unified Tabu Search Heuristic for Vehicle Routing Problems with Time Windows*, in *The Journal of the Operational Research Society*, volume 52(8), pp. 928–936, 2001, ISSN 01605682

[45] J.-F. Cordeau, M. Gendreau, A. Hertz, G. Laporte and J.-S. Sormany, *New Heuristics for the Vehicle Routing Problem*, in *Logistics Systems: Design and Optimization*, pp. 279–297, Springer US, 2005, ISBN 978-0-387-24971-1

[46] J.-F. Cordeau, M. Gendreau and G. Laporte, *A Tabu Search Heuristic for Periodic and Multi-depot Vehicle Routing Problems*, in *Networks*, volume 30(2), pp. 105–119, 1997

[47] J.-F. Cordeau, G. Laporte, J.-Y. Potvin and M. W. Savelsbergh, *Chapter 7 Transportation on Demand*, in C. Barnhart and G. Laporte (eds.), *Transportation*, *Handbooks in Operations Research and Management Science*, volume 14, pp. 429–466, Elsevier, 2007

[48] J.-F. Cordeau, G. Laporte, M. W. Savelsbergh and D. Vigo, *Chapter 6 Vehicle Routing*, in C. Barnhart and G. Laporte (eds.), *Transportation*, *Handbooks in Operations Research and Management Science*, volume 14, pp. 367–428, Elsevier, 2007

[49] P. Costanza, *Context-Oriented Programming in ContextL: State of the Art*, in *LISP50: Celebrating the 50th Anniversary of Lisp*, pp. 1–5, ACM, New York, NY, USA, 2008, ISBN 978-1-60558-383-9

[50] B. Crevier, J.-F. Cordeau and G. Laporte, *The Multi-Depot Vehicle Routing Problem with Inter-Depot Routes*, in *European Journal of Operational Research*, volume 176(2), pp. 756–773, 2007, ISSN 0377-2217

[51] G. A. Croes, *A Method for Solving Traveling-Salesman Problems*, in *Operations Research*, volume 6(6), pp. 791–812, 1958

[52] K. Czarnecki, M. Antkiewicz, C. H. P. Kim, S. Lau and K. Pietroszek, *Model-Driven Software Product Lines*, in *OOPSLA '05: Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 126–127, ACM, New York, NY, USA, 2005, ISBN 1-59593-193-7

[53] K. Czarnecki and S. Helsen, *Classification of Model Transformation Approaches*, 2003, unpublished

[54] G. B. Dantzig and J. H. Ramser, *The Truck Dispatching Problem*, in *Management Science*, volume 6(1), pp. 80–91, 1959, ISSN 00251909

[55] G. Dantzig and D. Fulkerson, *Minimizing the Number of Tankers to Meet a Fixed Sschedule*, in *Naval Research Logistics Quarterly*, volume 1, pp. 217–222, 1954

[56] S. Deelstra, M. Sinnema, J. V. Gurp and J. Bosch, *Model Driven Architecture as Approach to Manage Variability in Software Product Families*, in *Proceedings of the Workshop on Model Driven Architectures: Foundations and Applications*, pp. 109–114, Springer, 2003

[57] I. Deif and L. Bodin, *Extension of the Clarke and Wright Algorithm for Solving the Vehicle Routing Problem with Backhauling*, in K. A (ed.), *Proceedings of the Babson Conference on Software on Uses in Transportation and Logistic Management*, pp. 75–96, Babson Park, 1984

[58] U. Derigs, J. Gottlieb, J. Kalkoff, M. Piesche, F. Rothlauf and U. Vogel, *Vehicle Routing with Compartments: Applications, Modelling and Heuristics*, in *OR Spectrum*, February 2010, ISSN 0171-6468

[59] G. Desaulniers, J. Desrosiers, I. Ioachim, M. M. Solomon, F. Soumis and D. Villeneuve, *A Unified Framework for Deterministic Time Constrained Vehicle Routing and Crew Scheduling Problems*, in T. G. Crainic and G. Laporte (eds.), *Fleet Management and Logistics*, pp. 57–94, Springer-Verlag, New York, 1998, ISBN 0792381610

[60] J. Desrosiers, Y. Dumas, M. M. Solomon and F. Soumis, *Time Constrained Routing and Scheduling*, in C. M. G. N. M.O. Ball, T.L. Magnanti (ed.), *Handbooks in Operations Research and Management Science*, 8, pp. 35–139, Elsevier Science Publishers, 1995

[61] M. Dorigo, *Optimization, Learning and Natural Algorithms*, Ph.D. thesis, Politecnico di Milano, Italy, 1992

[62] M. Dror, *Note on the Complexity of the Shortest Path Models for Column Generation in VRPTW*, in *Operations Research*, volume 42(5), pp. 977–978, 1994

[63] L. M. A. Drummond, L. S. Ochi and D. S. Vianna, *An Asynchronous Parallel Metaheuristic for the Period Vehicle Routing Problem*, in *Future Generation Computer Systems*, volume 17(4), pp. 379–386, 2001, ISSN 0167-739X

[64] Y. Dumas, J. Desrosiers and F. Soumis, *The Pickup and Delivery Problem with Time Windows*, in *European Journal of Operational Research*, volume 54(1), pp. 7–22, 1991

[65] I. Dumitrescu, S. Ropke, J.-F. Cordeau and G. Laporte, *The Traveling Salesman Problem with Pickup and Delivery: Polyhedral Results and a Branch-and-Cut Algorithm*, in *Mathematical Programming*, volume 121(2), pp. 269–305, 2009, ISSN 0025-5610

[66] B. Eksioglu, A. V. Vural and A. Reisman, *The Vehicle Routing Problem: A Taxonomic Review*, in *Computers & Industrial Engineering*, volume 57(4), pp. 1472–1483, 2009

[67] L. M. Ellram and S. P. Siferd, *Total Cost of Ownership: A Key Concept in Strategic Cost Management Decisions*, in *Journal of Business Logistics*, volume 19(1), pp. 55–84, 1998

[68] J. Euchi and H. Chabchoub, *A Hybrid Tabu Search to Solve the Heterogeneous Fixed Fleet Vehicle Routing Problem*, in *Logistics Research*, April 2010, ISSN 1865-035X

[69] A. E. Fallahi, C. Prins and R. W. Calvo, *A Memetic Algorithm and a Tabu Search for the Multi-compartment Vehicle Routing Problem*, in *Computers & Operations Research*, volume 35(5), pp. 1725–1741, 2008, ISSN 0305-0548

[70] M. Fayad and D. C. Schmidt, *Object-Oriented Application Frameworks*, in *Communications of the ACM*, volume 40(10), pp. 32–38, 1997, ISSN 0001-0782

[71] R. E. Filman, T. Elrad, S. Clarke and M. Akşit (eds.), *Aspect-Oriented Software Development*, Addison-Wesley, 2004, ISBN 0-321-21976-7

[72] M. Fisher, *Chapter 1 Vehicle Routing*, in M. Ball, T. Magnanti, C. Monma and G. Nemhauser (eds.), *Network Routing*, *Handbooks in Operations Research and Management Science*, volume 8, pp. 1–33, Elsevier, 1995

[73] K. Fleszar, I. H. Osman and K. S. Hindi, *A Variable Neighbourhood Search Algorithm for the Open Vehicle Routing Problem*, in *European Journal of Operational Research*, volume 195(3), pp. 803–809, 2009, ISSN 0377-2217

[74] M. Fowler, *Inversion of Control Containers and the Dependency Injection pattern*, January 2004, `http://www.itu.dk/courses/VOP/E2005/VOP2005E/8_injection.pdf`

[75] M. Fowler, *Domain-Specific Languages*, Addison-Wesley, 2010, ISBN 0-321-71294-3

[76] W. B. Frakes and K. Kang, *Software Reuse Research: Status and Future*, in *IEEE Transactions on Software Engineering*, volume 31, pp. 529–536, 2005

[77] P. M. Francis, K. R. Smilowitz and M. Tzur, *The Period Vehicle Routing Problem and its Extensions*, in *The Vehicle Routing Problem: Latest Advances and New Challenges*, *Operations Research/Computer Science Interfaces Series*, volume 43, pp. 73–102, Springer US, 2008, ISBN 978-0-387-77777-1

[78] G. Freeman, D. Batory and G. Lavender, *Lifting Transformational Models of Product Lines: A Case Study*, in *Theory and Practice of Model Transformations*, *Lecture Notes in Computer Science*, volume 5063, pp. 16–30, Springer Berlin / Heidelberg, 2008, ISBN 978-3-540-69926-2, ISSN 0302-9743

[79] R. Fukasawa, H. Longo, J. Lysgaard, M. P. d. Aragão, M. Reis, E. Uchoa and R. F. Werneck, *Robust Branch-and-Cut-and-Price for the Capacitated Vehicle Routing Problem*, in *Mathematical Programming*, volume 106(3), pp. 491–511, 2006, ISSN 0025-5610

[80] B. Funke, T. Grünert and S. Irnich, *Local Search for Vehicle Routing and Scheduling Problems: Review and Conceptual Integration*, in *Journal of Heuristics*, volume 11(4), pp. 267–306, 2005, ISSN 1381-1231

[81] Y. Gajpal and P. Abad, *Multi-ant Colony System (MACS) for a Vehicle Routing Problem with Backhauls*, in *European Journal of Operational Research*, volume 196(1), pp. 102–117, 2009, ISSN 0377-2217

[82] M. Gamache, F. Soumis, G. Marquis and J. Desrosiers, *A Column Generation Approach for Large-Scale Aircrew Rostering Problems*, in *Operations Research*, volume 47(2), pp. 247–263, 1999

[83] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*, W. H. Freeman & Co Ltd, January 1979, ISBN 0716710455

[84] P. Garrido and C. Castro, *Stable Solving of CVRPs Using Hyperheuristics*, in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pp. 255–262, ACM, New York, NY, USA, 2009, ISBN 978-1-60558-325-9

[85] P. Garrido and M. Riff, *DVRP: a Hard Dynamic Combinatorial Optimisation Problem Tackled by an Evolutionary Hyper-Heuristic*, in *Journal of Heuristics*, volume 16, pp. 795–834, 2010, ISSN 1381-1231

[86] M. Gendreau, *Constraint Programming and Operations Research: Comments from an Operations Researcher*, in *Journal of Heuristics*, volume 8(1), pp. 19–24, 2002, ISSN 1381-1231

