Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage. [http://researchspace.auckland.ac.nz/feedback](http://researchspace.auckland.ac.nz/feedback)

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the [Library Thesis Consent Form](http://researchspace.auckland.ac.nz/library/thesis-consent-form) and [Deposit Licence](http://researchspace.auckland.ac.nz/library/deposit-licence).

Note: Masters Theses

The digital copy of a masters thesis is as submitted for examination and contains no corrections. The print copy, usually available in the University Library, may contain corrections made by hand, which have been requested by the supervisor.
Methodologies for the Development of Qualitative Spatial and Temporal Reasoning Applications

Carl Schultz

Abstract

A great variety of scientific, engineering-based, and commercial application domains are fundamentally grounded in concepts of space and time. Over the last three decades there has been significant interest in more human-focused and intuitive qualitative spatial and temporal reasoning (QSTR) methods, which address the inherent limitations of purely numerical approaches for reasoning about space and time. However, despite the extremely significant theoretical advances that have been made in the QSTR field, there is a distinct absence of commercial and industrial applications that utilise QSTR calculi. The central issue is that relatively little research has addressed the unique challenges of designing and developing QSTR-based applications in comparison to more traditional systems that employ numerical processing techniques.

The primary objective of this thesis is to support software engineering practitioners in the development of applications that utilise QSTR calculi. Five QSTR application case studies, which cover a range of diverse application domains, are presented and analysed throughout this thesis to motivate the development of effective methodologies. Furthermore, a comprehensive definition of QSTR applications is presented to provide a formal basis for establishing methodologies that address three major areas of QSTR application development: requirements specification, design, and validation.

Design methodologies are presented that enable developers to evaluate the efficacy of numerous QSTR calculi with respect to QSTR application functional requirements. Additionally, the design methodologies adapt object-oriented concepts and machine learning techniques to facilitate the development of custom, high level, application-specific qualitative relations and constraints.

Four key validation methodologies are adapted from well known techniques in software engineering: unit testing, integration testing, test coverage, and mutation testing. Furthermore, a novel metric called H-complexity is presented and used to define four additional test coverage classes that a developer can employ to assess the efficacy of a test suite.

Finally, a meta-validation methodology is established that enables developers and the QSTR community to empirically investigate the efficacy of QSTR validation techniques. Experiments are conducted using the meta-validation methodology and the results are analysed to identify the most effective utilisation of QSTR validation methodologies according to the software development process being employed.
For Dad.
Acknowledgements

I am deeply grateful to my supervisors Assoc. Prof. Robert Amor and Prof. Hans Guesgen for their generosity and enthusiastic support. I have found their patience and eager willingness to share their wealth of knowledge inspiring.

I would very much like to acknowledge the Tertiary Education Commission for awarding me the Bright Future Top Achiever Doctoral Scholarship, which provided financial support during my PhD.

I wish to thank my bright shining best friend Ljika for all of the beautiful fun adventures and excitement, the art and the music and the coffee, and for all the happiness that she has shared with me.

I would like to thank my Tetka Jelena, who is such a joy and a delight, for always looking after me.

And finally I would like to say a very special thank you to my dearest mother Merlene for her kindness, love, and support.
Contents

1 Introduction 1
1.1 The Application of Qualitative Spatial and Temporal Reasoning 1
1.2 Objectives of this Thesis 3
1.3 State of the Art in Applying QSTR Calculi 4
1.4 Contributions of this Thesis 6
1.5 Organisation of this Thesis 7

2 Qualitative Spatial and Temporal Reasoning 11
2.1 Introduction 11
2.2 Early QSTR Calculi 11
2.2.1 Allen’s Interval Algebra 12
2.2.2 Guesgen’s Spatial Orientation Calculus (Rectangle Algebra) 13
2.2.3 Region Connection Calculus 15
2.2.4 Other Early QSTR Calculi Motivated by Allen’s Calculus 17
2.3 Unifying, Generalising and Augmenting QSTR Calculi 17
2.4 Algebraic Formulation of QSTR Calculi 19
2.5 Underlying Domains of Interpretation 20
2.6 Reasoning using Qualitative Spatial and Temporal Calculi 21
2.6.1 Path Consistency, Algebraic Closure, and Weak Composition 22
2.6.2 Tractability 23
2.7 Conceptual Neighbourhoods 24
2.8 Case Studies of QSTR Applications 26
2.8.1 QtvLight: Qualitative Decision Support for Architectural Lighting 26
2.8.2 TreeSap: Geographic Information System 27
2.8.3 SPBD: Qualitative Interface for Robotic Programming by Demonstra-
tion 28
2.8.4 Ul-Qayyum’s Qualitative Image Retrieval System 31
2.8.5 SailAway: Maritime Qualitative Navigation 31
2.9 Summary 32
3 Theoretical Foundations of QSTR Applications

3.1 Introduction ................................................. 35
3.2 Formal Definition of QSTR Applications ......................... 36
3.3 Actors and Roles ............................................. 39
3.4 Ambiguity ................................................... 40
3.5 Fundamental Operations on Scenarios ......................... 40
3.6 Characteristics of QSTR Applications .................. 41
   3.6.1 Reasoning Across a Broad Range of Abstraction Levels .... 42
   3.6.2 Continuity Assumption and Neighbourhoods for Changing Scenarios 42
   3.6.3 Modelling Infinite Domains .......................... 42
   3.6.4 Reasoning About Objects in Multi-Dimensional Models .... 43
3.7 Summary .................................................... 43

4 Formal Software Requirements for QSTR Applications .... 45

4.1 Introduction ................................................. 45
4.2 External Interface Requirements .............................. 48
4.3 Specifying Functional Requirements with an Idealised Application .... 50
4.4 A Complete Enumeration of QSTR Tasks ....................... 51
   4.4.1 Deriving Standard QSTR Tasks Using Fundamental Operations .... 53
   4.4.2 Deriving QSTR Tasks for Unordered Scenarios .................. 56
   4.4.3 Deriving QSTR Tasks for Ordered Scenarios ................... 58
   4.4.4 Deriving QSTR Tasks for Partially Ordered Scenarios ....... 64
4.5 Characterising QSTR Application Execution Behaviour .......... 65
4.6 Performance Requirements ................................ 68
4.7 Summary .................................................... 70

5 Organising and Visualising QSTR Applications ............. 73

5.1 Introduction ................................................. 73
5.2 Organising Relations and Constraints Into Model Fragments .... 74
5.3 Mapping Object Oriented Concepts to the QSTR Application Domain ... 77
   5.3.1 Design Pattern: Fragment Definitions .................. 78
   5.3.2 Design Pattern: Fragment Generalisation ................ 80
5.4 Adapting UML Sequence Diagrams to Visualise QSTR Application Behaviour .... 81
5.5 Summary .................................................... 84

6 Designing Low Level Application Fragments: Selecting QSTR Calculi .... 87

6.1 Introduction ................................................. 87
6.2 Defining Functional Consistency ................................ 91
6.3 Determining Functional Consistency in Consistent Unambiguous Scenarios ... 92
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4 Determining Functional Consistency in Consistent Ambiguous Scenarios</td>
<td>96</td>
</tr>
<tr>
<td>6.5 Determining Functional Consistency in Inconsistent Scenarios</td>
<td>97</td>
</tr>
<tr>
<td>6.6 Functional Consistency Analysis Methodology</td>
<td>98</td>
</tr>
<tr>
<td>6.7 Generating the Strongest QSTR Theory from the Idealised Application</td>
<td>106</td>
</tr>
<tr>
<td>6.8 Summary</td>
<td>107</td>
</tr>
<tr>
<td>7 Designing High Level Application Fragments: Utilising Data-Based Metrics and Deriving High Level Neighbourhood Graphs</td>
<td>109</td>
</tr>
<tr>
<td>8 Implementing QSTR Applications</td>
<td>131</td>
</tr>
<tr>
<td>9 Test Classes and Test Quality Metrics for Validating QSTR Applications</td>
<td>143</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>9.2.1 Formal Definitions for QSTR Validation</td>
<td>146</td>
</tr>
<tr>
<td>9.3 Unit Testing</td>
<td>147</td>
</tr>
<tr>
<td>9.3.1 Merging Decision Tables</td>
<td>149</td>
</tr>
<tr>
<td>9.3.2 Equivalence Class: Decision Table Row Combinations</td>
<td>149</td>
</tr>
<tr>
<td>9.3.3 Equivalence Class: Interaction of Rows With Different Condition</td>
<td>150</td>
</tr>
<tr>
<td>9.3.4 Refined Unit Testing</td>
<td>151</td>
</tr>
<tr>
<td>9.4 Integration Testing</td>
<td>152</td>
</tr>
<tr>
<td>9.4.1 Interaction Between Expressions in The Same Constraint</td>
<td>153</td>
</tr>
<tr>
<td>9.4.2 Interaction Between Expressions in Different Constraints</td>
<td>153</td>
</tr>
<tr>
<td>9.5 Test Coverage for Test Set Evaluation</td>
<td>154</td>
</tr>
<tr>
<td>9.5.1 Sentence Interaction Graph</td>
<td>155</td>
</tr>
<tr>
<td>9.5.2 Test Coverage Methodology Overview</td>
<td>156</td>
</tr>
<tr>
<td>9.5.3 Test Coverage Metrics</td>
<td>156</td>
</tr>
<tr>
<td>9.6 Fault Seeding for Test Set Evaluation</td>
<td>160</td>
</tr>
<tr>
<td>9.6.1 QSTR Mutation Testing Methodology</td>
<td>160</td>
</tr>
<tr>
<td>9.6.2 Selecting and Evaluating QSTR Mutation Operators</td>
<td>161</td>
</tr>
<tr>
<td>9.6.3 \textbf{abs} Mutation Operator</td>
<td>163</td>
</tr>
<tr>
<td>9.6.4 \textbf{ror} Mutation Operator</td>
<td>164</td>
</tr>
<tr>
<td>9.6.5 \textbf{CRT} and \textbf{ICE} Mutation Operators</td>
<td>165</td>
</tr>
<tr>
<td>9.6.6 Method Overloading: \textbf{NEM} and \textbf{NR} Mutation Operators</td>
<td>166</td>
</tr>
<tr>
<td>9.6.7 Remove Relation</td>
<td>167</td>
</tr>
<tr>
<td>9.7 Summary</td>
<td>168</td>
</tr>
</tbody>
</table>

10 H-Complexity for Analysing and Validating QSTR Applications         169

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 Introduction</td>
<td>169</td>
</tr>
<tr>
<td>10.2 H-Complexity Theory</td>
<td>170</td>
</tr>
<tr>
<td>10.2.1 Homogeneous Sets</td>
<td>170</td>
</tr>
<tr>
<td>10.2.2 H-complexity: Measuring Complexity With H Sets</td>
<td>170</td>
</tr>
<tr>
<td>10.2.3 Calculating H-Complexity</td>
<td>171</td>
</tr>
<tr>
<td>10.2.4 Partial Basic Queries, Partial H Sets and Partial Scenarios</td>
<td>175</td>
</tr>
<tr>
<td>10.3 Using H-complexity to Analyse QSTR Applications</td>
<td>176</td>
</tr>
<tr>
<td>10.3.1 Relation Arity</td>
<td>176</td>
</tr>
<tr>
<td>10.3.2 Number of Query Variables</td>
<td>178</td>
</tr>
<tr>
<td>10.3.3 Allowable Relation States</td>
<td>181</td>
</tr>
<tr>
<td>10.4 Using H-complexity to Quantify Test Coverage</td>
<td>181</td>
</tr>
<tr>
<td>10.4.1 Activity 1: Identify Component Domains</td>
<td>183</td>
</tr>
<tr>
<td>10.4.2 Activity 2: Specify Conditions to Refine the Test Space</td>
<td>183</td>
</tr>
<tr>
<td>10.4.3 Activity 3: Calculate H-Complexity for Refined Test Spaces</td>
<td>183</td>
</tr>
</tbody>
</table>
10.4.4 Activity 4: Calculate the Class of a Given Test Instance ............... 186
10.4.5 Activity 5: H-Complexity Coverage Metrics .......................... 187
10.4.6 Comparison Between Adapted and H-Complexity Test Coverage Met-
rics ................................................................. 189
10.4.7 Utilising Adapted and H-Complexity Test Coverage Metrics ............ 193
10.5 Summary .......................................................... 194

11 Evaluating Validation Techniques ................................. 197
11.1 Introduction .......................................................... 197
11.2 Evaluation Methodology ........................................... 198
  11.2.1 Metrics for Assessing Test Approach Efficacy ...................... 199
  11.2.2 Discrete Sampling Theory ....................................... 202
11.3 Generating Random Subject QSTR Applications ......................... 204
  11.3.1 Random Variables for Representing the Distribution of QSTR Applica-
   tions ................................................................. 206
  11.3.2 Parameters for the Structured Model of QSTR Applications ....... 208
  11.3.3 Parameters for Constraints within a Fragment ...................... 209
  11.3.4 Parameters for Constraints between Fragments ...................... 210
11.4 An Experiment for Evaluating the Presented QSTR Validation Techniques ... 211
  11.4.1 Experiment Procedure ........................................... 211
  11.4.2 Experiment Configuration and Parameter Values for Generating Ran-
   dom QSTR Applications ........................................ 213
  11.4.3 Pilot Study for Estimating the Standard Deviation of Mutation Scores 215
  11.4.4 Experiment Results and Analysis .................................. 217
11.5 Summary .......................................................... 225

12 Conclusions ......................................................... 229
12.1 Future Research .................................................. 237
12.2 QSTR Application Development using Functional Consistency Analysis and
   Future Research .................................................. 237
12.3 Discussion of Limitations and Future QSTR Meta-Validation Research .... 238

References .............................................................. 241

Ai
  A.1 Unit Tests ....................................................... 257
  A.2 Vertex Tests ..................................................... 257
  A.3 Test Script ...................................................... 258
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.4.4 Activity 4: Calculate the Class of a Given Test Instance</td>
<td>186</td>
</tr>
<tr>
<td>10.4.5 Activity 5: H-Complexity Coverage Metrics</td>
<td>187</td>
</tr>
<tr>
<td>10.4.6 Comparison Between Adapted and H-Complexity Test Coverage Met-</td>
<td>189</td>
</tr>
<tr>
<td>rics</td>
<td></td>
</tr>
<tr>
<td>10.4.7 Utilising Adapted and H-Complexity Test Coverage Metrics</td>
<td>193</td>
</tr>
<tr>
<td>10.5 Summary</td>
<td>194</td>
</tr>
<tr>
<td>11 Evaluating Validation Techniques</td>
<td>197</td>
</tr>
<tr>
<td>11.1 Introduction</td>
<td>197</td>
</tr>
<tr>
<td>11.2 Evaluation Methodology</td>
<td>198</td>
</tr>
<tr>
<td>11.2.1 Metrics for Assessing Test Approach Efficacy</td>
<td>199</td>
</tr>
<tr>
<td>11.2.2 Discrete Sampling Theory</td>
<td>202</td>
</tr>
<tr>
<td>11.3 Generating Random Subject QSTR Applications</td>
<td>204</td>
</tr>
<tr>
<td>11.3.1 Random Variables for Representing the Distribution of QSTR App-</td>
<td>206</td>
</tr>
<tr>
<td>plications</td>
<td></td>
</tr>
<tr>
<td>11.3.2 Parameters for the Structured Model of QSTR Applications</td>
<td>208</td>
</tr>
<tr>
<td>11.3.3 Parameters for Constraints within a Fragment</td>
<td>209</td>
</tr>
<tr>
<td>11.3.4 Parameters for Constraints between Fragments</td>
<td>210</td>
</tr>
<tr>
<td>11.4 An Experiment for Evaluating the Presented QSTR Validation Techn-</td>
<td>211</td>
</tr>
<tr>
<td>iques</td>
<td></td>
</tr>
<tr>
<td>11.4.1 Experiment Procedure</td>
<td>211</td>
</tr>
<tr>
<td>11.4.2 Experiment Configuration and Parameter Values for Generating Ra-</td>
<td>213</td>
</tr>
<tr>
<td>ndom QSTR Applications</td>
<td></td>
</tr>
<tr>
<td>11.4.3 Pilot Study for Estimating the Standard Deviation of Mutation S-</td>
<td>215</td>
</tr>
<tr>
<td>cores</td>
<td></td>
</tr>
<tr>
<td>11.4.4 Experiment Results and Analysis</td>
<td>217</td>
</tr>
<tr>
<td>11.5 Summary</td>
<td>225</td>
</tr>
<tr>
<td>12 Conclusions</td>
<td>229</td>
</tr>
<tr>
<td>12.1 Future Research</td>
<td>237</td>
</tr>
<tr>
<td>12.2 QSTR Application Development using Functional Consistency Analysis and Future Research</td>
<td>237</td>
</tr>
<tr>
<td>12.3 Discussion of Limitations and Future QSTR Meta-Validation Research</td>
<td>238</td>
</tr>
<tr>
<td>References</td>
<td>241</td>
</tr>
<tr>
<td>Appendix A</td>
<td></td>
</tr>
<tr>
<td>A.1 Unit Tests</td>
<td>257</td>
</tr>
<tr>
<td>A.2 Vertex Tests</td>
<td>257</td>
</tr>
<tr>
<td>A.3 Test Script</td>
<td>258</td>
</tr>
</tbody>
</table>
List of Figures

2.1 An example scenario of blocks illustrating Guesgen’s qualitative relations between objects (reproduced from [78]). .................................................. 15
2.2 The conceptual neighbourhood of Allen’s basic interval relations. ............... 25
2.3 Screenshot of the interface used to specify a query in TreeSap GIS. ............... 28
2.4 Screenshot of the transparency method used in TreeSap GIS to visualise results of the query “Find all Roads near a Specific Building (black circle)” ............... 29
2.5 Screenshot of the display threshold method used in TreeSap GIS to visualise results of the query “Find all Roads near a Specific Building (black circle)” for threshold values 100%, 75%, and 50%.................................................. 29
2.6 Sonar configuration in the Pioneer 3DX robot sonar ring. ............................... 30

4.1 A selection of associated external interface QSTR problem characteristics, ordered according to their dependencies. ....................... 49
4.2 Use case relationships for basic QSTR tasks.............................................. 52
4.3 QSTR application behaviour during the execution of a software application. .. 66
4.4 A selection of associated functional and performance QSTR problem characteristics, ordered according to their dependencies. ....................... 68

5.1 Example adapting sequence diagram notation to allow classifiers to represent multiple fragments, and to allow objects to appear in multiple classifiers. ....... 82
5.2 Using standard sequence diagram notation to illustrate semantics of the leftmost multi-fragment classifier in Figure 5.1.................................................. 82
5.3 Example using standard sequence diagram notation for classifiers with single fragments.................................................. 84

6.1 Scenario for which the idealised application produces output α. Interval x must always be before z. .................................................. 89
6.2 Scenario for which the idealised application produces output β. Region z is necessarily connected to x. .................................................. 90
7.1 Mean, upper quartile, and lower quartile cohesion and coupling distances of image categories. Landscape with mountains is abbreviated to *lwm*.

7.2 Graph of the near and far clustering from Table 7.1.

7.3 Partial decision Tree representation of the fragment definition of *coast* images.

7.4 Derived neighbourhood graph from generated fragment definitions presented in Table 7.2.

7.5 Reference domain fragment $F_2$ containing relations $R_1, \ldots, R_8$ and $R'_1, \ldots, R'_6$ with simple ordered neighbourhoods (left). The fragment definition space (right) consists of fragment definitions that specify one relation from $R_1, \ldots, R_8$ and one relation from $R'_1, \ldots, R'_6$.

7.6 Refined fragment definition spaces. (a) direct path restrictions, (b) critical path restrictions, (c) equivalence class of direct paths, (d) equivalence class of critical paths in conjunction with a direct path restriction.

8.1 Architectural overview of a Prolog implementation of a QSTR application.

8.2 Architectural overview of a relational database implementation of a QSTR application.

9.1 Flow graph of the QSTR validation process.

9.2 Sentence Interaction Graph (SIG) of a selection of domain rules for architectural lighting.

9.3 Overview of the process of calculating test coverage.

9.4 Partial ordering of coverage criteria from weakest to strongest (if full coverage is achieved).

9.5 Sentence interaction graph of the example application.

9.6 Overview of the process of calculating mutation score.

10.1 Two graphs representing binary relations $R_1$ and $R_2$.

10.2 Graph consisting of three disconnected subgraphs. Subgraphs $\{a, b\}$ and $\{c, d\}$ correspond to basic queries, where $\{e, f, g\}$ contains more than one basic query as induced subgraphs.

10.3 Overview of the process of calculating H-complexity coverage scores (regarding the input “*get app design*”, note that the developer may only be analysing a portion of the full QSTR application).

11.1 Overview of evaluation methodology for assessing QSTR validation techniques.

11.2 The estimated probability mass function, $f_{X|R_1}$, of the total number of relations in a QSTR application, generated by first specifying the number of fragments, and then incrementally populating each fragment with a random number of relations, uniformly distributed between 1 and 10.
# List of Tables

2.1 The thirteen basic qualitative relations between pairs of time intervals in Allen’s Interval Algebra. .......................................................... 12

2.2 The nine basic qualitative relations between pairs of objects in Guesgen’s Rectangle Algebra. ......................................................... 14

2.3 The eight basic qualitative relations between pairs of regions in Region Connection Calculus. ......................................................... 16

3.1 Comparing the domains of model theory, QSTR applications, and the roles of QSTR application designers and users. ...................... 40

3.2 Permitted operations on QSTR scenarios based on the combination of application components that are variable. .......................... 41

4.1 Comparing the QSTR application requirements methodologies presented in this chapter with IEEE Recommended Practice for Software Requirements Specifications (IEEE 830-1998) [179]. ......................................................... 47

4.2 Tasks that can be performed on a scenario with respect to the underlying parameters. In the state diagram, states are black circles, tasks (composed of states) are ovals with a task label, terminating states are double-lined circles, arrows are state transitions annotated with QSTR operations, and the arrow with no source state is the task entry point. ......................................................... 55

4.3 Tasks that can be performed on an unordered set of scenarios with respect to the underlying parameters. ................................... 57

4.4 Icons illustrating the important permutations of scenarios in an ordered sequence that are used to define a set of basic QSTR tasks. ....... 59

4.5 Tasks that can be performed on an ordered sequence of scenarios with respect to the underlying parameters. ............................ 63

4.6 Tasks that can be performed on a partially ordered set of scenarios with respect to the underlying parameters. ............................ 65
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7</td>
<td>Enumerating relevant paths through the QSTR application statechart in Figure 4.3 to determine basic application behavioural patterns. Characteristic values that uniquely define each behavioural class are bolded in the right column.</td>
</tr>
<tr>
<td>4.8</td>
<td>Each of the six application behaviour classes derived in Table 4.7 can be uniquely determined by their change source and task category characteristics.</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparison of QSTR applications and libraries based on the number of fragments and the number of relations per fragment. For each application the fragments are ordered by the level of abstraction, from highest to lowest.</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison of QSTR applications and libraries based on the number of high, intermediate, and low level fragments, and the number of relations per fragment.</td>
</tr>
<tr>
<td>5.3</td>
<td>Comparison of the custom qualitative components of QSTR applications and libraries (i.e. excluding QSTR calculi), based on the number of high, intermediate, and low level fragments, and the number of relations per fragment.</td>
</tr>
<tr>
<td>5.4</td>
<td>Adapting standard UML class diagram notation and concepts [141] to the QSTR application domain.</td>
</tr>
<tr>
<td>5.5</td>
<td>Adapting standard UML sequence diagram notation and concepts [141] to the QSTR application domain.</td>
</tr>
<tr>
<td>6.1</td>
<td>Six cases, defined as allowable combinations of four conditions.</td>
</tr>
<tr>
<td>6.2</td>
<td>Determining functional consistency depending on whether the scenario is guaranteed to be consistent or the scenario is possibly inconsistent, and whether the scenario is unambiguous (the network is atomic) or ambiguous (the network is non-atomic).</td>
</tr>
<tr>
<td>6.3</td>
<td>Classes of scenarios for which the QSTR application using IA will be functionally inconsistent.</td>
</tr>
<tr>
<td>6.4</td>
<td>Classes of scenarios for which the QSTR application using RCC8 will be functionally inconsistent.</td>
</tr>
<tr>
<td>6.5</td>
<td>Classes of scenarios for which the tank-valve application by [126] using IA will be functionally inconsistent.</td>
</tr>
<tr>
<td>7.1</td>
<td>Near and far clusters formed using mean coupling distance thresholds. Near threshold is cohesion mean +10%, far threshold is greatest coupling mean −10%.</td>
</tr>
<tr>
<td>7.2</td>
<td>Fragment definitions generated using the J48 decision tree learning algorithm, implemented in Weka [168], with confidence 0.25.</td>
</tr>
<tr>
<td>7.3</td>
<td>QSTR application developer guidelines for deriving effective neighbourhood graphs.</td>
</tr>
<tr>
<td>9.1</td>
<td>Decision tables for conditions in the exists clause (upper) and not exists clause (lower) of the left hand side set expression in the example constraint.</td>
</tr>
</tbody>
</table>
9.2 Two three-dimensional matrices representing the left hand side set expression of the architectural lighting constraint in Example 9.3.1. ........................................ 150
9.3 Decision tables for conditions in the exists clause (upper) and not exists clause (lower) of the relevant portions of the example set expression that separate triggers and overrides. ................................................................. 152
9.4 One-dimensional matrix representing the example set expression after merging the decision tables in Table 9.3. ................................................................. 152
9.5 Adapted coverage criteria for the example application. ........................................ 159
9.6 Adapting mutation operators for imperative and OO software to QSTR applica-
tions. .............................................................................................................. 164

10.1 Basic queries are derived from different combinations of relation state tuples. . 174
10.2 Homogeneous (H) sets are derived from different combinations of basic queries. 174
10.3 Scenarios are derived from different combinations of H sets being non-empty. . 175
10.4 Common conditions for refining test spaces. ................................................... 184

11.1 Mean and standard deviation of mutation scores (to 4 decimal places) for five testing approaches resulting from the pilot study with sample size \( n_p = 30 \). The cell with the highest standard deviation (Random Testing, Instance Creation Expression) is highlighted. ............................................................. 216
11.2 Mean and standard deviation of the number of components per QSTR application (to 4 decimal places). ................................................................. 217
11.3 Mean and standard deviation of the number of components per QSTR application to 4 decimal places. ................................................................. 218
11.4 Mean and standard deviation of the number of mutants generated per QSTR application to 4 decimal places . ................................................................. 219
11.5 Linear correlation between the number of mutants generated and the number of different application components. ................................................................. 220
11.6 Estimated means (\( \bar{y}_S \sim \bar{y}_U \)) and standard deviations (\( s \sim S \)) of mutation scores for the five test approaches for four different fault types. For each fault type, highlighted cells contain the highest mean mutation score. ............... 221
11.7 Minimum, mean, standard deviation of, and maximum number of tests in a testset for each of the five test approaches, where one testset is used to test one application. ................................................................. 223
11.8 Mean and standard deviation of the functional efficacy percentage scores for each test approach with respect to different fault types. The highlighted cells indicate the highest functional efficacy scores. ......................... 223
11.9 Minimum, mean, standard deviation of, and maximum number of assertions in a testset for each of the five test approaches. ........................................... 224
11.10 Mean and standard deviation of the number of inference steps required to determine an expected test result, and the mean and standard deviation of the number of constraints with non-empty domains in the initial premise scenario of a test (thus requiring further inspection by a human oracle). 225
11.11 Mean and standard deviation of the processing time in minutes per application (including all mutants) required by each validation method. 225
Chapter 1

Introduction

1.1 The Application of Qualitative Spatial and Temporal Reasoning

Spatial and temporal concepts pervade a tremendous range of scientific, engineering-based, and commercial application domains. In particular, representing and reasoning about spatial and temporal information is a deeply intrinsic component of geographic information systems, project scheduling and operations management, robotics and automated navigation, banking and transaction management systems, architecture and construction information technology, and audio and visual media processing.

Standard software tools that assist in these fields often rely on numerical approaches for processing spatial and temporal information. However, numerical methods have the following four significant limitations [47, 148].

1. Humans find numerical information considerably less intuitive than coarse qualitative information.

2. Numerical methods cannot effectively deal with incomplete information.\footnote{Standard arithmetical operators can not be applied to unknown quantities, for example $x = 5+?$ is undefined.} Moreover, even if information about the uncertainty of parameter values is available, such as range restrictions or probability distributions, the complex interaction of many numerical formulae can make the resulting value estimates unpredictably sensitive to small changes in the initial parameters.

3. Numerical methods often employ an inappropriate level of precision resulting in very computationally expensive processing, whereas rough, approximate and general information supplied by a simpler processing method may be more suitable.
4. Numerical methods have fundamental limitations in their ability to model vague and subjective information. In particular, many concepts that involve vagueness and ambiguity cannot be immediately expressed in numerical units, for instance, a numerical description of dramatic or tense impressions of an environment would be meaningless as the notion of drama cannot be captured by relationships between numerical units.

In response to the limitations of a purely numerical approach to processing spatial and temporal information, researchers in artificial intelligence have developed ancillary methods of qualitative spatial and temporal reasoning (QSTR). Motivated by the related area of commonsense reasoning \cite{84,98}, the objective of developing QSTR calculi is to formalise intuitive notions of physical relationships such as size, orientation, topology, and distance, and temporal relationships such as ordering, coincidence, and duration \cite{47}. Over the last three decades researchers have made significant progress in the theoretical foundations and analysis of QSTR calculi. Specifically, while reasoning in many QSTR formalisms has been shown to be NP hard \cite{158}, maximal tractable subsets of well known calculi have been identified \cite{13,123,125} and automatic methods for finding tractable subsets have been developed \cite{11}, thus informing a user about the classes of problems that are practical to solve. Techniques have also been developed that greatly improve reasoning performance \cite{4,18}.

Despite this theoretically advanced state of the field, there is a distinct absence of commercial and industrial applications that make significant use of QSTR calculi \cite{124,162}. The central issue is that the QSTR community is yet to address the unique challenges of designing and developing QSTR applications in comparison to more traditional systems that employ numerical processing techniques. An application that utilises QSTR calculi is necessarily embedded in a domain which is accompanied by a specialised and unique underlying theory of concepts and rules. In addition to this, applications are required to perform a variety of specialised context-specific tasks. Thus, while QSTR calculi can be effectively employed in a wide range of application domains, in almost all cases no combination of pre-existing QSTR calculi will perfectly and completely satisfy all application requirements. Hence, the application designer will need to formalise domain knowledge, and design complex, heterogeneous models that build on top of a mix of different existing QSTR calculi. Facilitating this QSTR application software development process is a key challenge for the QSTR community.

This thesis directly addresses this challenge by supporting the development of QSTR-based applications. In particular, the primary concern of an application designer is to meet the specific requirements set out by the end users while minimising the cost of development. This requires:

- methodologies for selecting the most appropriate QSTR calculus according to the application-specific requirements,
- methodologies for developing suitable higher-level qualitative models that address the necessary application-specific tasks, and
1.2 Objectives of this Thesis

The primary aim of this thesis is to establish methodologies that facilitate the development of applications that utilise QSTR calculi. The specific concrete objectives of this thesis are as follows.

1. **Identify salient characteristics of the problems that can be effectively addressed by QSTR applications.** Currently only ad-hoc guidelines are available for the types of problems which QSTR calculi are capable of addressing. Characterising these problems will greatly assist a software developer in identifying QSTR as a potential solution, thus promoting the application of QSTR calculi.

2. **Establish a practical and effective definition of QSTR applications.** At present there is no comprehensive definition that is sufficiently expressive to unify existing QSTR applications. Furthermore, the definition must provide insights into the nature of applying QSTR calculi by emphasising significant properties and tasks that are common to a wide range of QSTR applications.

3. **Establish a software engineering-based methodology for designing and validating QSTR applications.** At present there are no methodologies for designing, implementing, analysing, and validating QSTR applications, and even researchers in the field currently develop QSTR applications in a very ad hoc manner.

4. **Integrate software engineering principles and practices into a QSTR development methodology.** Software developers should not need to have an expert understanding of QSTR calculi in order to create QSTR applications. The central problem is that methodologies in software engineering have primarily focused on supporting object oriented
Chapter 1. Introduction

(OO) software development [93] resulting in a significant absence of software engineering approaches for analysing and managing knowledge-based systems [96]. Design methodologies derived from concepts in software engineering are required to bridge the gap between expert QSTR logicians and software developers from other disciplines.

The following sections present a brief overview of the history of QSTR calculi and a review of the literature in applying QSTR. Section 1.4 will then present the contributions of this thesis that meet these objectives.

1.3 State of the Art in Applying QSTR Calculi

The development of QSTR calculi is very often motivated by potential application areas that require more coarse, intuitive reasoning. For example, Allen employed qualitative temporal calculi for planning systems [22], Frank developed qualitative approaches to facilitate querying in GIS [63], and Moratz and Wallgrun developed a qualitative orientation calculus to assist in robotic navigation [9]. However, it is well recognised in the community that there is an absence of commercial applications, developed by application domain experts, that heavily employ QSTR calculi [80, 162, 169].

A number of unifying QSTR frameworks are now being developed in an effort to make the field of QSTR more cohesive and accessible to software developers. As discussed in Section 2.4, Ligozat, Mitra, Condotta, and Renz [114, 136] have established a unifying algebraic formulation for comparatively analysing a variety of QSTR calculi. Two prominent libraries that provide fast implementations of QSTR calculi are SparQ [162] and GQR [70].

While developing a library of efficient and robust implementations of QSTR calculi is an essential step in making these formalisms accessible, this does not directly address the issues discussed in the introduction of this chapter. In particular, applications operate in specialised domains of concepts and rules. The software developer is thus faced with two essential tasks: selecting suitable QSTR calculi and formalising application-specific domain knowledge by developing custom qualitative relations that integrate with the existing QSTR calculi. Relatively little research in QSTR has focused directly on these issues that software developers are faced with when applying QSTR calculi. The most relevant and recent research in supporting the application of QSTR will now be reviewed. In [15] Renz and Schmid investigate the impact that QSTR calculus customisation has on the properties of reasoning. In [80] Hahmann and Gruninger investigate whether QSTR calculi in general meet the ontological requirements of applications. However, research is still required to identify the functional limitations of an application based on a specific QSTR calculus. Moreover, research is required to determine how the application’s efficacy in meeting functional requirements will vary if an alternative QSTR calculus is used.
Research is required to integrate the different aspects of QSTR application development. Researchers in the related field of qualitative reasoning (QR) have developed a workbench software application called Garp3 [39] to support the process of designing and reasoning with qualitative models.² Garp3 is an integrated development environment for designing and reasoning about qualitative models of physical systems. The motivation for Garp3 is identical to the problems that the QSTR field currently encounters, namely that wider audiences can be reluctant to employ the advanced methods for modelling qualitative physics that have been developed (although this problem does not appear to be as significant as with QSTR, for example [91]). The central aim of Garp3 is to overcome this inertia by supporting modellers in specifying and reasoning about qualitative models in a graphically based, user-friendly, homogeneous workbench. In QSTR, a recent research project by Bhatt, Dylla and Hois [37] focuses on integrating aspects of QSTR application development.³ Bhatt, Dylla and Hois [37] have established a framework for developing software tools that support the early stages of a design process for smart environments. Their framework greatly facilitates the specification of functional requirements for ambient-intelligence environments by integrating QSTR calculi, ontologies, and numerical data-models within the context of the popular architectural design tool ArchiCAD. This framework is analysed in Chapter 5 of this thesis.

A key issue is that QSTR calculi are not easily accessible to developers with a background in software engineering. In particular, prominent methodologies in software engineering have primarily focused on supporting object oriented (OO) software development [93] resulting in a significant absence of software engineering approaches for analysing and managing knowledge based systems [96]. Many current research projects are attempting to address this problem by adapting OO concepts to declarative systems such as software complexity [121], modularity metrics [96], refactoring [119] and so on. In addition to QSTR applications being in the minority class of software systems analysed within the software engineering community, software developers need particular support in managing properties that are central to QSTR applications. Specifically, there is a strong emphasis on continuity that impacts on the types of problems for which QSTR applications excel at addressing, and the common use of infinite models that influences the methods that developers can use for validation. Adapting concepts that are well known in the software engineering field would greatly support software developers in accessing and utilising QSTR research. Moreover, the software development approaches that need to be adapted are the product of research within the software engineering field and would similarly support application-directed research in QSTR.

²QR is distinct from QSTR as it is primarily concerned with treating scalar quantities in a qualitative discrete way, rather than directly modelling commonsense spatial and temporal relationships.
³Although their project does not appear to be specifically motivated by the Garp3 workbench.
1.4 Contributions of this Thesis

This thesis addresses all the objectives presented in Section 1.2. The major contributions of this thesis are as follows.

- **A comprehensive definition of QSTR applications is established.** A rigorous formal definition of QSTR applications is presented in Chapters 3, 4, and 5 which is both consistent with prominent characteristics of QSTR applications and capable of expressing all case study QSTR applications. Specifically, the low-level theoretical foundations of QSTR applications are established in Chapter 3. Pertinent higher level application concepts derived from the theoretical foundations consist of an exhaustive enumeration of basic reasoning tasks and general runtime behaviour in Chapter 4, and the definition of structural concepts (fragments) and important classes of constraints (definitions, generalisations, neighbourhoods, etc.) in Chapters 5 and 7. Finally, the precise roles of actors in the QSTR application development process, namely users and software developers, are formally defined in terms of the underlying theoretical foundations of QSTR applications in Chapter 3.