[87] M. Gendreau, A. Hertz and G. Laporte, *The Traveling Salesman Problem with Backhauls*, in *Computers & Operations Research*, volume 23(5), pp. 501–508, 1996, ISSN 0305-0548

[88] M. Gendreau, A. Hertz, G. Laporte and M. Stan, *A Generalized Insertion Heuristic for the Traveling Salesman Problem with Time Windows*, in *Operations Research*, volume 46(3), pp. 330–335, 1998

[89] M. Gendreau and J.-Y. Potvin, *Metaheuristics in Combinatorial Optimization*, in *Annals of Operations Research*, volume 140(1), pp. 189–213, 2005

[90] M. Gendreau, J.-Y. Potvin, O. Bräysy, G. Hasle and A. Løkketangen, *Metaheuristics for the Vehicle Routing Problem and Its Extensions: A Categorized Bibliography*, in *The Vehicle Routing Problem: Latest Advances and New Challenges*, *Operations Research/Computer Science Interfaces Series*, volume 43, pp. 143–169, Springer US, 2008, ISBN 978-0-387-77777-1

[91] F. Glover, *Tabu search — Part I*, in *ORSA Journal on Computing*, volume 1, pp. 190–206, 1989

[92] F. Glover, *Heuristics for Integer Programming Using Surrogate Constraints*, in *Decision Sciences*, volume 8(1), pp. 156–166, 1977

[93] A. Goel, *Vehicle Scheduling and Routing with Drivers' Working Hours*, in *Transportation Science*, volume 43(1), pp. 17–26, 2009, ISSN 1526-5447

[94] B. Golden, A. Assad, L. Levy and F. Gheysens, *The Fleet Size and Mix Vehicle Routing Problem*, in *Computers & Operations Research*, volume 11(1), pp. 49–66, 1984, ISSN 0305-0548

[95] B. L. Golden, S. Raghavan and E. A. Wasil (eds.), *The Vehicle Routing Problem: Latest Advances and New Challenges*, Springer, New York, 2008, ISBN 978-0-387-77777-1

[96] H. Gomaa and D. L. Webber, *Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model*, in *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*, 2004

[97] B. González-Baixauli, M. A. Laguna and Y. Crespo, *Product Lines, Features, and MDD*, in *Proceedings of EWMT Workshop*, Online proceedings, 2005

[98] C. Groër, B. Golden and E. Wasil, *A Library of Local Search Heuristics for the Vehicle Routing Problem*, in *Mathematical Programming Computation*, volume 2, pp. 79–101, 2010, ISSN 1867-2949

[99] J. V. Gurp, J. Bosch and M. Svahnberg, *On the Notion of Variability in Software Product Lines*, in *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, p. 45, IEEE Computer Society, Washington, DC, USA, 2001, ISBN 0-7695-1360-3

[100] G. Halmans and K. Pohl, *Communicating the Variability of a Software-Product Family to Customers*, in *Software and Systems Modeling*, volume 2(1), pp. 15–36, 2003, ISSN 1619-1366

[101] G. Hasle, *Transportation Management in Distributed Enterprises*, in *Human Systems Management*, volume 18, pp. 203–212, ISSN 0167-2533

[102] G. Hasle, 2011, SINTEF ICT, Oslo, Norway, personal communications

[103] G. Hasle and O. Kloster, *Industrial Vehicle Routing*, in G. Hasle, K.-A. Lie and E. Quak (eds.), *Geometric Modelling, Numerical Simulation, and Optimization*, pp. 397–435, Springer Berlin Heidelberg, 2007, ISBN 978-3-540-68783-2

[104] C. Hempsch and S. Irnich, *Vehicle Routing Problems with Inter-Tour Resource Constraints*, in *The Vehicle Routing Problem: Latest Advances and New Challenges*, *Operations Research/Computer Science Interfaces Series*, volume 43, pp. 421–444, Springer US, 2008, ISBN 978-0-387-77777-1

[105] J. H. Holland, *Outline for a Logical Theory of Adaptive Systems*, in *Journal of the ACM*, volume 9(3), pp. 297–314, 1962

[106] A. Hubaux and P. Heymans, *On the Evaluation and Improvement of Feature-Based Configuration Techniques in Software Product Lines*, in *31st International Conference on Software Engineering — Companion Volume*, IEEE, 2009, ISBN 978-1-4244-3495-4

[107] T. Ibaraki, S. Imahori, M. Kubo, T. Masuda, T. Uno and M. Yagiura, *Effective Local Search Algorithms for Routing and Scheduling Problems with General Time-Window Constraints*, in *Transportation Science*, volume 39(2), pp. 206–232, 2005, ISSN 1526-5447

[108] *IEEE standard 1061-1998 for a Software Quality Metrics Methodology*, Technical report, 1998

[109] S. Irnich, *A Unified Modeling and Solution Framework for Vehicle Routing and Local Search-Based Metaheuristics*, in *INFORMS Journal on Computing*, volume 20(2), pp. 270–287, 2008

[110] S. Irnich, *Resource Extension Functions: Properties, Inversion, and Generalization to Segments*, in *OR Spectrum*, volume 30, pp. 133–148, 2008

[111] S. Irnich and G. Desaulniers, *Shortest Path Problems with Resource Constraints*, in G. Desaulniers, J. Desrosiers and M. M. Solomon (eds.), *Column Generation*, pp. 33–65, Springer US, 2005, ISBN 978-0-387-25486-9

[112] S. Irnich, G. Desaulniers, J. Desrosiers and A. Hadjar, *Path-Reduced Costs for Eliminating Arcs in Routing and Scheduling*, in *INFORMS Journal on Computing*, volume 22(2), pp. 297–313, 2010

[113] S. Irnich, B. Funke and T. Grünert, *Sequential Search and its Application to Vehicle-routing Problems*, in *Comput. Oper. Res.*, volume 33(8), pp. 2405–2429, 2006, ISSN 0305-0548

[114] *ISO/IEC 13568:2002(E) Information Technology — Z Formal Specification Notation — Syntax, Type System and Semantics*, Technical report, 2002

[115] I. Jacobson, M. Griss and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997, ISBN 0-201-92476-5

[116] M. Jaring and J. Bosch, *Variability Dependencies in Product Family Engineering*, in *Software Product-Family Engineering*, *Lecture Notes in Computer Science*, volume 3014, pp. 81–97, Springer Berlin / Heidelberg, 2004, ISBN 978-3-540-21941-5

[117] M. Jha and L. O'Brien, *Identifying Issues and Concerns in Software Reuse in Software Product Lines*, in *ICSR '09: Proceedings of the 11th International Conference on Software Reuse*, pp. 181–190, Springer-Verlag, Berlin, Heidelberg, 2009, ISBN 978-3-642-04210-2

[118] R. E. Johnson and B. Foote, *Designing Reusable Classes*, in *Journal of Object-Oriented Programming*, volume 1(2), pp. 22–35, 1988

[119] N. Jozefowiez, F. Semet and E.-G. Talbi, *Multi-Objective Vehicle Routing Problems*, in *European Journal of Operational Research*, volume 189(2), pp. 293–309, 2008, ISSN 0377-2217

[120] J. Kallrath, *Modeling Languages in Mathematical Optimization*, Kluwer Academic Publishers, Norwell, MA, USA, 2004, ISBN 1402075472

[121] K. C. Kang, J. Lee and P. Donohoe, *Feature-Oriented Product Line Engineering*, in *IEEE Software*, volume 19, pp. 58–65, 2002, ISSN 0740-7459

[122] S. Kent, *Model Driven Engineering*, in *Integrated Formal Methods*, *Lecture Notes in Computer Science*, volume 2335, pp. 286–298, Springer Berlin / Heidelberg, 2002, ISBN 978-3-540-43703-1, ISSN 0302-9743

[123] P. Kilby and P. Shaw, *Vehicle Routing*, in F. Rossi, P. V. Beek and T. Walsh (eds.), *Handbook of Constraint Programming*, pp. 801–836, Elsevier B.V., Amsterdam, The Netherlands, 2006, ISBN 978-0444527264

[124] P. Kilby, P. Prosser and P. Shaw, *A Comparison of Traditional and Constraint-based Heuristic Methods on Vehicle Routing Problems with Side Constraints*, in *Constraints*, volume 5, pp. 389–414, 2000

[125] G. A. P. Kindervater and M. W. P. Savelsbergh, *Chapter 10 Vehicle routing: Handling Edge Exchanges*, in J. K. L. E. H. Aarts (ed.), *Local Search in Combinatorial Optimization*, pp. 337–360, John Wiley, & Sons Ltd., 1997, ISBN 978-0-471-94822-3

[126] S. Kirkpatrick, C. D. Gelatt, Jr. and M. P. Vecchi, *Optimization by Simulated Annealing*, in *Science*, volume 220, pp. 671–680, 1983

[127] A. G. Kleppe, J. Warmer and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003, ISBN 032119442X

[128] I. Kurtev, *Adaptability of model transformations*, Ph.D. thesis, University of Twente, Enschede, 2005

[129] G. Laporte, *The Traveling Salesman Problem, the Vehicle Routing Problem, and Their Impact on Combinatorial Optimization*, in *International Journal of Strategic Decision Sciences*, volume 1(2), pp. 82–92, 2010

[130] G. Laporte, *Fifty Years of Vehicle Routing*, in *Transportation Science*, volume 43(4), pp. 408–416, 2009, ISSN 1526-5447

[131] G. Laporte, M. Gendreau, J.-Y. Potvin and F. Semet, *Classical and Modern Heuristics for the Vehicle Routing Problem*, in *International Transactions in Operational Research*, volume 7(4–5), pp. 285–300, 2000, ISSN 0969-6016

[132] A. Larsen, O. B. Madsen and M. M. Solomon, *Recent Developments in Dynamic Vehicle Routing Systems*, in *The Vehicle Routing Problem: Latest Advances and New Challenges*, *Operations Research/Computer Science Interfaces Series*, volume 43, pp. 199–218, Springer US, 2008, ISBN 978-0-387-77777-1

[133] E. Lawler, *Combinatorial Optimization, Networks and Matroids*, Holt, Rinehart and Winston, 1976

[134] C. Lecluyse, T. V. Woensel and H. Peremans, *Vehicle Routing with Stochastic Time-dependent Travel Times*, in *4OR: A Quarterly Journal of Operations Research*, volume 7(4), pp. 363–377, 2009, ISSN 1619-4500