- **Methodologies for key stages in QSTR application development are established, based on well known principles and concepts in software engineering.** The methodologies established in this thesis help to bridge the gap between software engineers and logicians by specifically targeting the following pertinent areas of software development: requirements specification (Chapter 4), application design and implementation (Chapters 5, 6, 7, and 8), application validation (Chapters 9 and 10), and validation methodology evaluation (Chapter 11).

- **A mapping is established between QSTR application concepts and object oriented (OO) concepts in the software engineering domain.** Specifically, in Chapter 5 QSTR application domain analogs of prominent OO concepts are defined such as classes, aggregates, generalisations, and associations. This mapping is particularly significant as it readily enables the utilisation of popular OO software engineering development frameworks and tools such as the unified modelling language class diagrams. Hence, this makes QSTR application development considerably more accessible to practitioners in software engineering.

- **A methodology is established for selecting the most suitable QSTR calculus with respect to application-specific functional requirements.** In Chapter 6 a collection of formulae are derived that precisely identify classes of scenarios for which a QSTR application will produce incorrect output according to different input scenario conditions. This enables a QSTR application developer to analyse myriad QSTR calculi in order to select the ideal combination based on their specific task requirements.
• A novel metric called H-complexity is developed that quantifies the expressiveness of a relational language. H-complexity is a highly versatile metric that is employed to support QSTR application design, validation, analysis, and implementation. A formula and an algorithm is derived in Chapter 10 for calculating the H-complexity of restricted (or refined) and unrestricted relational languages. Chapter 10 establishes principles for utilising H-complexity to manage application development resources. Furthermore, in Chapter 10 a collection of novel test coverage metrics based on H-complexity is defined.

1.5 Organisation of this Thesis

The primary focus of this thesis is facilitating the development of software applications that utilise QSTR calculi. Being motivated by prominent principles and practices in software engineering, the central objective of this thesis is to establish software development methodologies that support QSTR application developers in three main areas of software development: specifying formal requirements of QSTR applications in Chapter 4, the design of QSTR applications in Chapters 5-8, and the validation of QSTR applications in Chapters 9-11. The chapters of this thesis are organised as follows.

Chapter 2 presents an overview of the field of qualitative spatial and temporal reasoning. This chapter provides a review of the QSTR literature and presents the major concepts and theoretical advances in the field. Most importantly, five QSTR application case studies are presented.

Chapter 3 presents a rigorous theoretical foundation for QSTR applications. Based on the QSTR application case studies and the major theoretical advances in the QSTR field presented in the previous chapter, Chapter 3 presents a formal definition of QSTR applications and a formal definition of the roles of QSTR application developers and users. Salient characteristics of QSTR applications are identified.

Chapter 4 establishes formal requirements methodologies for QSTR applications. The theoretical foundation of QSTR applications established in the previous chapter is used to derive methodologies for specifying external interface requirements, functional requirements, and performance requirements for QSTR applications. These requirements characterise the types of problems that QSTR applications are highly effective at addressing.

Chapter 5 establishes a fundamental mapping between the concepts in object oriented (OO) software and the QSTR application domain. A central concept of fragments is introduced, which forms the basis of the mapping between the OO and QSTR application domain. This mapping greatly enhances the accessibility of QSTR applications, as it allows the adoption of prominent software development tools from the Unified Modelling Language (UML) for designing, organising, and conceptualising QSTR applications. An analysis of the fragment

\[ \text{In general H-complexity can be used in any relational system.} \]
structure of the QSTR application case studies is presented, which highlights the hierarchical organisation of QSTR applications. These results motivate the design methodologies in Chapters 6 and 7.

Chapter 6 focuses on facilitating the design of low level fragments by establishing a methodology for selecting the most effective QSTR calculi. This chapter builds on the functional requirements methodology of specifying idealised applications presented in Chapter 4. A key concept called functional consistency is defined to assess the performance of a QSTR application with respect to functional requirements specified in the idealised application. The central focus of the chapter is the derivation of equations that identify the classes of scenarios (as expressions in the interpretation language) for which a QSTR application will produce output that differs from the idealised application. A developer can use these mathematical tools to analyse the efficacy of myriad QSTR calculi according to their specific task requirements.

Chapter 7 focuses on facilitating the design of high level fragments by establishing methodologies that utilise data-based metrics and derive neighbourhood graphs for high level fragments. This chapter builds on the formalisation of fragment definitions presented in Section 5.3. Methodologies are presented that enable the developer to assess the efficacy of fragment definitions with respect to field data and domain-specific contextual information, and to utilise the hierarchical interactions between fragments for deriving high level conceptual neighbourhoods. Importantly, the definition of conceptual neighbourhoods is extensively generalised to enable the design of high level neighbourhoods that are consistent with the structure of fragments in a QSTR application.

Chapter 8 presents approaches for implementing QSTR applications in software. Basic software architectures are presented that integrate highly specialised QSTR reasoners which implement standard QSTR calculi (selected using methodologies in Chapter 6) with implementations of high level custom fragments (designed using methodologies in Chapter 7). Strategies are presented for implementing the representational components of QSTR applications that are formally defined in Chapter 3. Furthermore, principles for implementing deductive closure and scenario consistency checking algorithms are presented using Prolog as an example.

Chapter 9 establishes methodologies for QSTR application validation, inspired by research in software engineering. Two prominent white-box testing methodologies are adapted from software engineering, namely unit testing and integration testing. Additionally, two methodologies for analysing the efficacy of a test suite at detecting faults are adapted from software engineering, namely test coverage and mutation testing. Test coverage and mutation testing can be employed by developers to determine whether an application has been sufficiently tested.

Chapter 10 establishes a QSTR application validation methodology based on a novel language complexity metric called H-complexity. While in the previous chapter validation methods were adapted from software engineering, in this chapter H-complexity (which is used in validation) is developed specifically for relation-based logical systems. H-complexity is de-
rived from the relational theoretic model that underlies QSTR applications presented in Chapter 3 by employing research in finite model theory. H-complexity is then used to develop validation tools to support QSTR application developers.

Chapter 11 establishes a QSTR meta-validation methodology for evaluating the efficacy of alternative QSTR validation approaches. The QSTR evaluation validation methodology employs mutation testing, presented in Chapter 9, to enable developers to conduct experiments that empirically investigate whether a given set of QSTR validation techniques are effective at detecting faults in QSTR application designs. An experiment is conducted using this methodology to evaluate the relative fault-detecting efficacy of the validation methods presented in Chapter 9. The results of the experiments are analysed to determine the most effective utilisation of the respective validation methodologies according to software development process being employed.

Chapter 12 presents the conclusions of this thesis. This chapter summarises the definition of QSTR applications established in this thesis. The most important aspects of the methodologies that are presented in this thesis are highlighted with respect to the three main areas of QSTR application software development that are addressed: QSTR application requirements, design, and validation. This chapter concludes by presenting future research directions for supporting the application of QSTR.
Chapter 2

Qualitative Spatial and Temporal Reasoning

2.1 Introduction

In this chapter an overview of the field of qualitative spatial and temporal reasoning (QSTR) is presented. QSTR calculi are used to represent and reasoning about scenarios that are described using qualitative relations between spatial and temporal objects. Section 2.2 reviews early prominent QSTR calculi. Section 2.3 presents research in combining and generalising QSTR calculi to reason about more complex models of space and time. Section 2.4 presents the reformulation of QSTR calculi as relation algebras, which assisted in the identification of salient properties that are shared by many QSTR calculi. Section 2.5 presents the notion of a domain of interpretation with respect to QSTR calculi. Section 2.6 presents the central concepts and major theoretical advances in reasoning using qualitative spatial and temporal calculi. Section 2.7 introduces the important notion of conceptual neighbours, which are used for a number of QSTR reasoning tasks. Section 2.8 presents five case studies of QSTR applications, which will be referred to throughout this thesis.

2.2 Early QSTR Calculi

A large variety of QSTR calculi have now been developed that are directly based on a particular constraint satisfaction encoding that was introduced in 1983 by Allen [23]. In this section a number of prominent, seminal QSTR calculi are reviewed that have contributed significantly to the development of the field.
### Basic Interval Algebra relations

<table>
<thead>
<tr>
<th>Relation</th>
<th>Pictorial representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$ is before $y$</td>
<td><img src="image" alt="before" /></td>
</tr>
<tr>
<td>$y$ is after $x$</td>
<td></td>
</tr>
<tr>
<td>$x$ meets $y$</td>
<td><img src="image" alt="meet" /></td>
</tr>
<tr>
<td>$y$ is met by $x$</td>
<td></td>
</tr>
<tr>
<td>$x$ overlaps $y$</td>
<td><img src="image" alt="overlap" /></td>
</tr>
<tr>
<td>$y$ is overlapped by $x$</td>
<td></td>
</tr>
<tr>
<td>$x$ starts $y$</td>
<td><img src="image" alt="start" /></td>
</tr>
<tr>
<td>$y$ is started by $x$</td>
<td></td>
</tr>
<tr>
<td>$x$ is during $y$</td>
<td><img src="image" alt="during" /></td>
</tr>
<tr>
<td>$y$ contains $x$</td>
<td></td>
</tr>
<tr>
<td>$x$ finishes $y$</td>
<td><img src="image" alt="finish" /></td>
</tr>
<tr>
<td>$y$ is finished by $x$</td>
<td></td>
</tr>
<tr>
<td>$x$ equals $y$</td>
<td><img src="image" alt="equal" /></td>
</tr>
</tbody>
</table>

**Table 2.1:** The thirteen basic qualitative relations between pairs of time intervals in Allen’s Interval Algebra.

### 2.2.1 Allen’s Interval Algebra

Allen significantly influenced the field of temporal reasoning by presenting a qualitative calculus of temporal intervals [1, 23], known as Allen’s Interval Algebra (IA). An important feature originally presented by Whitehead [166] (as cited in [34]) is to treat high-level concepts such as time intervals as the primitive unit to be reasoned about rather than mathematical abstractions such as time points. Based on a temporal model developed by Bruce [41] (Section 3.2, Definitions 5 and 6), Allen presented a set of thirteen jointly exhaustive, pairwise disjoint (JEPD) temporal relations that can hold between pairs of intervals: before, meets, overlaps, starts, during, contains, overlapped by, met by, after, and equals. The thirteen basic Allen relations, denoted $B$, are illustrated in Table 2.1.
Allen’s seminal contribution was the formulation of the task of reasoning about incomplete temporal information as a constraint satisfaction problem (CSP). A CSP network \( \mathcal{N} = (N, v) \) is a set of variables \( N \) representing objects in a scenario and a function \( v : N \times N \rightarrow 2^B \) that specifies the relations that possibly hold between two objects. If the scenario is unambiguous then each pair of objects is assigned to exactly one basic relation, i.e. for all \( x, y \in N, v(x, y) \in B \); the corresponding CSP network is referred to as an atomic network. In contrast, if the scenario is ambiguous then some pair of objects is assigned to multiple basic relations, i.e. there exists \( x, y \in N, |v(x, y)| \geq 2 \).

The primary reasoning mechanism in Allen’s Interval Algebra is relation composition. Given three objects, \( x, y, z \), the relations that can possibly hold between \( x \) and \( z \) are restricted by the relation \( R \in B \) that holds between \( x \) and \( y \), and the relation \( S \in B \) that holds between \( y \) and \( z \). Formally, let \( \circ : 2^B \times 2^B \rightarrow 2^B \) be the composition operator; compositional reasoning in IA requires that, for all \( x, y, z \), \( R(x, y) \wedge S(y, z) \rightarrow \bigvee_{T \in R \circ S} T(x, z) \). Allen encoded these constraints in a composition table, thus greatly simplifying compositional reasoning with an efficient look-up reference procedure. Each entry in the composition table specifies the composition of two basic relations \( R, S \in B \), and the composition of sets of basic relations \( \{R_1, \ldots, R_n\}, \{S_1, \ldots, S_m\} \in 2^B \) is the union of composition pairs from the Cartesian product of the sets of relations, \( \{R_1, \ldots, R_n\} \circ \{S_1, \ldots, S_m\} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} R_i \circ S_j \).

Allen presented an algorithm that eliminates local inconsistencies between relation triples, based on the standard CSP path-consistency algorithms ([120], particularly Section 4, and [116] PC-2 Section 7.5). To summarise, for all \( i, j, k \in N \), Allen’s path-consistency algorithm repeatedly updates \( v(i, j) \) to \( v(i, j) \cap (v(i, k) \circ v(k, j)) \) until no further updates occur. Allen’s algorithm executes in worst case polynomial time and is thus tractable. However, Allen observed that, while the algorithm is sound in the sense that no consistent relations are eliminated, the algorithm is incomplete, and therefore does not eliminate all inconsistent relations [23]. In Section 2.6 a more detailed review of the properties of Allen’s algorithm will be presented.

### 2.2.2 Guesgen’s Spatial Orientation Calculus (Rectangle Algebra)

In addition to qualitative temporal reasoning, Allen’s Interval Algebra has motivated a large number formalisms for reasoning about spatial objects and relationships in the area of qualitative spatial reasoning. The first qualitative spatial reasoning calculus that was directly based on Allen’s composition table approach was Guesgen’s orientation system [78], which is now referred to as either the rectangle algebra or the block algebra [2].

Guesgen presented a cognitively motivated one-dimensional spatial logic for representing relative spatial relationships between objects as opposed to employing absolute object positions. Guesgen adapted Allen’s temporal logic by recognising the direct isomorphism between the timeline and a one-dimensional spatial structure. This is used to define nine JEPD relationships
Chapter 2. Qualitative Spatial and Temporal Reasoning

<table>
<thead>
<tr>
<th>Basic Interval Algebra relations</th>
<th>Pictorial representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$ is left of $y$</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>$y$ is right of $x$</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>$x$ right side attached to $y$</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>$y$ left side attached to $x$</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>$x$ overlaps $y$</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>$y$ is overlapped by $x$</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>$x$ inside $y$</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>$y$ contains $x$</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>$x$ equals $y$</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Table 2.2: The nine basic qualitative relations between pairs of objects in Guesgen’s Rectangle Algebra.

based on Allen’s temporal relations: left of, right side attached to, overlaps, inside, contains, overlapped by, left side attached to, right of, and equals. Guesgen’s nine basic relations are illustrated in Table 2.2.

Following Allen’s approach, a composition table is provided that specifies the set of qualitative relations that can possibly hold over the composition of particular pairs of relations. Guesgen extended this orientation system to spatial relationships in higher dimensions by representing qualitative information on multiple orthogonal axes and reasoning about each axis independently; given two objects $x, y$ the relations are represented by the tuple

$$x (\text{horizontal axis, vertical axis, depth axis}) y.$$  

Guesgen provides an example of a scenario of blocks illustrated in Figure 2.1 [78]. A qualitative description of the scenario using Guesgen’s orientation relations is

$$x (\text{inside, left side (bottom) attached to, inside}) y,$$

$$y (\text{left of, inside, overlaps}) z.$$  

Reasoning is used to infer the relations that possibly hold between $x$ and $z$. 


2.2 Early QSTR Calculi

2.2.3 Region Connection Calculus

Another early and very prominent qualitative spatial reasoning formalism that was motivated by Allen’s compositional reasoning approach is the Region Connection Calculus (RCC) originally developed by Randell, Cui, and Cohn [135]. RCC is a family of spatial reasoning approaches based on qualitative topological relationships that can hold between arbitrary regular regions of uniform dimensionality. The most well known region calculus is the RCC8 which defines a composition table for a set of eight JEPD relationships: disconnected, externally connected, partially overlaps, tangential proper part, tangentially contains, non-tangential proper part, non-tangentially contains, and equals. The eight basic RCC8 relations are illustrated in Table 2.3.

Interestingly, a few years earlier Egenhofer had arrived at the same set of qualitative topological relationships. The primary distinctions between these reasoning systems are that (a) Egenhofer’s model of regions is based on point-set topology as opposed to using arbitrary topological spaces with regions as the primitive object, and (b) Egenhofer explicitly restricts the domain of discourse to connected regions in a two dimensional plane with no holes. These distinctions have important consequences for reasoning, in particular, in determining whether a partially described scenario is consistent. The impact that the choice of domain has on reasoning is discussed in Section 2.6.

RCC is also closely related to a family of QSTR calculi known as contact relation algebras. Ladkin and Maddux first presented the containment algebra\(^1\) [104] in the context of one dimensional qualitative temporal reasoning. When this algebra is employed for reasoning about

\(^1\)The containment algebra was developed by Ladkin and Hayes, see Acknowledgements in [104].
Table 2.3: The eight basic qualitative relations between pairs of regions in Region Connection Calculus.
regions of arbitrary dimension the calculus corresponds to a well known spatial reasoning system called RCC5 presented by Bennett [32].

2.2.4 Other Early QSTR Calculi Motivated by Allen’s Calculus

This section reviews a number of early qualitative temporal and spatial calculi that were motivated by Allen’s compositional reasoning approach. An overview of the general properties of early QSTR calculi can be found in [111] and [68].

In the area of qualitative temporal reasoning, Ladkin et al. [102] present a qualitative calculus of branching time. In addition to the thirteen primitive interval relations introduced by Allen, in branching time the endpoints of intervals can be unrelated. This yields eleven new qualitative relations such as unrelated, adjacent, touching, partially proceeds, and so on. Rodriguez et al. [140] present a reasoning mechanism for a qualitative calculus of relativistic time that defines qualitative relations between incomparable time intervals. Relativistic time can be used, for example, to reason about concurrency in a distributed system or to model and reason about relativity in a space-time continuum. The authors develop an Allen-based composition approach for an axiomatisation of relativistic time by Lamport [106].

With respect to the qualitative spatial domain, calculi are generally divided into three broad categories: orientation, mereotopology, and distance. Mukerjee and Joe [122] develop a system for reasoning about the relative orientation of objects by employing IA to reason about intervals along both orthogonally aligned and arbitrarily aligned spatial axes. Given two objects X and Y, the authors define the relative direction of Y with respect to an intrinsic front of X, and the relative position of Y with respect to the intersection of the projected lines of travel of X and Y. Transitive reasoning is used to determine the directional and positional relations between objects X and Z given the relations between X and Y, and Y and Z. Maddux introduces a family of compass algebras [117] (revised in [7]) that generalises Allen’s interval reasoning to allow spatial reasoning about cardinal directions in two dimensions. In [3] Frank also develops a calculus for reasoning about cardinal directions, which is strongly motivated by the need for intuitive reasoning in geographic information systems. Frank explores alternative approaches for partitioning space into regions labelled with cardinal directions, such as projecting cones from a point, defining half planes, and incorporating a neutral region. In [108] Ligozat extensively analyses the properties of cardinal direction calculi.

2.3 Unifying, Generalising and Augmenting QSTR Calculi

This section reviews research on the integration and generalisation of different QSTR calculi to enable the reasoning of relations between a broader range of objects. These calculi are highly
relevant to the application of QSTR, as it is necessary for application developers to combine and customise QSTR calculi in order to satisfy application-specific task requirements.

In the qualitative temporal reasoning domain, Vilain [157] developed an integrated qualitative relation calculus for reasoning about both time intervals and time points, which is based on Allen’s Interval Algebra [24] (as cited in [157]). It was shown that reasoning in this extended formalism is still tractable. Ladkin [101] introduced the notion of representing time intervals that consist of disconnected periods of time, referred to as non-convex intervals. Ligozat [112, 113] generalised both of these temporal extensions by developing a QSTR calculus that is capable of reasoning about time points, convex time intervals, and non-convex time intervals.

In [133] Pujari, Kumari, and Sattar present an interval calculus that combines Allen’s relations with qualitative information about the relative duration of time intervals, referred to as the INDU calculus. The relative duration between intervals is represented by the relations of an algebra of points (PA) [159]: <, =, and >. The INDU calculus exhibits extremely interesting and unusual behaviour [27]. In particular, determining whether an INDU scenario is consistent is significantly more complicated than determining the consistency of either IA or PA scenarios. The INDU calculus illustrates the important principle that integrated QSTR calculi in general do not inherit the properties of their constituent calculi.

In the domain of qualitative spatial reasoning, Frank [63] analyses a number of variant calculi that combine cardinal relations and distance relations. As with the INDU calculus the interaction between cardinal and distance calculi is non-trivial; in some cases the ordering of the composition of object pairs influences the resulting inferred information (i.e. composition is no longer associative) ([63] Sections 5.3 and 5.4). Despite this, Frank showed that spatial inferences are refined by combining distance and cardinal directions as opposed to reasoning about the spatial domains independently ([63] Section 8.2). Thus, combining QSTR calculi can reduce the amount of ambiguity produced by reasoning. Similarly, Hernandez [87] showed that combining topological and orientation relations in a two dimensional model of space can reduce reasoning ambiguity ([87] Section 5.2.3).

Recently Wolfl and Westphal [19] formalised the notion of integrating QSTR calculi. They define a loose integration as a formalism in which, firstly each constituent calculus is reasoned about independently, and secondly the inferred information is propagated between the calculi to eliminate further inconsistencies. Gerevini and Renz [72] have developed a modified path-consistency algorithm that determines the consistency of scenarios expressed using loosely integrated QSTR calculi. A tight integration defines a new set of basic relations that explicitly expresses all semantic interdependencies. Formally, given two QSTR calculi $X,Y$, the set of basic relations in the tight integration is $\{ R^X \cap R^Y | R^X \in B^X, R^Y \in B^Y, R^X \cap R^Y \neq \emptyset \}$, where $B^X, B^Y$ are the sets of basic relations in $X,Y$ respectively [19]. Interestingly, despite tight integrations often being considerably more expressive than loose integrations, in some cases reasoning in
tight integrations is still tractable. Finally, Wolfl and Westphal also define orthogonality as the lack of semantic dependencies. For example, the cardinal direction calculus corresponds to the Cartesian product of the Point Algebra with itself (representing the orthogonal $x$ and $y$ axes); this is an orthogonal calculus integration because any pair of point algebra relations is valid. Allen’s Interval Algebra also corresponds to Cartesian products of the Point Algebra with itself (each IA relation can be interpreted as a 4-tuple comparing the startpoints and endpoints of two intervals); this is a non-orthogonal calculus integration as certain combinations are invalid such as $(x^- > y^-, x^+ < y^-, \ldots)$, where $x^-$ is the startpoint of interval $x$ and $x^+$ is the endpoint of interval $x$. If calculi are orthogonal then their loose and tight integrations are equivalent.

### 2.4 Algebraic Formulation of QSTR Calculi

Early on, Ladkin and Maddux [99, 104] reformulated Allen’s Interval Algebra as a relation algebra as defined by Tarski [154] (as cited in [104]) which is a slight extension of a Boolean algebra. This reformulation lead to central insights into the underlying properties of Allen’s calculus. Most notably Ladkin and Maddux showed that Allen’s set of axioms is complete, meaning that all relevant formulae are derivable from the current set, and that the only scenario that satisfies all of Allen’s axioms is the set of intervals that have rational $\mathbb{Q}$ endpoints $i, j$ (i.e. endpoints that are interpreted on a dense, unbounded, linear ordering),

$\{(i, j) \in \mathbb{Q} \mid i < j\}$.

Thus, there exists a unique, valid scenario with discrete intervals, and it contains an infinite number of intervals.\(^2\) It follows that an algorithm can be developed that will visit any given valid interval in a finite amount of time, which is called recursive enumerability. The significance is that completeness and recursively enumerability imply decidability, proving that Allen’s calculus is considerably more practical for temporal reasoning than unrestricted first-order logic which is undecidable [104].

Intuitively, a scenario containing an infinite number of intervals is required to satisfy strong composition (or extensional composition [35]), whereas Allen based QSTR calculi typically reason using weak composition [6, 14, 114]. That is, compositional reasoning in an Allen-based QSTR calculus employs the rule

$$R(x, y) \land S(y, z) \rightarrow T_1(x, z) \lor \ldots \lor T_m(x, z),$$

where the consequent relations are taken from the composition table for $R, S$. In particular, this only asserts that the consequent is necessary but not sufficient,

$$R(x, y) \land S(y, z) \leftarrow T_1(x, z) \lor \ldots \lor T_m(x, z).$$

\(^2\)In model theoretic terminology this is referred to as $\mathfrak{K}_0$ categoricity, meaning that the theory has one infinite model up to isomorphism.
Strong composition requires that each entry in the composition table is both necessary and sufficient. If the Interval Algebra is interpreted in an infinite non-dense model such as the set of all integers then it no longer satisfies strong composition [100]. For example, given two intervals $x, z$ such that $x^+ = 2$ and $z^- = 3$ then before $(x, z) \rightarrow \exists y \cdot$ before $(x, y) \land$ before $(y, z)$ can only be satisfied in a dense domain, as no integer interval exists that has a duration that is less than 1.

Maddux, Ladkin, Ligozat, Condotta, Mitra, Renz, and others have used the perspective of relation algebras to unify the great variety of QSTR calculi [5, 55–57, 103, 104, 114, 136]. The relation algebra represents the properties of a QSTR calculus by defining operators on the relation symbols without referring to any particular scenarios, while the models of a corresponding first order theory represent the actual qualitative scenarios that are valid according to the calculus. Ligozat introduced the concept of weak representations [113] to connect the algebraic formulation to the models of the logical formulation. This is particularly relevant for the task of establishing a definition of QSTR applications that is consistent with QSTR calculi.

### 2.5 Underlying Domains of Interpretation

QSTR calculi are interpreted in a domain, or a universe $U$, of objects ( [114] Section 3.3; [5] Section 2.2; [136] Section 2.4; [6] Section 2.2). An example of a domain of interpretation is that an interval $x \in U$ is a pair $(w_1, w_2)$ on a linear ordering $w_1, w_2 \in W$ such as integers or rationals, where $w_1 < w_2$. Thus, the relation before in IA is interpreted as before $(x, y) \leftrightarrow x_{w_2} < y_{w_1}$. Some common domains are:

- an unbounded, dense partial ordering of points (e.g. cardinal directions, or any n-dimensional point calculus where $n > 1$);
- nonlinear (partial) ordering of time points (e.g. relativistic time);
- an unbounded, dense, linear ordering of points (e.g. end points of intervals for Allen’s calculus and time points);
- a circle split into $n$ intervals (e.g. cyclic intervals [28]);
- non-empty closed regular subsets of a topological space (e.g. RCC);
- set of all closed disks (circles) on a 2D plane;
- set of all closed, connected, arbitrarily shaped regions on a 2D plane.

The choice of domain has a large impact on the properties of the QSTR calculus, as presented in the following sections.
A common characteristic of QSTR calculi is that the universes are infinite. That is, an infinite number of temporal objects such as intervals or points on the timeline, and an infinite number of spatial objects such as regions in $n$-dimensional space are required to satisfy the axioms of QSTR calculi in the strong sense as discussed in Section 2.4.

### 2.6 Reasoning using Qualitative Spatial and Temporal Calculi

A network is consistent (or satisfiable) if there is an instantiation of the variables in some desired domain of interpretation that satisfies all of the qualitative constraints. If there are multiple possible relations that can hold between two objects $x, y$, then for each relation $R \in v(x,y)$ there must be some consistent instantiation of all objects that also satisfies $R$ (see Theorem 3.2 on minimal networks in [120] and Theorem 3.4 with its Corollary; [116] Section 7.4; [52] Section 2, page 67; feasible and minimal labels in [31], Section 2.2; more recently [55] Section 6 and [136] Sections 2.1 to 2.4). The task of determining whether a network is consistent is referred to as ISAT, and the task of determining the strongest implied relation between each pair of objects is called ISI [29, 73, 125, 158]. In [73] Golumbic and Shamir also define the all consistent solutions problem such that all combinations of possible relations must be consistent (which is stronger than the minimal labelling problem). All of these tasks are essentially equivalent (i.e. they are polynomially Turing-reducible) as proven in [73] Proposition 3.1 and [125] Proposition 2.1.

Allen’s algorithm is not capable of determining consistency of non-atomic interval algebra networks (i.e. networks where some pairs of objects have multiple possible relations). Vilain, Kautz, and van Beek [158, 159] proved that determining consistency for non-atomic interval algebra networks is NP-hard. Despite this, Allen’s algorithm is capable of determining consistency when interval algebra networks are atomic (i.e. networks where all pairs of objects have exactly one possible qualitative relation).

In general, Ligozat [5] proved that Allen’s algorithm can determine whether atomic networks are consistent for any calculus with a linearly ordered domain such as Allen’s interval algebra, or where the domain is a Cartesian product of linear orderings, such as cardinal directions [3, 108], $n$-point algebras [26] or the block algebra [2, 78]. Moreover, Nebel [123] proved that Allen’s algorithm can determine consistency of atomic RCC8 networks, which operate in a significantly less structured domain compared to linearly and partially ordered domains.

Unfortunately, there are some calculi for which Allen’s algorithm can not even decide consistency of atomic networks, such as the containment algebra ( [103] Theorem 5.9) and the INDU calculus [27, 133]. Renz and Ligozat [14] have proven that Allen’s algorithm is capable of determining whether an atomic network is consistent if there are no qualitative relations in
the calculus that can be partitioned into two or more finer qualitative relations; a QSTR calculus with this property is referred to as being closed under constraints.

2.6.1 Path Consistency, Algebraic Closure, and Weak Composition

Allen’s algorithm was an adaption of the path-consistency algorithms developed in the constraint satisfaction community ([120] and [116] PC-2 Section 7.5). However, it was noticed [5, 33, 35] that Allen’s algorithm does not give path-consistency and instead provides a weaker property called algebraic closure [14]. Path consistency requires that, for all triples of variables \(x, y, z\), any consistent instantiation of \(x, z\) can be extended to some consistent instantiation of \(y\) [120]; algebraic closure only requires that for all \(x, y, z\) \(R(x, y) \land S(y, z) \rightarrow T_1(x, z) \lor \ldots \lor T_n(x, z)\). In particular, as previously discussed in Section 2.4, for algebraic closure the consequent is a necessary but not a sufficient condition whereas path-consistency also requires that \(T_1(x, z) \lor \ldots \lor T_n(x, z) \rightarrow \exists y \cdot R(x, y) \land S(y, z)\) ([120] Section 4, Equation 4.2, and [116] Section 7.5). In set theoretic notation, strong composition is \(R \circ S = T_1 \cup \ldots \cup T_n\), whereas weak composition (\(\diamond\)) is the strongest union of relations \(T_1, \ldots, T_n\) such that \(R \circ S \subseteq T_1 \cup \ldots \cup T_n = R \diamond S\) [14] (Section 2.5). The term extensional has also been used in the QSTR literature [33, 107] to refer to QSTR calculi that satisfy strong composition. The significance of the distinction between algebraic closure and path-consistency is that algebraic closure is a weaker condition, and thus there may be inconsistent scenarios that path-consistency can detect which are determined to be consistent according to algebraic closure.

Extensionality is clearly an important computational property of any QSTR calculus, as it determines whether algebraic closure in fact gives path-consistency. For many QSTR calculi extensionality depends strongly on the domain of interpretation. For example, as presented in Section 2.4, Maddux and Ladkin [105] formally showed that if Allen’s interval algebra is interpreted in a dense, unbounded domain such as the set of all rational numbers then strong composition is satisfied. Ligozat generalised these results to non-convex interval algebras [113] and then subsequently to all QSTR calculi based on Cartesian products of linear orderings such as the compass algebra and the rectangle algebra [5] (Theorem 1 and specifically Lemma 6).

In the case of RCC, it was noted by Cui et al. [48] (Section 3.1) that RCC is not extensional if regions are interpreted as closed disks of the same size; a maximum of six such disks can be externally connected to a seventh central disk without any disks overlapping, although algebraic closure will not detect an inconsistency if a scenario declares seven non-overlapping disks externally connected to an eighth disk. More critically, however, researchers noticed that compositional reasoning in RCC does not always identify path inconsistencies in less restrictive interpretations ([33, 35]; [34] Section 9.1.2, 9.1.3 and 9.2.5). While Duntsch et al. [57] showed that if regions are interpreted as closed disks in the Euclidean plane then RCC is extensional, the nature of RCC extensionality was still not completely understood. Bennett proposed that
RCC might become extensional if the universal region (a region that is connected to every region) is removed, however this was disproved by Li and Ying (Example 4.1). Indeed, Li and Ying completely answered the question of RCC extensionality by proving that if the domain of interpretation allows disconnected regions and holes then RCC is not extensional. Moreover, they presented an exhaustive enumeration of compositional triads that are extensional only if the domain of interpretation satisfies particular conditions such as density and connectedness.

### 2.6.2 Tractability

As already discussed in Section 2.6, determining consistency in Allen’s calculus is an NP-hard problem [158, 159]. In response to this, researchers have focused on improving the efficiency of the algorithms for reasoning (e.g., Freksa [66, 67]) and finding tractable subsets of the consistency problem [10, 125, 159]. In Chapter 6 research results on tractable subclasses are used to identify classes of scenarios in which an application that utilises a QSTR calculus will produce erroneous output. Thus, tractable subclasses are not only relevant with respect to computational complexity, but are used to facilitate application validation.

Full expressiveness of Allen’s calculus implies that any combination of qualitative relations can be assigned to a pair of objects, where any relation in the given combination is a possibility. In the restricted reasoning problem ([125] and [126] Section 3.2.5), pairs of objects can only be assigned particular combinations of possible relations with the intention that reasoning in the prescribed subclass is tractable.

The continuous endpoint subclass, investigated in [30, 31, 159], is both intuitively appealing [126] (Section 3.2.5) and tractable. Informally, a set of relations are in the continuous endpoint subclass if it is possible to transform between all relations in the set by continuously deforming an interval endpoint, without passing through any other basic relations. Path-consistency solves the minimal labelling problem (ISI) for scenarios that only contain continuous endpoint relations. Unfortunately only \( \sim 1\% \) of the possible IA relation combinations are in this subclass.

Ladkin and Maddux [104] and van Beek [30] identified and analysed the pointisable subclass of Allen’s interval algebra which consists of \( \sim 2\% \) of the possible relation combinations. Any scenario of intervals in the pointisable subclass can be converted into a tractable single point algebra scenario. Thus, in general path-consistency can determine satisfiability of pointisable scenarios (ISAT), however it cannot find the minimally labelled network (ISI) [29, 104, 158].

In a landmark paper, Nebel and Burckert [125] identified the maximal tractable subclass of Allen’s calculus, referred to as the ORD-Horn subclass, for which algebraic closure is sufficient for determining consistency (ISAT). Moreover, they proved that the ORD-Horn subclass is the unique maximal tractable subclass (that contains all basic primitive relations). The ORD-Horn
subclass is much larger than the continuous endpoint and pointisable subclasses, consisting of \( \sim 10\% \) of the relation combinations.

Identifying tractable subclasses of prominent calculi has now become a central research topic in QSTR. For example, in [123] Nebel investigates the complexity properties of RCC and shows that algebraic closure is sufficient for determining whether an unambiguous scenario is consistent. In [138, 139] Renz and Nebel build on these results by identifying a maximal tractable subset of RCC for which algebraic closure is sufficient for determining consistency. In [13] Renz identifies all three maximal tractable subsets of RCC and develops theoretical tools for identifying maximal tractable subsets in arbitrary qualitative calculi.

In [110] Ligozat proved that tractability is a property of a concept called *pre-convexity* [112, 113] rather than ORD-Horn, although in Allen’s calculus the subclasses defined by these properties coincide. Ligozat used pre-convexity to identify tractable subclasses of all qualitative calculi based on cartesian products of linear orderings (including calculi represented by points on an \( n \)-dimensional plane such as the *compass algebra*) [109].

### 2.7 Conceptual Neighbourhoods

Continuity plays an important role in both qualitative reasoning about physical systems [167] and QSTR [49, 58, 74–76]. In QR, continuity refers to the value of a variable as a function of time. In QSTR, continuity refers to the relation between two objects in the domain of interpretation as a function of time [69, 85]. A relation is continuous if arbitrarily small changes in the relation can be observed with arbitrarily small changes in time. Intuitively, a relation must change in a smooth manner, as opposed to abruptly jumping between discrete states [167].

The relevance to qualitative reasoning and QSTR is that if a function \( f \) is continuous on some closed interval between \( a \) and \( c \), then for any value \( i \) between two values \( f(a) \) and \( f(c) \), there must be some value \( b \) such that \( f(b) = i \). That is, if \( f(a), f(b), f(c) \) are three qualitatively distinct states, then the scenario property represented by \( f \) must necessarily transition through \( f(b) \) in order to transition from \( f(a) \) to \( f(c) \).

In [67] Freksa employed continuity for concisely representing coarse qualitative temporal information. Freksa argues that an important type of coarse knowledge is characterised by omitting fine distinctions between similar relations. Freksa introduced the notions of conceptual neighbours and conceptual neighbourhoods; two relations are conceptual neighbours if they can directly transition between each other, without passing through any other relation [67] (Section 2.2). Transitions between relations with a change in time occur through object translations and deformations.\(^3\) A conceptual neighbourhood is a connected graph where vertices are qualitative relations and edges between vertices correspond to relations being conceptual neighbours; Figure 2.2 illustrates the conceptual neighbourhood of Allen’s interval relations. Thus, coarse

\(^3\)These may only be apparent changes with respect to an observer, refer to [64] Section 4, Point 8 (page 8).
knowledge is not simply a disjunction of arbitrary possible qualitative relations, but a disjunction of neighbouring relations. Freksa used this approach to efficiently reason about coarse information ( [67] Sections 4 and 5; also Section 5 in the revised version [66]; refer to [65] Section 4 for a brief overview).