[135] H. Li and A. Lim, *A Metaheuristic for the Pickup and Delivery Problem with Time Windows*, 2001, Department of Computer Science, National University of Singapore

[136] S. Lin and B. W. Kernighan, *An Effective Heuristic Algorithm for the Traveling-Salesman Problem*, in *Operations Research*, volume 21(2), pp. 498–516, 1973

[137] F.-H. Liu and S.-Y. Shen, *The Fleet Size and Mix Vehicle Routing Problem with Time Windows*, in *The Journal of the Operational Research Society*, volume 50(7), pp. 721–732, 1999, ISSN 01605682

[138] F. Marschall and P. Braun, *Model Transformations for the MDA with BOTL*, in *In Proceedings of the Workshop on Model Driven Architectures: Foundations and Applications*, pp. 25–36, Springer, 2003

[139] O. Martin, S. W. Otto and E. W. Felten, *Large-Step Markov Chains for the Traveling Salesman Problem*, in *Complex Systems*, volume 5, pp. 299–326, 1991

[140] T. Mens, K. Czarnecki and P. V. Gorp, *A Taxonomy of Model Transformations*, in J. Bezivin and R. Heckel (eds.), *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Dagstuhl, Germany, 2005, ISSN 1862-4405

[141] D. Mester and O. Bräysy, *Active Guided Evolution Strategies for Large-scale Vehicle Routing Problems with Time Windows*, in *Computers & Operations Research*, volume 32(6), pp. 1593–1614, 2005, ISSN 0305-0548

[142] D. Mester and O. Bräysy, *Active-guided Evolution Strategies for Large-scale Capacitated Vehicle Routing Problems*, in *Computers & Operations Research*, volume 34(10), pp. 2964–2975, 2007, ISSN 0305-0548

[143] J. Miller and J. Mukerji (eds.), *MDA Guide version 1.0.1 omg/2003-06-01*, Object Management Group, 2003

[144] N. Mladenovic and P. Hansen, *Variable Neighborhood Search*, in *Computers & Operations Research*, volume 24(11), pp. 1097–1100, 1997

[145] L. Moccia, J.-F. Cordeau and G. Laporte, *An Incremental Tabu Search Heuristic for the Generalized Vehicle Routing Problem with Time Windows*, in *Journal of the Operational Research Society*, 2011

[146] P. Mohagheghi and V. Dehlen, *Where Is the Proof? - A Review of Experiences from Applying MDE in Industry*, in *Model Driven Architecture — Foundations and Applications, Lecture Notes in Computer Science*, volume 5095, pp. 432–443, Springer Berlin / Heidelberg, 2010, ISBN 978-3-540-69095-5

[147] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt and J.-M. Jézéquel, *Weaving Variability into Domain Metamodels*, in *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pp. 690–705, Springer-Verlag, Berlin, Heidelberg, 2009, ISBN 978-3-642-04424-3

[148] D. Muthig and C. Atkinson, *Model-Driven Product Line Architectures*, in *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pp. 110–129, Springer-Verlag, London, UK, 2002, ISBN 3-540-43985-4

[149] L. Muyldermans and G. Pang, *On the Benefits of Co-collection: Experiments with a Multi-compartment Vehicle Routing Algorithm*, in *European Journal of Operational Research*, volume 206(1), pp. 93–103, 2010

[150] Y. Nagata, *Edge Assembly Crossover for the Capacitated Vehicle Routing Problem*, in *EvoCOP'07: Proceedings of the 7th European Conference on Evolutionary Computation in Combinatorial Optimization*, pp. 142–153, Springer-Verlag, Berlin, Heidelberg, 2007, ISBN 978-3-540-71614-3

[151] Y. Nagata and O. Bräysy, *Efficient Local Search Limitation Strategies for Vehicle Routing Problems*, in *EvoCOP'08: Proceedings of the 8th European Conference on Evolutionary Computation in Combinatorial Optimization*, pp. 48–60, Springer-Verlag, Berlin, Heidelberg, 2008, ISBN 3-540-78603-1, 978-3-540-78603-0

[152] Y. Nagata and O. Bräysy, *Edge Assembly-based Memetic Algorithm for the Capacitated Vehicle Routing Problem*, in *Networks*, volume 54(4), pp. 205–215, 2009, ISSN 0028-3045

[153] Y. Nagata, O. Bräysy and W. Dullaert, *A Penalty-based Edge Assembly Memetic Algorithm for the Vehicle Routing Problem with Time Windows*, in *Computers & Operations Research*, volume 37(4), pp. 724–737, 2010, ISSN 0305-0548

[154] N. Niu, J. Savolainen and Y. Yu, *Variability Modeling for Product Line Viewpoints Integration*, in *34th Annual IEEE Computer Software and Applications Conference*, July/Summer 2010

[155] L. Northrop, *Software Product Lines Essentials*, 2008, Software Engineering Institute, Carnegie Mellon University

[156] M. Nowak, O. Ergun and I. White, Chelsea C., *Pickup and Delivery with Split Loads*, in *Transportation Science*, volume 42(1), pp. 32–43, 2008

[157] *Object Management Group Unified Modeling Language Infrastructure Version 2.3*, 2010

[158] T. Oliveira, P. Alencar, D. Cowan, I. Filho and C. Lucena, *Enabling Model Driven Product Line Architectures*, in D. Akehurst (ed.), *Proceedings of Second European Workshop on MDA*, Computing Laboratory, University of Kent, Canterbury, UK, 2004

[159] I. Or, *Traveling Salesman-Type Problems and their Relation to the Logistics of Regional Blood Banking*, Ph.D. thesis, Department of Industrial Engineering and Management Sciences. Northwestern University, Evanston, IL, 1976

[160] I. H. Osman, *Metastrategy Simulated Annealing and Tabu Search Algorithms for the Vehicle Routing Problem*, in *Annals of Operations Research*, volume 41(4), pp. 421–451, 1993, ISSN 0254-5330

[161] S. Parragh, K. Doerner and R. Hartl, *A Survey on Pickup and Delivery Problems: Part I: Transportation Between Customers and Depot*, in *Journal für Betriebswirtschaft*, volume 58, pp. 21–51, 2008

[162] S. Parragh, K. Doerner and R. Hartl, *A Survey on Pickup and Delivery Problems: Part II: Transportation Between Pickup and Delivery Locations*, in *Journal für Betriebswirtschaft*, volume 58, pp. 81–117, 2008

[163] J. Partyka and R. Hall, *On the Road to Connectivity*, in *OR/MS Today*, volume 37(1), pp. 42–49, 2010

[164] D. Pisinger and S. Ropke, *A General Heuristic for Vehicle Routing Problems*, in *Computers & Operations Research*, volume 34, pp. 2403–2435, 2007

[165] K. Pohl, G. Böckle and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, 2005, ISBN 3540243720

[166] J. Potvin, G. Lapalme and J. Rousseau, *A Generalized k-opt Exchange Procedure for the MTSP*, in *Information Systems and Operations Research*, volume 27(4), pp. 474–481, 1989

[167] T. Ralphs, L. Kopman, W. Pulleyblank and L. Trotter, *On the Capacitated Vehicle Routing Problem*, in *Mathematical Programming*, volume 94, pp. 343–359, 2003, ISSN 0025-5610

[168] M. Reimann, K. Doerner and R. F. Hartl, *D-Ants: Savings Based Ants Divide and Conquer the Vehicle Routing Problem*, in *Computers & Operations Research*, volume 31(4), pp. 563–591, 2004, ISSN 0305-0548

[169] Y. Ren, M. Dessouky and F. Ordó nez, *The Multi-shift Vehicle Routing Problem with Overtime*, in *Computers & Operations Research*, volume 37(11), pp. 1987–1998, 2010, ISSN 0305-0548

[170] J. Renaud, G. Laporte and F. F. Boctor, *A Tabu Search Heuristic for the Multi-depot Vehicle Routing Problem*, in *Computers & Operations Research*, volume 23(3), pp. 229–235, 1996, ISSN 0305-0548

[171] P. P. Repoussis, C. D. Tarantilis, O. Bräysy and G. Ioannou, *A Hybrid Evolution Strategy for the Open Vehicle Routing Problem*, in *Computers & Operations Research*, volume 37(3), pp. 443–455, 2010, ISSN 0305-0548

[172] P. P. Repoussis, C. D. Tarantilis and G. Ioannou, *An Evolutionary Algorithm for the Open Vehicle Routing Problem with Time Windows*, in F. B. Pereira and J. Tavares (eds.), *Bio-inspired Algorithms for the Vehicle Routing Problem*, *Studies in Computational Intelligence*, volume 161, pp. 55–75, Springer Berlin / Heidelberg, 2009, ISBN 978-3-540-85151-6

[173] Y. Rochat and F. Semet, *A Tabu Search Approach for Delivering Pet Food and Flour in Switzerland*, in *The Journal of the Operational Research Society*, volume 45(11), pp. 1233–1246, 1994

[174] S. Ropke and D. Pisinger, *An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows*, in *Transportation Science*, volume 40(4), pp. 455–472, 2006, ISSN 1526-5447

[175] S. Ropke and D. Pisinger, *A Unified Heuristic for a Large Class of Vehicle Routing Problems with Backhauls*, in *European Journal of Operational Research*, volume 171(3), pp. 750–775, June 2006

[176] F. Rossi, P. v. Beek and T. Walsh, *Handbook of Constraint Programming*, Elsevier Science Inc., New York, NY, USA, 2006, ISBN 0444527265

[177] L.-M. Rousseau, M. Gendreau and G. Pesant, *Using Constraint-Based Operators to Solve the Vehicle Routing Problem with Time Windows*, in *Journal of Heuristics*, volume 8(1), pp. 43–58, 2002, ISSN 1381-1231

[178] A. L. Santos, K. Koskimies and A. Lopes, *A Model-Driven Approach to Variability Management in Product-Line Engineering*, in *Nordic Journal of Computing*, volume 13(3), pp. 196–213, 2006, ISSN 1236-6064

[179] D. Sariklis and S. Powell, *A Heuristic Method for the Open Vehicle Routing Problem*, in *The Journal of the Operational Research Society*, volume 51(5), pp. 564–573, 2000, ISSN 01605682