Neighbourhood graphs are used in a number of QSTR tasks such as envisioning and diagnosis [36, 49]. Envisioning is the process of simulating a sequence (or all possible sequences) of qualitative states as objects move (relative to other objects or relative to an observer) and deform [98]. Given two objects \( x, y \) with the relation \( R(x, y) \) in a scenario state \( s_1 \), in the succeeding state \( s_2 \) the relation between \( x, y \) either has not changed or the relation has transitioned into relation \( R'(x, y) \) such that \( R \) and \( R' \) are conceptual neighbours. Due to continuity it is not possible for the two objects to 'jump' into some other relation \( R'' \) which is not a conceptual neighbour of \( R \). Conversely, diagnosis is the process of simulating states that proceed \( s_1 \). Thus the neighbourhood graph reduces the number of ways that two objects can be related in scenario \( s_i \) based on information about the proceeding scenario \( s_{i-1} \) or information about the succeeding scenario \( s_{i+1} \).
2.8 Case Studies of QSTR Applications

In this section five case studies on QSTR applications are presented. The case studies highlight common properties of QSTR applications and thus will be referred to throughout the thesis to motivate and illustrate specific application development methodologies. The case studies have been selected to represent a range of application domains and task requirements.

2.8.1 QtvLight: Qualitative Decision Support for Architectural Lighting

QtvLight is a prototype design tool for architectural lighting [147, 148]. The central aim of this case study was to investigate whether qualitative logic taken from scientific literature can be formalised and accurately automated in a software system that employs QSTR calculi.

This case study focused on the work of two prominent researchers in architectural lighting, namely Flynn and Cuttle, who have studied the subjective responses elicited from various lighting configurations. Flynn has performed seminal studies on the relationships between luminances, luminous patterns and subjective response [61, 62]. Flynn aimed to establish basic guidelines on how to influence a range of non-visual effects with a lighting scheme and identified five key impressions: visual clarity, spaciousness, pleasantness, relaxation, and intimacy. It was found that the subjective responses are elicited by a number of intermediate qualitative lighting conditions. For example, to create a sense of relaxation the lighting designer could selectively place indirect luminaires around the periphery (e.g. wall sconces or accent lighting on wall art decorations) complemented with direct low intensity incandescent lamps placed over the area of occupancy [153]. Cuttle [51] proposes a set of central factors, such as ambient illuminance and the flow of light, that influence a person’s subjective impression of a lighting environment with the aim of supporting architectural design objectives in the creation of a lighting scheme. In this case study, the research by Flynn and Cuttle was formalised in order to interpret and reason about qualitative lighting information such as “bright uniform light across centrally located work surfaces, with some perimeter emphasis”.

The prototype software tool QtvLight was developed to assist an architect during the early stages of a project by analysing an electrical lighting installation and providing fast qualitative feedback on the subjective impressions that will be evoked. The tool accepts qualitative input describing building components and the lighting configuration, but can also accept numerical input (e.g. dimensions from a preliminary CAD design). The QSTR rectangle algebra [2, 78] was employed as it was found that the relations defined by projections of orthogonal axes were ideal for specifying the qualitative orientation of lighting features, walls, and furniture.

Given that the aim of the case study was to determine whether the research by Flynn and Cuttle could be accurately automated using QSTR calculi, the results of (human-based) experiments conducted by Flynn [61, 62] were used to validate the reasoning engine’s applicability to
2.8 Case Studies of QSTR Applications

2.8.1 Lighting Design

It was found that the reasoning engine correctly determines subjective impressions when strong responses are reported. When the response is only moderate the engine still determines the correct qualitative value, for example, if generally hazy is the correct inference then QtvLight simply infers hazy. When the response is neutral (i.e. lying on the boundary between two distinct qualitative values) QtvLight does not respond in a consistent manner between different impressions.

2.8.2 TreeSap: Geographic Information System

The software application TreeSap is a geographic information system (GIS) that facilitates qualitative querying [16, 17, 149]. The main objective of developing TreeSap was to investigate the application of QSTR to provide intuitive querying interfaces in order to address the problem that GIS querying support is often either very limited or requires a user to have knowledge in specialised areas such as database languages or set theory.

TreeSap employs an adaptation of the fuzzy proximity formalism presented in [77]. The possible distances between two objects are partitioned into seven qualitative regions: touching, very near, near, moderately near, moderately far, far, very far. Distance is measured as either straightline distance or the distance through a network such as roadways and ferry routes [16]. In order to accommodate the inherent vagueness in the regions of qualitative proximity, querying is made to be more flexible by ranking query results using the neighbourhood graph as presented in [75, 76]. Similarity \( \alpha_i \) between two relations is defined as the length of the shortest path between the relations in the neighbourhood graph. For example, given the query “Find all cafes near Symonds St”, cafes that are near Symonds St are assigned the similarity \( \alpha_0 \), cafes that are either very near or moderately near are assigned the similarity \( \alpha_1 \), and so on.

Figure 2.3 presents a screenshot of the query interface. The user builds their query as a hierarchical tree structure of conditions where nodes of the tree are search criteria. These criteria are described in terms of a subject, and its qualitative spatial relationship with another object. A query is built up in stages, and at each stage the user is presented with the results of the partial query, visualised on a map.

TreeSap conveys query results using two visualisation techniques. The first approach uses transparency to represent how well a feature fulfills the query criteria, as demonstrated in Figure 4.

---

4It must be noted that the case study was not addressing the question of whether the logic actually captures the concepts of subjective impressions, as this can only be addressed by field psychologists, domain experts, researchers and so on.

5Additional usability features are that (a) the interface is mouse-driven and thus does not require a user to learn or remember commands or keywords, (b) a query can never be malformed due to the nature in which it is built, (c) there is no possibility for erroneous or invalid input, such as incorrect data types being entered into a field, (d) the user is given immediate feedback through the mouse-driven interface, i.e. when the mouse moves over an interactive component, such as a button, the component lights up, and (e) all user communication and interaction (such as component highlighting, tool-tips, popup selection boxes, and messaging) occur near the mouse pointer, hence avoiding large eye and mouse movements between targets.
2.4. Features that fulfill the query criteria to a high degree are displayed completely opaquely, while features that are less relevant to the query are displayed transparently.\(^6\) A limitation of the transparency approach is that, while it can provide an instant overview of a query result, it does not effectively convey subtle trends and details. For example, the exact location with the highest solution quality is not always obvious, as subtle differences in transparency can be difficult to recognise. To accommodate for these limitations a second visualisation approach employs a user-controlled threshold to restrict the amount of information presented at a given time. All features that have a solution quality above the threshold are displayed opaquely and any features with a solution quality that do not meet this threshold are not displayed at all. A scenario is illustrated in Figure 2.5, where more roads are displayed as the threshold is lowered, thus dynamically revealing underlying patterns and trends in the query result.

2.8.3 SPBD: Qualitative Interface for Robotic Programming by Demonstration

Simple Programming by Demonstration (SPDB) [161] is a cognitively motivated framework for programming a robot to perform tasks by showing or demonstrating how the task can be accomplished rather than writing software instructions. The first objective of this case study was to investigate the applicability of QSTR calculi in facilitating robot-human interaction by representing and reasoning about information in an intuitive, qualitative manner as opposed to exposing the human demonstrator to raw numerical sensor data. The second objective was to investigate whether QSTR calculi can directly support the development of robot software by promoting a more modular software architecture; QSTR calculi were used to separate the

\(^6\)Transparency offers an intuitive and visually efficient technique for conveying qualitative information. Opaque features represent the solution to the query and are therefore the most important pieces of information being displayed; these features appropriately attract a user’s attention by being displayed more distinctly than non-solution features. By displaying neighbouring, non-solution features very faintly the user is implicitly given some spatial context to assist in the interpretation of the solution.
Figure 2.4: Screenshot of the transparency method used in TreeSap GIS to visualise results of the query “Find all Roads near a Specific Building (black circle)”.

Figure 2.5: Screenshot of the display threshold method used in TreeSap GIS to visualise results of the query “Find all Roads near a Specific Building (black circle)” for threshold values 100%, 75%, and 50%.
inherent logic of the tasks being demonstrated from the numerical parameters that are unique to a given robot such as the robot shape or movement speed.

The case study was conducted using a Pioneer 3DX robot equipped with a sonar sensor ring consisting of sixteen sonar devices, as illustrated in Figure 2.6. Distance measurements were mapped into one of five qualitative values: adjacent (0m to 0.3m), very near (0.3m to 1.0m), near (1.0m to 2.0m), moderately near (2.0m to 3.0m), and far (3.0m to 4.0m). Qualitative orientations of objects relative to the robot were inferred by identifying the sensors from which a given distance measurement was taken: left (sonar 0 and sonar 15), front left (sonar 1 and sonar 2), front (sonar 3 and sonar 4), front right (sonar 5 and sonar 6), right (sonar 7 and sonar 8), back right (sonar 9 and sonar 10), back (sonar 11 and sonar 12), and back left (sonar 13 and sonar 14). Qualitative relative turn values were used to track objects as the robot turns, for example, to interpret the instruction: “approach the wall and turn until the wall is parallel on the leftside”. The four relative turn qualitative values are turn parallel on left, turn to face, turn parallel on right, turn away. Artificial hysteresis was employed to prevent erratic behaviour at the boundary between qualitative values [161].

In the SPBD framework, the demonstrator initially performs a task by controlling the robot. During the execution of the task SPBD records sequences of qualitative information. Once the task example is complete, the human demonstrator refines the task description through a graphical interface by reviewing and editing the sequences of qualitative information. By repeating this process the human demonstrator builds up a library of subtasks. A future aim was that during subsequent demonstrations the robot could refer to the library to identify patterns in sensory traces and automatically replace matching low-level qualitative sequences with higher-level subtasks ([161] Section 8.2). The subtasks used to evaluate the framework were to loop in a rectangular path, approach and stop in front of an object, turn parallel to an object, and avoid an object that the robot is approaching ([161] Section 6.5). It was found that mapping sensory traces into human-readable qualitative values enables the human demonstrator to easily interpret and refine the description of a task, for example, by eliminating sensor noise, demonstration mistakes, and sensor data that is correct but irrelevant for the given task ([161] Section
Moreover, the modularity achieved by abstracting away from the numerical details makes the task description significantly more concise and flexible ([161] Section 5.1).

### 2.8.4 Ul-Qayyum’s Qualitative Image Retrieval System

In [134] Ul-Qayyum and Cohn from the Leeds Qualitative Spatial Reasoning Research group [176] present a method for image retrieval using qualitative spatial reasoning. Each image is divided into a 10 × 10 matrix of cells which are annotated with a semantic texture such as sky or grass based on research by Vogel and Schiele [160]. Qualitative relationships between all pairs of semantic concepts are calculated for an image. For example, a possible image property is \(sky > grass\) meaning that more cells are annotated with sky compared to grass. Nine semantic concepts are defined: sky, water, grass, trunks, foliage, field, rocks, flowers, and sand [134, 160]. Images are compared by considering the differences between the relations of each pair of semantic concepts. For example, two images \(x, y\) are considered to be qualitatively equivalent if, for all \(i, j \in \{\text{sky, . . . , sand}\}\), \(R_{xij} = R_{yij}\). This approach thus facilitates image retrieval by either providing a target image for which qualitatively equivalent images should be retrieved, or by expressing a qualitative description of an image, that is, a qualitative query.

Ul-Qayyum and Cohn investigate four different QSTR calculi. Relative size consists of three relations, \(<, =, >\), for comparing pairs of semantic concepts by the qualitative proportion of annotated cells. The touching approach consists of a single relation that compares pairs of semantic concepts by specifying whether any region of cells annotated with one texture type is adjacent to (or in contact with) a region of cells annotated with the other texture type. The Allen approach projects the maximum and minimum vertical extent of a texture type onto the vertical axis to define intervals; Allen’s interval relations are then used to specify the relation between two texture types. Finally, the chord approach classifies image rows by identifying the combination of texture types that appear at least once in a given row; a particular combination of texture types is referred to as a tone e.g. \{sky\} or \{sand, grass\}.

Higher-level image categories are also defined such as coastal image, waterscape or forest image. Vogel and Schiele have produced a manually annotated repository of images [160] with which Ul-Qayyum and Cohn validate their qualitative application. Each image in the repository is used as a target query image; the efficacy of the application retrieval is measured by the category of the returned images. For example, if a forest image is provided as the target, then the application is expected to primarily return other forest images.

### 2.8.5 SailAway: Maritime Qualitative Navigation

In [20, 170] Wolter et al. from the Spatial Cognition research group [177] present a QSTR application called SailAway for reasoning about the navigation paths of maritime vessels. SailAway complements the previous case studies by explicitly reasoning about agents and motion.
Chapter 2. Qualitative Spatial and Temporal Reasoning

SailAway scenarios consist of autonomously controlled sea vessels, referred to as agents, that follow real maritime navigation rules that were established to prevent vessel collisions. The specification of the regulations is qualitative in nature as the rules were developed to be comprehended and followed by humans. Thus, as part of the SailAway project, the right-of-way rules were formalised using a qualitative reasoning calculus called the Oriented Point Relation Algebra (OPRA) [8]. Vessel motion is modelled by transitions between OPRA relations with respect to the OPRA neighbourhood graph. Thus, SailAway simulates the behaviour of agents equipped with OPRA and the formalised right-of-way rules as they navigate to avoid collisions. One objective of the SailAway project was to determine whether a collision between the autonomous vessels is possible under the formal maritime navigation rules. It was found that in certain cases collisions that involve three vessels can occur.

2.9 Summary

In this chapter the primary concepts and theoretical advances in QSTR have been reviewed. Allen’s Interval Algebra has motivated a wide variety of qualitative approaches for reasoning about spatial and temporal relations between objects. In particular, Allen’s primary contribution was the formulation of the task of reasoning about incomplete temporal information as a constraint satisfaction problem (CSP). An unambiguous scenario, where exactly one qualitative relation is specified for each pair of objects, corresponds to an atomic CSP network. An ambiguous scenario, where multiple qualitative relations are specified for at least one pair of objects, corresponds to a non-atomic CSP network. The primary reasoning mechanism in QSTR calculi is relation composition; transitive constraints are encoded in a composition table, thus greatly simplifying compositional reasoning with an efficient look-up reference procedure.

QSTR calculi are combined and generalised by modelling more complex objects or by integrating different aspects of space and time. In general, QSTR calculi can be combined using either a loose or a tight integration. In a loose integration of calculi, calculus each constituent calculus is reasoned about independently, and then the inferred information is propagated between the calculi to eliminate further inconsistencies. In a tight integration a new set of basic relations that explicitly expresses all semantic interdependencies are defined. This is particularly relevant to this thesis as QSTR applications typically combine heterogeneous QSTR calculi.

Maddux and Ladkin’s algebraic formulation of Allen’s Interval Algebra was employed by the QSTR research community to unify a large number of disparate QSTR calculi. QSTR calculi are defined on domains of interpretation. The choice of domain has a large impact on the reasoning properties of a QSTR calculus.

Two main reasoning tasks of QSTR calculi are determining whether a given scenario is consistent and eliminating any inconsistent relations that were previously specified as a possibility. A scenario described using a QSTR calculus is consistent if it has an instantiation of the vari-
ables in the domain of interpretation that satisfies all of the qualitative constraints. A crucial property of a given QSTR calculus is whether Allen’s algorithm can determine the consistency of non-atomic networks and atomic networks. A major area of research in QSTR is determining the largest combinations of relations for which Allen’s algorithm can determine the consistency of non-atomic networks, known as determining the maximal tractable subsets of a calculus.

Conceptual neighbours reflect the transition sequences between relations in a QSTR calculus due to the property of continuity. Conceptual neighbourhoods are graphs where vertices are qualitative relations and edges represent conceptual neighbours. Conceptual neighbourhoods are used for qualitative tasks such as envisioning potential future qualitative scenarios and diagnosing possible prior qualitative scenarios.

Five case studies were presented that cover a range of application domains, namely architectural lighting, geographic information systems, robotics, image retrieval, and maritime navigation. Importantly, the case studies highlight common properties of QSTR applications and thus will be referred to throughout the thesis to motivate and illustrate specific application development methodologies.
Chapter 3

Theoretical Foundations of QSTR Applications

3.1 Introduction

The previous chapter introduced the field of QSTR and presented the state of the art in facilitating the application of QSTR calculi. As presented in Sections 2.4 and 2.6, while researchers have established an effective unifying definition of QSTR calculi, relatively few attempts have been made at defining the systems that apply QSTR calculi. The central problem is that a QSTR calculus is developed in order to formalise a very specific aspect of the relative relationship between spatial or temporal entities, thus addressing a range of tasks which involve particular modelling assumptions about space and time. In contrast to this, a QSTR application is developed in order to satisfy very focused and unique requirements, which involve specific user profiles, domain terminology, and tasks that the application must perform. Hence, QSTR applications are very different types of logical systems compared to QSTR calculi, and the motivation for their development differs significantly to the objectives of developing QSTR calculi. This highlights the need for a comprehensive definition of QSTR applications that explicates the role of QSTR calculi while being consistent with the results from the case studies presented in Section 2.8.

In order to formalise QSTR calculi and QSTR applications, the following definitions for sentences in a first-order theory are required.

**Definition 3.1.1.** Let $L$ be a first-order language that consists of a collection of relation symbols such as before (i.e. the signature) along with the standard first-order logical constants such as $\lor, \land, \to$, and so on. A theory $\Theta^L$ is a set of sentences $t_1^L, \ldots, t_n^L$ in the language $L$. A sentence $t^L$ is an expression $e^L$. An expression is either atomic, composed, a negation, or it contains a connective. An atomic expression is a relation in the language $L$, e.g. $x < y$. A composed expression is a constraint between variables that consists of multiple atomic expressions [120].
e.g. if \( x < y \) and \( y < z \) then \( x < z \) is the composed expression. A negation expression is \( \neg e^L \). A connective expression is either \( e^L_1 \land e^L_2 \) or \( e^L_1 \lor e^L_2 \).

**Definition 3.1.2.** Formally, a QSTR calculus \( A \) is a first order theory \( \Theta^A \) consisting of axioms specified in the first order language \( L \) derived from the signature of the relation symbols. An \( n \)-ary relation \( R^A \) (typically binary or ternary) in \( A \) can hold between \( n \) objects, \( R^A(x_1, \ldots, x_n) \). The axioms of a calculus are the sentences in \( L \) that define conversion and (weak) composition.

A qualitative scenario is a first-order expression that describes objects and relations between objects using the language of the calculus. This can be thought of in set theoretic terms, where objects are atoms in the universe \( U \) and relations \( R \) with arity \( a_R \) are sets that consist of tuples of atoms, \( R \subseteq U^a_R \) [136]. For example, a simple scenario that describes the relationship between two buildings in Auckland, the Ferry Building and Britomart, is the expression

\[
\text{near} = \{(\text{Ferry Building}, \text{Britomart})\}.
\]

Most QSTR calculi come with (sometimes implicit) domains of interpretation [114], for example, a scenario of Allen’s calculus is interpreted as a linear ordering \( W \) together with a subset \( U \) of the intervals \((w^-, w^+)\), \( w^- < w^+ \) on \( W \). The set of scenarios that are consistent with the axioms of a qualitative calculus are denoted as \( \text{Mod}(\Theta^A) \) (i.e. the models of the \( A \) theory). The set of scenarios that use the language of the qualitative calculus, but are not necessarily consistent, are denoted as \( \text{Int}(\Theta^A) \).

This chapter establishes a formal theoretical foundation of QSTR applications. The following section presents a formal definition of QSTR applications. In Section 3.3 this theoretical foundation is used to formally define the roles of QSTR application users and developers. In Section 3.4 the expression of scenario ambiguity with respect to QSTR applications is formalised. Section 3.5 derives three fundamental operations that can be performed on QSTR application scenarios. Finally, in Section 3.6 the formal definitions are used, in conjunction with an analysis of the case studies, to identify four central characteristics of QSTR applications.

### 3.2 Formal Definition of QSTR Applications

QSTR applications employ QSTR calculi to represent and reason about spatial configurations of objects and temporal configurations of events. When presented with a qualitative description of a scenario the application checks for particular patterns in the scenario and then produces output.

A standard paradigm in computer science is that a software program is equivalent to a mathematical function which accepts input symbols and responds by producing output symbols.

---

1 In model theory these are called interpretations, not to be confused with the definition of a domain of interpretation.
Adopting this paradigm, let \( \Theta \) be a QSTR application’s theory for describing scenarios. Thus the input to a QSTR application is a scenario \( s \in \text{Int}(\Theta) \), that is, an expression in the language of \( \Theta \). Let \( O = \{a, b, c, \ldots\} \) be a set of symbols representing possible QSTR application outputs. QSTR applications can be developed to process individual scenarios or collections of scenarios.

**Definition 3.2.1.** A QSTR application that processes individual scenarios is a function \( \theta : \text{Int}(\Theta) \rightarrow 2^O \) that accepts a scenario described in \( \Theta \) and generates a set of outputs. A QSTR application that processes an ordered sequence of scenarios is a function \( \theta : \text{Int}(\Theta)^n \rightarrow 2^O \) that accepts a sequence of length \( n \) of scenarios described in \( \Theta \) and generates a set of outputs. A QSTR application that processes an unordered collection of scenarios is a function \( \theta : 2^{\text{Int}(\Theta)} \rightarrow 2^O \) that accepts a set of scenarios described in \( \Theta \) and generates a set of outputs. If output symbols are mutually exclusive then \( \theta(\cdot) \in O \).

QSTR applications can incorporate a number of different QSTR calculi. If \( A_1, \ldots, A_n \) are QSTR calculi that are employed by a QSTR application then the theory \( \Theta \) that the application uses to describe scenarios contains each constituent calculus theory, \( \Theta \supseteq \Theta_{A_1} \cup \ldots \cup \Theta_{A_n} \). When the application is given a scenario (or collection of scenarios) to process, the reasoning services of a constituent QSTR calculus \( A \) can be employed by isolating the portion of the scenario that matches the language of \( A \) and executing the path-consistency algorithm (or any other relevant QSTR reasoning service as discussed in 2.6; refer also to [162]), that is, calculus \( A \) is used to process the induced sub-expression \( e^A \) from the full scenario expression \( e \). Thus, applications employ a loose integration of calculi [19] as presented in Section 2.3. It must be noted that a calculus \( A \) can be a tight integration of other calculi [19], although in general this must be provided by the QSTR community [169]. The customised, application-specific relations and rules \( \Theta \setminus (\Theta_{A_1} \cup \ldots \cup \Theta_{A_n}) \) will typically operate on an input scenario \( s \) after the reasoning services of the constituent QSTR calculi have been applied.

Given application \( \theta \) that accepts scenarios from \( \Theta \), observe that each output \( i \in O \) is a theory, denoted \( \Theta_i \), that uses the same language as \( \Theta \). Models of the theory \( \Theta_i \) correspond to scenarios where the application is required to produce output \( i \): for all scenarios \( s \in \text{Int}(\Theta) \), \( i \in \theta(s) \iff s \in \text{Mod}(\Theta_i) \). This leads to the following general, homogeneous definition of QSTR applications that uniformly describes constituent QSTR calculi and the custom application-specific relations and rules.

**Definition 3.2.2.** A QSTR application \( \mathcal{A} = (R, \Theta) \) is a finite set of relation symbols \( R \) with arities \( a_R \) for each \( R \in R \) and constraints between the relation symbols in the form of a set of sentences \( \Theta \) that express the models of the application.

All QSTR applications share one important constraint by virtue of the crisp boolean nature of qualitative concepts. By definition, the effect of introducing a qualitative concept is to distinguish between object tuples (the definition of a “quality” is a “distinguishing characteristic” [174]). The most elementary distinction possible is a bipartition of the tuples. Thus, for
each relation type \( R \in \mathcal{R} \) with arity \( a_R \), and for each tuple of arity \( a_R \), the relation either holds or does not hold. Furthermore, qualitative concepts do not necessarily apply to all tuples. For example, a piece of furniture can occlude light from a directed source shining onto a particular surface. However, if the directed source was pointing away from the surface, then occlusion is no longer meaningful. Therefore, a third relation state, not applicable, must also be included. Thus, for each relation symbol \( R \in \mathcal{R} \) in the signature, a QSTR application model \( s \) requires three relations, \( R^+ \) (holds), \( R^- \) (does not hold), and \( R^- \) (not applicable), that satisfy the following axiom, where \( \triangle \) is the symmetric difference operator (the set theoretic equivalent of mutual exclusion).

**Axiom 3.2.3.** For each relation \( R \in \mathcal{R} \) in a QSTR application’s signature, a model \( s \) contains three relations, \( R^+ \), \( R^- \), and \( R^- \) such that \( R^+ \triangle R^- \triangle R^- = U^{a_R} \). The allowable states of relation \( R \) are denoted \( A_R \subseteq \{+,-,\sim\} \).

**Example 3.2.4.** The QSTR application QtvLight presented in Section 2.8 is defined as follows. Let \( BA \) be the QSTR calculus of Guesgen’s block algebra as presented in Section 2.2. Let \( \Theta^X \) be a theory of simple custom qualitative relations such as directed at and occluded [148]. Let \( \Theta^{IFC} \) be an ontology that formally describes concepts and rules from the application domain of construction and architecture.\(^2\) Let \( \Theta^Y \) be a formalisation of the qualitative rules of Flynn [61] on the subjective responses of lighting configurations. The theory of QtvLight is \( \Theta = \Theta^BA \cup \Theta^X \cup \Theta^{IFC} \cup \Theta^Y \). Functionally, QtvLight accepts a scenario that describes a room with the qualitative orientation relations of furniture, walls, doors, light fixtures and so on, and then produces the subjective responses \( O = \{\text{clear, hazy, spacious,} \ldots\} \) elicited by the lighting configuration, \( \theta^{QtvLight} : \text{Int}(\Theta) \rightarrow 2^O \).

**Example 3.2.5.** The QSTR application SailAway presented in Section 2.8.5 is defined as follows. Let \( OPRA \) be a calculus of oriented points [8]. It must be noted that the theory \( \Theta^{OPRA} \) also specifies the conceptual neighbours of each OPRA relation. Let \( \Theta^X \) be a formalisation of a selection of maritime right-of-way rules. Let \( \Theta^Y \) be a theory of simple custom qualitative relations such as collision at rear, and vessel types such as motorVessel. The theory of SailAway is \( \Theta = \Theta^{OPRA} \cup \Theta^X \cup \Theta^{IFC} \cup \Theta^Y \). Functionally, SailAway accepts a scenario describing the initial qualitative orientation of vessels \( s_1 \in \text{Int}(\Theta) \), simulates (envisions) subsequent future states in which all vessels obey maritime rules \( s_1, \ldots, s_n \in \text{Mod}(\Theta)^n \) (each transition is annotated with a vessel action such as turn starboard), eliminates all states that involve collisions, and then reports the final collision free sequence of scenarios \( O = \text{Mod}(\Theta)^n \), thus \( \theta^{SailAway} : \text{Int}(\Theta) \rightarrow O \).

\(^2\)The Industry Foundation Classes (IFC) [175] is an open ontology for the building industry.
Example 3.2.6. Nokel [126] (page 46, Figure 23) presents this example to illustrate the logical limitations of using binary relations. This example shows that, although the qualitative calculus is not capable of ruling out abnormal scenarios based on the binary CSP encoding, applications developed using the calculus can discriminate between relevant scenarios. The scenario consists of three valves attached to a tank such that exactly one valve must be open at any time. Let $\Theta_{IA}$ be a theory of Allen’s Interval Algebra, let output symbol $\gamma$ indicate abnormal valve operation, and let $\Theta_{IA}^\gamma$ consist of two sentences where $x, y, z$ are the intervals when valves are open:

- $\text{overlaps} (x, y) \lor \text{starts} (x, y) \lor \text{during} (x, y) \lor \text{finishes} (x, y)$ (intervals when valves are open intersect)
- $\text{before} (x, z) \land \neg \exists y \cdot \text{meets} (y, z)$ (all valves are closed after $x$ and before $z$).

The qualitative application will detect abnormal behaviour in any temporally consistent schedule of open valves.

3.3 Actors and Roles

In this section the role of the application developer is formally defined in contrast to the role of the application user. Given formal software requirements, a QSTR application developer is responsible for establishing the relevant qualitative concepts and the class of valid scenarios. This involves two key aspects: selecting suitable QSTR calculi, selecting a set of custom relation symbols and encoding a suitable set of constraints; suitability means satisfying specific test criteria and conditions on metrics that imply that the software requirements have been met. From a functional perspective, for each output in $O$ specified in the formal requirements, the developer must specify the class of scenarios, as expressions in the language used in $\Theta$, for which the application must produce the output; in general, the developer must specify sentences $\Theta_R$ that determine the properties of each custom relation $R$.

A QSTR application user constructs scenarios in a QSTR application by specifying a scenario $s \in \text{Int} (\Theta)$ and employing reasoning to accomplish tasks such as determining scenario consistency $s \in \text{Mod} (\Theta)$, envisioning potential future scenarios $s_1, \ldots, s_n \in \text{Mod} (\Theta)^n$, and so on. A complete set of basic QSTR application task types with respect to this formal definition of QSTR applications is presented in Chapter 4. Table 3.1 summarises the relationship between model theory, QSTR applications, and actor roles.
### Chapter 3. Theoretical Foundations of QSTR Applications

<table>
<thead>
<tr>
<th>Model Theory</th>
<th>QSTR Application Domain</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language $L$</td>
<td>Specification of useful qualitative relations.</td>
<td>QSTR application designer</td>
</tr>
<tr>
<td>Theory $\Theta^L$ based on $L$</td>
<td>Constraints that determine the interaction between the relations.</td>
<td></td>
</tr>
<tr>
<td>Scenario $s \in \text{Int}(\Theta^L)$ based on $L$</td>
<td>Using a QSTR application to represent and reason about objects.</td>
<td>QSTR application user</td>
</tr>
</tbody>
</table>

Table 3.1: Comparing the domains of model theory, QSTR applications, and the roles of QSTR application designers and users.

### 3.4 Ambiguity

Often parts of the user’s scenario are indefinite or unknown, and reasoning with the application constraints is used to help resolve this ambiguity. For each relation $R \in \mathbf{R}$, the user can place $a_R$-tuples of objects from $U$ in a fourth indefinite set, $R^2$ which is mutually exclusive with the three corresponding definite sets. This partial scenario is a shorthand for specifying a set of scenarios $s_1, \ldots, s_n$ each representing a possible complete scenario, $s \leftrightarrow s_1 \lor \ldots \lor s_n$.

From an application-level perspective, complete scenarios are unambiguous and fully observable up to the precision admitted by the relations of a given calculus;\(^3\) partial scenarios are ambiguous and only partially observable. With respect to a constraint satisfaction problem of scenarios (refer to Section 2.2.1), complete scenarios correspond to atomic networks, whereas partial scenarios correspond to non-atomic networks.

### 3.5 Fundamental Operations on Scenarios

In this section a complete set of fundamental scenario operations that can be performed using a QSTR application is derived. In Chapter 4 these operations are combined to enumerate a set of basic purely qualitative tasks.

The set of possible operations that can performed on a partial scenario will now be determined by identifying QSTR application components that are variable during application runtime. Based on Definition 3.2.2, QSTR application components are symbols in $L$, models of the language $\Theta^L$, the partial scenario universe $U$, and the collection of sets that interpret the relation symbols $R^+, R^-, R^\sim, R^?$.

The relation symbols in $L$ and the theory $\Theta^L$ are determined at QSTR application design time and are thus fixed when reasoning about scenarios. Once a partial scenario has been specified, either the user has declared all of the relevant objects, and hence the set of objects $U$ is also fixed, or objects may appear and disappear from the set, for example, in dynamic scenarios. Thus, the only component that is variable in all scenarios is the set of possible complete scenarios defined by $R^+, R^-, R^\sim, R^?$. Furthermore, in some applications the

\(^3\)This is consistent with the definition of weak representations [5], Section 3.1, Remark 2. Also, refer to the last paragraph of Section 2.4 in this thesis.
3.6 Characteristics of QSTR Applications

This section presents four central properties of QSTR applications. These properties are derived from the formal definition of QSTR applications presented in Section 3.2 and a review of the QSTR application case studies presented in Section 2.8. The significance of these characteristics is that, in order to facilitate the development of QSTR applications, methodologies are required to directly support the software developer in these four areas.

---

Table 3.2: Permitted operations on QSTR scenarios based on the combination of application components that are variable.

<table>
<thead>
<tr>
<th>application design</th>
<th>D</th>
<th>C</th>
<th>L</th>
<th>U</th>
<th>Rα</th>
<th>R</th>
<th>permitted operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>scenario design</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>✓</td>
</tr>
<tr>
<td>reasoning</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>✓</td>
</tr>
</tbody>
</table>

A set of objects $U$ may also be variable. This leaves only three fundamental operations that can be performed on a qualitative partial scenario:

- selecting subsets of tuples in the partial scenario,
- refining the partial scenario by removing a particular complete scenario, and
- editing the set of objects in the scenario.

Therefore, all QSTR application tasks can be defined as a series of tuple selections, partial scenario refinements and scenario universe edits. Table 3.2 illustrates a comparison between actor roles, variable components, and permitted operations on QSTR application and scenario components. The contents Table 3.2 are organised as follows: the left hand column assigns actor roles to the available variables; $D$ represents the domain of interpretation as defined in Section 2.5; components that are variable are represented by $v$ and components that are constant are represented by $c$; operations that can be performed are represented by ✓ and operations that cannot be performed are represented by ✗; partial scenario models distinguish between definite relations $R^\alpha$ where $\alpha = \{+, -, \sim\}$, and indefinite relations, $R^\beta$. 

---
3.6.1 Reasoning Across a Broad Range of Abstraction Levels

QSTR applications often employ a broad range of abstraction levels in the same model. As defined in Section 3.2, a QSTR application theory $\Theta$ consists of

- theories from QSTR calculi, which describe scenarios in a low-level way, and
- custom application-specific relations, which typically describe higher-level qualitative concepts as patterns in qualitative described scenarios.

For example, the QtvLight application (Section 2.8.1) reasons about very abstract high-level emotional responses with respect to low-level physical spatial relations between furniture and light fixtures. In contrast, a numerical ray tracing simulator stores and processes numerical descriptions of ray traces, luminance values, and three-dimensional shapes. QSTR application developers require special techniques for rapidly designing and validating models that have a very layered and hierarchical structure. This is addressed in Chapters 5 and 7.

3.6.2 Continuity Assumption and Neighbourhoods for Changing Scenarios

QSTR relies heavily on the concept of continuity, stating that temporal and spatial objects cannot morph and translate discontinuously, but must change in a continuous fashion (refer to Section 2.7). A fundamental relationship exists between continuity and compositional reasoning, which is the prominent reasoning mechanism for standard QSTR calculi. Moreover, continuity is used directly in critical QSTR tasks such as envisioning [49]. As discussed in Section 2.7, continuity is formally defined using conceptual neighbours and neighbourhood graphs [65, 67].

The standard definition of conceptual neighbours is as follows [45]. Relations $R_1$ and $R_2$ are conceptual neighbours if it is possible for $R_1$ to hold over a tuple of objects at one point in time, and for $R_2$ to hold over the tuple at a later time, with no other mutually exclusive relation holding over the tuple in between. A neighbourhood graph has one node for each relation $R \in R$ and an edge between two nodes if the corresponding relations are neighbours. From an application perspective, this turns out to be too weak to define the conceptual neighbours of high-level relations (such as subjective impressions) in a way that is consistent with the interaction between high level and low level relations. Chapter 7 generalises the definition of conceptual neighbours to apply to QSTR applications, and presents a methodology for designers to customise their conceptual neighbour definitions.

3.6.3 Modelling Infinite Domains

As presented in Section 2.5, QSTR applications typically have infinite domains, in contrast to, for example, relational database models and constraint satisfaction programming (CSP) models
which typically have finite domains [14, 104]. This significantly complicates the process of validating a specific QSTR calculi’s reasoning mechanism so that even expert logicians find this to be a non-trivial task [169].

When considering the perspective of QSTR applications, two further problems are that QSTR applications are significantly more complicated than a given calculus, and the application designers are not necessarily expert logicians. Thus, more practical software engineering based approaches to validating constraints over infinite domains are required for QSTR applications. Chapters 9 and 10 present methodologies for validating QSTR applications based on prominent software engineering techniques.

3.6.4 Reasoning About Objects in Multi-Dimensional Models

QSTR applications very often model multi-dimensional structures. Prominent tasks that use qualitative reasoning, particularly composition, apply transitivity to determine whether a scenario is consistent, and thus rely on relations having an ordering. In qualitative reasoning about physical systems (refer to Section 1.3) relations map to scalar one dimensional quantities, and thus have an obvious total order. On the other hand, spatial scenarios often apply at least two dimensions, thus admitting only partial orderings. Temporal scenarios can also apply multiple dimensions in the form of branching and parallel time streams, resulting in a partial ordering of events.

Multi-dimensional models significantly complicate the design of qualitative reasoning methods, as the designer needs to determine the structure of the partial ordering to employ transitivity. The challenges raised by this characteristic are outside the scope of this thesis.

3.7 Summary

In this chapter a rigorous formal definition of QSTR applications has been presented. QSTR applications are logical systems that respond to specific patterns in scenarios that are described using pertinent QSTR calculi. Formally, a QSTR application consists of a set of relations (the signature) and a first-order theory for describing the interaction between relations (i.e., the theory expresses constraints between relations). A QSTR scenario is a first-order expression that describes objects and relations using the signature of the application.

It was observed that all QSTR applications share one important constraint by virtue of the crisp boolean nature of qualitative concepts. For each QSTR application relation, a scenario expressed using the application requires three mutually exclusive, jointly exhaustive sets of tuples: holds (+), not holds (−), and not applicable (∼). In order to express ambiguity in a scenario, a fourth indefinite (?) set is required.
This theoretical foundation has been employed to formally define the roles of QSTR application users and developers. Users specify scenarios in the language of the QSTR application and request the performance of certain tasks on the scenarios by the application. Developers are responsible for selecting suitable QSTR calculi, selecting a set of custom relation symbols, and encoding a suitable set of constraints. These aspects of application development are directly addressed by the methodologies presented in Chapters 6 and 7.

The theoretical foundation has also been used to identify a fundamental set of basic QSTR application operations that can be performed on scenarios. The basic set of tasks consists of selecting objects from scenarios, refining scenarios by reducing the ambiguity of relations, and modifying the set of scenario objects. In the following chapter these basic tasks are used to enumerate a complete set of purely qualitative application tasks.

Finally, the formal definitions have been used, in conjunction with an analysis of the case studies, to identify four central characteristics of QSTR applications that must be specifically addressed by the development methodologies: reasoning across a broad range of abstraction levels; the continuity assumption and the ubiquity of neighbourhoods for reasoning about changing scenarios; modelling infinite domains; reasoning about objects in multi-dimensional models of space and time. The application development methodologies for requirements specification, design, and validation established in the following chapters are directly based on the characteristics of QSTR applications highlighted in this chapter.
Chapter 4

Formal Software Requirements for QSTR Applications

4.1 Introduction

In the previous chapter the salient characteristics and fundamental properties of QSTR applications were established. In particular, the chapter formally defined the roles of QSTR application developers and users, and derived a fundamental set of basic QSTR application operations that can be performed on scenarios. In addition to understanding the underlying properties of QSTR applications, it is crucial to identify the types of problems that QSTR applications are highly effective at addressing. This is accomplished by defining the relevant types of formal software requirements that characterise QSTR applications.

According to many standard software engineering practices, formal software requirements are necessary for high quality software development and validation [43, 152]. Hence, a formal software requirements methodology for QSTR applications will greatly enhance the software engineering community’s accessibility to QSTR calculi. Moreover, the software development approaches being adapted are the product of research within the software engineering field. Therefore, establishing a formal requirements methodology also supports the application focused research in QSTR.

This chapter establishes the salient types of formal QSTR application requirements that a developer can use to determine how QSTR can effectively address a given problem. As noted in Section 1.1, the focus of the application requirements in this chapter is on the potential QSTR components of an application; the requirements for other aspects of a system that are not directly related to QSTR can be handled by the standard requirements specification techniques for that domain.

The two main issues when establishing a set of core types of formal requirements are

- ensuring that all of the characteristics are critically relevant, and
• ensuring that no significant characteristics have been omitted.

The remainder of this chapter is organised as follows. Section 4.2 presents the important aspects of external interface requirements for QSTR applications. The important aspects of functional requirements are presented in Sections 4.3-4.5. Specifically, Section 4.3 presents a novel functional requirements methodology in which the developer defines an idealised application. Section 4.4 presents an enumeration of basic QSTR application tasks to support a developer in specifying functional requirements. Section 4.5 uses the set of tasks enumerated in the previous section to characterise the general runtime behaviour of QSTR applications. Finally, Section 4.6 presents salient types of performance requirements for QSTR applications.

The software engineering community has developed the IEEE 830-1998 standard which presents guidelines for producing high quality formal software requirements in the form of a software requirements specification (SRS) document [179]. Table 4.1 compares the main recommendation sections of the IEEE 830-1998 standard, Considerations for producing a good SRS with the methodologies presented in this chapter. Table 4.1 illustrates that all significant and relevant sections of IEEE 830-1998 are addressed in this requirements methodology.

---

1The remaining aspects are not relevant to QSTR application requirements, namely prototyping, SRS evolution, and embedding project requirements in SRS.
**4.1 Introduction**

Main topics in IEEE 830-1998

<table>
<thead>
<tr>
<th>Nature of the SRS. Must address external interfaces, functionality, performance, system attributes, and design constraints.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison with QSTR application requirements methodology</td>
</tr>
<tr>
<td>External interfaces, functionality and performance are directly addressed in this chapter. System attributes and design constraints are not addressed as they are non-functional and depend heavily on context (e.g. details about the particular operating system).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Characteristics of a good SRS. Must be correct, unambiguous, verifiable, complete, internally consistent, ranked, modifiable, and traceable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctness is addressed in Section 4.2 with respect to related project documentation and domain knowledge; all requirement types are formally defined and thus unambiguous; the requirements are verifiable (refer to Chapters 9-11); tasks are derived by considering all permutations of underlying parameter conditions and operations thus addressing functional completeness. The remaining SRS characteristics are not QSTR specific.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Joint preparation of SRS. Both customer and developer must jointly author SRS.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The methodologies for eliciting requirements presented in this chapter are presented in a high level way with visual components that abstract from the underlying theoretical details. This facilitates communication between the two parties.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Embedding design in SRS. Only identifying required design constraints and not project a specific design.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Because QSTR applications are declarative systems, the distinction between design constraints and specifying designs is different to OO models. A very direct mapping between requirements and design is acceptable and often necessary (refer to Chapters 5-7). The presented requirements do not project implementation details.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment of the SRS. Role is to restrict valid designs without specifying design or implementation details.</th>
</tr>
</thead>
</table>

| Table 4.1: Comparing the QSTR application requirements methodologies presented in this chapter with IEEE Recommended Practice for Software Requirements Specifications (IEEE 830-1998) [179]. |
4.2 External Interface Requirements

QSTR applications address problems by modelling relevant phenomena and reasoning over those models. Thus, in order to specify an application’s inputs and outputs, some basic aspects of the model need to be determined. This section presents application characteristics that can be used to formalise the external interfaces of the QSTR reasoning engine.

The external interface characteristics are derived from the formal definition of QSTR applications by identifying meaningful combinations of underlying parameter properties and operations. In addition to this, a number of characteristics have been motivated from a review of QSTR research literature [47, 68, 80, 87, 111], such as spatial and temporal entities, and spatial granularity.

Two parameters that all QSTR applications have, and that all user scenarios in an application necessarily share, are relations $R$ and the relation structure specified using a set of constraints, $\Theta$. The following application characteristics about the domain being modelled determine the type of QSTR calculi that can be used for reasoning, and support the developer in establishing basic relations and constraints.

**Spatial and Temporal Entities.** The context of the problem in terms of the entities that need to be modelled. This includes time points, time intervals, spatial points, spatial intervals (directed or undirected), and spatial regions.

**Spatial Granularity.** The spatial context of the problem primarily defined by the scale. Basic categories of environments for which existing QSTR calculi have been designed to reason about, ranging from smallest to largest, are: desktop, indoor [9], outdoor (e.g. a sports field or an open area; Section 5 in [165]), neighbourhood [12, 170], geographical [3, 60, 63, 143], and astronomical [140].

**Spatial Dimensionality.** The number of dimensions required to model the necessary spatial relationships, typically a combination of one, two, or three dimensions, and may also model arbitrary dimensions.

Once the basic spatial relations have been established, the designer needs to consider more abstract relations between objects in the problem domain.

**Object relationships.** Objects in a model can have simple relationships, with only superficial or limited interaction, for example, a GIS application that describes the qualitative spatial relationships between arbitrary features in terms of orientation and proximity. Alternatively, model elements can have complex relationships, with a lot of significant interaction and strong dependencies, for example, a town planning GIS application that incorporates a high degree of semantic content about the types of

---

2References provide examples of QSTR calculi that employ a context-based granularity.
4.2 External Interface Requirements

![Diagram](image)

**Figure 4.1:** A selection of associated external interface QSTR problem characteristics, ordered according to their dependencies.

- **multiple scenarios**
- **scenario relations**
- **universe relations**
- **object mappings**

Determining further details about the type of complex object relations is part of the model design process rather than requirements specification. Complex interactions between QSTR application components are addressed in Chapter 5.

Other important characteristics about the domain are highlighted by the analysis of the theoretical foundations in the following section on functional requirements. Figure 4.1 illustrates the sequence that a developer should consider these characteristics based on the dependencies between each property. Specifically, **scenario relations** requires the model to include **multiple scenarios**, and if the universes between scenarios are **disjoint** then there is no mapping between objects across scenarios and thus the application automatically has complete information about the mapping (i.e. the mapping is **definite**).

**Multiple scenarios.** A QSTR application can model a real-world scene from a number of different perspectives simultaneously by maintaining multiple scenarios (in the formal sense); this includes reasoning over instances of a dynamic scene. Alternatively QSTR applications can model a real-world scene with a single (formal) scenario; this corresponds to a single perspective of a real-world scene that does not change.

**Scenario relations.** A collection of scenarios can be unordered, partially ordered, or totally ordered, as discussed in Section 4.4.

**Universe relations.** The universes from a collection of scenarios can be either disjoint (so that objects are uniquely modelled by each scenario), partially overlap (some, but not all objects are shared between scenarios, for example if objects are entering and leaving a dynamic scene), or completely overlap (all scenarios have equivalent universes).
**Object mappings.** The mappings between the universes of a collection of scenarios are either completely known, or not completely known (for example, they may need to be inferred) during QSTR application run time.

These characteristics also help to define the scope of functional requirements.

### 4.3 Specifying Functional Requirements with an Idealised Application

Functional requirements define the set of tasks that a system must be capable of performing, and how the system will behave during execution [141, 179]. These requirements are specified as the inputs, behaviours, and outputs of system components. This section establishes a novel approach for specifying functional requirements by defining an *idealised application*. Idealised applications greatly facilitate the selection of appropriate QSTR calculi to incorporate into a QSTR application. In Chapter 6 mathematical tools are presented that allow a developer to efficiently analyse myriad QSTR calculi with respect to the idealised application. Moreover, once a developer has selected the most suitable QSTR calculus $A$, an algorithm is presented in Chapter 6 for automatically generating the corresponding theory $\Theta^A$ that most closely models the idealised application.

QSTR calculi are, by design, less expressive abstractions of an interpretation language (although the interpretation language may only be implicitly specified [114]). A developer defines an idealised application by specifying how the QSTR application would be required to behave if it were to receive perfect information about the scenario.

**Definition 4.3.1.** Let $C$ be an interpretation language. The *idealised application* on $C$ is $\theta^C : \text{Int}(\Theta^C) \rightarrow 2^O$. For each output $i \in O$ there is a corresponding *idealised theory* $\Theta^C_i$. The definition of idealised applications is extended to collections of scenarios in the natural manner (refer to Definition 3.2.1).

The idealised application is simply a mapping between patterns in scenarios and combinations of output symbols. It is the responsibility of the developer to define an appropriate mapping which reflects their functional requirements.

The significance of this approach is that the developer specifies an idealised application by ignoring both the limitations on the availability of scenario information during runtime, and the abstractive nature of qualitative systems. Alternatively, the developer can declare different rules in various qualitative and quantitative languages; these heterogeneous rules can then be translated (i.e. normalised or unified) into one or more common interpretation languages to define an idealised application.
To define an idealised application, firstly the developer must select a sufficiently expressive interpretation language $C$. The QSTR community has identified a number of useful domains of interpretation, as presented in Section 2.5. Having selected the interpretation language, the developer must specify an idealised theory $\Theta^C_i$ for each output symbol $i$ that the application is required to produce. Section 3.6.1 observed that intermediate and high-level custom qualitative relations are defined as patterns in low-level scenario descriptions provided by QSTR calculi. Because the idealised application is used for assisting the developer in selecting QSTR calculi, the only output symbols $i$ that should be considered for the idealised application are those that are defined as patterns directly on a low-level scenario description. Thus, in many cases output $i$ will only represent an intermediate inference rather than the ultimate, highest-level application output.

**Example 4.3.2.** In the QtvLight application (refer to Section 2.8.1) rather than defining an idealised theory for a subjective impression such as *spacious*, a more effective approach is to define idealised theories for intermediate inferences such as *bright* ambient illumination, occludes, and directed at; this is because the subjective impressions are entirely defined by intermediate qualitative relations as opposed to low level scenario descriptions in the block algebra [148].

Further examples of idealised applications will be presented in Chapter 6, which presents mathematical tools for selecting QSTR calculi that meet the functional requirements specified by the idealised application.

### 4.4 A Complete Enumeration of QSTR Tasks

A crucial aspect of functional requirements is specifying the tasks that an application is required to perform [141, 179]. This section derives an essential set of basic, purely qualitative tasks from the underlying theoretical foundations; a number of these qualitative tasks are also commonly referred to in the QSTR literature such as qualitative envisioning and diagnosis e.g. [49]. This derivation defines the exact extent to which QSTR can be applied. This establishes a standard with which a software developer can determine, firstly, whether or not QSTR is applicable to their problem, and secondly, what specific qualitative tasks may be suitable. Figure 4.2 illustrates a use case relationship diagram which summarises the complete set of QSTR tasks. Use cases are represented by ellipses, and relationships are represented by arrows. White use cases apply to all models, dark grey use cases apply to unordered and partially ordered scenarios, and light grey use cases apply to ordered scenarios.

---

3Use cases are employed in *use case diagrams* to illustrate the relationship between actors and tasks [141].
Figure 4.2: Use case relationships for basic QSTR tasks.
4.4 A Complete Enumeration of QSTR Tasks

4.4.1 Deriving Standard QSTR Tasks Using Fundamental Operations

A task is defined as a sequence of operations on a mathematical structure. A set of basic QSTR tasks is established by considering sequences of operations that can be performed on the parameters of a scenario in a QSTR application. As presented in Chapter 3, an application has a set of relations $\mathbf{R}$ and constraints $\Theta$. A scenario $s$ consists of a universe $U_s$ and a set of relation state sets $\mathbf{R}_s$. The fundamental operations that can be applied to scenarios are selection, refinement and editing the universe (adding or removing objects). When working with multiple scenarios, in addition to the parameters $\mathbf{R}$ and $\Theta$, there is now a set of partial scenarios $S$, where each scenario $s_i \in S$ has a universe $U_{s_i}$ and a set of relation states $\mathbf{R}_{s_i}$, and there is a mapping function $h$ that maps objects between scenarios, $h : X \rightarrow 2^X$ where $X = \bigcup_{s_i \in S} U_{s_i}$.

The select operator can only be performed on application parameters, i.e. $\mathbf{R}$, $\Theta$, $h$, and $U_s$. The only variables are $U$ and $h$. This limits the number of task categories that can be performed, which are enumerated below and summarised in Table 4.2. The most basic task is to simply execute a selection operation.

**Query.** Isolates relevant subsets of a model. The querying task applies the selection operation. Selection criteria can be specified using any of the available parameters ($\mathbf{R}$, $\Theta$, $h$, $U_s$) and selection can be used to return subsets, again, from any of the available parameters.

**Example 4.4.1.** In the Image Retrieval application (refer to Section 2.8.4) the authors use a qualitative description of images to provide a method for searching through an image database based on semantic content. The database represents a scenario $s$ that contains image objects $U_s$. Queries accept relations $\mathbf{R}$ that represent semantic image concepts and return image objects from $U_s$ that satisfy the semantic criteria.

**Example 4.4.2.** In TreeSap (refer to Section 2.8.2) scenario $s$ contains geographic data and qualitative relations between geographic features in $U_s$. Queries accept relations $\mathbf{R}$ that represent qualitative spatial concepts and return GIS objects from $U_s$ that satisfy the semantic criteria.

In many cases, a QSTR application user may not have complete information about the criteria of the query that they want to execute, for example, a robot reasoning with noisy sensor readings [58]. It may be the case that certain conditions are more flexible than others, and moreover, if a query returns no results, then it would be highly desirable for a user to be able to relax the conditions of their query in an intuitive way, e.g. [76, 149]. A QSTR application can broaden a selection operation by also accepting relations that are within a threshold neighbourhood distance of the relations that the user specified in their query.
Query relaxation. Accept relations that are within a threshold neighbourhood distance of the given target relation. Because selection is a fundamental operation, all QSTR tasks can be relaxed using this approach, such as relaxed consistency checking and relaxed inference.

Conceptual neighbourhoods (refer to Section 2.7) provide an ideal mechanism for query relaxation because they encode the structure of relations based on continuous change. Thus, relations that are physically similar will have a smaller distance through the neighbourhood graph.

The next basic task is applying refinement and universe edits.

Modify. Changes a partial scenario by either eliminating possible scenarios (refinement) or by adding or removing objects from the universe (universe edit).

The next task builds on the previous querying task by testing conditions on the returned subsets, for example to check the consistency of a scenario with respect to the application constraints.

Check consistency. Ensures that the model does not break any application constraints. The model is contradictory if an inconsistent subset (with respect to some application constraint) contains definite tuples, or if an indefinite subset still violates the constraint regardless of how the indefiniteness is resolved. Conditions can involve any of the available parameters (R, Θ, h, Us).

The next basic task is to execute a check consistency task and then refine the model based on the condition results.

Infer. Accepts a partial qualitative description of a model as premise information and infers as much about the indefinite components of the model as possible (typically by composing relations to approximate path-consistency), i.e. deductive closure. Inference typically applies the check consistency and modify tasks to identify and eliminate inconsistent possible scenarios from a partial scenario description, that is, moving tuples out of indefinite relations R_i and into definite relations.

The four basic tasks derived in this section are very general and can be employed in any QSTR application. The following sections build on these basic tasks by formally characterising more specialised QSTR tasks for models that contain two or more scenarios (refer to Section 3.2). In these cases reasoning is applied to the relationship between partial scenarios in a group, and the type of ordering between scenarios is critical for determining the tasks that can be performed.
### Table 4.2: Tasks that can be performed on a scenario with respect to the underlying parameters.

<table>
<thead>
<tr>
<th>Task</th>
<th>Parameters involved</th>
<th>Formal description</th>
<th>State diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Query.</strong> Isolates relevant subsets of a model.</td>
<td>Select: $R, \Theta, h, U_s$</td>
<td>Select from $R, \Theta, U_s$ such that conditions are satisfied (specified using $R, \Theta, U_s$).</td>
<td>select</td>
</tr>
<tr>
<td><strong>Query relaxation.</strong> Accept relations that are within a threshold neighbourhood distance of the given target relation.</td>
<td>Select: $R, \Theta, G, h, U_s$</td>
<td>Select from $R, \Theta, U_s$ such that conditions are satisfied (specified using $R, \Theta, U_s$), and relations are within threshold distance through specified graph in $G$.</td>
<td>edit $U_s$</td>
</tr>
<tr>
<td><strong>Modify.</strong> Changes a partial scenario.</td>
<td>Modify: $h, U_s, R_s$</td>
<td>Refine $R$, by eliminating possible scenarios or edit the universe $U$, by adding or removing objects.</td>
<td>refine</td>
</tr>
<tr>
<td><strong>Check consistency.</strong> Ensures that the model does not break any application constraints.</td>
<td>Query: $R, \Theta, h, U_s$</td>
<td>Execute a query (using $R, \Theta, U_s$) and then test conditions on the result (specified using $R, \Theta, U_s$).</td>
<td>pass, fail</td>
</tr>
<tr>
<td><strong>Infer.</strong> Manipulates the model based on premise information.</td>
<td>Check: $R, \Theta, h, U_s$</td>
<td>Execute a consistency check. Terminate if check passes, or fails and cannot be corrected. Otherwise, modify the model to correct the fail and repeat from start.</td>
<td>modify, pass, fail, uncorrectable</td>
</tr>
</tbody>
</table>

In the state diagram, states are black circles, tasks (composed of states) are ovals with a task label, terminating states are double-lined circles, arrows are state transitions annotated with QSTR operations, and the arrow with no source state is the task entry point.
4.4.2 Deriving QSTR Tasks for Unordered Scenarios

A QSTR application can model a scene from a number of different perspectives simultaneously. This section addresses the case where the partial scenarios representing different perspectives have no inherent order. For example, an application might be used to integrate alternative perspectives of a scenario taken from different autonomous agents. Another example is a police service using a QSTR application to help isolate the position of an emergency by integrating different pieces of partial information in the form of partial scenarios such as “an accident has occurred in the Auckland Domain, near the motorway”.

Inference is used to reason about the relationship between the different partial scenarios. Because there is no ordering, scenarios can only be merged together or split up by modifying the partial scenario universes.

*Merge.* Finds either the union or the intersection of a set of scenarios. This task is particularly useful when the mapping of objects between scenarios is not known, or when different perspectives are compared for consistency. The key challenge is to identify objects that appear in multiple scenarios by applying matching criteria with respect to qualitative relation states and the relative perspectives of the agents involved.

Once partial scenarios have been merged, single scenario tasks described in the previous section can be applied, such as consistency checking and querying. When merging multiple conflicting scenarios with respect to the relations that hold over particular tuples, any conflict resolution approach such as voting can be employed through querying, consistency checking and inference.

*Split.* The inverse of merging. Divides a scenario into a set of possibly overlapping scenarios. For example, splitting can be used when agents do not need to maintain global information about a scenario and instead can efficiently specialise in certain parts of a scenario.

When performing the splitting task, the designer needs to specify criteria for dividing objects between different scenarios and agents. Table 4.3 presents the relationship between the tasks and the underlying application parameters.
<table>
<thead>
<tr>
<th>Task</th>
<th>Parameters involved</th>
<th>Formal description</th>
<th>State diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge</td>
<td>Query: R, G, h, U_i</td>
<td>Finds either the union or the intersection of a set of scenarios. Given set of $s_1, ..., s_n$ such that $s_1, ..., s_{n-1}$ will be merged into $s_n$. Execute query $(\bigcap_{i=1}^{n-1} U_i)$ or $(\bigcup_{i=1}^{n-1} U_i)$ as required and modify $U_i$ to be result. Execute queries for translating relations for each $s_1, ..., s_{n-1}$ via neighbourhood graphs (to account for the difference in perspective of each $s_i$ and $s_n$) and modify $R_{sn}$ to be result. Execute queries to determine if objects in $U_n$ are identical (e.g. if they have the same relation states in $R_{sn}$) and modify $h$ and $U_n$.</td>
<td></td>
</tr>
<tr>
<td>Split</td>
<td>Query: R, G, h, U_i</td>
<td>Divides a scenario into a set of possibly overlapping scenarios. Given set of scenarios $s_1, ..., s_n$ such that $s_1$ will be divided into scenarios $s_2, ..., s_n$. For each scenario $s_i$ from $s_2, ..., s_n$ execute a query to determine objects in $U_{i1}$ relevant for $s_i$, and assign those objects to universe $U_{si}$. Execute queries to translate relations from $s_i$ into each $s_j$, $2 \leq i \leq n$, via neighbourhood graphs (to account for the difference in perspective of $s_1$ and each $s_j$) and modify $R_{si}$ to be result.</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Tasks that can be performed on an unordered set of scenarios with respect to the underlying parameters.
4.4.3 Deriving QSTR Tasks for Ordered Scenarios

A QSTR application can model a scene as a sequence of partial scenarios representing either a change in time, such as a dynamic scene where objects are moving around, or space, such as increasing the granularity of a GIS map. QSTR tasks for sequences of scenarios require neighbourhood graphs. A complete set of basic tasks is established by considering different permutations of scenario types, as illustrated in Table 4.4. Each row depicts a sequence of scenarios. White triangles represent scenarios that are not modified in a task and black triangles represent variable scenarios. Three dots represent that the sequence includes one or more scenarios of the type before and after the dots. Arrows represent consistency checks; the scenario at the head of the arrow is checked for consistency with all scenarios in the sequence up to the scenario at the tail of the arrow.

*Check forward consistency.* Ensures that the first scenario in a sequence of scenarios is valid with respect to neighbourhood graphs. In particular, forward consistency means that tuples from the first scenario always change relation state in a consistent way as the sequence progresses.

*Check backward consistency.* Ensures that the last scenario in a sequence of scenarios is valid with respect to neighbourhood graphs. In particular, backward consistency means that tuples from the last scenario always change relation state in a consistent way as the sequence progresses.

Forward and backward consistency are flexible tasks that can be combined to check consistency in a range of different ways. In isolation they only check the consistency of one scenario with respect to a subsequent or consequent sequence of scenarios. This can be useful when a user is only interested in testing and reasoning about particular key scenarios with respect to other scenarios. It can also be used to check that certain scenarios are consistent in a sequence that has missing or very ambiguous scenarios, for example, due to momentary occlusions of a robot’s sensors. Ensuring total consistency requires checking that each scenario in a sequence is both forward and backward consistent with the other scenarios.

The check sequence consistency task can be combined with the modify task to generate potential future and past scenarios.

*Envision.* Generates potential successor scenarios based on the conceptual neighbourhood graph. Envisioning is with respect to either time, by forecasting into the future, or space, by increasing the resolution of the model. Given a scenario, envisioning to depth $n$ is the set of consistent sequences of length $n$.

*Diagnose.* The inverse of envisioning. Generates potential predecessor scenarios based on the conceptual neighbourhood graph. Also, refer to [36].
Table 4.4: Icons illustrating the important permutations of scenarios in an ordered sequence that are used to define a set of basic QSTR tasks.
Furthermore, contextual information can be used to determine the most likely sequence of scenarios.

Refined envision (diagnose). Selects a subset of the set of consistent sequences. Refined envisioning makes it possible to generate scenarios that are a greater number of steps away from the initial scenario. Refined diagnosis is the inverse of refined envisioning.

Contextual information includes conditional probabilities with respect to the current scenario state (e.g. a cup is very likely to fall if it is not on top of some other object like a table), conditional probabilities with respect to previous scenario states (e.g. trajectories), and domain knowledge about the movement patterns and behaviour of specific objects (e.g. in a predator-prey scenario it is more likely that a predator will follow prey, rather than simply follow a trajectory [165]).

The fifth permutation of scenario types corresponds to modifying intermediate scenarios with respect to the rest of sequence.

Complete sequence. Accepts an incomplete sequence of scenarios (i.e. a sequence that has gaps where scenarios are very ambiguous) and applies a combination of check sequence consistency and modify tasks to determine potential scenarios that can complete the sequence.

Finally, adjacent scenarios in a sequence can be combined into a single scenario, or divided into multiple scenarios.

Ordered merge. Finds the union or intersection of adjacent scenarios in a sequence. Changing space and time are two parameters:

1. Merge snapshots of a dynamic scene. For example, a robot attempting to label dynamic objects across a sequence of sensor readings by referring to conceptual neighbourhoods to decide what sequence of qualitative relations is more likely to belong to a single object, such as correctly labeling which object is the 'coffee cup' and which object is the 'spoon' in each scenario snapshot.

2. Merge snapshots of a scene taken at different granularities. For example, combining satellite images taken at different heights, such as correctly labelling which object is a particular 'mountain' and which object is a particular 'house' in each scenario snapshot.

The ordered merge task can be used to resolve ambiguity about the mapping between objects in adjacent scenarios. It can also be used to compress adjacent scenarios into one scenario by abstracting from unnecessary relation transition information such as the ordering between certain transitions of different objects. For example, if objects \((A,B)\) transition from relation
near to far between scenarios \( s_1 \) and \( s_2 \), and objects \((C,D)\) transition from relation *left of* to *in front of* between scenarios \( s_2 \) and \( s_3 \), then the two scenarios \( s_2 \) and \( s_3 \) can be merged into \( s' \) where both transitions for objects \((A,B)\) and \((C,D)\) occur simultaneously (this is a lossy process where the ordering between transitions is removed).

**Ordered split.** The inverse of ordered merge. This is primarily used for injecting an ordering between transitions that occur in different objects, such as reversing the process of order merging scenarios \( s_2 \) and \( s_3 \) in the previous example.

The ordered split task can be used to increase the granularity between transitions. The designer will need to impose an ordering between transitions that have occurred simultaneously in a scenario, for example, by employing domain knowledge. Table 4.5 presents the relationship between the tasks and the underlying application parameters.
<table>
<thead>
<tr>
<th>Task</th>
<th>Parameters involved</th>
<th>Formal description</th>
<th>State diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check forward consistency. Ensures that the first scenario in a sequence of scenarios is valid with respect to neighbourhood graphs.</td>
<td>Query: ( R, G, h, U_s ) Modify: ( U_s, R_s )</td>
<td>Given sequence ( s_1 \ldots s_n ). For each relation ( R ), for each tuple ( x ) from ( R' ) and ( R' ) in scenario ( s_1 ): ( s_1 ) is forward consistent if ( x ) is in ( R_1^+ \ldots R_n^+ ) in at least one possible scenario for each ( s_2, \ldots, s_n ), respectively AND path ( R \cdot R_1^+ \ldots R_n^+ ) is in a neighbourhood graph in ( G ).</td>
<td><img src="query" alt="Diagram" /></td>
</tr>
<tr>
<td>Check backward consistency. Ensures that the last scenario in a sequence of scenarios is valid with respect to neighbourhood graphs.</td>
<td>Query: ( R, G, h, U_s )</td>
<td>Given sequence ( s_1 \ldots s_n ). For each relation ( R ), for each tuple ( x ) from ( R' ) and ( R' ) in scenario ( s_n ): ( s_n ) is backward consistent if ( x ) is in ( R_1^- \ldots R_n^- ) in at least one possible scenario for each ( s_1, \ldots, s_{n-1} ), respectively AND path ( R_1^- \ldots R_n^- ) is in a neighbourhood graph in ( G ).</td>
<td><img src="fail" alt="Diagram" /></td>
</tr>
<tr>
<td>Envision. Generates potential successor scenarios based on the conceptual neighbourhood graph.</td>
<td>Check: ( R, G, h, U_s ) Modify: ( U_s, R_s )</td>
<td>Given sequence ( s_1 \ldots s_n ). ( s_n ) is an envisionment of ( s_1 \ldots s_{n-1} ) if it is backward consistent. Otherwise, modify ( s_n ) to make it backward consistent (note: ( s_n ) can be initialised by placing all tuples in indefinite sets).</td>
<td><img src="modify" alt="Diagram" /></td>
</tr>
<tr>
<td>Diagnose. Generates potential predecessor scenarios based on the conceptual neighbourhood graph.</td>
<td>Check: ( R, G, h, U_s ) Modify: ( U_s, R_s )</td>
<td>Given sequence ( s_1 \ldots s_n ). ( s_1 ) is a diagnosis of ( s_2 \ldots s_n ) if it is forward consistent. Otherwise, modify ( s_1 ) to make it forward consistent (note: ( s_1 ) can be initialised by placing all tuples in indefinite sets).</td>
<td><img src="chk" alt="Diagram" /></td>
</tr>
<tr>
<td>Refined Envision (Diagnose). Selects a subset of the set of consistent sequences.</td>
<td>Check: ( R, G', h, U_s ) Modify: ( U_s, R_s )</td>
<td>Identical to normal envisioning, except rather than testing for a consistent path in ( G ), refer to another source to determine acceptable paths ( G' ) (called refined check consistency). (note: refined diagnosis uses refined forward consistency)</td>
<td><img src="modify" alt="Diagram" /></td>
</tr>
<tr>
<td>Complete sequence. Determines scenarios that can be inserted into a sequence consistently.</td>
<td>Check: ( R, G, h, U_s ) Modify: ( U_s, R_s )</td>
<td>Given sequence ( s_1 \ldots s_n ). Repeatedly execute forward and backward consistency checks. If they fail, then modify the scenario that caused the fail. Terminate when all checks pass, or an uncorrectable failure occurs. Weak: only check forward consistency of ( s_1 ) and backward consistency of ( s_n ) (but modify any intermediate scenario on consistency fail). Strong: check forward and backward consistency of all scenarios, from ( s_1 ) to ( s_n ).</td>
<td><img src="chk" alt="Diagram" /></td>
</tr>
<tr>
<td>Task</td>
<td>Parameters involved</td>
<td>Formal description</td>
<td>State diagram</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Ordered merge. Finds the union or intersection of adjacent scenarios in a sequence.</td>
<td>Query: R, G, h, U, s</td>
<td>Given sequence s₁, ..., sₙ such that s₁,...,sₙ₋₁ will be merged into sₙ. Execute query (⎨₁,...ₙ₋₁ ∪ Uₙ) or (∪₁,...ₙ₋₁ Uₙ) as required and modify Uₙ to be result. Execute queries for translating relations for each s₁,...,sₙ₋₁ via neighbourhood graphs (to account for the difference in perspective of each sᵢ and sₙ, or to prioritise relations to resolve conflicts) and modify Rₙ to be result. Execute queries to determine if objects in Uₙ are identical (e.g. if they have the same relation states in Rₙ) and modify h and Uₙ.</td>
<td>modify</td>
</tr>
</tbody>
</table>

**Ordered split.** Divides a scenario by ordering transitions that occur in different objects.

<table>
<thead>
<tr>
<th>Task</th>
<th>Parameters involved</th>
<th>Formal description</th>
<th>State diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Query: R, G, h, U, s</td>
<td>Given sequence s₁, ..., sₙ, such that s₁ will be divided into scenarios s₂,...,sₙ. For each scenario sᵢ from s₂,...,sₙ execute a query to determine objects in Uᵢ relevant for sᵢ and assign those objects to universe Uᵢ. Execute queries to translate relations from sᵢ into each sᵢ, 2≤i≤n, via neighbourhood graphs (to account for the difference in perspective of s₁ and each sᵢ, or to order relations) and modify Rₙ to be result.</td>
<td>query</td>
</tr>
</tbody>
</table>

Table 4.5: Tasks that can be performed on an ordered sequence of scenarios with respect to the underlying parameters.
4.4.4 Deriving QSTR Tasks for Partially Ordered Scenarios

A QSTR application can model multiple scenario sequences such that different sequences focus on modelling different, possibly overlapping, universes. Sequences may join at points where they share a scenario, and branch where the scenario universes differ. An application may model different sequences simultaneously for a range of purposes. For example, in a model of a dynamic traffic light scenario, the sequence may branch to represent two possibilities, that a car enters the scenario, or does not enter, at a particular instant in time. Another example is splitting a universe to allow autonomous agents to specialise in particular scenario objects, and ignore their relations with other objects. Finally, the relationship between some dynamic scenario objects may simply be irrelevant, such as the spatial relations between people and furniture in different rooms of a house.

Unordered merge and split tasks presented in Section 4.4.2 above can be used to convert between partially ordered and ordered sequences, as shown in Table 4.6. Merge can be used to combine partially ordered scenarios into an ordered scenario, for example, if multiple agents are mapping an area, then the QSTR application can combine two scenario sequences at the points where robots are co-located. Split can be used to make a sequence branch into two partially ordered sequences, for example, dividing a soccer scenario sequence into multiple sequences to allow robot soccer players to specialise in reasoning about particular objects.
### 4.5 Characterising QSTR Application Execution Behaviour

This section establishes a template for the behaviour of QSTR applications based on the purely qualitative tasks from Section 4.4. This template can be used by software developers to characterise QSTR applications by explicitly incorporating behaviour into task requirements.

Figure 4.3(left) illustrates the statechart diagram describing the generic behaviour of QSTR applications during execution, derived from the tasks in Table 4.2; circles represent states, arrows represent possible state transitions, and the arrow annotations describe the effect of the transition on the QSTR application. States represent how the application is manipulating the model. During the model development state, inference tasks refine the model and edit scenario universes. During the model querying state, querying and check consistency tasks isolate and compare relevant parts of the model. Model changed occurs when an agent external to the reasoning process modifies the model, such as a user updating geographical data, or a sensor delivering new information. Figure 4.3(right) shows the underlying low level model operations that are performed in each state.

<table>
<thead>
<tr>
<th>Task</th>
<th>Parameters involved</th>
<th>Formal description</th>
<th>State diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Merge.</td>
<td>Query: R, G, h, U,</td>
<td>Given set of sequences $s_1, \ldots, s_n$ such that all sequences are the same length $k$, and $s_1^i, \ldots, s_{i-1}^i$ will be merged into $s_i$. Execute a query to form $k$ sets of scenarios $S_1, \ldots, S_k$ respectively, for $1 \leq i \leq k$. Merge the set $S_i$ into the $i$th scenario $s_i$ of sequence $s_n$.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Merge: $h, U, R$</td>
<td></td>
<td>merge</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>query</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>split</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>query</td>
</tr>
<tr>
<td>Sequence Split.</td>
<td>Query: R, G, h, U,</td>
<td>Given set of sequences $s_1, \ldots, s_n$ such that all sequences are the same length $k$, and $s_1$ will be divided into $s_2, \ldots, s_n$. For each scenario $s_i$ in $s_1$, execute a query to gather a set of scenarios $s_2^i, \ldots, s_n^i$ taken from the $i$th position of sequences $s_2, \ldots, s_n$, respectively. Split $s_i$ into $s_2^i, \ldots, s_n^i$.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Split: $U, R$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: Tasks that can be performed on a partially ordered set of scenarios with respect to the underlying parameters.