[180] M. W. P. Savelsbergh and M. Sol, *The General Pickup and Delivery Problem*, in *Transportation Science*, volume 29, pp. 17–29, 1995

[181] M. W. P. Savelsbergh, *The Vehicle Routing Problem with Time Windows: Minimizing Route Duration*, in *INFORMS Journal on Computing*, volume 4(2), pp. 146–154, 1992

[182] J. Savolainen, J. Kuusela, M. Mannion and T. Vehkomäki, *Combining Different Product Line Models to Balance Needs of Product Differentiation and Reuse*, in *ICSR '08: Proceedings of the 10th International Conference on Software Reuse*, pp. 116–129, Springer-Verlag, Berlin, Heidelberg, 2008, ISBN 978-3-540-68062-8

[183] I. Schaefer, *Variability Modelling for Model-Driven Development of Software Product Lines*, in *Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, 2009

[184] V. Schmid, K. F. Doerner, R. F. Hartl, M. W. P. Savelsbergh and W. Stoecher, *A Hybrid Solution Approach for Ready-Mixed Concrete Delivery*, in *Transportation Science*, volume 43(1), pp. 70–85, 2009

[185] L. Schrage, *Formulation and Structure of More Complex/Realistic Routing and Scheduling Problems*, in *Networks*, volume 11(2), pp. 229–232, 1981

[186] A. Schrijver, *On the History of Combinatorial Optimization (Till 1960)*, in G. N. K. Aardal and R. Weismantel (eds.), *Discrete Optimization*, *Handbooks in Operations Research and Management Science*, volume 12, pp. 1–68, Elsevier, 2005

[187] D. C. Sharp, *Component-Based Product Line Development of Avionics Software*, in *Proceedings of the First Conference on Software Product Lines : Experience and Research Directions*, pp. 353–370, Kluwer Academic Publishers, Norwell, MA, USA, 2000, ISBN 0-79237-940-3

[188] P. Shaw, *Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems*, in *Lecture Notes in Computer Science*, pp. 417–431, Springer-Verlag, 1998

[189] T. Solakivi, L. Ojala, J. Töyli, H.-M. Hälinen, H. Lorentz, K. Rantasila and T. Naula, *Finland State of Logistics*, 2009, Finnish Ministry of Transport and Communications, Helsinki

[190] M. M. Solomon, *Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints*, in *Operations Research*, volume 35(2), pp. 254–265, 1987, ISSN 0030364X

[191] K. Sörensen, M. Sevaux and P. Schittekat, *"Multiple Neighbourhood" Search in Commercial VRP Packages: Evolving Towards Self-Adaptive Methods*, in *Adaptive and Multilevel Metaheuristics*, pp. 239–253, Springer US, 2008

[192] J. M. Spivey, *The Z notation: A Reference Manual*, Oriel College, Oxford, England, 2001

[193] M. Svahnberg, J. van Gurp and J. Bosch, *A Taxonomy of Variability Realization Techniques: Research Articles*, in *Software — Practice and Experience*, volume 35(8), pp. 705–754, 2005, ISSN 0038-0644

[194] E. Taillard, P. Badeau, M. Gendreau, F. Guertin and J.-Y. Potvin, *A Tabu Search Heuristic for the Vehicle Routing Problem with Soft Time Windows*, in *Transportation Science*, volume 31(2), pp. 170–186, 1997

[195] E.-G. Talbi, *Metaheuristics: From Design to Implementation*, Wiley Publishing, 2009, ISBN 0470278587, 9780470278581

[196] C. D. Tarantilis, *Solving the Vehicle Routing Problem with Adaptive Memory Programming Methodology*, in *Computers & Operations Research*, volume 32(9), pp. 2309–2327, 2005, ISSN 0305-0548

[197] R. Tawhid and D. Petriu, *Integrating Performance Analysis in the Model Driven Development of Software Product Lines*, in *Model Driven Engineering Languages and Systems*, *Lecture Notes in Computer Science*, volume 5301, pp. 490–504, Springer Berlin / Heidelberg, 2010, ISBN 978-3-540-87874-2, ISSN 0302-9743

[198] F. A. Tillman, *The Multiple Terminal Delivery Problem with Probabilistic Demands*, in *Transportation Science*, volume 3, pp. 192–204, 1969

[199] F. A. Tillman and R. W. Hering, *A Study of a Look-ahead Procedure for Solving the Multiterminal Delivery Problem*, in *Transportation Research*, volume 5, pp. 225–229, 1971

[200] P. Toth and D. Vigo, *Branch-and-bound Algorithms for the Capacitated VRP*, pp. 29–51, 2001, ISBN 0-89871-498-2

[201] P. Toth and D. Vigo, *A Heuristic Algorithm for the Symmetric and Asymmetric Vehicle Routing Problems with Backhauls*, in *European Journal of Operational Research*, volume 113(3), pp. 528–543, 1999, ISSN 0377-2217

[202] P. Toth and D. Vigo (eds.), *The Vehicle Routing Problem*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001, ISBN 0-89871-498-2

[203] S. Trujillo, A. Zubizarreta, J. De Sosa and X. Mendialdua, *Is Model Variability Enough?*, in *Proceedings of the 1st International Workshop on Model-Driven Product Line Engineering*, pp. 43–48, Online proceedings, Twente, The Netherlands, 2009

[204] P. H. Vance, C. Barnhart, E. L. Johnson and G. L. Nemhauser, *Airline Crew Scheduling: A New Formulation and Decomposition Algorithm*, in *Operations Research*, volume 45(2), pp. 188–200, 1997

[205] V. Černý, *Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm*, in *Journal of Optimization Theory and Applications*, volume 45(1), pp. 41–51, January 1985

[206] T. Vidal, T. G. Crainic, M. Gendreau, N. Lahrichi and W. Rei, *A Hybrid Genetic Algorithm for Multi-Depot and Periodic Vehicle Routing Problems*, Technical Report 34, CIRRELT, 2010

[207] M. Voelter and I. Groher, *Product Line Implementation using Aspect-Oriented and Model-Driven Software Development*, in *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pp. 233–242, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-2888-0

[208] S. Voß and D. L. Woodruff, *Optimization Software Class Libraries*, in *Optimization Software Class Libraries*, *Operations Research/Computer Science Interfaces Series*, volume 18, pp. 1–24, Springer US, 2002, ISBN 978-1-4020-7002-0, ISSN 1387-666X

[209] C. Voudouris and E. Tsang, *Guided Local Search*, Technical report, University of Essex, UK, 1995

[210] K. E. Wiegers, *Software Requirements, Second Edition*, Microsoft Press, 2003, ISBN 0735618798

[211] H. P. Williams, *Model Building in Mathematical Programming*, Wiley, 4th edition, 1999, ISBN 0471997889

[212] T. V. Woensel, L. Kerbache, H. Peremans and N. Vandaele, *Vehicle Routing with Dynamic Travel Times: A Queueing Approach*, in *European Journal of Operational Research*, volume 186(3), pp. 990–1007, 2008

[213] S. Wøhlk, *A Decade of Capacitated Arc Routing*, in *The Vehicle Routing Problem: Latest Advances and New Challenges*, *Operations Research/Computer Science Interfaces Series*, volume 43, pp. 29–48, Springer US, 2008, ISBN 978-0-387-77777-1

[214] D. H. Wolpert and W. G. Macready, *No Free Lunch Theorems for Optimization*, in *IEEE Transactions on Evolutionary Computation*, volume 1(1), pp. 67–82, 1997

[215] A. Wren and A. Holliday, *Computer Scheduling of Vehicles from One or More Depots to a Number of Delivery Points*, in *Operational Research Quarterly (1970-1977)*, volume 23(3), pp. 333–344, 1972, ISSN 00303623

[216] H. Xu, Z.-L. Chen, S. Rajagopal and S. Arunapuram, *Solving a Practical Pickup and Delivery Problem*, in *Transportation Science*, volume 37(3), pp. 347–364, 2003, ISSN 1526-5447

[217] E. E. Zachariadis and C. T. Kiranoudis, *An Open Vehicle Routing Problem Metaheuristic for Examining Wide Solution Neighborhoods*, in *Computers & Operations Research*, volume 37(4), pp. 712–723, 2010, ISSN 0305-0548

[218] E. E. Zachariadis and C. T. Kiranoudis, *A Strategy for Reducing the Computational Complexity of Local Search-based Methods for the Vehicle Routing Problem*, in *Computers & Operations Research*, volume 37(12), pp. 2089–2105, 2010, ISSN 0305-0548

APPENDIX 1 TYPE INDEX

activatingCapabilities : $(\mathrm{seq}\, Activity \times Activity) \pfun \mathbb{F}\, Capability$ 139
activatingDeltas : $(Capability \times Resource) \pfun PartialResourceDelta$ 139
activeWaitingAt : $(Activity \times Resource) \pfun \mathbb{Z}$ 131
activities : $\mathbb{F}\, Activity$ 113
Activity 113
actorOf : $\mathrm{paths}\, Activity \times Activity \pfun Actor$ 166
actors : $\mathbb{F}\, Actor$ 113
Add 167
AddActivityGroup 156
AddBounds 157
AddCapability 156
AddCapabilityProhibition 156
AddCapabilityRequirement 156
AddTransitionalResourceFunction 157
Ansatz 113
assigned : $\mathbb{F}\, Activity$ 113
$\mathrm{bounds}\, X == \mathbb{F}\, X \pfun Interval$ 145
capabilities : $\mathbb{F}\, Capability$ 125
Capability 124
CapabilityBinding 125
CapabilityCompletenessRule 127
$CapabilityDelta == Activity \pfun \mathbb{Z}$ 124
capabilityDeltas : $Capability \pfun CapabilityDelta$ 125
capabilityExistence : $Activity \pfun \mathbb{F}\, Capability$ 128
CapabilityExistenceRuleBinding 128
CapabilityExistenceRuleRestriction 128
capabilityNonexistence : $Activity \pfun \mathbb{F}\, Capability$ 128
CapabilityNonexistenceRuleBinding 128
CapabilityNonexistenceRuleRestriction 129
capabilityParents : $Capability \pfun Capability$ 135
CapabilityProblem 129
CapabilityProjection 127
CapabilityRestriction 126
$CapabilityValue == Capability \pfun \mathbb{Z}$ 125
capabilityValues : $Activity \pfun CapabilityValue$ 125
CreateActivity 154
CreateActor 154
CreateCapability 155
CreateResource 155
deactivatingCapabilities : $Activity \pfun \mathbb{F}\, Capability$ 139
$DependentProfitDelta == (\mathbb{F}\, PartialResourceValue \times Activity \times Activity) \pfun \mathbb{Z}$ 150
$DependentResourceDelta == (\mathbb{F}\, PartialResourceValue \times Activity \times Activity) \pfun \mathbb{Z}$ 149
Disjointness 116