This section establishes a template for the behaviour of QSTR applications based on the purely qualitative tasks from Section 4.4. This template can be used by software developers to characterise QSTR applications by explicitly incorporating behaviour into task requirements.

Figure 4.3(left) illustrates the statechart diagram describing the generic behaviour of QSTR applications during execution, derived from the tasks in Table 4.2; circles represent states, arrows represent possible state transitions, and the arrow annotations describe the effect of the transition on the QSTR application. States represent how the application is manipulating the model. During the model development state, inference tasks refine the model and edit scenario universes. During the model querying state, querying and check consistency tasks isolate and compare relevant parts of the model. Model changed occurs when an agent external to the reasoning process modifies the model, such as a user updating geographical data, or a sensor delivering new information. Figure 4.3(right) shows the underlying low level model operations that are performed in each state.
This template captures all QSTR application behavioural sequences, hence QSTR applications can be characterised with respect to behaviour by identifying a set of significant, distinct paths through the state diagram. Six basic cases are illustrated in Table 4.7. The following characteristics are sufficient for determining an application’s behavioural class, as illustrated in Table 4.8.

**Change source.** Model can be changed either internally via automated reasoning, externally via some agent external to the application, both, or neither (when the model is never changed during its lifetime).

**Task categories.** QSTR tasks fall into three basic categories, either querying the model, developing the model, or both.

The following two characteristics can help a developer determine the behavioural class if they are uncertain about the values of change source and task categories.

**Model dependency.** The duration of dependency on the model during an application session. A problem may only have initial dependency, where all the information required for processing is initially available, for example, checking the qualitative consistency of an existing spatial database. Alternatively, other problems require information that is not initially available and thus the dependency on the working model is ongoing, for example, checking the consistency of a spatial database whenever modifications are made.
<table>
<thead>
<tr>
<th>State diagram path</th>
<th>Behaviour description</th>
<th>Example</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>startup --- model development --- shutdown</td>
<td>All information required for reasoning is available before using the application.</td>
<td>Checking the qualitative consistency of an existing spatial database.</td>
<td>Chg. src.: internal Tasks: develop Dependency: initial Lifetime: change</td>
</tr>
<tr>
<td>startup --- model querying --- shutdown</td>
<td>The user will update the model, and employ automated reasoning tasks. The user will not query the model.</td>
<td>Iteratively check the qualitative consistency of a developing spatial database as data is added.</td>
<td>Chg. src.: both Tasks: develop Dependency: ongoing Lifetime: change</td>
</tr>
<tr>
<td>startup --- model querying --- model development --- shutdown</td>
<td>Application specialises in qualitative querying. All qualitative information is initially available, hence no inference is required.</td>
<td>GIS application that reads from a database of qualitative spatial relations between features on a map. Users can execute qualitative queries.</td>
<td>Chg. src.: none Tasks: query Dependency: ongoing Lifetime: no change</td>
</tr>
<tr>
<td>startup --- model querying --- model development --- shutdown</td>
<td>Equivalent to the previous behaviour, except that the model may change during execution.</td>
<td>In addition to the above example, the user can modify the qualitative relations inferred from the numerical database.</td>
<td>Chg. src.: external Tasks: query Dependency: ongoing Lifetime: change</td>
</tr>
<tr>
<td>startup --- model querying --- model development --- shutdown</td>
<td>Application specialises in qualitative querying. Not all qualitative information is immediately available, hence application must infer qualitative relations.</td>
<td>GIS application reads numerical database of features. Checks qualitative consistency (e.g. ‘the school is safely accessible’). Users can execute qualitative queries.</td>
<td>Chg. src.: internal Tasks: dev. + query Dependency: ongoing Lifetime: change</td>
</tr>
<tr>
<td>startup --- model querying --- model development --- shutdown</td>
<td>Application provides inference and querying services.</td>
<td>In addition to the above example, the user can modify the qualitative relations inferred from the numerical database.</td>
<td>Chg. src.: both Tasks: dev. + query Dependency: ongoing Lifetime: change</td>
</tr>
</tbody>
</table>

Table 4.7: Enumerating relevant paths through the QSTR application statechart in Figure 4.3 to determine basic application behavioural patterns. Characteristic values that uniquely define each behavioural class are bolded in the right column.
change source

<table>
<thead>
<tr>
<th>task categories</th>
<th>none</th>
<th>internal</th>
<th>external</th>
<th>both</th>
</tr>
</thead>
<tbody>
<tr>
<td>develop</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
<td><img src="image4.png" alt="Diagram" /></td>
</tr>
<tr>
<td>query</td>
<td><img src="image5.png" alt="Diagram" /></td>
<td><img src="image6.png" alt="Diagram" /></td>
<td><img src="image7.png" alt="Diagram" /></td>
<td><img src="image8.png" alt="Diagram" /></td>
</tr>
<tr>
<td>both</td>
<td><img src="image9.png" alt="Diagram" /></td>
<td><img src="image10.png" alt="Diagram" /></td>
<td><img src="image11.png" alt="Diagram" /></td>
<td><img src="image12.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Table 4.8: Each of the six application behaviour classes derived in Table 4.7 can be uniquely determined by their change source and task category characteristics.

Figure 4.4: A selection of associated functional and performance QSTR problem characteristics, ordered according to their dependencies.

*Model lifetime.* The state of the model during an application session. Once initialised the working model may never change, for example, bootstrapping a robot with a qualitative description of an environment on which the robot runs qualitative queries for accomplishing navigation tasks. Alternatively, the working model may change, for example, if a robot performs simultaneous qualitative location and mapping. Figure 4.4 illustrates the dependencies between the values of these characteristics.

### 4.6 Performance Requirements

Environmental characteristics specify properties about scenarios, the QSTR language, and about how the model is changed. The designer can then determine the impact that these properties
will have on the memory and processing time required to perform the necessary application tasks.

*Query complexity.* The descriptive complexity of the QSTR language. This specifies the computational complexity needed to process the required queries (and hence all tasks in general) in the worst case.

*Scenario size.* Amount of information stored in relations approximated with the order of magnitude of the number of objects in a typical use case. For example, small scenarios have up to 100 objects, large scenarios have up to 1000 objects, very large scenarios have up to 10,000 objects, some scenarios that read information from large databases may process millions of objects, and so on.

Because many QSTR calculi are NP-complete, reasoning with certain large scenarios is intractable. This is a topic of ongoing research within the QSTR community [4, 18, 70], although for well known calculi such as Allen’s interval calculus and RCC researchers have determined exactly which fragments are tractable to compute [13, 125]. Chapter 8 discusses methodologies and tools to support the designer in managing reasoning performance. Furthermore, Chapter 10 presents a methodology for characterising the complexity of QSTR application expressions (such as constraints and queries) referred to as H-complexity.

The combination of the following characteristics about the changes that a model will undergo during an application session helps to determine how critical resource management will be, and whether the application is likely to be able to perform the required tasks within the necessary resource constraints. As illustrated in Figure 4.4 the values between these characteristics are independent, however they are associated with characteristics in the previous section.

*Model change.* Models either change monotonically or non-monotonically. Importantly, non-monotonic changes require previous reasoning to be reversed, and the designer will need to ensure that these tasks can be performed within the required resource constraints.

*Model stability.* The frequency of changes that occur to a model. Models are either stable and changes occur rarely, or volatile and changes occur frequently.

*Change scope.* The impact of a change under typical conditions can be either local, effecting isolated fragments of a model, or global, effecting large portions of a model. A series of local changes may be efficiently managed by using iterative reasoning updates, whereas global changes may be more efficiently managed by abandoning the previous model and reasoning from premises in a new model.

An application that has a stable model, with local monotonic changes is likely to require less resources to manage model changes than an application that has a volatile model with global non-monotonic changes.
4.7 Summary

In this chapter, categories for formal QSTR application requirements have been established in the areas of external interface requirements, functional requirements, and performance requirements. The characteristics within each category can be employed by a software developer to classify the types of problems that need to be addressed by QSTR applications. The IEEE standard for developing software requirements IEEE 830-1998 has been reviewed to ensure that all of the presented characteristics are relevant to software engineering and that no important concepts were omitted from consideration.

External interface requirements have been derived by considering how QSTR applications can model a problem domain. The QSTR application domain being modelled can be characterised by the types of spatial and temporal entities, the level of granularity, and the dimensionality; these requirements help to determine the type of QSTR calculi that could be effective for a given problem. Object relationships characterise more abstract, application-specific relations between objects in the problem domain. In problems that involve modelling multiple scenarios the pertinent requirements characteristics are the relationships between scenarios and each scenario’s universe.

A methodology has been presented in which the developer specifies the functional requirements of an application by defining an idealised application in a sufficiently expressive interpretation language. An idealised application is specified by ignoring both the limitations on the availability of scenario information during runtime, and the abstractive nature of qualitative systems.

An enumeration of basic QSTR application tasks was presented to support a designer in specifying functional requirements. The set of tasks was derived from possible sequences of operations on the underlying mathematical structure of QSTR applications. General tasks that apply to any problem domain are query, modify, check consistency and infer. When problem domains are modelled using multiple scenarios, additional tasks were derived that apply to the relationship between scenarios. Unordered and partially ordered scenario tasks are merge and split. Ordered scenario tasks are check forward and backward consistency, envision, diagnose, complete sequence, ordered merge and ordered split.

A statechart template has been established that represents all basic QSTR application behavioural sequences. This template was used to derive six basic classes of behaviour by identifying significant, distinct paths through the state diagram. Behavioural characteristics that define the classes are change source, task categories, model dependency and model lifetime.

Finally performance requirements has been presented to assist a developer in determining the impact that particular properties will have on the available resources during task execution. Query complexity and scenario size characterise the computational complexity of QSTR tasks.
Model change, model stability and change scope characterise how the QSTR application model is expected to be modified during task execution.
Chapter 5

Organising and Visualising QSTR Applications

5.1 Introduction

In Chapter 3 a theoretical foundation for QSTR applications was defined. Based on these underlying mathematical semantics a formal requirements methodology for QSTR applications was established in the previous chapter. However, while providing a sound formal basis for applying QSTR and characterising the problems that can be addressed, a methodology is still required that directly supports developers in the organisation and design of QSTR applications. In particular, it is natural for a designer to want to arrange relations into appropriate categories and compartmentalise domain concepts in a process of information chunking; this provides a more cognitively manageable QSTR application, and hence directly improves the efficacy of the design process [40, 44]. On the other hand, structuring relations at this level is not required for the mechanical process of automated reasoning. Moreover, it is highly desirable that the underlying theory be kept as clean and simple as possible. Thus, methodologies for relation organisation are analogous to design patterns in software engineering [152], as they define a standard approach for modelling some structure that is common to a range of QSTR application domains as opposed to complicating the underlying semantics.

This chapter presents a mapping between the concepts in object oriented software engineering and the QSTR application domain. This greatly enhances the accessibility of QSTR applications, as it allows the adoption of prominent software development tools such as the Unified Modelling Language (UML) for designing QSTR applications. Section 5.2 presents a methodology for organising relations in a model using the concept of fragments; furthermore an analysis of the case study QSTR applications is conducted with respect to the organisation of fragments. Section 5.3 presents the details of the mapping between object oriented (OO) domain concepts and the central concepts in QSTR applications. Section 5.4 adapts UML se-
sequence diagrams to enable a developer to visualise the flow of information between components in a QSTR application.

5.2 Organising Relations and Constraints Into Model Fragments

Certain relations in QSTR application models can often be grouped together, because they refer to a similar aspect of a domain at the same abstraction level. Moreover, relations within a group often share constraints such as mutual exclusivity, symmetry, having inverse pairs, and so on. In the area of qualitative reasoning (QR) about physical systems the term *model fragment* refers to modular, partial models composed of different components that can be reused and extended in other models [91]. The concept of fragments is ideal for structuring QSTR applications: a QSTR fragment is a cohesive group of relations and their constraints (expressed as theories, as defined in Chapter 3). In general, a QSTR calculus corresponds to a fragment. However it is the responsibility of the developer to determine the precise meaning of cohesive, according to the specific application context, when organising custom qualitative relations and their associated theories into fragments.

**Example 5.2.1.** In the QtvLight application (refer to 2.8.1), Guesgen’s block algebra [2, 78] forms a natural fragment as it can be reused in a wide range of QSTR applications. The set of qualitative relations that define the subjective impressions of a lit environment also forms a natural fragment, which is distinct from the relations that pertain to varying qualitative degrees of ambient illumination [148].

Simple fragment metrics such as the number of fragments and the number of relations per fragment are highly effective in characterising QSTR applications. Specifically, fragment analysis can provide a heuristic measure of the overall relative application complexity, and can highlight important structural and operational properties. Table 5.1 presents a comparison of the QSTR application case studies, along with the SparQ library and the AmI Framework (developed by Bhatt, Dylla and Hois [37]; refer to Section 1.3) with respect to the number of fragments and the number of relations per fragment. It must be noted that multiple values or formulae for the relation count in Table 5.1 represent different variations of the associated calculus; for example, the oriented point calculus [8] supports variable granularity $m$. Table 5.2 presents the distribution of fragments and relations at high, intermediate, and low abstraction levels based on typical scenario input (low level fragments) and expected application output (high level). Table 5.3 presents the distribution of custom application-specific fragments and relations (i.e. excluding the application of existing QSTR calculi); the mean number of relations per fragment is 31 (standard deviation 30) and the mean number of fragments per application is 6 (standard deviation 5).
It can be observed from Tables 5.1, 5.2, and 5.3 that QtvLight is the most \textit{structurally} complex application with respect to the range of abstraction levels and the number of potential fragment interactions, although many of the fragments contain very few relations. In contrast, TreeSap GIS is very shallow with respect to the levels of abstraction. The quantity of fragments and the range of abstraction levels also highlights an operational distinction between TreeSap and QtvLight; in TreeSap the higher level relations which correspond to patterns on low level relations are dynamically defined by users during querying, whereas in QtvLight the higher level relations are built into the design of the application. It is also interesting to observe in the applications QtvLight, Image Retrieval, and SailAway that the low level QSTR calculi tend to contain more relations than higher level fragments by an order of magnitude. This suggests a more universal principle that higher level fragments tend to be significantly less expressive compared to low level relations. This is consistent with the definition of QSTR applications given in Section 3.2, specifically, that high level relations represent patterns in lower level relations. This demonstrates that the hierarchical nature of QSTR applications is reflected in the organisation of fragments and the distribution of relations.

The QSTR fragment is a simple and powerful concept that provides an intuitive mechanism for organising QSTR relations. In the following sections fragments form the basis of a mapping between object oriented concepts and the QSTR application domain.
Table 5.1: Comparison of QSTR applications and libraries based on the number of fragments and the number of relations per fragment. For each application the fragments are ordered by the level of abstraction, from highest to lowest.

Table 5.2: Comparison of QSTR applications and libraries based on the number of high, intermediate, and low level fragments, and the number of relations per fragment.
5.3 Mapping Object Oriented Concepts to the QSTR Application Domain

The object oriented (OO) paradigm is one of the most popular approaches for organising and developing applications in the software engineering community [92, 141]. Many prominent software modelling languages, such as the Unified Modelling Language (UML) unified process, are used for specifying and visualising primarily object oriented (OO) software systems [92, 141]. In particular, UML class diagrams visualise the static components of an OO software system, specifically focusing on the structure of objects and how their operations are organised [141].

1. OO software design paradigms such as UML class diagrams can potentially provide a powerful interface for software engineers to develop and utilise QSTR techniques; this requires a mapping from OO concepts such as classes, object attributes, object operations and class relationships, to the QSTR application domain. The central challenge which prevents the direct employment of OO design tools for QSTR application development is that OO software is imperative whereas QSTR applications are rule-based systems.

The key for associating QSTR applications with OO systems is to specify fragments as the analogue of OO classes. This establishes a fundamental mapping between OO software and QSTR applications, and forms a basis for adapting other prominent OO concepts. Thus, the analogue to class diagrams in the QSTR domain is the fragment structure of the qualitative model. Table 5.4 summarises the adaptation of OO class diagram notation for the QSTR domain. OO classes are analogous to fragments in a QSTR application model such as the Allen Interval Algebra.

Table 5.3: Comparison of the custom qualitative components of QSTR applications and libraries (i.e. excluding QSTR calculi), based on the number of high, intermediate, and low level fragments, and the number of relations per fragment.

<table>
<thead>
<tr>
<th>Application</th>
<th>Total fragments (median relations per fragment, total)</th>
<th>High level fragments (median relations per fragment, total)</th>
<th>Intermediate fragments (median relations per fragment, total)</th>
<th>Low level fragments (median relations per fragment, total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Retrieval</td>
<td>6 (6, 33)</td>
<td>1 (6, 6)</td>
<td>-</td>
<td>4 (6, 27)</td>
</tr>
<tr>
<td>TreeSap GIS</td>
<td>2 (7, 14)</td>
<td>-</td>
<td>-</td>
<td>2 (7, 14)</td>
</tr>
<tr>
<td>QtvLight</td>
<td>17 (2.5, 90)</td>
<td>1 (10, 10)</td>
<td>6 (2.5, 18)</td>
<td>10 (2.5, 62)</td>
</tr>
<tr>
<td>SailAway</td>
<td>5 (3.5, 15)</td>
<td>1 (1, 1)</td>
<td>-</td>
<td>3 (4, 14)</td>
</tr>
<tr>
<td>AmI Framework</td>
<td>4 (3, 15)</td>
<td>-</td>
<td>-</td>
<td>4 (3, 15)</td>
</tr>
<tr>
<td>SPBD</td>
<td>4 (4.5, 21)</td>
<td>-</td>
<td>1 (4, 4)</td>
<td>3 (5, 17)</td>
</tr>
</tbody>
</table>

5.3 Mapping Object Oriented Concepts to the QSTR Application Domain

The object oriented (OO) paradigm is one of the most popular approaches for organising and developing applications in the software engineering community [92, 141]. Many prominent software modelling languages, such as the Unified Modelling Language (UML) unified process, are used for specifying and visualising primarily object oriented (OO) software systems [92, 141]. In particular, UML class diagrams visualise the static components of an OO software system, specifically focusing on the structure of objects and how their operations are organised [141].

1. OO software design paradigms such as UML class diagrams can potentially provide a powerful interface for software engineers to develop and utilise QSTR techniques; this requires a mapping from OO concepts such as classes, object attributes, object operations and class relationships, to the QSTR application domain. The central challenge which prevents the direct employment of OO design tools for QSTR application development is that OO software is imperative whereas QSTR applications are rule-based systems.

The key for associating QSTR applications with OO systems is to specify fragments as the analogue of OO classes. This establishes a fundamental mapping between OO software and QSTR applications, and forms a basis for adapting other prominent OO concepts. Thus, the analogue to class diagrams in the QSTR domain is the fragment structure of the qualitative model. Table 5.4 summarises the adaptation of OO class diagram notation for the QSTR domain. OO classes are analogous to fragments in a QSTR application model such as the Allen Interval Algebra.

2. An OO class represents a discrete model concept [141] and thus a possible alternative is to view QSTR relations as equivalent to OO classes. However, QSTR relations are usually very tightly associated with at least one other QSTR relation which conflicts with the concept of OO class cohesion, and OO classes are complex entities that have attributes and operations whereas relations are relatively simple structures.

UML has been selected as an OO modelling paradigm to demonstrate the mapping between QSTR applications and OO concepts due to its prominence in software engineering. It must be noted however that any OO modelling paradigm could have been chosen with similar results.
classes are analogous to QSTR constraints between fragments. Two very common types of fragment constraints map directly to two types of OO relationships. These are presented as design patterns in the following sections.\(^3\) In both cases, relations in one fragment are tightly associated to relations in a collection of other fragments because they refer to the same concept in the domain, but at different levels of abstraction. Section 5.3.1 presents fragment definitions which are analogous to OO aggregation associations [141]. Section 5.3.2 presents fragment generalisations which are analogous to OO generalisations [141].

### 5.3.1 Design Pattern: Fragment Definitions

The lower level relations represent properties or attributes, and specific combinations of these properties realise some particular higher level concepts.

**Therefore:**

Designate the higher level fragment as the abstraction domain, and the lower level fragment as the reference domain. For each abstraction domain relation, select a subset of reference domain relations that together describe or define the higher level relation; this subset is a definition of the higher level relation. There can be more than one definition for each higher level concept. The developer should define each subset as a minimal subset; if any of the lower level relations are removed from the definition then the subset no longer accurately describes the higher level concept. This encourages the developer to create multiple precise definitions that can overlap, rather than a smaller number of fuzzy definitions.

For each definition, specify a constraint of the form:

<conjunction of reference domain relations in the definition> implies <the higher level relation>.

**Example 5.3.1.** In the Image Retrieval application (refer to Section 2.8.4) a mountain image can be defined as an image with more mountains than sky, and more sky than grass. To express this, the designer can define two fragments, one for qualitative image categories, including the relation *mountain*, and another for qualitative differences in features of an image, including “*mountain > sky*” and “*sky > grass*”. The conjunction between fragments is then implemented with the constraint

\[
x \in R^{+}_{\text{mountain}>\text{sky}} \land x \in R^{+}_{\text{sky}>\text{grass}} \rightarrow x \in R^{+}_{\text{mountain}}
\]

---

\(^3\)This thesis follows the Portland Form design pattern format [50]:

<problem description (paragraph text)>

**Therefore:**

<solution description (paragraph text)>

<solution examples>.
5.3 Mapping Object Oriented Concepts to the QSTR Application Domain

<table>
<thead>
<tr>
<th>Standard Class Diagram Concept</th>
<th>Analogous QSTR Concept and Key Differences to Standard Class Diagrams</th>
<th>UML Symbol Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class. Defines a modelled concept as a category of objects.</td>
<td>OO classes are fragments. OO attributes are fragment relations, and OO operations are fragment constraints.</td>
<td>Subjective Impression&lt;br&gt;relaxing&lt;br&gt;cozy&lt;br&gt;pleasant&lt;br&gt;dramatic&lt;br&gt;tense&lt;br&gt;mutually exclusive&lt;br&gt;symmetric&lt;br&gt;neighbourhood graph</td>
</tr>
<tr>
<td>Aggregation association. Represents part-whole relationships between classes.</td>
<td>Fragment definitions of the form: &lt;conjunction of low level relations&gt; [\subseteq] &lt;high level relation&gt;</td>
<td>Features&lt;br&gt;Image Category</td>
</tr>
<tr>
<td>Generalisation. Represents taxonomic relationships between classes.</td>
<td>Fragment generalisations of the form: &lt;disjunction of low level relations&gt; [\subseteq] &lt;high level relation&gt;</td>
<td>Temporal Interval&lt;br&gt;Semi-Intervals</td>
</tr>
<tr>
<td>Class associations. In general associations describe connections between classes.</td>
<td>Relations from each fragment appear together in a constraint.</td>
<td>Ambient Illumination&lt;br&gt;Subjective Impression</td>
</tr>
</tbody>
</table>

**Table 5.4:** Adapting standard UML class diagram notation and concepts [141] to the QSTR application domain.


5.3.2 Design Pattern: Fragment Generalisation

The difference between two fragments is granularity, so that relations in one fragment are a coarse, incomplete, ambiguous, or generalised representation of relations in another, more fine grained fragment.

Therefore:

Designate the higher level fragment as the abstraction domain, and the lower level fragment as the reference domain. For each abstraction domain relation, select a subset of reference domain relations that individually represent the same concept as the abstraction domain relation, but to a more precise degree; this subset is a specialisation. There is always exactly one specialisation subset for each higher level relation. Furthermore, two specialisations for two different high level relations can overlap. The developer should define each subset as a maximal subset; if any lower level relations are not included in the subset then in no way do they refine the higher level relation. This ensures that a specialisation represents all possible refinements of a high level concept, and tends to prevent the developer ruling out potential, albeit improbable, refinements, which would compromise reasoning soundness. Following this strategy, a developer can clearly identify when a high level relation is too coarse or general (i.e. the specialisation subset is too large), and may decide to either partition the overly general relation into different relations within the abstraction domain, or introduce an entirely new intermediate abstraction layer fragment.

For each specialisation, specify a constraint of the form:

\[
\text{<disjunction of reference domain relations in the specialisation>} \implies <\text{the higher level relation}>.
\]

Example 5.3.2. Consider the incomplete temporal information that “Mozart is older than Beethoven”. In Freka’s semi-interval calculus \[66\], a time interval \( t_1 \) is older than time interval \( t_2 \) if \( t_1 \) started before \( t_2 \). This semi-interval knowledge says nothing about the relationship between the endings of the two time intervals. Thus, the high level semi-interval relation older than can potentially be refined to one of the following interval relations: before, meets, overlaps, finished by, or contains. The disjunction of relations is implemented with the constraint

\[
(t_1, t_2) \in \text{before}^+ \lor \ldots \lor (t_1, t_2) \in \text{contains}^+ \rightarrow (t_1, t_2) \in \text{olderThan}^+.
\]
5.4 Adapting UML Sequence Diagrams to Visualise QSTR Application Behaviour

UML sequence diagrams visualise the exchange of messages between classifier roles, such as classes, and across association roles, such as dependencies [141]. The link between fragments and OO classes is adopted from the previous section, and thus the analogue to UML sequence diagrams in the QSTR domain is the flow of inferred information between fragments of a model.

The main focus of sequence diagrams is to explicitly order events. However, QSTR tasks typically apply to groups of fragments, and information flows can potentially occur between all pairs of fragments within a group. Moreover, the ordering of the flow of information between groups of fragments is often irrelevant. Thus, the critical problem in adapting sequence diagrams to QSTR applications is managing groups of fragments in the diagram. Although sequence diagrams have some notational features that support these cases such as coregions and the parallel combined fragment [178] they are extremely cumbersome to specify for QSTR applications, especially during the early application design stage when the particular fragments in a given group are likely to change.

To address this, a simple notational device will now be presented for specifying easily modifiable groups of fragments: groups are defined by specifying multiple fragments in a classifier as illustrated in Figure 5.1. It must be noted that this is inconsistent with standard sequence diagram notation as only one class type is allowed to be specified for each classifier.\(^4\) The semantics of multiple fragments in a single classifier are that incoming and outgoing tasks apply to all fragments in the group, and reflexive tasks apply between all pairs of fragments within the group, as illustrated in Figure 5.2. Square brackets on the lifelines indicate that tasks can occur in any order. Messages within the loop frame continue to be sent and received until deductive closure is achieved.

Table 5.5 summarises the adaptation of concepts from standard sequence diagrams to the QSTR domain.\(^5\) In QSTR sequence diagrams, fragments are classifiers and labels represent a subset of tuples that have particular conditions with respect to the relations in the fragment.\(^6\) For example, given a tuple labelled room\(_{\text{obj}}\), the label furniture\(_{\text{obj}}\) in the proxim-

---

\(^4\)If the designer needs to use standard UML software tools for creating sequence diagrams then there are many methods for representing collections of fragments that are consistent with standard sequence diagram notation. For example, the designer can create a descriptive classifier name such as low level spatial fragments, and then specify the particular fragments using UML notes. Alternatively, the collection of fragments could be placed into a package which the classifier can refer to. The approach specified in this thesis is the most direct and flexible.

\(^5\)The same adaptations apply to collaboration diagram notation, i.e. classes represent groups of fragments and messages represent tasks.

\(^6\)The purpose of sequence diagrams is to assist in the design process rather than formally specify a design, and therefore the designer should never express condition details that obscure the main design aspect being illustrated. An informal condition description is likely to be more appropriate, such as “furniture is in the room”, rather than a complete list of detailed formal first order logic expressions.
Chapter 5. Organising and Visualising QSTR Applications

Figure 5.1: Example adapting sequence diagram notation to allow classifiers to represent multiple fragments, and to allow objects to appear in multiple classifiers.

Figure 5.2: Using standard sequence diagram notation to illustrate semantics of the leftmost multi-fragment classifier in Figure 5.1.
<table>
<thead>
<tr>
<th>Standard Sequence Diagram Concept</th>
<th>Analogous QSTR Concept and Key Differences to Standard Sequence Diagrams</th>
<th>UML Symbol Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Classifier</strong>. Roles of particular objects in the interaction such as actors, class instances and so on.</td>
<td>Roles of sets of objects, either (a) actors, or (b) objects in one or more fragments. The same set of objects can appear in multiple classifiers, and a single classifier can represent multiple fragments. Conditions are used to specify the object subset.</td>
<td>![UML Symbol Example]</td>
</tr>
<tr>
<td><strong>Lifeline</strong>. Represents object existence and activation.</td>
<td>Existence means that a set of objects exists in a scenario (objects may enter and leave a scenario). Activation means that the set of objects is being reasoned about with respect to the particular fragment classifier.</td>
<td>![UML Symbol Example]</td>
</tr>
<tr>
<td><strong>Message</strong>. Flow of control and information, such as method calls, signals, object creation, and so on.</td>
<td>Messages are tasks representing the logical flow of information (rather than necessarily strictly ordered), such as envision, query, consistency check and so on.</td>
<td>![UML Symbol Example]</td>
</tr>
<tr>
<td><strong>Destruction</strong>. Object is removed from memory.</td>
<td>Set of objects is removed from the scenario. E.g. modelling objects leaving a scenario, such as people walking out of a room. E.g. system infers that particular objects can not exist and must be removed to make the scenario consistent, such as reasoning that certain events can not have taken place during an investigation of a crime scenario.</td>
<td>![UML Symbol Example]</td>
</tr>
</tbody>
</table>

**Table 5.5**: Adapting standard UML sequence diagram notation and concepts [141] to the QSTR application domain.

An entity fragment is the set of tuples such that the relations furniture and in room_obj hold, that is, furniture_obj∈furniture+ and (furniture_obj,room_obj)∈in+.

Lifelines indicated by vertical dashed lines represent the period when a set of objects is present in the scenario, and when they are involved in tasks, indicated by vertical solid double lines. Sequence diagram messages indicated by arrows represent QSTR tasks. Object destruction indicated by a cross at the end of a lifeline represents an object being removed from the scenario. The ordering of messages between particular fragments can be visualised using standard sequence diagram notation [141], as illustrated in Figure 5.3.

Because QSTR systems are rule based, the developer will typically have limited control over the actual sequence that information is propagated. Therefore, QSTR sequence diagrams are an effective and useful tool for specifying and communicating the logical sequence of information propagation rather than the actual sequence.
Chapter 5. Organising and Visualising QSTR Applications

1. user provides premise information
2. proximity infers missing relations
3. proximity propagates information to subjective impressions
4. user queries subjective impressions

Figure 5.3: Example using standard sequence diagram notation for classifiers with single fragments.

5.5 Summary

This chapter presented methodologies that directly support developers in organising and designing QSTR application relations and the associated relation theories. The primary contribution of this chapter has been to establish a mapping between object oriented (OO) domain concepts (such as classes, object attributes, object operations and class relationships) and the central concepts in QSTR applications. This enables software engineers to readily employ the plethora of popular OO design tools such as the Unified Modelling Language class diagrams, thus making QSTR application development significantly more accessible to software engineers with a background in OO design.

The central insight that forms the basis of the mapping is that the QSTR application analogue of OO classes is a QSTR fragment. A fragment is defined in this chapter as a cohesive, modular collection of relations and the corresponding relation theories (or constraints) that specify relation conversion, composition, and conceptual neighbours. QSTR calculi such as Allen’s Interval Algebra or Guesgen’s orientation calculus (refer to Section 2.2) are fragments.

Furthermore, it has been shown that organising QSTR applications into fragments facilitates application analysis. Simple fragment metrics such as the number of fragments and the number of relations per fragment have been shown to be effective in characterising the QSTR case study applications, by providing a heuristic measure of overall complexity and by highlighting salient operational and organisational properties. For example, it was observed that in the TreeSap case study application the more abstract, higher level relations are dynamically defined by users during querying, whereas in QtvLight the more abstract relations of subjective impression are built into the design of the application.
It has been demonstrated in this chapter that applying certain UML diagrams directly to QSTR applications is not practical, particularly UML diagrams that concern the flow of information between software components such as UML sequence diagrams. This is due to the imperative nature of OO software as opposed to the more declarative, rule-based nature of QSTR applications. UML sequence diagrams have been adapted by presenting a simple notational device that allows a developer to specify easily modifiable groups of fragments. The adapted sequence diagrams emphasise the flow of information between groups of fragments and abstract from the ordering of information flow within a group of fragments.

Fragment analysis of the case study applications has highlighted the very hierarchical organisational structure of QSTR applications; QSTR applications consist of low level fragments that represent and reason about relatively concrete spatial and temporal scenario concepts such as QSTR calculi, and high level fragments that encode more abstract and typically highly application-specific concepts such as subjective impressions or image categories. The following chapter presents a methodology that enables the selection of the most effective QSTR calculi in order to facilitate the design and development of low level fragments. Chapter 7 subsequently presents a variety of methodologies that support QSTR application developers in the organisation and design of high level fragments, motivated by machine learning, OO cohesion and coupling, and QSTR conceptual neighbourhoods.
Chapter 6

Designing Low Level Application Fragments: Selecting QSTR Calculi

6.1 Introduction

The previous chapter introduced the concept of QSTR fragments for organising QSTR applications, and highlighted the distinction between low level fragments, which include QSTR calculi, and high level fragments. A fundamental aspect of all QSTR applications is the utilisation of QSTR calculi as low level fragments for representing and reasoning about spatial configurations of objects and temporal configurations of events. Specifically, in Section 3.2 QSTR applications were defined as systems that identify and respond to specific patterns in qualitatively described scenarios that are provided by QSTR calculi. It is therefore essential that the developer is equipped with methodologies that enable the selection of the most effective QSTR calculi with respect to the given application task requirements, as defined in Chapter 4, for modelling spatial and temporal scenarios.

Section 4.2 identified crucial ontological factors, formally defined as the external interface requirements, that are pertinent for the selection of QSTR calculi. As presented in Section 1.3, a number of research results from the QSTR community can be employed for supporting the developer in addressing the external interface requirements of the application. Specifically, in [80] Hahmann and Gruninger investigate whether QSTR calculi that focus on topological relations meet the ontological requirements of applications in general. Ligozat [111] and Freksa [68] have identified alternative models of space, the results of which can be used to facilitate the selection of the most ontologically effective topology- and orientation-based QSTR calculi. Moreover, in [5] Ligozat studies a family of QSTR calculi that are interpreted on Cartesian products of linear orderings. Finally, in [15] Renz and Schmid investigate the impact that QSTR calculus customisation has on the properties of reasoning.

However, despite valuable research being available that can assist in addressing external interface requirements, relatively little research in QSTR has focused directly on other extremely
challenging issues faced by software developers that are interested in selecting and employing specific QSTR calculi. Firstly, research is required to identify the functional limitations of an application based on a specific QSTR calculus. This requires a methodology for determining the classes of scenarios, described using a given QSTR calculus, for which a qualitative application is guaranteed to produce erroneous output with respect to the functional requirements (as defined in Section 3.2 and 4.3). Secondly, the developer requires a methodology to determine how the application’s efficacy in meeting functional requirements will vary if an alternative QSTR calculus is used.

In Section 4.3 a methodology was established that enables QSTR application developers to specify functional requirements by defining an idealised application. Idealised applications specify how a QSTR application would be required to behave if it were to receive perfect information about the scenario. As formalised in Definition 4.3.1, idealised applications are defined using a sufficiently expressive interpretation language $C$. Each pattern that an application is required to respond to by producing output $i \in O$ is formalised by the developer as a corresponding idealised theory $\Theta^C_i$.

In this chapter, mathematical tools are presented that enable a developer to efficiently evaluate the functional efficacy of myriad QSTR calculi. Having defined an idealised application $\theta^C$ in the interpretation language $C$, the developer can select a number of QSTR calculi $A_1, \ldots, A_n$ and analyse the performance of the corresponding applications $\theta^{A_1}, \ldots, \theta^{A_n}$ against the idealised application $\theta^C$ using the mathematical analysis tools presented in this chapter. Finally, the chosen $\theta^{A_i}$ application is generated using an algorithm presented in Section 6.7 as an approximation of $\theta^C$, i.e. the strongest theory $\Theta^A_i$ for each $\Theta^C_i$ (up to 3-consistency, as discussed in Section 6.7).

**Definition 6.1.1.** Theory $\Theta^A_i$ is the strongest $A$ theory with respect to theory $\Theta^C_i$ iff there is no other set $\Theta'^A$ of sentences in $A$ such that $\text{Mod}(\Theta^C_i) \subseteq \text{Mod}(\Theta'^A) \subset \text{Mod}(\Theta^A_i)$.

To summarise,

1. The developer defines the idealised application $\theta^C$ based on formal software requirements (Section 4.3).

2. Theoretical analysis tools presented in this chapter are used to identify classes of scenarios where the output of the corresponding QSTR applications $\theta^{A_1}, \ldots, \theta^{A_n}$ is guaranteed to be incorrect.