end : *Activity* 113
GenericRoutingProblem 148
groupBounds : bounds *Activity* 146
grouping $X == \mathbb{F}(\mathbb{F}\, X)$ 144
GroupingBinding 145
GroupingProblem 147
GroupingProjection 145
GroupingRestriction 145
GroupingRuleBinding 146
GroupingRuleRestriction 146
groups : grouping *Activity* 145
groupValues : grouping *Activity* 145
IdentityPartialResourceProjection 133
IdentityPartialResourceValues 133
InitRoutingProblem 158
Injectivity 116
Interval 145
isActive : $\mathbb{P}(Activity \pfun CapabilityValue \times Activity \times Capability)$ 126
isZero : $\mathbb{P}(Activity \pfun CapabilityValue \times Activity \times Capability)$ 126
$M : \mathbb{Z}$ 122
next : $(\mathrm{seq}\, X \times X) \pfun X$ 115
NonnegativeCapabilityRule 128
ParentPartialResourceProjection 136
ParentPartialResourceValues 135
PartialResourceBinding 131
PartialResourceDelta == *ResourceDelta* 130
partialResourceDeltas : $Resource \pfun PartialResourceDelta$ 131
partialResourceLowerBounds : $(Capability \times Resource) \pfun \mathbb{Z}$ 131
PartialResourceProblem 138
PartialResourceProjection 133
PartialResourceRestriction 132
PartialResourceRule 134
PartialResourceRuleBinding 134
PartialResourceRuleRestriction 134
partialResourceUpperBounds : $(Capability \times Resource) \pfun \mathbb{Z}$ 134
PartialResourceValue == $(Capability \times Resource) \pfun \mathbb{Z}$ 130
partialResourceValues : $Activity \pfun PartialResourceValue$ 131
partialValueAt : $(Activity \times Capability \times Resource) \pfun \mathbb{Z}$ 131
paths $X == \{ f : Actor \rel \mathrm{seq}_1\, X \}$ 113
prev : $(\mathrm{seq}\, X \times X) \pfun X$ 115
ProfitDelta == $(Activity \times Activity) \pfun \mathbb{Z}$ 147
ProperAnsatz 116
Remove 168
ResetRoutingProblem 166
Resource 118

ResourceBinding 118
ResourceConstrainedProblem 122
ResourceDelta == $(Activity \times Activity) \pfun \mathbb{Z}$ 118
resourceDeltas : $Resource \pfun ResourceDelta$ 118
resourceDeltaStacks : $Activity \pfun ResourceDeltaStacks$ 139
ResourceDeltaStacks == $Resource \pfun Stack[PartialResourceDelta]$ 139
resourceLowerBounds : $(Activity \times Resource) \pfun \mathbb{Z}$ 118
ResourceProjection 120
ResourceRestriction 119
ResourceRule 121
ResourceRuleBinding 120
ResourceRuleRestriction 120
resources : $\mathbb{F}\, Resource$ 118
ResourceSlackBinding 121
ResourceSlackProjection 122
ResourceSlackRestriction 122
resourceSlackValues : $Activity \pfun ResourceValue$ 121
resourceUpperBounds : $(Activity \times Resource) \pfun \mathbb{Z}$ 120
ResourceValue == $Resource \pfun \mathbb{Z}$ 118
resourceValues : $Activity \pfun ResourceValue$ 118
Reverse 169
routes : $\mathsf{paths}\, Activity$ 113
SetParent 155
SetStrictLowerBound 155
SimpleGrouping 146
single : $\mathbb{F}\, X \pfun X$ 141
StackResourceBinding 139
StackResourceProblem 143
StackResourceProjection 142
StackResourceProjectionUtilities 140
StackResourceRestriction 141
start : *Activity* 113
strictLowerBoundResources : $\mathbb{F}\, Resource$ 118
$\_ \,\mathsf{to}\, \_ \sim \_ : X \times X \times \mathsf{seq}\, X \to \mathsf{seq}\, X$ 166
TryInsertActivity 172
unassigned : $\mathbb{F}\, Activity$ 113
Uniqueness 115
Update 168
UpdateResources 167
valueOrZero : $(Activity \pfun CapabilityValue \times Activity \times Capability) \pfun \mathbb{Z}$ 126

APPENDIX 2 THE Z NOTATION: A PRIMER

This appendix is for a reader uninitiated to the Z notation. In this short introduction, we go through the principles and central elements of the language, and illustrate the basic usage of notation. This introduction is largely based on the Z reference manual by Spivey [192].

The Z notation is a formal specification language. Formal specifications use mathematical notation to describe in a precise way the properties an information system must have without specifying exactly how these properties are achieved. They, then, state *what* the system must do without expressing *how* it has to be done. Expressing these properties has been subject to research, and formal approaches emerged from a mathematical theory: Z has its roots in formal set theory and typed expressions. An early version of this language was given by Abrial et al. [1] in 1980, where they note a need for exhaustive and unambiguous language for communicating central ideas of a design without sacrificing the details. We note that this principle of *abstraction* prevails in the usage of Z today; a formal specification can be given without getting bogged down to details of implementation and without dragging through the heavy rigor of algebraic formalization.

As with every typed system, the concept of **type** is defined. In Z, there are three kinds of basic types: *set types*, *Cartesian product types*, and *schema types*. These types can be composed into more complex types by defining new types by giving them *members* of the basic types or other composed types.

Sets and set types are the basic building block of the type system of Z. Any set of objects of the same type t is itself an object in the set type $\mathbb{P}\, t$. Two sets of the same type $\mathbb{P}\, t$ are equal exactly if they have the same members. Sets can be written, for example, by listing their elements, as in $\{\, 2, 4, 6, 8, 10 \,\}$. This list has the type $\mathbb{P}\, \mathbb{Z}$, a set of integers.

Tuples and Cartesian product types can be illustrated with the following example. If x, y, and z are three objects of types t, u, and v, respectively, then the ordered triple (x, y, z) is an object of type $t \times u \times v$.

Binary relations and functions, some of the most important mathematical objects in Z, are modeled by their *graphs*. The graph of a binary relation is the set of ordered pairs for which it holds: for example the graph of a relation $\_ < \_$ on integers contains the pairs $(0, 1)$, $(0, 2)$, $(-5, 15)$, but not $(5, 5)$ or $(85, 6)$. The notation $X \leftrightarrow Y$, meaning the set of binary relations between the sets X and Y, is defined as a synonym for the set $\mathbb{P}(X \times Y)$ of subsets of the set $X \times Y$ of ordered pairs. A function is a special kind of relation: the set $X \to Y$ relates each member of X, the *domain*, to exactly one member of Y, the *range*. The notation $f(x)$ can be used to express this unique element of Y. It is notable that relations and functions have the same type $\mathbb{P}(X \times Y)$.

A *variable* is simple arbitrary name. A *signature* is a collection of variables, each with a type. Signatures are created with *declarations* and they provide means for making mathematical statements, expressed by *predicates*. For example, the declaration $x, y : \mathbb{Z}$ creates a signature of two variables x and y, both of type $\mathbb{Z}$. In this signature, a predicate $x < y$ can be used to express the *property* that the value

of x is less than the value of y. Note that two different predicates may express the same property. A **schema** is a signature together with a set of properties over the signature. For instance, the given example can be written as a schema as

$$
\begin{array}{|l}
\hline
A \\
\hline
x : \mathbb{Z} \\
y : \mathbb{Z} \\
\hline
x < y \\
\hline
\end{array}
$$

This schema defines the type A. The first part above the horizontal line of the schema corresponds to the signature, and the second part to the properties of the schema. Here x and y are the *components* of A. The properties of a schema express facts that must remain true under every value, or *binding*, of the variables of that schema. These properties are similar to assertions in some programming languages. Although they must be true at all times, it is essential to realize that they cannot ensure correctness of the system: ensuring completeness and correctness of the invariants is naturally left up to the writer of the specification.

Many mathematical constructions are independent of the elements from which the construction starts: for example, sequences of numbers and characters are both same kinds of objects. Z notation offers a generic construct for these situations. The constructions allow us to specify families of concepts independently of their concrete realizations. **A generic schema** is defined as follows:

$$
\begin{array}{|l}
\hline
G[X] \\
\hline
S : \mathbb{F}\, X \\
\hline
\#S < 4 \\
\hline
\end{array}
$$

The given example defines a schema with a set of some yet undefined elements with the restriction that there must be less than four of these elements in the set. We may now provide a type for the generic schema to obtain a concrete schema, for example, as follows:

$$H \mathrel{\widehat=} G[\mathbb{Z}]$$

This definition results in the following schema:

$$
\begin{array}{|l}
\hline
H \\
\hline
S : \mathbb{F}\, \mathbb{Z} \\
\hline
\#S < 4 \\
\hline
\end{array}
$$

A generic definition provides means for constructing generic operations, such as the following example where the *last* operator is defined for generic nonempty finite sequences.

$$
\begin{array}{|l}
\hline
[X] \\
last : \ \mathrm{seq}_1 X \to X \\
\hline
\forall s : \ \mathrm{seq}_1 X \bullet last\, s = s(\#s) \\
\hline
\end{array}
$$

New types can also be created by combining existing schemas by **schema calculus**. Given a schema of type B as

$$
\begin{array}{|l}
\hline
B \\
x : \mathbb{Z} \\
\hline
x > 5 \\
\hline
\end{array}
$$

we may create a new schema C, by *horizontal schema definition*, as follows:

$$C \mathrel{\widehat=} A \land B$$

This schema is the logical conjunction of the schemas A and B. It can be written in a regular schema format as follows:

$$
\begin{array}{|l}
\hline
C \\
x : \mathbb{Z} \\
y : \mathbb{Z} \\
\hline
x < y \land \\
x > 5 \\
\hline
\end{array}
$$

This resulting schema combines the two schemas by combining their signatures and expressing the predicate as a conjunction of the predicates of both schemas. Note that two schemas cannot be combined if they contain a signature with differing types of variables of the same name.