3. An algorithm presented in Section 6.7 automatically generates the chosen qualitative application $\theta^{A_i}$.

For brevity, in this chapter the variable arguments of relations will be specified as a subscript, $R^A(x_1, \ldots, x_n) \leftrightarrow R^A_{x_1 \ldots x_n}$. Two illustrative examples will now be presented. The following mathematical terminology is used to present the results in this section: the main results are formally
presented as theorems; partial results that build up to theorems are formally presented as propositions; properties that are important but relatively obvious are formally presented as lemmas.

**Example 6.1.2.** Let \( \Theta_C \) be a theory of intervals with rational endpoints, and let \( \Theta_C^\alpha \) (the theory for producing output \( \alpha \)) consist of the sentence \( y^- - x^+ > 5 \land y^- - z^- = 2 \land y^+ - z^+ = 2 \). Figure 6.1 illustrates a scenario that satisfies this expression. The corresponding \( IA \) sentence \( t_{IA}^\alpha \in \Theta_{IA}^\alpha \) would be \( before (x, y) \land overlappedBy (y, z) \). Applying the composition table for \( before (x, y) \) and \( overlappedBy (y, z) \) gives \( before, meets, overlaps, starts, during \) for \( (x, z) \). However, it can easily be inferred that \( z^- > x^+ \), that is, the sole correct relation between \( x \) and \( z \) is \( before \), giving us the strongest sentence \( t_{IA}^\alpha \) for generating output \( \alpha \). The primary task is determining the classes of scenarios for which \( \Theta_{IA}^\alpha \) erroneously produces \( \alpha \).

In Example 6.1.2 the problem of generating the strongest \( IA \) theory for output \( \alpha \) can be solved by reasoning about the interpretation language and creating more expressions (e.g. creating an interval between \( x \) and \( z \)). However, the primary objective of QSTR is to move away from reasoning about complex interpretation languages such as metric spaces and instead reason on a more abstract qualitative level. Furthermore, the situation is more complex when weak composition does not equal composition (refer to Section 2.6) in the QSTR calculus as shown in the next example.

**Example 6.1.3.** To use a well known example for regions with holes [107] illustrated in Figure 6.2, let \( \Theta_C \) be a theory of regular closed spatial regions (where \( i(x) \) is the interior of \( x \)), and let \( \Theta_C^\beta \) consist of a single sentence that is a conjunction of the following expressions:

- \( w = x \cup y \)
- \( x \subset w \land x \not\subset i(w) \)
- \( y \subset i(w) \)
- \( i(x) \cap i(y) = \emptyset \land x \cap y \neq \emptyset \)
- \( i(y) \cap i(z) = \emptyset \land y \cap z \neq \emptyset \)
Figure 6.2: Scenario for which the idealised application produces output $\beta$. Region $z$ is necessarily connected to $x$.

After using the map $\varphi^{RCC8}$ on each atomic expression we get the following $RCC8$ expressions:

- $(\text{NTPP} (x, w) \lor \text{TPP} (x, w) \lor \text{EQ} (x, w) \lor \text{NTPP} (y, w) \lor \text{TPP} (y, w) \lor \text{EQ} (y, w))$
- $\text{TPP} (x, w)$
- $\text{NTPP} (y, w)$
- $\text{EC} (x, y)$
- $\text{EC} (y, z)$

Applying the composition table for $\text{EC} (x, y)$ and $\text{EC} (y, z)$ gives $\text{DC}$, $\text{EC}$, $\text{PO}$, $\text{TPP}$, $\text{TPPi}$, $\text{EQ}$ although the relations $\text{DC} (x, z)$ and $\text{EC} (x, z)$ can never occur. Thus, the strongest qualitative theory that is still complete for $\Theta_C^\beta$ would exclude these relations between $x$ and $z$. Again, the primary task is determining the classes of scenarios for which $\Theta_C^{RCC8}$ erroneously produces $\beta$.

Example 6.1.4. Nokel [126] (page 46, Figure 23) presents this example to illustrate the logical limitations of using binary relations; this example was also used in Section 3.2 to highlight the expressiveness of QSTR applications. The scenario consists of three valves attached to a tank such that exactly one valve must be open at any time. Formally, let $\Theta_C^\gamma$ be a theory of intervals with rational endpoints, let output symbol $\gamma$ indicate abnormal valve operation, and let $\Theta_C^\gamma$ consist of two sentences where $x, y, z$ are the intervals when valves are open:

- $y^- \in [x^-, x^+] \lor y^+ \in (x^-, x^+]$ (y intersects a semi-open interval bound by x),
- $x^+ < z^- \land \neg \exists y \cdot y^+ = z^-$ (all valves are closed after x and before z).

As shown in Example 3.2.6, the corresponding Interval Algebra sentences are

- overlaps $(x, y) \lor \text{starts} (x, y) \lor \text{during} (x, y) \lor \text{finishes} (x, y)$,
- $\text{before} (x, z) \land \neg \exists y \cdot \text{meets} (y, z)$.

The qualitative application will detect abnormal behaviour in any temporally consistent schedule of open valves.

\footnote{In general, an interval $i$ consists of a startpoint $i^-$ and an endpoint $i^+$.}
6.2 Defining Functional Consistency

For each output \( i \in O \), the task of determining functional consistency is to identify scenario classes for which \( \theta^A \) erroneously produces \( i \) (if any). Let \( \varphi^A \) map scenario \( s \), an expression in the interpretation language of \( \Theta^C \), to the corresponding scenario in \( A \) (the inverse map is denoted as \( (\varphi^A)^{-1} \)). The interpretation language, by design, is always more expressive than the qualitative language, therefore

\[
\varphi^A (\varphi^A)^{-1} (R^A) \rightarrow R^A.
\]

Conversely, QSTR calculi are defined as abstractions of the interpretation language, hence

- if the mapping \( \varphi^A \) is lossless then \( R^C \rightarrow (\varphi^A)^{-1} \varphi^A (R^C) \), otherwise
- the mapping is lossy, \( R^C \leftarrow (\varphi^A)^{-1} \varphi^A (R^C) \).

**Definition 6.2.1.** A QSTR application \( \theta^A \) is functionally consistent iff it produces the correct output for all input scenarios: for all \( s \in \text{Int}(\Theta^C) \), \( \theta^C (s) = \theta^A (\varphi^A (s)) \). The application is functionally inconsistent for some scenario \( s \in \text{Int}(\Theta^C) \) if it erroneously produces output given \( s \), \( \theta^C (s) \neq \theta^A (\varphi^A (s)) \).

Given the idealised theory \( \Theta^C_i \) that defines the classes of scenarios for which the idealised application produces output \( i \), and a QSTR calculus \( A \), let \( \Theta^A_i \) be the strongest possible \( A \) theory, \( \text{Mod}(\Theta^C_i) \subseteq \text{Mod}(\Theta^A_i) \). Determining functional consistency entails determining the classes of scenarios \( \text{Mod}(\Theta^A_i) \setminus \text{Mod}(\Theta^C_i) \), by specifying a set of expressions in the interpretation language \( C \). In this chapter it is proved that functional consistency can be determined by using reference information about QSTR calculi, which is readily available from research in the QSTR community, such as the map \( \varphi^A \) and properties of weak composition.

The cases that need to be considered for determining functional consistency will now be explicitly presented. The cases that are relevant to functional consistency are derived from different combinations of the following four conditions:

- a given scenario \( s \) is consistent in the domain of interpretation, \( s \in \text{Mod}(\Theta^C) \);
- given calculus \( A \) the corresponding qualitative scenario \( \varphi^A (s) \) is determined to be consistent, \( \varphi^A (s) \in \text{Mod}(\Theta^A) \);
- the idealised application \( \theta^C \) produces output \( i \) in the scenario, \( s \in \text{Mod}(\Theta^C_i) \);
- the strongest \( A \) application \( \theta^A \) produces \( i \), \( s \in \text{Mod}(\Theta^A_i) \).

The above four conditions can produce \( 2^4 = 16 \) different possible cases. However, the following properties apply to the conditions based on the properties of QSTR calculi and QSTR applications:
Chapter 6. Designing Low Level Application Fragments: Selecting QSTR Calculi

<table>
<thead>
<tr>
<th>Cases given scenario s</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario is consistent: $s \in \text{Mod}({\theta}^{c})$</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>QSTR calculus determines consistency: $q^{(4)}(s) \in \text{Mod}({\theta}^{q})$</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Idealised application produces output $\alpha$: $s \in \text{Mod}({\theta}^{a}_{c})$</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>QSTR application produces output $\alpha$: $q^{(4)}(s) \in \text{Mod}({\theta}^{q}_{a})$</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.1: Six cases, defined as allowable combinations of four conditions.

- idealised application output is only specified for consistent scenarios;\(^{2}\)
- QSTR application output is only specified for QSTR consistent scenarios;
- QSTR calculi never erroneously decide that a scenario is inconsistent (i.e. QSTR calculi and sound but not complete, refer to Section 2.2);
- QSTR applications never erroneously omit output (i.e. QSTR applications are complete but not sound; this is discussed further in Section 12.2).

These properties only allow six cases out of the possible sixteen. Table 6.1 presents the six key cases that need to be considered for determining functional consistency. In cases 1 to 4 the qualitative application is functionally consistent, and in cases 5 and 6 it is functionally inconsistent. In cases 1,2,3 and 6 the qualitative calculus correctly decides whether or not the scenario is consistent, and in cases 4 and 5 the qualitative calculus incorrectly decides that the scenario is consistent. The following two sections derive mathematical tools that determine functional consistency for scenarios that are guaranteed to be consistent (case 6 in Table 6.1). Section 6.5 presents tools for determining functional consistency for both consistent and inconsistent scenarios. This approach separates the tasks of determining scenario consistency, which is handled by the calculus, and determining functional consistency, which is handled by the mathematical tools developed in this chapter.

### 6.3 Determining Functional Consistency in Consistent Unambiguous Scenarios

This section derives functional consistency expressions for consistent unambiguous scenarios; in unambiguous scenarios every pair of objects has exactly one qualitative relation.\(^{3}\) From

\(^{2}\)This is due to a completeness property of QSTR applications, in the sense of guaranteeing that if the idealised application produces output $\alpha$ for scenario $s$ then the QSTR application will also produce output $\alpha$. This is discussed further in Section 12.2.

\(^{3}\)The corresponding constraint networks are called atomic.
an application-level perspective these scenarios are completely observable up to the precision admitted by the relations of a given calculus.

Firstly atomic expressions are considered. In general, if some atomic expression $e^A \in t^A$ does not completely match the corresponding expression $e^C \in t^C$ then there must exist some tuple that is a model of $e^A$ and not $e^C$. In these scenarios $\Theta^A$ will give the incorrect output. It is straightforward to check if an atomic expression $e^C$ is equivalent to the corresponding $e^A$, i.e. for all $x, y$, $\varphi^A (R^C_{xy}) \rightarrow R^C_{xy}$, unless $R^C_{xy}$ is equivalent to the boundary conditions of the mapped $R^A$, then there is some scenario that is a model of $R^A_{xy}$ and not $R^C_{xy}$.

**Lemma 6.3.1.** An atomic expression $e^A = \varphi^A (e^C)$ is functionally consistent with $e^C$ for all scenarios in $\text{Mod} (\Theta^C)$ iff the QSTR calculus mapping is lossless.

**Proof.** If mapping $\varphi^A$ is lossy then there is some scenario that is a model of $R^A_{xy}$ and not $R^C_{xy}$. By definition $R^C_{xy} \rightarrow \varphi^A (R^C_{xy})$, thus if a mapping is lossy then there exists $x, y$ such that $\varphi^A (R^C_{xy}) \land \neg R^C_{xy}$. Moreover, all tuples in $\varphi^A (R^C_{xy})$ must be consistent with $\Theta^C$ for scenarios with two objects otherwise the qualitative relation is trivially inconsistent. Thus, there must be some consistent scenario with exactly two objects $x, y$ such that $\varphi^A (R^C_{xy}) \rightarrow R^C_{xy}$ which is the definition of functional inconsistency (refer to Definition 6.2.1).

The class of invalid scenarios is found by taking the complement of the interpretation language relation,

$$\neg R^C_{xy} \land (\varphi^A)^{-1} \varphi^A (R^C_{xy}) \quad (6.1)$$

Determining whether composed expressions are functionally consistent is not as straightforward. If $R^C \circ S^C = T^C$ then the functionally inconsistent scenarios satisfy $\neg T^C_{xy} \land (\varphi^A)^{-1} \varphi^A (T^C_{xy})$. However, this cannot be checked directly because there is no atomic expression in $t^C$ that corresponds to the composed expression $T^C$. Firstly it is observed that whenever $x, y, z$ is a model of $\Theta^C$ such that $R^A_{xy}$ and $S^A_{yz}$ then $T^A_{xz}$ is a valid possibility by weak composition. Alternatively, it must be checked whether $T^A$ admits any scenarios that satisfy $R^A_{xy}$, $S^A_{yz}$, $T^A_{xz}$ which are not actually models of $\Theta^C$. It must be noted that, in the following propositions, the expression $T^A_{xz} = \cap_{x, y, z} R^A_{xy} \circ S^A_{yz}$ is a set-theoretic form of Allen’s algebraic closure condition which is repeated for all variable pairs $x, z$ until a fixpoint is reached (refer to the last paragraph in Section 2.2).

**Proposition 6.3.2.** Given variables $x, z$, if all possible atomic arguments $R^A_{xy} = \varphi^A (R^C_{xy})$, $S^A_{yz} = \varphi^A (S^C_{yz})$ are lossless in sentence $t^C$ then $T^A_{xz}$ (in the strongest corresponding A sentence $t^A$) is equivalent to QSTR composition, $T^A_{xz} = \cap_{x, y, z} R^A_{xy} \circ S^A_{yz}$.

**Proof.** From qualitative composition, $\varphi^A (R^C) \circ \varphi^A (S^C) = T^A$, and because we have the strongest $A$ sentence then $T^A \subseteq T^A$ (i.e. some composed qualitative relations may have been removed, see
A relation $T^A$ is removed from $T^A$ if, for all $x,y,z$, $R^A_{xy} \land S^C_{yz} \land \neg T^A_{xz}$. The map $\varphi^A$ is lossless for $R^C$ and $S^C$ so a relation $T^A$ is removed from $T^A$ if, for all $x,y,z$, $\varphi^A(R^C_{xy}) \land \varphi^A(S^C_{yz}) \land \neg T^A_{xz}$. But this contradicts the composition, i.e. no such $T^A$ exists. Therefore, no relations are removed and $T^A = T^A$.

Using Lemma 6.3.1 and Proposition 6.3.2 it can be proved that if output axioms in $t^C_i$ correspond to the composition table, then composition table reasoning is guaranteed to produce the correct application output (in atomic consistent networks). To illustrate the idea of the proof, consider the following example.

**Example 6.3.3.** Let $\Theta^C$ be a theory of regular closed spatial regions, and let $t^C_i$ consist of two expressions $R^C, S^C$ that are lossless for $RCC8$, $\varphi^{RCC8}(R^C_{xz}) \leftrightarrow EC_{xz}$ and $\varphi^{RCC8}(S^C_{yz}) \leftrightarrow EC_{yz}$. The composed expression $T^C_{RCC8} = \ldots \lor EC_{xy} \lor \ldots$ is lossy because of weak composition; given a scenario where $y$ fills a hole of $x$ (satisfying $T^C_{RCC8}$), there is no $z$ that is externally connected to both regions (i.e. for all $z$, $EC_{xz} \leftrightarrow EC_{yz}$; refer to Figure 6.2 above), so $T^C_{RCC8}$ admits tuples that are not models of $\Theta^C$ and $t^C_i$. Thus, if an unambiguous scenario satisfies $t^C_{RCC8}$ and not $t^C_{EC}$ then it must be due to the deficiencies of weak composition. But then those scenarios are not consistent, and it has been assumed that all scenarios are consistent for the present argument; i.e. case 6 in Table 6.1 is currently being addressed. Therefore, if the unambiguous scenario is guaranteed to be consistent then $t^C_{RCC8}$ is functionally consistent.

**Proposition 6.3.4.** Given sentence $t^A$, if all possible atomic arguments $R^A$, $S^A$ are functionally consistent with respect to $t^C_i$ then composition $T^A = R^A \triangleleft S^A$ is also functionally consistent for consistent, unambiguous scenarios.

**Proof.** All scenarios are assumed to be consistent. Therefore, it needs to be proved that the only unambiguous scenarios that are models of $t^A$ and not $t^C_i$ are inconsistent and thus will not occur. That is, it needs to be proved that if $x,y,z$ is a model of $t^A$ such that $T^A_{xz} = T^A_{xz} \lor \ldots \lor T^A_{xz}$ is true in $t^A_i$, and there is no $y$ such that $R^C_{xy} \lor S^C_{yz}$ is true in $t^C_i$, then $x,z$ is not a model of the interpretation language $\Theta^C$ i.e. it is an inconsistent scenario.

Because the arguments are functionally consistent, then by Lemma 6.3.1 all pairs of arguments are lossless. Thus, by Proposition 6.3.2 $T^A$ corresponds to QSTR composition. If weak composition is equal to composition then $\varphi^A(R^C) \circ \varphi^A(S^C) = T^A$ is a theorem of $\Theta^A$. Thus $T^A_{xz} \land \neg (R^C_{xy} \land S^C_{yz})$ is not a model of $\Theta^A$ (instead this corresponds to case 3 in Table 6.1) proving the proposition for calculi with strong composition. If weak composition does not equal composition then $\varphi^A(R^C) \circ \varphi^A(S^C) \subset T^A$ is a theorem of $\Theta^A$, but exactly those scenarios where $T^A_{xz} \land \neg (R^C_{xy} \land S^C_{yz})$ are inconsistent with $\Theta^C$, meaning that $T^A$ is only functionally inconsistent for inconsistent scenarios (giving case 5 in Table 6.1). Thus, $T^A$ is functionally consistent if the unambiguous scenario is consistent. □
Proposition 6.3.4 shows that if arguments \( R^A \) and \( S^A \) are lossless, then composition \( T^A \) is functionally consistent. An expression can now be derived, based on Equation 6.1, that specifies an approximation\(^4\) of the class of functionally inconsistent scenarios for binary composition by identifying the scenarios where either of the arguments \( R_C^1 \), \( R_C^2 \) are lossy,

\[
\bigvee_{i=1,2} \left[ \neg R_C^i (x_i, x_{i+1}) \land (\varphi^A)^{-1} \varphi^A (R_C^i (x_i, x_{i+1})) \right].
\]

The following proposition states that this equation is only an approximation.

**Proposition 6.3.5.** If arguments are lossy then composition is not necessarily functionally inconsistent.

**Proof.** To prove this proposition it must be shown that \( R_C \subset (\varphi^A)^{-1} \varphi^A (R_C) \) (i.e. the mapping is lossy) and that \( R_C \circ S_C = (\varphi^A)^{-1} \varphi^A (R_C) \circ S_C \) (i.e. composition is functionally consistent). This is shown with a simple example. Let \( R_C = \{ (a, b) \} \), \( S_C = \{ (b, c) \} \), \( (\varphi^A)^{-1} \varphi^A (R_C) = \{ (a, b), (a', b') \} \). Then \( R_C \circ S_C = (\varphi^A)^{-1} \varphi^A (R_C) \circ S_C = \{ (a, c) \} \).

The following theorem enables a much better approximation to be derived for specifying the class of scenarios where \( T^A \) is functionally inconsistent.

**Theorem 6.3.6.** A composed expression \( T^A \) is functionally inconsistent (for consistent unambiguous scenarios) iff the arguments \( R_C^1, \ldots, R_C^n \) are lossy such that tuples are introduced that can compose with tuples in other arguments.

**Proof.** Follows from Propositions 6.3.4 and 6.3.5. □

For \( n \)-length composition chains, the expression that more closely approximates the class of functionally inconsistent scenarios is

\[
\bigvee_{i=1 \ldots n} \left[ \neg R_C^i (x_i, x_{i+1}) \land \left( \bigwedge_{j=1 \ldots n} (\varphi^A)^{-1} \varphi^A (R_C^j (x_j, x_{j+1})) \right) \right]. \tag{6.2}
\]

The intersection over the sequence of \( j \) ensures that a composition chain satisfies the qualitative relations; the negated relation is a missing link in the composition chain with respect to the interpretation language, hence causing functional inconsistency. The disjunction over the sequence \( i \) reflects Proposition 6.3.4.

Equation 6.2 still only approximates the precise set of all functionally inconsistent scenarios; the precise equation must also check for all possible combinations of \( k \) missing links, for \( 1 < k < n \). For example, it may be the case that two chains exist (in the following let \( R_i^A = (\varphi^A)^{-1} \varphi^A (R_C^i) \)),

\(^4\)That is, the approximation contains all scenarios for which the QSTR application will be functionally inconsistent.
\[ -R_2^C \land (R_1^A \land R_2^A \land R_3^A) \]
\[ -R_3^C \land (R_1^A \land R_2^A \land R_3^A) \]

while there is no chain such that \(-R_2^C \land -R_3^C \land (R_1^A \land R_2^A \land R_3^A)\). Further research is currently being conducted to determine whether this level of precision is useful to a developer and practically feasible to report.

Finally, if two variables \(x, y\) are constrained indirectly then some sequence of atomic expressions must exist such that \(R_{C_{xi}}, \ldots, R_{C_{jy}}\) ([120] Section 3). Thus, functional inconsistency is defined by the loss in atomic expressions given by Equation 6.1, and \(n\)-length composition chain expressions given by Equation 6.2 for \(n = 2 \ldots m\) where \(m\) is the longest acyclic path.

### 6.4 Determining Functional Consistency in Consistent Ambiguous Scenarios

This section considers the case where multiple qualitative relations can be assigned to pairs of objects. That is, the application may be presented with an ambiguous scenario description because the scenario is only partially observable. When a pair of objects is assigned to multiple qualitative relations it means that each relation is a possibility based on the current available information about the scenario. Unless otherwise specified by the developer, an application should produce output \(i\) if some combination of the possible relations is a model of the theory \(t^A_i\) (i.e. the ambiguous scenario is consistent with \(t^A_i\)). When weak composition equals composition the case is equivalent to the one described in the previous section.

**Proposition 6.4.1.** Given sentence \(t^A_i\), if all possible atomic arguments \(R^A, S^A\) are functionally consistent with respect to \(t^C_i\) and weak composition equals composition then composition \(T^A\) is also functionally consistent (for consistent ambiguous scenarios).

**Proof.** See the proof of Proposition 6.3.4. In particular, \(\phi^A(R^C) \circ \phi^A(S^C) = T^A\) is a theorem of \(\Theta^A\) and thus \(T^A_{xz} \land - (R^C_{xy} \land S^C_{yz})\) is not a model of \(\Theta^A\). \(\square\)

The above proof highlights that, in addition to the classes of scenarios specified in the previous section, an application will be functionally inconsistent when some combination of relations in the ambiguous scenario is a model of \(t^A_i\) when in fact the combination is not actually possible in \(\Theta^C\) (even though the scenario is consistent). When weak composition equals composition this cannot happen because \(T^A_{xz} \rightarrow \exists y \cdot R^C_{xy} \land S^C_{yz}\). To illustrate this, consider an extension to the application in Example 6.3.3 so that it produces \(\beta\) when three regions are externally connected.

---

\(^{5}\)The corresponding constraint networks are called non-atomic.
Let scenario $s$ consist of three regions $a, b, c$, where $b$ fills the hole of $a$, and $c$ is externally connected to $b$. A consistent partial description of the scenario is EC$(a, b)$ and EC$(b, c)$. Given this ambiguous description of $s$ RCC8 will erroneously infer that EC$(a, c)$ is a possibility causing the application to erroneously produce $\beta$, even though the scenario is consistent.

**Theorem 6.4.2.** If some atomic expression $e^A \in t^A_i$ is a non-extensional relation $T^A$ when $T^A \in \phi^A(R^C) \land \phi^A(S^C)$ then a class of ambiguous consistent scenarios for which $t^A_i$ is functionally inconsistent is $\phi^A(R^C_{xy}) \land \phi^A(S^C_{yz}) \land \neg (\phi^A)^{-1} T^A_{xz}$ such that $T^A \in \bigcap_{x} R^A_{xy} \land S^A_{yz} R^A \land S^A$.

**Proof.** Follows from Theorem 4.5 in [107].

The explicit non-extensional composition triads are given by the QSTR community, e.g. [107]. The triads can thus be directly referenced to provide the developer with a meaningful class of scenarios for which their $A$ application will be functionally inconsistent. Importantly, it must be observed that this not only identifies the scenarios for which an $A$ application is functionally inconsistent, but it also specifies the exact erroneous output.

## 6.5 Determining Functional Consistency in Inconsistent Scenarios

This section considers the case where the scenario descriptions presented to the application are inconsistent in a way that is undetectable to the QSTR calculus, i.e. case 5 in Table 6.1. This covers a large number of real world application situations where there are conflicting sources of information about a scenario, such as sensor errors or inconsistent human reports about a scene. From an application perspective it is not necessarily a problem if a scenario is undetectably inconsistent; the application may still produce the correct output when given such an inconsistent scenario, even though the application incorrectly assumes that the scenario is consistent. Unfortunately, it is not possible to predict which erroneous output will be given when scenarios are potentially inconsistent.

**Theorem 6.5.1.** Given an inconsistent scenario that is determined to be consistent by calculus $A$, whether an $A$ application is functional inconsistent for some output $i$ is independent of the theory for producing output $i$.

**Proof.** Let $s$ be an inconsistent scenario that is determined to be consistent by calculus $A$. Let $t^A_i$ be some sentence for producing output $i$. Let $s' = \phi^A(s) \land t^A_i$. Then $s'$ is undetectably inconsistent by $A$ and the application $\theta^A$ will erroneously produce output $i$.

Thus, regardless of the theory for producing $i$ some inconsistent scenario exists such that the QSTR application will erroneously produce $i$. Therefore, functional inconsistency is entirely dependent on the limitations of the calculus rather than the definition of the application.
Formally, given sentence $r^A_i$ for producing output $i$, the application will be functionally inconsistent for scenario $s$ if

$$s \notin \text{Mod} \left( \Theta^C \right) \land \varphi^A(s) \in \text{Mod} \left( \Theta^A \right) \land \varphi^A(s) \in \text{Mod} \left( r^A_i \right).$$

Despite this, the developer can still be informed about the classes of scenarios for which their application may produce erroneous output. Firstly the case of unambiguous scenarios will be addressed. Algebraic closure is sufficient for determining consistency in unambiguous scenarios using many important calculi, including $IA$ and $RCC8$. If the calculus has this property then case 5 will never arise with unambiguous scenarios, and the functionally inconsistent class of scenarios is defined by the equations given in Section 6.3. However, many calculi do not have this property [14], as presented in Section 2.6. Renz and Ligozat [14] have shown that algebraically closed but inconsistent scenarios can be constructed if a qualitative relation can be partitioned; in these scenarios a QSTR application may also be functionally inconsistent.

Determining whether an ambiguous scenario is consistent is in general an NP-hard problem for all QSTR calculi. However, some of the most prominent advances in QSTR research have been to identify maximal tractable subsets of a given calculus (e.g. this has been completely solved for $IA$ [125] and $RCC8$ [13]). The class of scenarios for which the application may be functionally inconsistent is precisely the intractable subsets of the calculus. Thus, these important theoretical results can be directly employed to inform a developer about the scenarios in which their application may produce erroneous output.

### 6.6 Functional Consistency Analysis Methodology

This section summarises the results from the previous sections and illustrates how functional consistency can be employed by the developer. The two major cases for functional consistency are when scenarios are consistent (case 6 in Table 6.1) and when scenarios are undetectably inconsistent by the QSTR calculus (case 5 in Table 6.1). In both of these cases scenarios can be either unambiguous or ambiguous. Table 6.2 specifies the classes of scenarios for which an application is functionally inconsistent with respect to these conditions; partitionable relations are defined in [14], and non-extensional triads and intractable subclasses are identified by the QSTR community, e.g. [107] and [13, 125].

Interestingly, functional consistency analysis can readily assist a developer in determining the required level of precision to achieve a given task. Specifically, let $\theta^A$ be an application that erroneously produces two mutually exclusive outputs $\alpha, \beta$ in some scenario $s$, $\theta^A(s) = \{\alpha, \beta\}$. The developer can refine the relevant classes of scenarios in the design of the application by defining a new output symbol $\gamma$ to represent ambiguous application states, $\alpha \land \beta \leftrightarrow \gamma$. This
6.6 Functional Consistency Analysis Methodology

Table 6.2: Determining functional consistency depending on whether the scenario is guaranteed to be consistent or the scenario is possibly inconsistent, and whether the scenario is unambiguous (the network is atomic) or ambiguous (the network is non-atomic).

<table>
<thead>
<tr>
<th>Case given a scenario</th>
<th>Scenario is unambiguous</th>
<th>Scenario is ambiguous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario is consistent</td>
<td>Equations 6.1, 6.2</td>
<td>Equations 6.1, 6.2 If weak composition ≠ composition then also non-extensional triads</td>
</tr>
<tr>
<td>Scenario is undetectably inconsistent</td>
<td>If algebraic closure ≠ path-consistency then partitionable relations</td>
<td>Intractable subclasses</td>
</tr>
</tbody>
</table>

provides a natural and efficient discretisation of the input space based on the expressiveness of a qualitative calculus, for example, as an alternative to defining fuzzy membership functions.

The three examples from the Introduction section of this chapter will now be revisited to illustrate the functional consistency methodology.

Example 6.6.1. The functionally inconsistent scenarios for the QSTR application that employs Interval Algebra (IA) in Example 6.1.2 are presented in Table 6.3. Firstly for consistent, unambiguous scenarios, Equation 6.1 is employed for each pair of variables directly constrained by an atomic expression in the interpretation language \( C \), namely \( (x, y) \) and \( (y, z) \). From the idealised application,

\[
\begin{align*}
R^C_{xy} &= y^--x^+ > 5, \\
R^C_{yz} &= (y^--z^- = 2) \land (y^+ - z^+ = 2).
\end{align*}
\]

From the IA map,

\[
(\phi^{IA})^{-1} \phi^{IA} \left( R^C_{xy} \right) = (\phi^{IA})^{-1} \left( \text{before}_{xy} \right) = y^- > x^+,
\]

\[
(\phi^{IA})^{-1} \phi^{IA} \left( R^C_{yz} \right) = (\phi^{IA})^{-1} \left( \text{overlappedBy}_{yz} \right) = y^- > z^- \land y^+ > z^+ \land z^+ > y^-.
\]

Substituting into Equation 6.1,
\neg R^C_{xy} \land (\varphi^{IA})^{-1} R^C_{xy} = (y^- - x^+ \leq 5 \land y^- > x^+) = 0 < y^- - x^+ \leq 5
\neg R^C_{yz} \land (\varphi^{IA})^{-1} R^C_{yz} = (y^- - z^- \neq 2) \land (y^+ - z^+ \neq 2)
\land y^- > z^- \land y^+ > z^+ \land z^+ > y^-.

The functional inconsistency of the composed expression for variables \((x, z)\) is given by Equation 6.2,

\neg T^C_{xz} \land (\varphi^{IA})^{-1} T^C_{xz} = [0 < y^- - x^+ \leq 5]
\lor [(y^- - z^- \neq 2) \land (y^+ - z^+ \neq 2)
\land y^- > z^- \land y^+ > z^+ \land z^+ > y^-].

Secondly for consistent, ambiguous scenarios, in IA weak composition is equivalent to strong composition [5, 104], and hence the functionally inconsistent scenarios are equivalent to the above class.

Thirdly for undetectably inconsistent, unambiguous scenarios, in IA atomic, algebraically closed networks are consistent [5, 158, 159] (refer to Section 2.6 in this thesis) and hence this case cannot occur.

Finally for inconsistent, ambiguous scenarios, the unique, maximal tractable subset of IA is the ORD-Horn class [125] (refer to Section 2.6.2 in this thesis); any scenarios that are outside of this class are potentially undetectably inconsistent, and therefore the application may also be functionally consistent in these scenarios.

**Example 6.6.2.** The functionally inconsistent scenarios for the QSTR application that employs Region Connection Calculus (RCC8) in Example 6.1.3 are presented in Table 6.4. For consistent, unabmiguous scenarios, Equation 6.1 is employed for each pair of variables directly constrained by an atomic expression in the interpretation language \(C\), namely \((x, y), (y, z), (x, w)\) and \((y, w)\). In the following \(\bot\) (falsum) means logically unsatisfiable. From the idealised application,
### Table 6.3: Classes of scenarios for which the QSTR application using IA will be functionally inconsistent.

<table>
<thead>
<tr>
<th>Case given a scenario</th>
<th>Description</th>
<th>Functionally Inconsistent Scenarios</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario is consistent, unambiguous</td>
<td>Equations 6.1, 6.2</td>
<td>0 ( &lt; (y^r - x^r) \leq 5 )</td>
</tr>
<tr>
<td>Scenario is consistent, ambiguous</td>
<td>Equations 6.1, 6.2 (weak composition = composition)</td>
<td>(as above)</td>
</tr>
<tr>
<td>Scenario is undetectably inconsistent, unambiguous</td>
<td>(algebraic closure = path-consistency)</td>
<td>(none)</td>
</tr>
<tr>
<td>Scenario is undetectably inconsistent, ambiguous</td>
<td>intractable subclasses</td>
<td>Satisfies ( \Theta^\text{IA}_u ) and is outside ORD-Horn subclasses</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
R_{xy}^C &= (w = x \cup y) \land (i(x) \cap (y) = 0) \land (x \cap y \neq 0), \\
R_{yz}^C &= (i(y) \cap (z) = 0) \land (y \cap z \neq 0), \\
R_{yw}^C &= (w = x \cup y) \land (x \subset w \land x \notin i(w)), \\
R_{yw}^C &= (w = x \cup y) \land (y \subset i(w)).
\end{align*}
\]

From the RCC8 map,

\[
\begin{align*}
(\varphi^{\text{RCC8}})^{-1} \varphi^{\text{RCC8}}(R_{xy}^C) &= (\varphi^{\text{RCC8}})^{-1}(\text{EC}_{xy}) \\
&= (i(x) \cap (y) = 0) \land (x \cap y \neq 0), \\
(\varphi^{\text{RCC8}})^{-1} \varphi^{\text{RCC8}}(R_{yz}^C) &= (\varphi^{\text{RCC8}})^{-1}(\text{EC}_{yz}) \\
&= (i(y) \cap (z) = 0) \land (y \cap z \neq 0), \\
(\varphi^{\text{RCC8}})^{-1} \varphi^{\text{RCC8}}(R_{yw}^C) &= (\varphi^{\text{RCC8}})^{-1}(\text{TPP}_{yw}) \\
&= x \subset w \land x \notin i(w), \\
(\varphi^{\text{RCC8}})^{-1} \varphi^{\text{RCC8}}(R_{yw}^C) &= (\varphi^{\text{RCC8}})^{-1}(\text{NTPP}_{yw}) \\
&= y \subset i(w).
\end{align*}
\]
Substituting into Equation 6.1,

\[ -R^C_{xy} \land (\varphi^{RCC8})^{-1} \varphi^{RCC8} (R^C_{xy}) = ((w \neq x \cup y) \lor (i(x) \cap i(y) \neq \emptyset) \lor (x \cap y = \emptyset)) \]
\[ \land (i(x) \cap i(y) = \emptyset) \land (x \cap y \neq \emptyset) \]
\[ = (w \neq x \cup y) \land (i(x) \cap i(y) = \emptyset) \land (x \cap y \neq \emptyset), \]

\[ -R^C_{yz} \land (\varphi^{RCC8})^{-1} \varphi^{RCC8} (R^C_{yz}) = ((i(y) \cap i(z) \neq \emptyset) \lor (y \cap z = \emptyset)) \]
\[ \land (i(y) \cap i(z) = \emptyset) \land (y \cap z \neq \emptyset), \]
\[ = \bot, \]

\[ -R^C_{yw} \land (\varphi^{RCC8})^{-1} \varphi^{RCC8} (R^C_{yw}) = ((w \neq x \cup y) \lor - (x \subset w) \lor x \subset i(w)) \]
\[ \land x \subset w \land x \not\subset i(w) \]
\[ = (w \neq x \cup y) \land x \subset w \land x \not\subset i(w), \]

\[ -R^C_{yw} \land (\varphi^{RCC8})^{-1} \varphi^{RCC8} (R^C_{yw}) = ((w \neq x \cup y) \lor - (y \subset i(w))) \]
\[ \land y \subset i(w) \]
\[ = (w \neq x \cup y) \land y \subset i(w). \]

The functional inconsistency of the 2-length composition chain expressions are given by Equation 6.2,

\[ -T^C_{xz} \land (\varphi^{RCC8})^{-1} \varphi^{RCC8} (T^C_{xz}) = (w \neq x \cup y) \land (i(x) \cap i(y) = \emptyset) \land (x \cap y \neq \emptyset), \]
\[ \lor \bot, \]

\[ -T^C_{wz} \land (\varphi^{RCC8})^{-1} \varphi^{RCC8} (T^C_{wz}) = (w \neq x \cup y) \land (y \subset i(w)) \]
\[ \lor y \subset i(w). \]

The functional inconsistency of the 3-length composition chain expressions are given by Equation 6.2,

\[ -T^C_{xz} \land (\varphi^{RCC8})^{-1} \varphi^{RCC8} (T^C_{xz}) = (w \neq x \cup y) \land x \subset w \land x \not\subset i(w) \]
\[ \lor (w \neq x \cup y) \land y \subset i(w) \]
\[ \lor \bot, \]

\[ -T^C_{wz} \land (\varphi^{RCC8})^{-1} \varphi^{RCC8} (T^C_{wz}) = (w \neq x \cup y) \land x \subset w \land x \not\subset i(w) \]
\[ \lor (w \neq x \cup y) \land (i(x) \cap i(y) = \emptyset) \land (x \cap y \neq \emptyset) \]
\[ \lor \bot. \]
Secondly for consistent, ambiguous scenarios, for RCC8 all non-extensional triads have been identified (and enumerated) by Li and Ying [107]; given a composed output expression \( e_{ij} \in \Theta_\beta \), a scenario \( s \) is functionally inconsistent if it satisfies the arguments of a non-extensional triad, \( R_{ik} \land S_{kj} \) such that the composed output expression \( e_{ij} \in R_{ik} \circ S_{kj} \) is neither explicitly specified in \( s \) nor actually satisfiable in \( s \), \( \neg e_{ij} \) (thus RCC8 will erroneously reason that \( e_{ij} \) is a possibility and the application will erroneously produce \( \beta \)).