So far the only variables which have appeared in the predicate part have been components of the schema. However, Z contains also *global variables* — variables declared outside[1] any schema. These global variables can be constructed by **axiomatic definitions**. In fact, symbols such as $<$ are really just global variables defined in the Z toolkit.

For illustration, consider an example where we wish to denote a number range with two numbers and the symbol "$..$" between them. This can be done by axiomatic definition as follows:

$$
\begin{array}{|l}
\_ \mathinner{..} \_ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{P}\,\mathbb{Z} \\
\hline
\forall a, b : \mathbb{Z} \bullet \\
\qquad a \mathinner{..} b = \{\, k : \mathbb{Z} \mid a \le k \le b \,\}
\end{array}
$$

In this schema, we define that the number range contains all the numbers between the two given points.

[1] For a more thorough view on *scoping* on Z, we refer to the Z reference manual.

This definition can now be used in any schema requiring number ranges, for example as in the following:

$$
\begin{array}{|l}
\hline
[X] \\
\hline
tail : \operatorname{seq}_1 X \to \operatorname{seq} X \\
\hline
\forall s : \operatorname{seq}_1 X \bullet \\
\quad tail\, s = (\lambda n : 1 \mathinner{..} \#s - 1 \bullet s(n+1)) \\
\hline
\end{array}
$$

The generic schema defines an operation for obtaining all the elements of a sequence except the first. The operation selects the elements of that sequence by iterating n from 1 (using the defined operator) up to the number of elements in the sequence minus one, and returning the element at position $n + 1$ at each step.

Abbreviation definition may be used to define a new global constant, in which the identifier on the left becomes a global constant; its value is given by the expression on the right and its type is the same as the type of the expression. For example, the definition

$$\text{complex} == (\mathbb{Z} \times \mathbb{Z})$$

introduces a type for complex numbers that is effectively a pair of integers.

Sequential systems introduce one additional use case for formal definitions. Using the schemas describing sequential execution, we may introduce the notion of state to the formalizations. This enables us to express operations against the system specification.

A schema that expresses change in values of a given schema are introduced using **delta convention**. An example of such a schema is given below. Recall that the schema C defined two integers, x, and y.

$$
\begin{array}{|l}
\hline
IncreaseValues \\
\hline
\Delta C \\
step? : \mathbb{Z} \\
result! \\
\hline
x' = x + step? \\
y' = y + step? \\
result! = y' \\
\hline
\end{array}
$$

The schema defines an operation on the schema C and denotes with the delta (Δ) that it alters the values of the C. The inputs to this operation are *decorated* with the symbol "?", and its outputs with the symbol "!". The predicates of the schema define that both x and y of the C are increased by the input "step?". The values after the operation are decorated using the symbol "'". Similarly to delta convention, **xi convention** (Ξ) can be used to signal operation that does not alter the values of the target schema.

Operations described by schemas can be combined using **sequential composition and piping**. The following operation is defined as increase in values

twice using composition.

$$IncreaseTwice \mathrel{\widehat=} IncreaseValues \fatsemi IncreaseValues$$

The following operation uses the result of the first increase as an input to the second using piping.

$$IncreaseWithResult \mathrel{\widehat=} IncreaseValues \gg IncreaseValues[result?/step?]$$

Note that the name of the input in the second operation is changed to match the output of the first operation using **renaming**.

APPENDIX 3 GENERIC STACK IN Z NOTATION

This appendix defines a data structure with stack semantics as the Z specification [114] does not contain one.

A stack is a sequence of a given type of elements.

$$
\begin{array}{|l}
\hline
Stack[X] \\
elements : \operatorname{seq} X \\
\hline
\end{array}
$$

Adding to a stack is done through push operation, given by the following schema:

$$
\begin{array}{|l}
\hline
[X] \\
push : (Stack[X] \times X) \to Stack[X] \\
\hline
\forall s : Stack[X];\ x : X \bullet \\
\quad push(s, x) = \{ s' : Stack[X] \mid \\
\qquad s'.elements = s.elements \cup \{\#s.elements + 1 \mapsto x\} \bullet \\
\qquad (s, x) \mapsto s' \} \\
\hline
\end{array}
$$

Removing from a stack is done through pop operation, given by the following schema:

$$
\begin{array}{|l}
\hline
[X] \\
pop : Stack[X] \to (Stack[X] \times X) \\
\hline
\forall s : Stack[X] \bullet \\
\quad pop(s) = \{ s' : Stack[X];\ x : X \mid \\
\qquad x = last\, s.elements \land \\
\qquad s.'elements = s.elements \setminus \{\#s.elements \mapsto last\, s.elements\} \bullet \\
\qquad s \mapsto (s', x) \} \\
\hline
\end{array}
$$

Acquiring the topmost element from a stack without removing it is done through peek operation, given by the following schema:

$$
\begin{array}{|l}
\hline
[X] \\
peek : Stack[X] \to X \\
\hline
\forall s : Stack[X] \bullet peek(s) = last\, s.elements \\
\hline
\end{array}
$$

APPENDIX 4 DETAILED COMPUTATIONAL RESULTS

instance	veh.	dist.	time (s)	instance	veh.	dist.	time (s)
lc1_4_1	40	7152.06	10.78	lr2_4_1	12	11923.18	219.02
lc1_4_2	40	7151.26	57.43	lr2_4_2	10	10297.37	321.20
lc1_4_3	38	7452.90	80.28	lr2_4_3	9	10166.35	596.97
lc1_4_4	33	7463.41	482.60	lr2_4_4	7	7704.60	590.45
lc1_4_5	41	7345.60	383.66	lr2_4_5	8	9588.95	535.14
lc1_4_6	41	7260.40	96.42	lr2_4_6	7	9592.63	486.29
lc1_4_7	40	7151.56	13.52	lr2_4_7	7	8440.87	590.57
lc1_4_8	40	7362.17	417.02	lr2_4_8	7	7291.66	591.84
lc1_4_9	39	8336.17	409.31	lr2_4_9	7	9753.11	519.23
lc1_4_10	40	7962.45	490.41	lr2_4_10	8	9278.40	581.14
lc2_4_1	14	4751.24	96.85	lrc1_4_1	40	9647.17	536.48
lc2_4_2	14	5557.73	82.73	lrc1_4_2	38	8766.44	65.84
lc2_4_3	14	6615.17	295.84	lrc1_4_3	32	8195.80	77.64
lc2_4_4	13	5313.72	397.84	lrc1_4_4	24	6477.63	144.14
lc2_4_5	14	5458.98	100.76	lrc1_4_5	38	9822.61	72.38
lc2_4_6	13	4525.08	498.74	lrc1_4_6	36	8718.08	308.29
lc2_4_7	13	4848.46	268.74	lrc1_4_7	35	8748.45	489.86
lc2_4_8	13	4714.27	559.92	lrc1_4_8	32	8406.59	282.27
lc2_4_9	14	4989.09	143.51	lrc1_4_9	32	8777.39	582.85
lc2_4_10	14	5325.62	109.63	lrc1_4_10	28	7656.10	182.80
lr1_4_1	46	12894.84	406.72	lrc2_4_1	17	9022.69	177.82
lr1_4_2	36	10233.53	329.36	lrc2_4_2	13	7634.34	384.48
lr1_4_3	29	8838.07	333.69	lrc2_4_3	12	7385.16	599.37
lr1_4_4	21	7051.70	187.25	lrc2_4_4	9	6567.90	591.01
lr1_4_5	33	10747.34	425.09	lrc2_4_5	13	7546.18	323.63
lr1_4_6	34	10058.57	187.70	lrc2_4_6	12	7473.18	302.94
lr1_4_7	27	8578.75	455.18	lrc2_4_7	12	7521.35	436.17
lr1_4_8	19	6788.92	523.98	lrc2_4_8	12	7406.35	356.02
lr1_4_9	29	10233.20	130.44	lrc2_4_9	11	6281.48	598.71
lr1_4_10	25	8555.04	364.15	lrc2_4_10	8	7390.19	575.91