Thirdly for undetectably inconsistent, unambiguous scenarios, in RCC8 atomic, algebraically closed networks are consistent [123] (refer to Section 2.6 in this thesis) and hence this case cannot occur.

Finally for inconsistent, ambiguous scenarios, the maximal tractable subsets of RCC8 are \( C_8, Q_8, \) and \( \hat{H}_8 \) [13]; any scenarios that are outside of this class are potentially undetectably inconsistent, and therefore the application may also be functionally consistent in these scenarios.

**Example 6.6.3.** The functionally inconsistent scenarios for the QSTR application that employs Interval Algebra (IA) in Example 6.1.4 are presented in Table 6.3. Firstly for consistent, unambiguous scenarios, Equation 6.1 is employed for each pair of variables directly constrained by an atomic expression in the interpretation language \( C \), namely \((x,y)\) and \((y,z)\). From the idealised application,

\[
R^C_{xz} = (x^+ < z^-),
\]
\[
R^C_{yz} = (y^+ = z^-).
\]

From the IA map,

\[
\left( \varphi^{IA} \right)^{-1} \varphi^{IA} (R^C_{xz}) = \left( \varphi^{IA} \right)^{-1} (\text{before}_{xz}) = (x^+ < z^-),
\]
\[
\left( \varphi^{IA} \right)^{-1} \varphi^{IA} (R^C_{yz}) = \left( \varphi^{IA} \right)^{-1} (\text{meets}_{yz}) = (y^+ = z^-).
\]

Substituting into Equation 6.1,
### Table 6.4: Classes of scenarios for which the QSTR application using RCC8 will be functionally inconsistent.

<table>
<thead>
<tr>
<th>Case given a scenario</th>
<th>Description</th>
<th>Functionally Inconsistent Scenarios</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scenario is consistent, unambiguous</strong></td>
<td>Equations 6.1, 6.2</td>
<td><img src="" alt="Diagram" /></td>
</tr>
</tbody>
</table>
| **Scenario is consistent, ambiguous** | Equations 6.1, 6.2 and non-extensional triads | As above, and if satisfies some composed expression $\Theta^{RCC8} \subseteq \Theta^{\text{RCC8}_\beta}$ with any of the following:  
- $DC_{yz} \land EC_{yz} \land (EC_{xz} \lor PO_{xz} \lor TPP_{xz})$  
- $DC_{yz} \land TPP_{yz} \land (EC_{xz} \lor PO_{xz} \lor TPP_{xz})$  
- $EC_{yz} \land DC_{yz} \land (EC_{xz} \lor PO_{xz} \lor TPP_{yz})$  
- (etc.) |
| **Scenario is undetectably inconsistent, unambiguous** | (algebraic closure = path-consistency) | (none) |
| **Scenario is undetectably inconsistent, ambiguous** | intractable subclasses | Satisfies $\Theta^{RCC8} \subseteq \Theta^{\text{RCC8}_\beta}$ and is outside $C_k, Q_k$, and $\hat{H}_8$ |
Case given a scenario | Description | Functionally Inconsistent Scenarios
---|---|---
Scenario is consistent, unambiguous | Equations 6.1, 6.2 | no loss

Scenario is consistent, ambiguous | Equations 5.1, 5.2 (weak composition = composition) | (as above)

Scenario is undetectably inconsistent, unambiguous | (algebraic closure = path-consistency) | (none)

Scenario is undetectably inconsistent, ambiguous | intractable subclasses | Satisfies $\Theta^IA_x$ and is outside ORD-Horn subclasses

Table 6.5: Classes of scenarios for which the tank-valve application by [126] using IA will be functionally inconsistent.

The functional inconsistency of the composed expression for variables $(x, y)$ is given by Equation 6.2,

\[
\neg C_{xz} \land (\varphi^IA)^{-1} C_{xz} = (x^+ \geq z^-) \land (x^+ < z^-)
\]

\[= \bot,\]

\[
\neg C_{yz} \land (\varphi^IA)^{-1} C_{yz} = (y^+ \neq z^-) \land (y^+ = z^-)
\]

\[= \bot.\]

Thus, there is no functional inconsistency in the first case. The remaining functionally inconsistent scenarios are similar to the classes derived in Example 6.6.1 above.
6.7 Generating the Strongest QSTR Theory from the Idealised Application

Given $\theta^C$ and a QSTR calculus $A$ the developer requires generate an application $\theta^A$ that most closely approximates $\theta^C$ as specified in Definition 6.1.1. One naive approach for producing a theory $\Theta^A_i$ from $\Theta^C_i$ is to simply map each atomic expression $e^C$ in each sentence $t^C$ into the corresponding qualitative atomic expression $e^A$ in the sentence $t^A$. This is not sufficient as shown in Examples 6.1.2 and 6.1.3. It is thus necessary to ensure that each relation $T^A_1, \ldots, T^A_n$ in a composed expression $\varphi^A (R^C) \circ \varphi^A (S^C)$ is consistent with $\Theta^C_i$, i.e. for all $x, y, z$, $R^C_{xy} \land S^C_{yz} \rightarrow (\varphi^A)^{-1} (T^A_{xz} \lor \ldots \lor T^n_{xz})$. It is observed that this is the application-level analogue of weak composition, and hence Allen’s algorithm can be adapted to generate QSTR applications from idealised applications. The relations that can hold between triples of variables are constrained by entries in the composition table. Thus, the composition table is employed to minimise the reasoning performed in the domain of interpretation. Moreover, the QSTR community have developed tools such as Hets [169] for proving the theorems in the composition tables of QSTR calculi. These tools can be readily employed for implementing the final step in the algorithm.

For brevity, in the following connective expressions are treated as sets of expressions to be interpreted in the obvious way, e.g. a disjunctive expression $e = \{e_1, \ldots, e_n\}$ is $e_1 \lor \ldots \lor e_n$.

Algorithm 6.7.1. For each $t^C \in \Theta^C_i$ perform the following to create $t^A$ and add to $\Theta^A_i$.

1. Copy $t^C$ to $t^A$ and replace all atomic $e^C \in t^C$ with $\varphi^A e^C$.
2. For all $(x, z) \notin \{x, z | \text{exists atomic } e^C_{xz} \in t^C\}$ create a disjunctive expression $e^A_{xz} = \{R^A_{xz}, R^A \text{ is a relation in } A\}$.
3. Update each disjunctive $e^A_{xz}$ using the following operator until a fixpoint is reached:

$$e^A_{xz} := e^A_{xz} \land \{T^A_{xz}\}$$

$$R^A_{xy}, S^A_{yz} \in t^A, T^A \in R^A \diamond S^A,$$

$$\exists x, y, z : (\varphi^A)^{-1} (R^A_{xy} \land S^A_{yz} \land T^A_{xz}) \text{ is true in } t^C.$$
Proof. It must be proved that given any three variables \( x, y, z \) the algorithm finds the strongest expression \( e_{xz}^A \) for constraining those variables with respect to \( y \). Firstly if there is an atomic expression \( e_{xz}^C \) that constrains \( x, z \), then the map \( \phi^A \) will give the strongest expression \( e_{xz}^A \) (step 1). If no atomic expression exists between \( x, z \), and at least one pair of atomic expressions \( e_{xy}^C \) and \( e_{yz}^C \) exist that constrain variables \( x, y \) and \( y, z \) respectively then the algorithm will find the strongest expression \( e_{xz}^A \) for constraining \( x, z \) with respect to \( y \) using \( \psi^A \) (weak composition and \( \mathcal{I}^C \) model check in step 3). Building on this, if no atomic expression exists between two variables \( x, p_k \), and \( k \) atomic expressions \( e_{xp_1}^C, \ldots, e_{xp_{k-1}p_k}^C \) exist that constrain variables \((x, p_1), \ldots, (p_{k-1}, p_k)\) then for each \( i \) such that \( 1 < i \leq k \) the algorithm will eventually find the strongest expression \( e_{xp_i}^A \) with respect to any one of \( p_j \) for \( 1 < j < i \) (repeating step 3 until a fixpoint). If no chain of atomic expressions exists between two variables \( x, z \) then those variables are independent [120].

Corollary 6.7.3. Given \( \Theta_i^C \) and QSTR calculus \( A \), a theory \( \Theta_i^A \) may exist that is stronger than the theory \( \Theta_i^A \) produced by Algorithm 6.7.1.

Proof. By [120] it is possible that there is no \( x, y, w, z \) such that \( R_{xy}^A \land R_{yw}^A \land R_{wz}^A \land T_{xz}^A \) is true in \( \mathcal{I}^C \) while each \( R_{ij}^A \) is the strongest expression in \( A \) with respect to any other variable \( k \), \( R_{ik}^A \land R_{kj}^A \land R_{ij}^A \).

Just as Allen’s algebraic closure algorithm can not guarantee global consistency, guaranteeing the strongest \( A \) application would require extending the model check in the algorithm (step 3) to test for all variable quadruples, quintuples and so on. Based on the NP-hardness of Allen’s algorithm [158] it is conjectured that the problem of generating the strongest \( A \) application is worst case intractable.

6.8 Summary

In this chapter mathematical tools have been presented for analysing the functional performance of QSTR calculi with respect to the specific task requirements of an application. This builds on a methodology introduced in Section 4.3 that enables a developer to specify the functional requirements of an application by defining an idealised application in a sufficiently expressive interpretation language.

The primary contribution of this chapter is the derivation of simple equations that identify the classes of scenarios (as expressions in the interpretation language) for which a QSTR application will produce output that differs from the idealised application; this key concept was defined as functional consistency. It has been proved that functional consistency can be determined by analysing readily available properties of QSTR calculi such as weak composition and the effectiveness of algebraic closure. The developer can use these mathematical tools to analyse the efficacy of myriad QSTR calculi according to their specific task requirements. Moreover,
this establishes a basic foundation for further research into methodologies that facilitate the selection of QSTR calculi.

Furthermore, an adapted path-consistency algorithm has been presented for generating a QSTR calculus-based application up to 3-consistency from a fine-grained idealised application. It is conjectured that the problem of generating the strongest QSTR application from an idealised application is worst case intractable.

This chapter has presented a methodology that facilitates the selection of QSTR calculi, thus supporting the design of low level fragments in QSTR applications. The following chapter addresses the challenge of designing customised high level fragments.
Chapter 7

Designing High Level Application Fragments: Utilising Data-Based Metrics and Deriving High Level Neighbourhood Graphs

7.1 Introduction

In the previous chapter, a methodology was presented for designing low level fragments by selecting functionally effective QSTR calculi. This chapter is concerned with the design of customised high level fragments which formalise domain concepts and rules. Specifically, using the definition of QSTR applications established in Section 3.2, the developer must declare relevant relations $R \in \mathbb{R}$ and define theories $\Theta_R \subseteq \Theta$ in order to satisfy the formal software requirements defined in Chapter 4.

QSTR applications are typically organised in a hierarchical structure, as presented in Section 3.6 and supported by the fragment analysis in Section 5.2. Intuitively, relations in high level fragments are defined as patterns in the relations of lower level fragments; this was formalised using fragment definitions in Section 5.3. It is therefore necessary for methodologies to directly focus on enabling the analysis of hierarchical interactions between fragments if they are to facilitate the development of customised, application specific fragments. Furthermore, as presented in Sections 3.6 and 2.7, conceptual neighbourhoods are a central component of QSTR applications. Conceptual neighbourhoods are used to perform almost all tasks over multiple scenarios, as shown in Section 4.4. It is therefore imperative that a designer has a methodology for defining neighbourhoods over higher level relations to enable the performance of QSTR tasks using custom application-specific fragments.
Chapter 7. Designing High Level Application Fragments: Utilising Data-Based Metrics and Deriving High Level Neighbourhood Graphs

Example 7.1.1. In the QtvLight application (refer to 2.8.1), a number of subjective impressions, such as spacious, are defined using intermediate level fragments, such as the illuminance pattern on surfaces with relations uniform and non-uniform. The developer requires a methodology for analysing the interaction between these fragments in order to determine whether their particular formalisation of the domain knowledge satisfies the software requirements.

Example 7.1.2. The Image Retrieval application (refer to Section 2.8.4) defines high level image categories, such as coasts, waterscapes, fields, and forests, based on combinations of lower level qualitative relations, such as sky>grass. The developer may be need to define a conceptual neighbourhood graph that determines the adjacency of images, for example, to enable query image ranking according to high level image categories. Thus, a methodology is required to analyse and guide the design of the fragment definitions between image categories and the qualitative relations of image textures.

The developer formalises high level relations by referring to three primary sources of information: (1) Field research and studies in the domain, for example, Flynn’s rules about lighting configurations and subjective impressions [61, 62]; (2) Industry consensuses in the form of qualitative regulations and codes, for example, performance-based building codes [148] or maritime right-of-way rules as formalised in SailAway (refer to Section 2.8.5); (3) Client and user preferences, that is, qualitative requirements prescribed directly by the client or the intended user of the application.

These sources of domain information are obtained by additional field data or other contextual information. Methodologies are required that enable a developer to utilise field data and additional contextual information in order to design fragment interactions in a manner that is consistent with the application domain. Specifically, the findings from field research are typically reported with details of the study data (or contact information for accessing the details) for the purposes of verification. For example, the Image Retrieval application employed a manually annotated database of images developed by Vogel and Schiele [160], and the QtvLight application employed scenario configurations from the studies performed by Flynn [61]. Industry consensuses and client preferences are typically established in context of other domain information. For example, it is reasonable to suggest that waterscape images are more similar to coasts than desert images.

In this chapter three methodologies are presented that guide the design of high level fragments by facilitating the analysis of hierarchical fragment interactions. Section 7.2 presents a methodology that utilises field data by adapting the software engineering concepts of cohesion and coupling. Section 7.3 employs machine learning classification on field data to analyse hierarchical relationships between fragments, and to derive high level conceptual neighbourhoods based on the fragment definitions. Finally, Section 7.4 presents a methodology that extensively
generalises the definition of neighbourhoods to enable the design of high level neighbourhoods that are consistent with the structure of fragments in a QSTR application.

### 7.2 Analysing Fragment Interaction using Coupling and Cohesion

This section adapts software engineering concepts of object-oriented cohesion and coupling for QSTR applications, which enable a developer to analyse the organisation of a QSTR application in terms of fragment interactions. In QSTR applications, coupling is a measure of how effectively a set of low level relations distinguishes between different high level relations; cohesion is a measure of how effectively a set of low level relations defines a high level relation [145]. Coupling and cohesion are calculated by using a dataset of exemplary objects (either in one large scenario or spread across a set of scenarios) that reflect the correct interactions between fragments, such as the detailed results of field studies.

**Example 7.2.1.** Let $R_{\text{coast}}, R_{\text{forest}}$ be image category relations in a high level fragment $F_{\text{category}}$, and let $R_{\text{trunks}<\text{water}}, R_{\text{trunks}=\text{water}}, R_{\text{trunks}>\text{water}}$ be image texture relations in a low level fragment $F_{\text{trunks,water}}$. The developer has specified two simple fragment definitions $R_{\text{trunks}<\text{water}} \rightarrow R_{\text{coast}}$ and $R_{\text{trunks}>\text{water}} \rightarrow R_{\text{forest}}$. By using a database [160] of images that have been manually assigned a high level image category, the developer can analyse the efficacy of the specified fragment definitions.

The coupling and cohesion of high level relations are both calculated by measuring the distance between relations in the defining lower level fragments. Let the set of tuples in relation $R$ for scenario $s$ be denoted $s(R)$. Given a scenario $s$ that contains two object tuples $x, y$, the distance between $x$ and $y$ with respect to fragment $F = \{R_1, \ldots, R_n\}$ is the distance of the shortest path between the relations $R_i, R_j \in F$ through the neighbourhood graph of $F$ such that $x \in s(R_i)$ and $y \in s(R_j)$. Assuming all edges in the neighbourhood graph have a unit weight, if $R_i = R_j$ then the distance is 0, if $R_i$ is adjacent to $R_j$ then the distance is 1, and so on. If $N^d_F : F \times F \rightarrow \mathbb{N}_0$ is a function that determines the length of the shortest path between two relations in fragment $F$ (where $\mathbb{N}_0$ is the set of naturals including 0) then

$$\text{distance}(x, y, F, s) \equiv N^d_F(R_i, R_j) \quad \text{where} \quad R_i, R_j \in F \quad \text{and} \quad x \in s(R_i), y \in s(R_j).$$

If the two object tuples $x, y$ in the above scenario are also elements of a high level relation $R_\alpha$ then the cohesion of $R_\alpha$ is influenced by the distance between $R_i$ and $R_j$. If $R_i = R_j$ then $R_\alpha$ is relatively cohesive with respect to $x, y$ in the given scenario; if the distance between $R_i$ and $R_j$ is 1 then $R_\alpha$ is slightly less cohesive with respect to $x, y$ in the given scenario, and so on.

---

$^1$Neighbourhood edges can be weighted, thus influencing the total cost of the shortest path between two relations.
Alternatively, if the two object tuples \( x, y \) in the given scenario are elements of two different high level relations, \( x \in R_\alpha \) and \( y \in R_\beta \), then the coupling between \( R_\alpha \) and \( R_\beta \) is influenced by the distance between \( R_i \) and \( R_j \). If \( R_i = R_j \) then \( R_\alpha \) and \( R_\beta \) are strongly (or tightly) coupled with respect to \( x, y \) in the given scenario; if the distance between \( R_i \) and \( R_j \) is 1 then \( R_\alpha \) and \( R_\beta \) are more weakly (or loosely) coupled with respect to \( x, y \) in the given scenario, and so on.

The algorithm for calculating coupling accepts a set of scenarios \( S \), two high level relations \( R_\alpha, R_\beta \) and a set of low level fragments \( F_1, \ldots, F_n \). For each scenario \( s \), for each pair of tuples in \( x \in s(R_\alpha), y \in s(R_\beta) \) the algorithm calculates the cumulative distance between \( x, y \) with respect to each low level fragment \( F_1, \ldots, F_n \) and stores the total distance value in a set \( D \). The algorithm returns statistics on the final set of distance values such as the mean and variance. Two high level relations have low coupling if the distances have a high mean and a low variance. Ideally two relations that are significantly distinct in the application domain should have low coupling.

**Algorithm 7.2.2.** coupling\((S,R_\alpha,R_\beta,F_1,\ldots,F_n)\)

\[
\text{let } D = \{\} \\
\text{for all } s \in S \\
\quad \text{for all } x \in s(R_\alpha), y \in s(R_\beta) \\
\quad \quad \text{let } d = 0 \\
\quad \quad \text{for all } F \in \{F_1,\ldots,F_n\} \\
\quad \quad \quad d = d + \text{distance}(x, y, F, s) \\
\quad D = D \cup d \\
\text{return statistics}(D)
\]

**Example 7.2.3.** Following from Example 7.2.1, let scenario \( s \) be defined as \( s(R_{\text{coast}}) = \{a\} \), \( s(R_{\text{forest}}) = \{b, c\} \), \( s(R_{\text{trunks<water}}) = \{a, b\} \), and \( s(R_{\text{trunks>water}}) = \{c\} \). The coupling algorithm is executed: coupling\((\{s\}, R_{\text{coast}}, R_{\text{forest}}, F_{\text{trunks<water}})\); the resulting raw coupling distances are \( D = \{0, 2\} \), the coupling distance mean is 1 and the standard deviation is \( \sim 1.41 \).

The algorithm for calculating cohesion is a special case of the coupling algorithm where \( R_\beta = R_\alpha \). A relation is highly cohesive if the distances have a low mean and a low variance. Ideally relations should be cohesive, as this indicates that all objects for which the relation holds share a similar set of defining low level relations.

**Algorithm 7.2.4.** cohesion\((S,R_\alpha,F_1,\ldots,F_n)\)

\[
\text{return coupling}(S, R_\alpha, R_\alpha, F_1, \ldots, F_n)
\]

**Example 7.2.5.** Following from Example 7.2.1, let scenario \( s \) be defined as \( s(R_{\text{coast}}) = \{a, b, c\} \), \( s(R_{\text{trunks<water}}) = \{a, b\} \), and \( s(R_{\text{trunks>water}}) = \{c\} \). The cohesion algorithm is executed: cohesion\((\{s\}, R_{\text{coast}}, F_{\text{trunks<water}})\); the resulting raw cohesion distances are \( D = \{0, 1, 1\} \), the cohesion distance mean is \( \frac{2}{3} \) and the standard deviation is \( \sim 0.58 \).
Figure 7.1 presents the mean, upper quartile, and lower quartile of the cohesion and coupling distances for each image category in the actual Image Retrieval application (reimplemented by the author of this thesis, refer to Section 2.8.4) with a dataset of approximately 700 annotated images [134, 160]. It can be observed that the definitions of certain image categories appear to be faulty. In particular, the image categories fields and landscape with mountains (lwm) have lower mean coupling distances to other categories compared to mean category cohesion; that is, a field image is likely to be more similar to a sky cloud image than another randomly chosen field image. This could indicate to the developer that the qualitative approach used to categorise images may need to be improved depending on specific task requirements. Interestingly, although the coupling distance between two image categories is symmetric, the relative similarity between image categories depends strongly on the reference image category. For instance, using coasts as the reference image category, coast images are equidistant to coasts and waterscapes on average. However, using waterscapes as the reference image category, waterscape images are more similar to waterscapes than coasts on average. Similarly, from the perspective of landscapes with mountains, the waterscape, forest and coast categories are very similar, however from the perspective of forests, the waterscape and forest categories are the most distant from forest images.

In order to summarise the interaction between the high level relations, cohesion and coupling can be used to form basic clusters with respect to each image category. For example, using forests as a reference image category, two clusters are near forests and far from forests based on mean coupling distances. Near clusters contain image categories that have a mean coupling distance within a specified threshold percentage of the mean cohesion distance. Far clusters contain image categories that have a mean coupling distance within a specified threshold percentage of the highest mean coupling distance. Table 7.1 presents near and far clustering using a threshold of 10%. Figure 7.2 illustrates a graph representing the clustering information contained in Table 7.1. In Figure 7.2 the image category abbreviations are: coasts=co, waterscapes=wa, forests=fo, fields=fi, landscapes with mountains=lw, sky clouds=sk; solid arrows represent near clusters and dashed arrows represent image categories that are neither near nor far. A number of interesting properties can be observed when the clustering graph is interpreted as a neighbourhood graph, such that solid arrows represent strong neighbours and dashed arrows represent weak neighbours. It is intuitively reasonable that waterscapes and coasts are conceptual neighbours, and that landscapes with mountains and sky clouds are neighbours. Forests weakly neighbour fields due to the relative frequency of foliage, however fields do not neighbour forests and instead neighbour landscapes with mountains and sky clouds. This is due to the relatively large amount of sky that appears in images from these three categories.
Chapter 7. Designing High Level Application Fragments: Utilising Data-Based Metrics and Deriving
High Level Neighbourhood Graphs

Figure 7.1: Mean, upper quartile, and lower quartile cohesion and coupling distances of image categories. Landscape with mountains is abbreviated to lwm.

<table>
<thead>
<tr>
<th>Image</th>
<th>Threshold mean coupling</th>
<th>Images near (mean coupling &lt; near threshold)</th>
<th>Images Far from (mean coupling &gt; far threshold)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coasts</td>
<td>19.25 29.58</td>
<td>Waterscapes</td>
<td>Forests</td>
</tr>
<tr>
<td>Waterscapes</td>
<td>21.78 24.89 (none)</td>
<td>(none)</td>
<td>Forests, fields</td>
</tr>
<tr>
<td>Forests</td>
<td>13.86 29.58 (none)</td>
<td>Coasts</td>
<td>(none)</td>
</tr>
<tr>
<td>Fields</td>
<td>21.80 25.50</td>
<td>Sky clouds, lwm</td>
<td>Forests, waterscapes, coasts</td>
</tr>
<tr>
<td>lwm</td>
<td>14.00 21.88</td>
<td>Sky clouds</td>
<td>Forests, waterscapes, coasts</td>
</tr>
<tr>
<td>Sky clouds</td>
<td>6.75 23.52</td>
<td>(none)</td>
<td>Forests</td>
</tr>
</tbody>
</table>

Table 7.1: Near and far clusters formed using mean coupling distance thresholds. Near threshold is cohesion mean +10%, far threshold is greatest coupling mean −10%.
7.2 Analysing Fragment Interaction using Coupling and Cohesion

Figure 7.2: Graph of the near and far clustering from Table 7.1.

7.2.1 Utilising Coupling and Cohesion to Support High Level Fragment Design

The ways in which a developer can utilise coupling and cohesion depend on three parameters: (a) the neighbourhood graph of the high level fragment \( F \), (b) the neighbourhood graph of the low level fragments \( F_1, \ldots, F_n \), and (c) fragment definitions that associate \( F \) with \( F_1, \ldots, F_n \). If information is available for any two of these parameters then the developer can employ coupling and cohesion to infer information about the third unknown parameter.

Firstly the metrics can be used to assess the accuracy of the neighbourhood graph of the high level fragment \( F \) by determining whether the neighbours of \( R \in F \) have a higher coupling than more distant relations. That is, information about parameters (b) and (c) can be used to confirm (a). A full neighbourhood consistency test ensures that, from the perspective of each \( R \in F \), all partial orderings of the other relations \( R' \in F \) based on the coupling between \( R, R' \) (with a tolerance for equal coupling) approximate the neighbourhood graph of \( F \). Furthermore, coupling can be used to assign or evaluate neighbour weightings in the neighbourhood graph of \( F \) where edges are annotated with non-uniform costs; a low coupling yields a high neighbourhood weight.

Secondly, the developer can apply the metrics to analyse the accuracy of the neighbourhood graphs of the low level fragments. That is, information about parameters (a) and (c) can be used to confirm (b). This involves measuring the cohesion (resp. coupling) of two objects that are known to be similar (resp. distinct) according to the application domain. If the objects have low cohesion (resp. high coupling) then this is an indication that the low level fragment’s neighbourhood graph is inconsistent with contextual information.

Finally, cohesion and coupling can be used to analyse fragment definitions. That is, information about parameters (a) and (b) can be used to confirm (c). If a given fragment definition of
Chapter 7. Designing High Level Application Fragments: Utilising Data-Based Metrics and Deriving High Level Neighbourhood Graphs

R to $R_1, \ldots, R_n$ yields a low cohesion for $R$ then this is an indication that the fragment definition is faulty, particularly if another subset of relations $R'_1, \ldots, R'_n$ is identified that provides a more cohesive fragment definition for $R$. Similarly, if the coupling between two distinct relations $R, R'$ is unusually high then this is an indication that one or both of the fragment definitions are faulty. Conversely, cohesion and coupling can provide evidence that supports the developer’s fragment definitions in the case where each relation $R$ is highly cohesive and each pair of distinct relations $R, R'$ is loosely coupled. Furthermore, high level relations may not be precisely defined by low level relations, and instead may only be related as a rule of thumb. The developer can use cohesion and coupling to quantify the efficacy of the particular subset of low level relations as a heuristic definition of a high level relation, for example, to support the design of probabilistic qualitative reasoning systems.

7.2.2 Accuracy and Precision for Analysing Fragment Definitions

Coupling and cohesion measure the similarity and difference between fragments relative to other objects in a dataset. However, the developer is likely to have information about the properties of a prototypical object for a given high level relation. That is, coupling and cohesion are measured with respect to all low level relations in a defining fragment. To complement this, two further metrics, accuracy and precision\(^2\), are presented that measure fragment interactions with respect to the particular low level relations within defining fragments.

Example 7.2.6. Following from Example 7.2.1, let scenario $s$ be defined as $s(R_{\text{coast}}) = \{a, b, c\}$, $s(R_{\text{trunks<water}}) = \{a, b, c\}$. Assume that the developer has specified the erroneous fragment definition $R_{\text{trunks>water}} \rightarrow R_{\text{coast}}$. The cohesion algorithm is executed: $\text{cohesion}(\{s\}, R_{\text{coast}}, F_{\text{trunks,water}})$; the results show that $R_{\text{coast}}$ is maximally cohesive, despite none of the images representing a prototypical coast image according to the (erroneous) fragment definition. Hence a design fault remains undetected.

Accuracy and precision are a measure of the average discrepancy and variance (respectively) between an inferred relation $R$ and the expected relation $R'$ according to the existing fragment definition $R_1 \land \ldots \land R_n \rightarrow R'$. Formally, accuracy and precision are calculated as the mean and variance of the coupling between dataset objects and the prototype object from the specified fragment definition. The algorithm for calculating accuracy and precision accepts a set of scenarios $S$, a high level relation $R_\alpha$, and the relations $R_1, \ldots, R_n$ that define $R_\alpha$. A prototype object $k$ is created. For each scenario in $S$, $k$ is added to each defining relation $R_1, \ldots, R_n$ and a new prototype relation $R'_\alpha$. The algorithm returns the statistics of the coupling between the dataset $R_\alpha$ and the prototype $R'_\alpha$. High accuracy (resp. precision) is equivalent to the dataset and prototype relations having a high mean (low variance) coupling distance. High accuracy and precision indicate that the fragment definition is consistent with the data.

\(^2\)Accuracy and precision are related to the concepts of bias and error in statistics.
7.3 Generating High Level Neighbourhoods using Decision Tree Learning

Algorithm 7.2.7. accuracy\(\text{Precision}(S, R_\alpha, R_1, \ldots, R_n)\)

- let \(F_1, \ldots, F_n\) where \(R_1 \in F_1, \ldots, R_n \in F_n\)
- let \(S' = \{\}\)
- let \(k\) be a unique constant
- for \(s \in S\)
  - for \(R \in R_\alpha', R_1, \ldots, R_n\)
    - \(s'(R) = s(R) \cup \{k\}\)
- \(S = S \cup s'\)
- return coupling\((S', R_\alpha, R_\alpha', F_1, \ldots, F_n)\)

Example 7.2.8. Using the scenario in Example 7.2.6, the accuracy algorithm is executed: \(\text{accuracy}\left(\{s\}, R_{\text{coast}}, R_{\text{trunks} > \text{water}}\right)\); the resulting raw accuracy distances are \(D = \{2, 2, 2\}\), the accuracy distance mean is 2 and the standard deviation is 0.

7.3 Generating High Level Neighbourhoods using Decision Tree Learning

In this section machine learning techniques are employed for generating fragment definitions from datasets, and deriving conceptual neighbourhoods based on the generated definitions. Decision trees succinctly express fragment definitions, for example, Figure 7.3 illustrates a succinct definition of a coast image. Each vertex in the tree is annotated with a reference domain relation and a state (i.e. holds, not holds, and not applicable), and each leaf is annotated with an abstraction domain relation and a state. A path from the root to a leaf represents a fragment definition, where tuples in the relation state at each vertex in the path are conjoined to be an improper subset of the tuples in the relation state at the leaf, e.g. \(R_{\text{clouds} < \text{water}} \land R_{\text{clouds} < \text{sand}} \rightarrow R_{\text{coast}}\). Unspecified conditions are independent of the conditions given in the path and can take any value. In the above example all paths accept the relations \((\text{mountain} < \text{clouds})\), \((\text{mountain} = \text{clouds})\) and \((\text{mountain} > \text{clouds})\).

Decision tree learning is the process of building a model that accepts a set of input variable values and then predicts the value of a target variable [168]. The developer can employ decision tree learning to generate fragment definitions from field data. The developer can then

![Decision Tree](image_url)

**Figure 7.3:** Partial decision Tree representation of the fragment definition of coast images.
Table 7.2: Fragment definitions generated using the J48 decision tree learning algorithm, implemented in Weka [168], with confidence 0.25.

| Images | cl | wa | sa | sn | fo | fl | tr | sk | sa | co | fo | go | gr | sn | sn | mn |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Coasts | <  | <  | <  | <  | <  | <  | <  | >  | >  | <  | >  | >  | >  | >  | <  | 33.80% |
| Waterscapes | <  | <  | <  | >  | >  | >  | >  | >  | >  | >  | >  | >  | >  | >  | >  | 1.40% |
| Forest | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | 0.70% |
| Fields | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | 1.80% |
| lwm | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | <  | 0.90% |

Table 7.2: Fragment definitions generated using the J48 decision tree learning algorithm, implemented in Weka [168], with confidence 0.25.

Similar to the neighbourhood derivation using the clustering approach from coupling and cohesion, the generated fragment definitions can be employed to derive a conceptual neighbourhood of high level relations. In general, neighbourhoods can be derived from any fragment definition as the following example illustrates.

**Example 7.3.1.** Let $R_{\text{coast}}$, $R_{\text{forest}}$ be image category relations in a high level fragment $F_{\text{category}}$, and let $R_{\text{trunks}<\text{water}}$, $R_{\text{trunks} \equiv \text{water}}$, $R_{\text{trunks}>\text{water}}$ be image texture relations in a low level fragment $F_{\text{trunks,water}}$. The developer has specified two simple fragment definitions $R_{\text{trunks}<\text{water}} \rightarrow R_{\text{coast}}$ and $R_{\text{trunks}>\text{water}} \rightarrow R_{\text{forest}}$. Intuitively, $R_{\text{coast}}$ is conceptual neighbours with $R_{\text{forest}}$ if it is possible to start from the low level definition $R_{\text{coast}}$, namely $R_{\text{trunks}<\text{water}}$, make incremental changes to

---

**Chapter 7. Designing High Level Application Fragments: Utilising Data-Based Metrics and Deriving High Level Neighbourhood Graphs**

compare the resulting generated fragment definitions with their manually formalised fragment definitions. Table 7.2 presents fragment definitions generated by the J48 decision tree learning algorithm (with confidence parameter value 0.25), implemented in Weka [168]. In Table 7.2 the image texture abbreviations are: clouds=cl, water=wa, sand=sa, flowers=fl, foliage=fo, snow=sn, trunks=tr, sky=sk, grass=gr, mountains=mn. Sky clouds was excluded as it was found to have a large number of low proportion definitions. The dataset employed was Vogel and Schiele’s annotated collection of images [160].

Similar to the neighbourhood derivation using the clustering approach from coupling and cohesion, the generated fragment definitions can be employed to derive a conceptual neighbourhood of high level relations. In general, neighbourhoods can be derived from any fragment definition as the following example illustrates.

**Example 7.3.1.** Let $R_{\text{coast}}$, $R_{\text{forest}}$ be image category relations in a high level fragment $F_{\text{category}}$, and let $R_{\text{trunks}<\text{water}}$, $R_{\text{trunks} \equiv \text{water}}$, $R_{\text{trunks}>\text{water}}$ be image texture relations in a low level fragment $F_{\text{trunks,water}}$. The developer has specified two simple fragment definitions $R_{\text{trunks}<\text{water}} \rightarrow R_{\text{coast}}$ and $R_{\text{trunks}>\text{water}} \rightarrow R_{\text{forest}}$. Intuitively, $R_{\text{coast}}$ is conceptual neighbours with $R_{\text{forest}}$ if it is possible to start from the low level definition $R_{\text{coast}}$, namely $R_{\text{trunks}<\text{water}}$, make incremental changes to
the low level relations, and eventually transition into the low level definition of $R_{\text{forest}}$, namely $R_{\text{trunks}} \bowtie \text{water}$, without passing through the definition of some other high level relation. It is observed that $R_{\text{trunks}} \bowtie \text{water}$ does not correspond to any other high level relation and thus it is possible to move from $R_{\text{trunks}} \bowtie \text{water}$ to $R_{\text{trunks}} \bowtie \text{water}$ without defining some other high level relation. It follows that $R_{\text{coast}}$ and $R_{\text{forest}}$ are neighbours with respect to their underlying fragment definitions. Now, let $R_{\text{lake}}$ be added to the high level fragment containing $R_{\text{coast}}$ and $R_{\text{forest}}$, and let there be a fragment definition $R_{\text{trunks}} \bowtie \text{water} \rightarrow R_{\text{lake}}$; $R_{\text{coast}}$ and $R_{\text{forest}}$ are no longer neighbours.