JYVÄSKYLÄ STUDIES IN COMPUTING

1 Ropponen, Janne, Software risk management - foundations, principles and empirical findings. 273 p. Yhteenveto 1 p. 1999.
2 Kuzmin, Dmitri, Numerical simulation of reactive bubbly flows. 110 p. Yhteenveto 1 p. 1999.
3 Karsten, Helena, Weaving tapestry: collaborative information technology and organisational change. 266 p. Yhteenveto 3 p. 2000.
4 Koskinen, Jussi, Automated transient hypertext support for software maintenance. 98 p. (250 p.) Yhteenveto 1 p. 2000.
5 Ristaniemi, Tapani, Synchronization and blind signal processing in CDMA systems. - Synkronointi ja sokea signaalinkäsittely CDMA järjestelmässä. 112 p. Yhteenveto 1 p. 2000.
6 Laitinen, Mika, Mathematical modelling of conductive-radiative heat transfer. 20 p. (108 p.) Yhteenveto 1 p. 2000.
7 Koskinen, Minna, Process metamodelling. Conceptual foundations and application. 213 p. Yhteenveto 1 p. 2000.
8 Smolianski, Anton, Numerical modeling of two-fluid interfacial flows. 109 p. Yhteenveto 1 p. 2001.
9 Nahar, Nazmun, Information technology supported technology transfer process. A multi-site case study of high-tech enterprises. 377 p. Yhteenveto 3 p. 2001.
10 Fomin, Vladislav V., The process of standard making. The case of cellular mobile telephony. - Standardin kehittämisen prosessi. Tapaustutkimus solukkoverkkoon perustuvasta matkapuhelintekniikasta. 107 p. (208 p.) Yhteenveto 1 p. 2001.
11 Päivärinta, Tero, A genre-based approach to developing electronic document management in the organization. 190 p. Yhteenveto 1 p. 2001.
12 Häkkinen, Erkki, Design, implementation and evaluation of neural data analysis environment. 229 p. Yhteenveto 1 p. 2001.
13 Hirvonen, Kullervo, Towards better employment using adaptive control of labour costs of an enterprise. 118 p. Yhteenveto 4 p. 2001.
14 Majava, Kirsi, Optimization-based techniques for image restoration. 27 p. (142 p.) Yhteenveto 1 p. 2001.
15 Saarinen, Kari, Near infra-red measurement based control system for thermo-mechanical refiners. 84 p. (186 p.) Yhteenveto 1 p. 2001.
16 Forsell, Marko, Improving component reuse in software development. 169 p. Yhteenveto 1 p. 2002.
17 Virtanen, Pauli, Neuro-fuzzy expert systems in financial and control engineering. 245 p. Yhteenveto 1 p. 2002.
18 Kovalainen, Mikko, Computer mediated organizational memory for process control. Moving CSCW research from an idea to a product. 57 p. (146 p.) Yhteenveto 4 p. 2002.
19 Hämäläinen, Timo, Broadband network quality of service and pricing. 140 p. Yhteenveto 1 p. 2002.
20 Martikainen, Janne, Efficient solvers for discretized elliptic vector-valued problems. 25 p. (109 p.) Yhteenveto 1 p. 2002.
21 Mursu, Anja, Information systems development in developing countries. Risk management and sustainability analysis in Nigerian software companies. 296 p. Yhteenveto 3 p. 2002.
22 Seleznyov, Alexandr, An anomaly intrusion detection system based on intelligent user recognition. 186 p. Yhteenveto 3 p. 2002.
23 Lensu, Anssi, Computationally intelligent methods for qualitative data analysis. 57 p. (180 p.) Yhteenveto 1 p. 2002.
24 Ryabov, Vladimir, Handling imperfect temporal relations. 75 p. (145 p.) Yhteenveto 2 p. 2002.
25 Tsymbal, Alexey, Dynamic integration of data mining methods in knowledge discovery systems. 69 p. (170 p.) Yhteenveto 2 p. 2002.
26 Akimov, Vladimir, Domain decomposition methods for the problems with boundary layers. 30 p. (84 p.). Yhteenveto 1 p. 2002.
27 Seyukova-Rivkind, Ludmila, Mathematical and numerical analysis of boundary value problems for fluid flow. 30 p. (126 p.) Yhteenveto 1 p. 2002.
28 Hämäläinen, Seppo, WCDMA Radio network performance. 235 p. Yhteenveto 2 p. 2003.
29 Pekkola, Samuli, Multiple media in group work. Emphasising individual users in distributed and real-time CSCW systems. 210 p. Yhteenveto 2 p. 2003.
30 Markkula, Jouni, Geographic personal data, its privacy protection and prospects in a location-based service environment. 109 p. Yhteenveto 2 p. 2003.
31 Honkaranta, Anne, From genres to content analysis. Experiences from four case organizations. 90 p. (154 p.) Yhteenveto 1 p. 2003.
32 Raitamäki, Jouni, An approach to linguistic pattern recognition using fuzzy systems. 169 p. Yhteenveto 1 p. 2003.
33 Saalasti, Sami, Neural networks for heart rate time series analysis. 192 p. Yhteenveto 5 p. 2003.
34 Niemelä, Marketta, Visual search in graphical interfaces: a user psychological approach. 61 p. (148 p.) Yhteenveto 1 p. 2003.
35 You, Yu, Situation Awareness on the world wide web. 171 p. Yhteenveto 2 p. 2004.
36 Taatila, Vesa, The concept of organizational competence – A foundational analysis. - Perusteanalyysi organisaation kompetenssin käsitteestä. 111 p. Yhteenveto 2 p. 2004.

37 LYYTIKÄINEN, VIRPI, Contextual and structural metadata in enterprise document management. - Konteksti- ja rakennemetatieto organisaation dokumenttien hallinnassa. 73 p. (143 p.) Yhteenveto 1 p. 2004.
38 KAARIO, KIMMO, Resource allocation and load balancing mechanisms for providing quality of service in the Internet. 171 p. Yhteenveto 1 p. 2004.
39 ZHANG, ZHEYING, Model component reuse. Conceptual foundations and application in the metamodeling-based systems analysis and design environment. 76 p. (214 p.) Yhteenveto 1 p. 2004.
40 HAARALA, MARJO, Large-scale nonsmooth optimization variable metric bundle method with limited memory. 107 p. Yhteenveto 1 p. 2004.
41 KALVINE, VIKTOR, Scattering and point spectra for elliptic systems in domains with cylindrical ends. 82 p. 2004.
42 DEMENTIEVA, MARIA, Regularization in multistage cooperative games. 78 p. 2004.
43 MAARANEN, HEIKKI, On heuristic hybrid methods and structured point sets in global continuous optimization. 42 p. (168 p.) Yhteenveto 1 p. 2004.
44 FROLOV, MAXIM, Reliable control over approximation errors by functional type a posteriori estimates. 39 p. (112 p.) 2004.
45 ZHANG, JIAN, Qos- and revenue-aware resource allocation mechanisms in multiclass IP networks. 85 p. (224 p.) 2004.
46 KUJALA, JANNE, On computation in statistical models with a psychophysical application. 40 p. (104 p.) 2004.,
47 SOLBAKOV, VIATCHESLAV, Application of mathematical modeling for water environment problems. 66 p. (118 p.) 2004.
48 HIRVONEN, ARI P., Enterprise architecture planning in practice. The Perspectives of information and communication technology service provider and end-user. 44 p. (135 p.) Yhteenveto 2 p. 2005.
49 VARTIAINEN, TERO, Moral conflicts in a project course in information systems education. 320 p. Yhteenveto 1p. 2005.
50 HUOTARI, JOUNI, Integrating graphical information system models with visualization techniques. - Graafisten tietojärjestelmäkuvausten integrointi visualisointitekniikoilla. 56 p. (157 p.) Yhteenveto 1p. 2005.
51 WALLENIUS, EERO R., Control and management of multi-access wireless networks. 91 p. (192 p.) Yhteenveto 3 p. 2005.
52 LEPPÄNEN, MAURI, An ontological framework and a methodical skeleton for method engineering – A contextual approach. 702 p. Yhteenveto 2 p. 2005.
53 MATYUKEVICH, SERGEY, The nonstationary Maxwell system in domains with edges and conical points. 131 p. Yhteenveto 1 p. 2005.
54 SAYENKO, ALEXANDER, Adaptive scheduling for the QoS supported networks. 120 p. (217 p.) 2005.
55 KURJENNIEMI, JANNE, A study of TD-CDMA and WCDMA radio network enhancements. 144 p. (230 p.) Yhteenveto 1 p. 2005.
56 PECHENIZKIY, MYKOLA, Feature extraction for supervised learning in knowledge discovery systems. 86 p. (174 p.) Yhteenveto 2 p. 2005.
57 IKONEN, SAMULI, Efficient numerical methods for pricing American options. 43 p. (155 p.) Yhteenveto 1 p. 2005.
58 KÄRKKÄINEN, KARI, Shape sensitivity analysis for numerical solution of free boundary problems. 83 p. (119 p.) Yhteenveto 1 p. 2005.
59 HELFENSTEIN, SACHA, Transfer. Review, reconstruction, and resolution. 114 p. (206 p.) Yhteenveto 2 p. 2005.
60 NEVALA, KALEVI, Content-based design engineering thinking. In the search for approach. 64 p. (126 p.) Yhteenveto 1 p. 2005.
61 KATASONOV, ARTEM, Dependability aspects in the development and provision of location-based services. 157 p. Yhteenveto 1 p. 2006.
62 SARKKINEN, JARMO, Design as discourse: Representation, representational practice, and social practice. 86 p. (189 p.) Yhteenveto 1 p. 2006.
63 ÄYRÄMÖ, SAMI, Knowledge mining using robust clustering. 296 p. Yhteenveto 1 p. 2006.
64 IFINEDO, PRINCELY EMILI, Enterprise resource planning systems success assessment: An integrative framework. 133 p. (366 p.) Yhteenveto 3 p. 2006.
65 VIINIKAINEN, ARI, Quality of service and pricingin future multiple service class networks. 61 p. (196 p.) Yhteenveto 1 p. 2006.
66 WU, RUI, Methods for space-time parameter estimation in DS-CDMA arrays. 73 p. (121 p.) 2006.
67 PARKKOLA, HANNA, Designing ICT for mothers. User psychological approach. – Tieto- ja viestintätekniikoiden suunnittelu äideille. Käyttäjäpsykologinen näkökulma. 77 p. (173 p.) Yhteenveto 3 p. 2006.
68 HAKANEN, JUSSI, On potential of interactive multiobjective optimization in chemical process design. 75 p. (160 p.) Yhteenveto 2 p. 2006.
69 PUTTONEN, JANI, Mobility management in wireless networks. 112 p. (215 p.) Yhteenveto 1 p. 2006.
70 LUOSTARINEN, KARI, Resource , management methods for QoS supported networks. 60 p. (131 p.) 2006.
71 TURCHYN, PAVLO, Adaptive meshes in computer graphics and model-based simulation. 27 p. (79 p.) Yhteenveto 1 p.
72 ZHOVTOBRYUKH, DMYTRO, Context-aware web service composition. 290 p. Yhteenveto 2 p. 2006.

73 KOHVAKKO, NATALIYA, Context modeling and utilization in heterogeneous networks. 154 p. Yhteenveto 1 p. 2006.
74 MAZHELIS, OLEKSIY, Masquerader detection in mobile context based on behaviour and environment monitoring. 74 p. (179 p). Yhteenveto 1 p. 2007.
75 SILTANEN, JARMO, Quality of service and dynamic scheduling for traffic engineering in next generation networks. 88 p. (155 p.) 2007.
76 KUUVA, SARI, Content-based approach to experiencing visual art. - Sisältöperustainen lähestymistapa visuaalisen taiteen kokemiseen. 203 p. Yhteenveto 3 p. 2007.
77 RUOHONEN, TONI, Improving the operation of an emergency department by using a simulation model. 164 p. 2007.
78 NAUMENKO, ANTON, Semantics-based access control in business networks. 72 p. (215 p.) Yhteenveto 1 p. 2007.
79 WAHLSTEDT, ARI, Stakeholders' conceptions of learning in learning management systems development. - Osallistujien käsitykset oppimisesta oppimisympäristöjen kehittämisessä. 83 p. (130 p.) Yhteenveto 1 p. 2007.
80 ALANEN, OLLI, Quality of service for triple play services in heterogeneous networks. 88 p. (180 p.) Yhteenveto 1 p. 2007.
81 NERI, FERRANTE, Fitness diversity adaptation in memetic algorithms. 80 p. (185 p.) Yhteenveto 1 p. 2007.
82 KURHINEN, JANI, Information delivery in mobile peer-to-peer networks. 46 p. (106 p.) Yhteenveto 1 p. 2007.
83 KILPELÄINEN, TURO, Genre and ontology based business information architecture framework (GOBIAF). 74 p. (153 p.) Yhteenveto 1 p. 2007.
84 YEVSEYEVA, IRYNA, Solving classification problems with multicriteria decision aiding approaches. 182 p. Yhteenveto 1 p. 2007.
85 KANNISTO, ISTO, Optimized pricing, QoS and segmentation of managed ICT services. 45 p. (111 p.) Yhteenveto 1 p. 2007.
86 GORSHKOVA, ELENA, A posteriori error estimates and adaptive methods for incompressible viscous flow problems. 72 p. (129 p.) Yhteenveto 1 p. 2007.
87 LEGRAND, STEVE, Use of background real-world knowledge in ontologies for word sense disambiguation in the semantic web. 73 p. (144 p.) Yhteenveto 1 p. 2008.
88 HÄMÄLÄINEN, NIINA, Evaluation and measurement in enterprise and software architecture management. - Arviointi ja mittaaminen kokonais- ja ohjelmistoarkkitehtuurien hallinnassa. 91 p. (175 p.) Yhteenveto 1 p. 2008.
89 OJALA, ARTO, Internationalization of software firms: Finnish small and medium-sized software firms in Japan. 57 p. (180 p.) Yhteenveto 2 p. 2008.
90 LAITILA, ERKKI, Symbolic Analysis and Atomistic Model as a Basis for a Program Comprehension Methodology. 321 p. Yhteenveto 3 p. 2008.
91 NIHTILÄ, TIMO, Performance of Advanced Transmission and Reception Algorithms for High Speed Downlink Packet Access. 93 p. (186 p.) Yhteenveto 1 p. 2008.
92 SETÄMAA-KÄRKKÄINEN, ANNE, Network connection selection-solving a new multiobjective optimization problem. 52 p. (111p.) Yhteenveto 1 p. 2008.
93 PULKKINEN, MIRJA, Enterprise architecture as a collaboration tool. Discursive process for enterprise architecture management, planning and development. 130 p. (215 p.) Yhteenveto 2 p. 2008.
94 PAVLOVA, YULIA, Multistage coalition formation game of a self-enforcing international environmental agreement. 127 p. Yhteenveto 1 p. 2008.
95 NOUSIAINEN, TUULA, Children's involvement in the design of game-based learning environments. 297 p. Yhteenveto 2 p. 2008.
96 KUZNETSOV, NIKOLAY V., Stability and oscillations of dynamical systems. Theory and applications. 116 p. Yhteenveto 1 p. 2008.
97 KHRIYENKO, OLEKSIY, Adaptive semantic Web based environment for web resources. 193 p. Yhteenveto 1 p. 2008.
98 TIRRONEN, VILLE, Global optimization using memetic differential evolution with applications to low level machine vision. 98 p. (248 p.) Yhteenveto 1 p. 2008.
99 VALKONEN, TUOMO, Diff-convex combinations of Euclidean distances: A search for optima. 148 p. Yhteenveto 1 p. 2008.
100 SARAFANOV, OLEG, Asymptotic theory of resonant tunneling in quantum waveguides of variable cross-section. 69 p. Yhteenveto 1 p. 2008.
101 POZHARSKIY, ALEXEY, On the electron and phonon transport in locally periodical waveguides. 81 p. Yhteenveto 1 p. 2008.
102 AITTOKOSKI, TIMO, On challenges of simulation-based globaland multiobjective optimization. 80 p. (204 p.) Yhteenveto 1 p. 2009.
103 YALAHO, ANICET, Managing offshore outsourcing of software development using the ICT-supported unified process model: A cross-case analysis. 91 p. (307 p.) Yhteenveto 4 p. 2009.
104 KOLLANUS, SAMI, Tarkastuskäytänteiden kehittäminen ohjelmistoja tuottavissa organisaatioissa. - Improvement of inspection practices in software organizations. 179 p. Summary 4 p. 2009.
105 LEIKAS, JAANA, Life-Based Design. 'Form of life' as a foundation for ICT design for older adults. - Elämälähtöinen suunnittelu. Elämänmuoto ikääntyville tarkoitettujen ICT tuotteiden ja palvelujen suunnittelun lähtökohtana. 218 p. (318 p.) Yhteenveto 4 p. 2009.

106 Vasilyeva, Ekaterina, Tailoring of feedback in web-based learning systems: Certitude-based assessment with online multiple choice questions. 124 p. (184 p.) Yhteenveto 2 p. 2009.

107 Kudryashova, ElenaV., Cycles in continuous and discrete dynamical systems. Computations, computer assisted proofs, and computer experiments. 79 p. (152 p.) Yhteenveto 1 p. 2009.

108 Blackledge, Jonathan, Electromagnetic scattering and inverse scattering solutions for the analysis and processing of digital signals and images. 297 p. Yhteenveto 1 p. 2009.

109 Ivannikov, Andriy, Extraction of event-related potentials from electroencephalography data. - Herätepotentiaalien laskennallinen eristäminen EEG-havaintoaineistosta. 108 p. (150 p.) Yhteenveto 1 p. 2009.

110 kalyakin, Igor, Extraction of mismatch negativity from electroencephalography data. - Poikkeavuusnegatiivisuuden erottaminen EEG-signaalista. 47 p. (156 p.) Yhteenveto 1 p. 2010.

111 Heikkilä, Marikka, Coordination of complex operations over organisational boundaries. 265 p. Yhteenveto 3 p. 2010.

112 Fekete, Gábor, Network interface management in mobile and multihomed nodes. 94 p. (175 p.) Yhteenveto 1 p. 2010.

113 Kujala, Tuomo, Capacity, workload and mental contents - Exploring the foundations of driver distraction. 146 p. (253 p.) Yhteenveto 2 p. 2010.

114 Lugano, Giuseppe, Digital community design - Exploring the role of mobile social software in the process of digital convergence. 253 p. (316 p.) Yhteenveto 4 p. 2010.

115 Kampylis, Panagiotis, Fostering creative thinking. The role of primary teachers. - Luovaa ajattelua kehittämässä. Alakoulun opettajien rooli. 136 p. (268 p.) Yhteenveto 2 p. 2010.

116 Toivanen, Jukka, Shape optimization utilizing consistent sensitivities. - Muodon optimointi käyttäen konsistentteja herkkyyksiä. 55 p. (130p.) Yhteenveto 1 p. 2010.

117 Mattila, Keijo, Implementation techniques for the lattice Boltzmann method. - Virtausdynamiikan tietokonesimulaatioita Hila-Boltzmann -menetelmällä: implementointi ja reunaehdot. 177 p. (233 p.) Yhteenveto 1 p. 2010.

118 Cong, Fengyu, Evaluation and extraction of mismatch negativity through exploiting temporal, spectral, time-frequency, and spatial features. - Poikkeavuusnegatiivisuuden (MMN) erottaminen aivosähkönauhoituksista käyttäen ajallisia, spektraalisia, aikataajuus - ja tilapiirteitä. 57 p. (173 p.) Yhteenveto 1 p. 2010.

119 Liu, Shenghua, Interacting with intelligent agents. Key issues in agent-based decision support system design. 90 p. (143 p.) Yhteenveto 2 p. 2010.

120 Airaksinen, Tuomas, Numerical methods for acoustics and noise control. - Laskennallisia menetelmiä akustisiin ongelmiin ja melunvaimennukseen. 58 p. (133 p.) Yhteenveto 2 p. 2010.

121 Weber, Matthieu, Parallel global optimization Structuring populations in differential evolution. - Rinnakkainen globaalioptimointi. Populaation rakenteen määrittäminen differentiaalievoluutiossa. 70 p. (185 p.) Yhteenveto 2 p. 2010.

122 Väärämäki, Tapio, Next generation networks, mobility management and appliances in intelligent transport systems. - Seuraavan sukupolven tietoverkot, liikkuvuuden hallinta ja sovellutukset älykkäässä liikenteessä. 50 p. (111 p.) Yhteenveto 1 p. 2010.

123 Viukari, Leena, Tieto- ja viestintätekniikka-välitteisen palvelun kehittämisen kolme diskurssia. - Three discourses for an ICT-service development . 304 p. Summary 5 p. 2010.

124 Puurtinen, Tuomas, Numerical simulation of low temperature thermal conductance of corrugated nanofibers. - Poimutettujen nanokuitujen lämmönjohtavuuden numeerinen simulointi matalissa lämpötiloissa . 114 p. Yhteenveto 1 p. 2010.

125 Hiltunen, Leena, Enhancing web course design using action research . - Verkko-opetuksen suunnittelun kehittäminen toimintatutkimuksen keinoin . 192 p. Yhteenveto 2 p. 2010.

126 Aho, Kari, Enhancing system level performance of third generation cellular networks through VoIP and MBMS services. 121 p. (221 p.). Yhteenveto 2 p. 2010.

127 Häkkinen, Markku, Why alarms fail. A cognitive explanatory model. 102 p. (210 p.). Yhteenveto 1 p. 2010.

128 Pennanen, Anssi, A graph-based multigrid with applications. - Graafipohjainen monihilamenetelmä sovelluksineen. 52 p. (128 p.). Yhteenveto 2 p. 2010.

129 Ahlgren, Riikka, Software patterns, organizational learning and software process improvement. 70 p. (137 p.). Yhteenveto 1 p. 2011.

130 Nikitin, Sergiy, Dynamic aspects of industrial middleware architectures 52 p. (114 p.). Yhteenveto 1 p. 2011.

131 Sindhya, Karthik, Hybrid Evolutionary Multi-Objective Optimization with Enhanced Convergence and Diversity. 64 p. (160 p.). Yhteenveto 1 p. 2011.

132 Mali, Olli, Analysis of errors caused by incomplete knowledge of material data in mathematical models of elastic media. 111 p. Yhteenveto 2 p. 2011.
133 Mönkölä, Sanna, Numerical Simulation of Fluid-Structure Interaction Between Acoustic and Elastic Waves. 136 p. Yhteenveto 2 p. 2011.
134 Puranen, Tuukka, Metaheuristics Meet Metamodels. A Modeling Language and a Product Line Architecture for Route Optimization Systems. 270 p. Yhteenveto 1 p. 2011.