Figure 7.4 illustrates the neighbourhood graph derived from the generated fragment definitions presented in Table 7.2. In Figure 7.4 the image category abbreviations are: 

- coasts=$co$
- waterscapes=$wa$
- forests=$fo$
- fields=$fi$
- landscapes with mountains=$lw$
- sky clouds=$sk$

Edges are annotated with the probability of a high level relation transition according to a given low level relation transition; solid arrows represent strong neighbours with total transition probability greater than 70% and dashed arrows represent weak neighbours with total transition probability between 50% and 70%. The derived neighbourhood is significantly more connected than the neighbourhood graph derived from coupling clusters, illustrated in Figure 7.2. This is due to the standard QSTR definition of conceptual neighbours being too weak for describing the interaction between fragments at the application-level; this issue is thoroughly addressed in the following section.

Edges of the derived neighbourhood are annotated with the probability of a high level relation transition according to particular low level relation transitions. For example, the key difference between waterscapes and forest images in terms of conflicting low level relations is that waterscapes have $\text{clouds} \bowtie \text{water}$, whereas forests have $\text{clouds} = \text{water}$, as presented in Table 7.2. Thus, a low level description of a waterscape image is guaranteed to transition into a low level description of a forest image if the proportion of clouds to water changes from $\text{clouds} \bowtie \text{water}$ to $\text{clouds} = \text{water}$; in the domain of image textures such a transition would require that either more clouds are introduced or some water is removed.

In many cases multiple low level transition paths exist between high level image categories that depend on the particular starting image description. For example, a large proportion of coast images that undergo a low level transition from $\text{clouds} \bowtie \text{sand}$ to $\text{clouds} = \text{sand}$ will transition into waterscape images; the low level description of these images is presented in the first row of Table 7.2. However, coast images that start with a description given in the fourth row of Table 7.2 that undergo this change will not transition into waterscapes. Transition probability is calculated as the proportion of images in the database from categories $R_i, R_j$ that conflict on the proportion of a given pair of textures; importantly, the only textures that are compared are those that define each image category. For example, the pairs of textures tested when calculating the transition probabilities between coast and waterscape are (clouds, water), (clouds, sand), (foliage, snow), and (foliage, flowers). Thus $\sim 50\%$ of coast and waterscape images conflict on the proportion of clouds and sand, and $\sim 45\%$ conflict on the proportion of foliage and snow.
Figure 7.4: Derived neighbourhood graph from generated fragment definitions presented in Table 7.2.
7.4 Designing Consistent High Level Neighbourhood Graphs

To ensure that reasoning is sound, the designer necessarily requires their expected neighbourhood for relations in a higher-level abstraction domain fragment to be consistent with the neighbourhood of relations in the associated reference domain fragment. As illustrated in Example 7.3.1, two high level relations should not be defined as conceptual neighbours if no transition path between low level relations exists that does not pass through the definition of a third high level relation. Alternatively, the designer should be able to derive neighbourhoods for a group of relations based solely on fragment definitions if no other neighbour information is available.

However, a number of issues arise when applying the canonical neighbour definition to derive neighbourhoods from the relationship between fragments. In many cases the standard definition of conceptual neighbours permits all high level concepts to be neighbours, producing a neighbourhood graph that is ineffective for all neighbourhood based tasks such as query result ranking or envisioning. This was illustrated by comparing Figures 7.4 and 7.2 in the previous section; the neighbourhood graph derived from fragment definitions was significantly more connected than the neighbourhood derived from coupling clusters.

This section presents a methodology for defining high level neighbourhoods that are consistent with the structure of fragments in a QSTR application. As illustrated in Example 7.3.1, conceptual neighbours are defined using fragment definitions. Two types of conditions are presented for refining the definition of conceptual neighbours: path restrictions and equivalence classes. A consistent neighbourhood graph can be generated once the developer has decided on an appropriate definition of a conceptual neighbour.

Example 7.4.1. This example will be used throughout this section to illustrate the problem of employing the standard neighbourhood definition to higher level fragments, and to illustrate methods for overcoming these limitations. Figure 7.5 illustrates a reference domain fragment $F_2$ that includes eight mutually exclusive relations $R_1, \ldots, R_8$ and six mutually exclusive relations $R'_1, \ldots, R'_6$ each having a simple totally ordered conceptual neighbourhood. Each vertex in the grid illustrated in Figure 7.5 (right) represents a valid conjunction of relations in $F_2$, called the fragment definition space. An abstraction domain fragment $F_1$ contains three relations $R_x$ (black), $R_y$ (striped), $R_z$ (grey) that have fragment definitions in $F_2$ illustrated in the fragment definition space (right).

7.4.1 Methodology for Deriving a High Level Neighbourhood Graph

The developer can construct a high level conceptual neighbourhood graph $G_{F_\alpha}$ by applying the following procedure.

1. Define an abstraction domain fragment $F_\alpha$ and a set of reference domain fragments $F_{\beta_1}, \ldots, F_{\beta_n}$.
2. Define low level neighbourhood graphs \( G_{F_{\beta_1}}, \ldots, G_{F_{\beta_n}} \).

3. For each high level relation \( R \in F_\alpha \) define fragment definitions \( e_R \) such that \( \sigma(e_R) \subseteq F_{\beta_1} \times \ldots \times F_{\beta_n} \).

4. Decide on the appropriate definition of conceptual neighbours \( N_{F_\alpha} \).

5. Construct the neighbourhood graph \( G_{F_\alpha} \) such that

   (a) there is exactly one vertex for each high level relation \( R \in F_\alpha \), and

   (b) for each pair of high level relations \( R, R' \in F_\alpha \) there is an edge between the corresponding vertices iff \( N_{F_\alpha}(R, R') \).

In practice, steps 1 and 2 will require the developer to select a pair of appropriate fragments that have already been defined in the application. A methodology for configuring the neighbour test is described in the following sections, and summarised in Table 7.3. The designer can automate step 5 with a simple nested for-loop algorithm that executes the neighbour test on each pair of high level relations. Table 7.3 presents guidelines to help the developer select the appropriate neighbour definition. The guidelines relate a problem that the designer can experience when deriving neighbourhood graphs, the appropriate actions defined in the following sections, and associated effects of the action.

### 7.4.2 Formally Defining Conceptual Neighbours as Transitions Between Low Level Relations

Conceptual neighbours are derived from fragment definitions by firstly defining three types of transitions.
### Problem with current derived neighbourhood

<table>
<thead>
<tr>
<th>Relations that clearly lie between two relations have no effect on their neighbour status.</th>
<th>Path restriction: only accept shortest paths through reference domain.</th>
<th>Transitions will now monotonically approach the target definition. This avoids paths that ‘side step’ obstructions by taking impractical and counter-intuitive transitions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interesting and important cores of relations are obstructed by other relations, but this has no effect on their neighbour status.</td>
<td>Path restriction: only accept paths that change core low level relations. Core relations are those that conflict between every definition of two high level relations.</td>
<td>Transitions now only change the prototypical core of a high level relation when determining neighbour status. This avoids irrelevant and uninformative paths that lie in the vast and sparse ‘transition space’ between these critical points.</td>
</tr>
<tr>
<td>Relations that clearly lie between two relations have no effect on their neighbour status, even after applying path restrictions.</td>
<td>Equivalence classes: pick a restricted set of paths that strongly determines neighbour status and define the set as an equivalence class of paths.</td>
<td>Ordering of transitions is ignored when determining whether the path between two relations is obstructed. The neighbour test is strengthened to guarantee that no matter what order the transitions are made, no obstruction will occur.</td>
</tr>
<tr>
<td>Relations have multiple definitions, and the neighbour test is too coarse grained. The user needs more information about boundary cases where relations are sometimes or partially neighbours.</td>
<td>Neighbour probability: include probability scores with neighbour status.</td>
<td>Probabilities are a measure of the strength of neighbour status where high probability indicates that relations are closer neighbours than a low probability; the user can get an indication of the likelihood that a transition to some other relation will be interrupted.</td>
</tr>
<tr>
<td>Sometimes ( R_1 ) is a neighbour of ( R_2 ) when ( R_2 ) is not a neighbour of ( R_1 ), i.e. the neighbour test must be symmetric.</td>
<td>Symmetric neighbours: consider both transition directions between relations before determining their shared neighbour status.</td>
<td>Neighbour test now involves both transition directions. Care must be taken in understanding and communicating to users what the neighbour test now entails, as one transition direction can completely hide the neighbour status of the other direction. For example, saying ( R_1 ) is a neighbour of ( R_2 ) does not necessarily mean that it is possible to directly transition from ( R_1 ) to ( R_2 ) (the other transition direction may be the critical factor in determining neighbour status).</td>
</tr>
</tbody>
</table>

| Table 7.3: QSTR application developer guidelines for deriving effective neighbourhood graphs. |
Definition 7.4.2. Neighbourhood transition: Given a neighbourhood graph \( G_F \) of fragment \( F \) and two relations \( R_x, R_y \in F \), a neighbourhood transition \( \Delta G_F (R_x, R_y) \) is a sequence of relations \( \{R_x, R_1, \ldots, R_j, R_y\} \) that is a path in \( G_F \). It must be noted that there may be more than one path, and that paths can contain cycles.

Definition 7.4.3. Fragment definition transition: Given relation \( R \), and a fragment definition expression \( e = R_1 \land \ldots \land R_n \rightarrow R \), let \( \sigma (e_R) = \{R_1, \ldots, R_n\} \), i.e. the function \( \sigma \) collects the subset of reference domain relations (refer to Section 5.3.1). A fragment definition transition from \( \sigma (e_R) \) to \( \sigma (e_{R'}) \) is a sequence of fragment definitions, \( (\sigma (e_R), \ldots, \sigma (e_{R'}) \) such that adjacent fragment definitions in the sequence differ by exactly one pair of neighbouring low level relations.

Example 7.4.4. Consider the fragment definitions space in Figure 7.5. One transition from the fragment definition \( \{R_2, R'_1\} \) to \( \{R_5, R'_3\} \) is \( \{R_2, R'_1, R_3, R'_1, R_4, R'_1, R_5, R'_1, R_5, R'_2, R_5, R'_3\} \), and another transition is \( \{R_2, R'_1, R_3, R'_2, R_5, R'_2, R_4, R'_2, R_5, R'_3\} \).

Consider the set of all possible fragment definition transitions between two high level relations. The ordering of some particular low level changes is essential. In particular, by Definition 7.4.2 transitions can not violate the continuity assumption by skipping relations in the low level neighbourhood graph \( G_F \), e.g. in Example 7.4.1 a transition \( \ldots, \{R_2, R'_1\}, \{R_4, R'_1\}, \ldots \) is invalid. Other changes can occur in any order, e.g. in Example 7.4.1 the transition from \( R_3 \) to \( R_4 \) is completely independent of the transition from \( R'_1 \) to \( R'_2 \) and these transitions can occur in any order. Thus, a class of transitions can be succinctly expressed by representing a high level transition as a partial ordering of low level transitions.

Definition 7.4.5. Fragment definition transition class: Given two high level relations \( R, R' \in F_\alpha \), and fragment definitions \( e_R \) and \( e_{R'} \), a fragment definition transition class \( \Delta e_R, e_{R'} \ (R, R') \) is a set of transitions from \( \Delta G_{F_\alpha} (R, R') \) such that:

1. for each relation \( R_i \in \sigma (e_R) \) there is some \( R_j \in \sigma (e_{R'}) \), where \( R_i, R_j \in F_\beta \), such that there is a neighbourhood transition \( \Delta G_{F_\beta} (R_i, R_j) \), and

2. for each relation \( R_j \in \sigma (e_{R'}) \) there is some \( R_i \in \sigma (e_R) \), where \( R_i, R_j \in F_\beta \), such that there is a neighbourhood transition \( \Delta G_{F_\beta} (R_i, R_j) \), and

Example 7.4.6. In Example 7.4.1 one class of transitions from \( \{R_2, R'_1\} \) to \( \{R_5, R'_3\} \) is \( \{(R_2, R_3, R_4, R_5), (R'_1, R'_2, R'_3)\} \). Another class is \( \{(R_2, R_1, R_2, R_3, R_4, R_5), (R'_1, R'_2, R'_3)\} \).

Therefore, a transition class \( \Delta e_R, e_{R'} \ (R, R') \) specifies a partial ordering of incremental changes at the lower fragment definition level (i.e. from \( \Delta G_{F_\beta} \) required to move from the fragment definition of \( R \) to the fragment definition of \( R' \).
7.4 Designing Consistent High Level Neighbourhood Graphs

**Definition 7.4.7. Conceptual neighbours:** \( R, R' \in F \) are conceptual neighbours \( N_F (R, R') \) in the standard sense (refer to Section 2.7) iff there is some fragment definition transition 
\[
(\sigma (e_R), \ldots, \sigma (e_{R'})) \in \Delta e_{R,R'} \quad \text{such that} \quad \exists \ R'' \in F \quad \text{where} \quad \sigma (e_{R''}) \notin (\sigma (e_R), \ldots, \sigma (e_{R'})).
\]

That is, \( N(R, R') \) is true if it is possible to start from a low level fragment definition of \( R \), make incremental changes, and eventually transition into \( R' \) without passing through another relation’s fragment definition.

**Example 7.4.8.** The relations \( R_x \) and \( R_y \) in Example 7.4.1 are conceptual neighbours according to this definition because there exists a transition class 
\[
\Delta e_{R,R'} \,(R, R') = \{(R_2, R_3, R_4, R_5, R_6), (R_4', R_3', R_2', R_1', R_2')\}\]
that contains a fragment definition sequence 
\[
\{(R_2, R_4'), \ldots, \{R_2, R_1'\}, \ldots, \{R_6, R_1'\}, \{R_6, R_2'\}\}
\]
which does not include any of \( R_z \)'s fragment definitions.

The problem with the standard conceptual neighbour definition is that two relations possibly being neighbours results in relations almost always being neighbours. Thus the standard definition is too weak for defining high level neighbourhoods. In order to develop more accurate neighbour definitions two conditions are imposed. The first condition restricts the set of paths considered for determining neighbour status, and the second condition groups paths together into equivalence classes.

7.4.3 Refining Conceptual Neighbours: Path Restrictions

The developer can avoid impractical and counter-intuitive neighbourhoods by restricting the set of paths used to determine whether two relations are neighbours. Two types of path conditions are *direct* path restrictions and *critical* path restrictions. The *direct path restriction* requires that low level neighbourhood transition sequences take a shortest path in \( \Delta G_{\hat{R}} \,(R_i, R_j) \). This ensures that all transitions monotonically approach the target fragment definition. Figure 7.6(a) illustrates the admissible paths with this restriction; \( R_x \) and \( R_y \) are no longer neighbours.

The *critical path restriction* requires that paths only include relations that are guaranteed to conflict between the high level fragment definitions of two relations. Intuitively, certain relations in the reference domain will be very important cues for interpreting higher-level concepts, while some (probably most) combinations will lie in the vast fragment definition space between these critical points.

**Example 7.4.9.** Comparing the fragment definitions of *waterscapes* and *fields* in Table 7.2, it can be observed that a *waterscape* image and a *field* image are guaranteed to differ in the qualitative relations between \((\text{clouds, water})\) and \((\text{clouds, sand})\); these are *critical* relations. Alternatively, the images will not necessarily differ in the qualitative relations between \((\text{clouds, trunks})\) and \((\text{sky, flowers})\); these are not critical relations.
Chapter 7. Designing High Level Application Fragments: Utilising Data-Based Metrics and Deriving High Level Neighbourhood Graphs

Figure 7.6: Refined fragment definition spaces. (a) direct path restrictions, (b) critical path restrictions, (c) equivalence class of direct paths, (d) equivalence class of critical paths in conjunction with a direct path restriction.

The necessarily conflicting relations are the low level relations that primarily distinguish between high level relations and thus the critical transition paths can be a useful indicator of conceptual neighbour status.

Example 7.4.10. As illustrated in Figure 7.5, one fragment definition of $R_x$ is $\{R_4', R_1\}$ and one fragment definition of $R_y$ is $\{R_1', R_7\}$. Therefore when transitioning from $R_x$ to $R_y$ it is possible that $R_4'$ is already satisfied and no transition through the neighbourhood of relations $R_1', \ldots, R_6'$ is required. In contrast, regardless of the $R_x$ fragment definition, a transition through the relations in $R_1, \ldots, R_8$ will always be required. Figure 7.6(b) illustrates the admissible transition paths through the critical paths where transitions are always required, and shows that $R_x$ and $R_y$ are no longer neighbours.

7.4.4 Refining Conceptual Neighbours: Transition Path Equivalence Classes

The following example demonstrates that the standard neighbour definition can give counter-intuitive results.

Example 7.4.11. As illustrated in Figure 7.5, the transitions required to get from a fragment definition of $R_x$ to a fragment definition of $R_y$ always include $\{(R_3, R_4), (R_4, R_5), (R_5, R_6)\}$. The transitions to get from a fragment definition of $R_x$ to a fragment definition of $R_y$ always include $\{(R_4, R_5), (R_5, R_6)\}$, which is a subset of $R_x$’s required transitions. Because $R_x$’s required transi-
tions are a proper subset, in some applications it might be intuitive to view $R_z$ as an intermediate relation between $R_x$ and $R_y$. However, the standard definition does not distinguish this special case of intermediate relations.

A developer can control these cases of intermediate relations by grouping a set of transition paths into an equivalence class. This has the effect of removing the ordering of transitions taken in the set of paths in the equivalence class. An alternative perspective is that, previously the neighbour test checked whether it was possible to avoid intermediate relations, whereas now it is checking the stronger condition whether it is guaranteed to avoid intermediate relations within the equivalence class of paths. The developer employs equivalence classes by selecting restricted sets of paths (otherwise the neighbour test will fail whenever there are three or more relations).

**Example 7.4.12.** Figure 7.6(c) illustrates defining the direct paths as an equivalence class. Figure 7.6(d) illustrates defining the critical paths as an equivalence class so that the conflicting paths must be guaranteed to be unobstructed, and at least one path through the non-conflicting paths must be unobstructed.

### 7.4.5 Ensuring Conceptual Neighbours are Symmetric

It is observed that neighbour status is no longer necessarily symmetric when the designer restricts the set of transition paths. Although asymmetric neighbourhood graphs can be employed for performing neighbourhood based QSTR tasks, a developer may require neighbour status to be symmetric. The following three variations on the neighbour definition ensure symmetry, ordered from strongest to weakest. Two relations can be defined as symmetric neighbours if:

- both directions are unobstructed (conjunction),
- either direction is unobstructed (disjunction), and
- no single intermediate relation obstructs both directions (restricted disjunction).

The first variant is the strongest, stating that both directions must be clear before the relations are considered neighbours. The second variant weakens this by only requiring one of the directions to be unobstructed. The third variant states that no single obstruction occurs in both directions, so that two high level relations are not neighbours if the fragment definition of a third relation obstructs both relevant transition paths; this is useful in cases where the designer wants the third relation to represent a guaranteed intermediate concept in between two relations.

### 7.4.6 Dealing With Multiple Fragment Definitions

If a high level relation $R$ has multiple fragment definitions $e_{1,R}, \ldots, e_{n,R}$ then some of the fragment definitions may permit $R$ to be a neighbour to some other high level relation $R'$, while other
fragment definitions do not. It may be the case that only a single pair of fragment definitions out of many possible pairs allows two relations to be neighbours, so that in practice it is unlikely that the two relations will be neighbours, and most transitions between them will be obstructed.

The approach of deriving neighbourhoods using fragment definitions facilitates an effective and intuitive solution to this problem by annotating probabilities to conceptual neighbours. This allows a designer to develop more accurate neighbourhood graphs that are consistent with the fragment interactions. If all fragment definitions of a relation \( R \) are equally likely to be used, the probability \( P \) of employing a particular fragment definition \( x \in \sigma(e_R) \) is \( P(x) = |\sigma(e_R)|^{-1} \) (otherwise probability function \( P \) is given, for example, having been derived from domain knowledge). The probability that two high level relations \( R, R' \in F_\alpha \) are neighbours, \( P(N_{F_\alpha}(R, R')) \), is the sum of the probabilities of selecting pairs of fragment definitions \( e_R, e_{R'} \) from each relation that are neighbours, \( \Sigma P(e_R) P(e_{R'}) \) for all \( R_i \in \sigma(e_R), R_j \in \sigma(e_{R'}) \) such that \( N_{F_\beta}(R_i, R_j) \).

### 7.5 Summary

This chapter has presented methodologies that support the design of application-specific custom relations and fragments. The methodologies for high level fragment design enable the developer to evaluate the efficacy of fragment definitions with respect to field data and domain-specific contextual information, and to utilise the hierarchical interactions between fragments for deriving high level conceptual neighbourhoods.

In QSTR applications, coupling is a measure of how effectively a set of low level relations distinguishes between different high level relations; cohesion is a measure of how effectively a set of low level relations defines a high level relation. Coupling and cohesion are calculated by using a dataset of exemplary objects (either in one large scenario or spread across a set of scenarios) that reflect the correct interactions between fragments, such as the detailed results of field studies. A variation of cohesion and coupling has been presented that measures the accuracy and precision of the fragment definitions as the average discrepancy and variance between inferred relations and expected relations. A clustering approach based on cohesion and coupling information is employed to derive neighbourhood graphs. To demonstrate this, a high level neighbourhood graph for image categories in the Image Retrieval QSTR application have been presented.

The machine learning technique of decision tree learning has been employed for generating fragment definitions from datasets, and deriving conceptual neighbourhoods based on the generated definitions. Decision trees succinctly express fragment definitions; each vertex in the tree is annotated with a reference domain relation and a state (i.e. holds, not holds, and not applicable) and each leaf is annotated with an abstraction domain relation and a state. A path from the root to a leaf represents a fragment definition, where tuples in the relation state at each vertex in the path are conjoined to be an improper subset of the tuples in the relation state at the
leaf. It has been demonstrated that a developer can employ decision tree learning to generate fragment definitions from field data, and then compare the resulting fragment definitions with their manually formalised fragment definitions.

Finally, this chapter has presented a methodology that enables developers to define and utilise high level neighbourhoods that are consistent with the structure of fragments. The standard definition of conceptual neighbours was shown to be ineffectively weak when employed in the wider context of QSTR applications; specifically, when using the standard conceptual neighbour definition two relations possibly being neighbours results in relations almost always being neighbours. Thus, a central focus of this chapter has been to extensively generalise the definition of conceptual neighbourhoods to enable the design of high level neighbourhoods that are consistent with the interactions between fragments in a QSTR application. In order to develop more accurate neighbour definitions two conditions are imposed. The first condition restricts the set of paths considered for determining neighbour status, and the second condition groups paths together into equivalence classes.
Chapter 8

Implementing QSTR Applications

8.1 Introduction

Once the application developer has produced a suitable QSTR system design as addressed in Chapters 5-7, they must then implement the application in software. Highly specialised QSTR reasoners such as SparQ [162] and GQR [70] have now been developed that rapidly compute path-consistent deductive closure and consistency checks for well known QSTR fragments such as Allen’s interval calculus and RCC. These should be employed to compute the networks of relations for low level fragments that represent the imported established QSTR calculi. However, these reasoners can not be used to implement the designers custom fragments that build on the lower level established QSTR calculi. This chapter presents approaches for implementing the general QSTR reasoning component of the application.

The chapter is organised as follows. Section 8.2 presents different types of implementation languages that the designer can use, particularly focusing on declarative and relational databases approaches. Section 8.3 presents approaches for implementing the representational components of QSTR application. In particular, the software must represent objects, relations, and scenarios, where a scenario requires representing that each relation either holds, does not hold, is not applicable, or is indefinite for each object tuple. Section 8.4 presents an approach for implementing a general reasoning algorithm that can process an incomplete scenario or provide a querying mechanism. Finally, Section 8.5 discusses the modification of the design in order to satisfy resource constraints imposed at the level of implementation, particularly memory limitations.

8.2 Languages and Architecture

Different implementation platforms provide a tradeoff between ease of implementation and control. This thesis considers two approaches to implementation that employ well known and
widely used languages, namely declarative languages and relational database systems. Other approaches that are not discussed in this thesis include employing ontology editors that have integrated reasoners, such as the Protege Ontology Editor and Knowledge Acquisition System [127] that facilitates the utilisation of reasoners such as RacerPro [79], for example, see [37].

In all cases, specialised QSTR reasoners such as SparQ [162] and GQR [70] should be used to process fragments of the application that employ established QSTR calculi such as Region Connection Calculus. Thus, QSTR applications employ a loose integration of QSTR calculi as defined in [19] (refer to Section 2.3 of this thesis). From a software engineering perspective, integrating QSTR calculi in a loose manner [19] is the most efficient approach with respect to the development resources required for application design and validation. In particular it enables modular software development, thus facilitating software reuse and extensibility [43, 93].

8.2.1 Declarative Languages: Prolog

The first-order logic specification of qualitative relations and constraints as defined in Section 3.2 can be implemented using general purpose declarative languages such as Prolog [38]. These languages provide an inbuilt query engine that the developer can use for their application’s reasoning. The developer can use these languages by implementing QSTR relations as predicates and QSTR relation states for object tuples as tuple assertions.

Figure 8.1 illustrates an architectural overview of the main components in a Prolog software implementation; rectangles represent the main components of the implementation including the Prolog reasoning system and arrows indicate information flow. The Prolog reasoning software stores relations in Prolog sessions and processes Prolog code. The reasoning planner issues the Prolog reasoner with predicates and assertions that encode scenarios and implement deductive closure and consistency checks. Once reasoning has been performed, any software application can then access the locally consistent spatial and temporal networks through the QSTR application’s main interface.

8.2.2 Relational Database Languages: SQL

The underlying set theoretical basis for qualitative systems described in Chapter 3 leads to a natural implementation in languages that also have a set theoretic definition such as relational databases [150]. The developer can use these languages by implementing QSTR relations as database relations (or tables) and QSTR relation states for object tuples as database relation tuples (or table entries). This requires expertise in areas such as database query optimisation and data storage. By working on a relational database platform the developer automatically

---

1The position taken in this thesis is that tightly integrated QSTR calculi, such as the QSTR calculus of topology and qualitative region sizes developed in [19], are developed by the QSTR community rather than QSTR application developers. A tightly integrated calculus is viewed in the same way as any standard QSTR calculus such as IA or RCC: as a modular low level fragment, available in SparQ or GQR, that can be utilised in a QSTR application.
benefits from internal database query optimisers, scalable architectures that support very large data volumes and efficient inbuilt mechanisms for storage and retrieval. Moreover, by directly operating on data in the database environment we avoid the need to adapt and convert data between different programming platforms, thus further reducing development cost and processing time.

Figure 8.2 illustrates an architectural overview of the main components in a database implementation; rectangles represent the main components of the implementation, the cylinder represents a database system and arrows indicate information flow. The database stores relations and processes SQL queries. The query planner issues the database with the SQL queries that implement deductive closure and consistency checks. While in many cases the planner will be a software program that automatically manages the execution of the deductive closure algorithms, the planner can also simply be a human that manually issues the SQL queries through a graphical or command-line database interface. Once reasoning has been performed, any software application can then access the locally consistent spatial and temporal networks.
8.3 Encoding Relations and Scenarios

This section presents an approach for implementing the representational components of QSTR applications using declarative languages and database systems. The representational components are relations, scenario objects, and object tuples that take relation states in scenarios.

For declarative languages, each QSTR relation is encoded as a set of predicates, one predicate for each allowable relation state. Note that one of the states can be encoded implicitly, so that if a tuple does not appear in the explicitly represented states then it must take the implicit state by default. For example, the following Prolog code implements two binary relations $R_1$ and $R_2$ that have the allowable states holds, not holds and indefinite.

```
:-dynamic rel_1/2, no_rel_2/2,
    rel_2/2, no_rel_2/2.
```

Scenarios are specified by asserting object tuples into the predicates. For example, the following Prolog code specifies a scenario with three objects $a$, $b$, and $c$ where $(a, b) \in R_1^+$ and $(b, c) \in R_2^-$. 

```
assert(rel_1(a,b)), assert(no_rel_2(b,c)).
```

In this example, the tuples $(a, a), (b, b), (c, c), (a, c), (b, a)$ and so on are indefinite with respect to $R_1$ and $R_2$ because they have not been explicitly assigned to any state.

Similarly for database systems, each QSTR relation is encoded as a database relation. For example,

```
rel_1( id1 INTEGER, id2 INTEGER, 
    PRIMARY KEY(id1, id2))
```

```
no_rel_1( id1 INTEGER, id2 INTEGER, 
    PRIMARY KEY(id1, id2)).
```

Scenarios are specified by populating the database relations appropriately.

8.4 Implementing General Reasoning

All QSTR tasks require some form of reasoning. As presented in Section 4.4, Chapter 4, the four basic QSTR tasks are querying, relaxed querying, consistency checking and computing deductive closure. To perform these tasks, the designer must implement constraints and develop reasoning software that can check the consistency of scenarios and execute rules to perform deductive closure. This section presents an approach for implementing each of these components that are required for QSTR application reasoning.
8.4 Implementing General Reasoning

8.4.1 Implementing Constraints

Qualitative domain rules that are used to infer new information from premise information are represented as QSTR constraints. In this section a uniform representation of constraints will be defined to facilitate constraint implementation in both database and declarative languages. The uniform constraint representation is based on the set-theoretic definition of QSTR scenarios presented in Chapter 3. It must be noted that, in general, any suitable first-order implementation preferred by the developer or required by the given QSTR application requirements can be employed.

Set expressions form new sets by collecting elements from existing sets according to specified conditions. The conditions used to build a set can be divided into three categories. For each tested element (or tuple of elements) a condition either allows the element into the new set, or rejects the element. It follows that each condition from a set expression either

- **triggers** the expression (a set is empty if these conditions are never satisfied),
- **overrides** the expression (even if the expression has been triggered, if these conditions hold then they stop an object from being admitted into the set), or
- is **independent** (has no effect).

Thus, set expressions take the general form

\[
\text{select tuple: exists tuple: trigger conditions and not exists tuple: override conditions.}
\]

For a set expression to be non-empty, it must have one or more triggers that are not overridden. A constraint is implemented as a comparison between two set expressions, e.g. \( X \subseteq Y \) where \( X, Y \) are set expressions that take the general form above.

As a programming construct this format has a number of desirable design properties. Most importantly, it is a modular construct that facilitates uniform analysis and testing by structuring the set expressions. Moreover, the construct is orthogonal as conditions can themselves contain nested selections, and it supports both bottom-up and top-down design as nested selections can be either well-developed units or unspecified stubs. Furthermore, the set expression structure is readily accessible to OO and database software developers. Firstly, the construct is simple and only requires a basic background knowledge of logic and set theory. Secondly, the set expression structure is syntactically similar to the portion of Structured Query Language (SQL) that is equivalent to first-order logic (i.e. the portion that excludes aggregates and arithmetic operations).
In general, constraints of this form are implemented by firstly gathering the tuples that satisfy the conditions of each expression and then comparing the two sets of tuples. For declarative languages, this can be accomplished by implementing the expression conditions, testing whether the constraint is satisfied, and if it is not, then updating the scenario. For example, the following Prolog code implements the constraint that binary relation $R_0$ is disjoint from the binary relations $R_1$ and $R_2$.

```
( (rel_1(A1,B1);rel_2(A1,B1)), % if either R1 or R2 holds
  \+no_rel_0(A1,B1), % and R0 is indefinite
  assert(no_rel_0(A1,B1))), % then set R0 to not hold.
```

This condition must then be executed for all matching tuples. In Prolog this can be accomplished by the `setof` predicate as follows.

```
updateRule_0:-
  setof(_, ((rel_1(A1,B1);rel_2(A1,B1)), \+no_rel_0(A1,B1), assert(no_rel_0(A1,B1))), L), % collect all results into set L
  \+(L=[]). % predicate is true if at least one tuple was asserted
% (i.e. predicate is true if the scenario was modified)
```

For relational query languages such as SQL this can be accomplished by directly implementing the constraints as queries. For example, the above constraint is implemented with the following SQL query.

```
INSERT INTO no_rel_0
SELECT * FROM rel_1
UNION SELECT * FROM rel_2
EXCEPT SELECT * FROM no_rel_0
```

### 8.4.2 Inference Rules and Deductive Closure

All QSTR tasks are a sequence of queries and model updates. In particular, qualitative inference (i.e. used for incomplete models) is an extension of a querying task, where the query result is fed back into the model. For example, four database operations are required to satisfy the following rule about temporal intervals:

\[
before_y \subseteq \bigcap_{z \in during_y} before_z,
\]

1. **Query:** Gather all elements in $before_y$. 

2. **Query**: Gather all elements in *during*.

3. **Query**: Gather all *before* sets associated with each element from result (2).

4. **Update**: Add elements from (1) to each set from (3).

The significance is that all QSTR tasks can be implemented as a sequence of queries and updates. This highlights the critical influence that query optimisation has on QSTR reasoning performance, as querying underlies every QSTR task performed by QSTR applications.

In general, whenever a QSTR scenario is modified, all of the constraints that refer to any of the modified relations must be checked. Thus, the designer needs to set up an inference procedure that executes inference rules, records the relations that were modified, and then repeat the execution of rules that have one of the modified relations in their domains. The following pseudocode outlines the general reasoning algorithm.

1. Maintain a **TODO** list of inference rules.
2. While the **TODO** list is not empty:
   a) Select the next rule on the list to be executed, and execute the rule.
   b) If the database was modified, determine the affected rules and add to the **TODO** list.

Each rule can only appear once in the **TODO** list, that is, the list is an ordered set. The two important subprocedures are determining the ordering of rule execution (step 2(a)) and identifying the affected rules (step 2(b)).

In step 2(a), determining the sequence of rule execution is an issue of optimisation. The aim of query optimisation, given a set of rules, is to determine the optimal rule execution sequence that minimises the number of executed rules in the worst case. Because the execution of a rule can modify the database, previously executed rules may need to be recomputed, i.e. reasoning is often non-linear. For further information the developer can refer to the related problem of variable and value ordering in constraint satisfaction [142] (chapter 5). The next step is to perform a finer level of optimisation by interleaving inference rule execution to minimise the number of actual database queries required. For this, the developer can refer to research in database optimisation [150].

In step 2(b), the **TODO** list can be updated in the following three ways. A simple brute force approach is to revisit every constraint whenever the scenario is modified.

1. If any of the rules modify the database, re-execute all rules.

This is the simplest approach for implementing a QSTR reasoner. It can be built very quickly and is thus ideal for implementing a simple reasoner at the application prototyping stage of development. Moreover, it is satisfactory for applications that do not require extensive reasoning, or only encode small scenarios. In Prolog, the `infer` predicate is recursively defined as follows.
infer :- ( updateRule_0; % update is true if scenario is modified
    updateRule_1;
    ... 
    updateRule_41 ), infer.
infer. % finished updating scenario, return true.

The next approach takes the relationship between constraints into account.

2. When constraint \( c \) executes, add all constraints that refer to any relation to \( c \).\(^2\)

This approach requires the designer to generate a relation constraint graph before reasoning can take place. Compared to the previous brute force approach, only adding those constraints that share relations reduces the number of constraints that are redundantly added back onto the \( TODO \) list, thus improving reasoning performance. Moreover, compared to the following approach, it does not require the reasoner to remember which relations were modified, only which constraints were executed.

A more focused approach tracks the specific relations that are modified and refers directly to the constraint domains to determine which constraints should be revisited.

3. When relation \( R \) is modified, add all constraints that contain \( R \) in their domain to the \( TODO \) list.

This requires the designer to firstly generate a reference lookup table that maintains the domains of each constraint. When a constraint executes, the modified relations are recorded and are used to determine the constraints that are added back to the \( TODO \) list. This approach only replaces those constraints that are directly associated with the modified relations, thus further reducing the number of redundantly replaced constraints. However, software that can track modified relations during reasoning and can revisit constraints based on their domains has a greater cost for development.

### 8.4.3 Checking Consistency

The final reasoning task is determining whether a scenario is consistent with the domain rules incorporated into the QSTR application design. Specifically, a QSTR application scenario is inconsistent if an object tuple takes no state for some relation, or more than one state for some relation. That is, if the following axiom is violated then the scenario is inconsistent,

\[
\forall R \in \mathbf{R} \cdot U^{\alpha R} = R^+ \triangle R^- \triangle R^\sim \triangle R^2.
\]

\(^2\)In Chapter 9 this notion is formalised as a Sentence Interaction Graph (SIG).
Reasoning can discover that a scenario is inconsistent if a rule is found that can not be satisfied by some tuple without violating the above axiom.

For declarative languages, each relation predicate must be declared to be a QSTR relation type, for example,

\[
\text{relation}(	ext{rel}_1, 2).
\]
\[
\text{relation}(	ext{rel}_2, 2).
\]

The consistency check must then be implemented by ensuring that, for each relation, no tuple takes more than one state.\(^3\) In Prolog, the built in predicate call can be used to accomplish this, for example,

\[
\text{relation}(P, 2),\ % \text{get the next relation type}
\]
\[
\text{atom_concat}(\text{no}_-, P, \text{NoP}),\ % \text{get the not holds state}
\]
\[
\% \text{check if any tuples satisfy both the holds and not holds states}
\]
\[
\text{call}(P, X, Y), \text{call}(\text{NoP}, X, Y).\]

This condition must be executed for all matching tuples. As previously shown, in Prolog this can be accomplished by the `setof` predicate,

\[
\text{invalid}(L) : -
\text{setof}([P, X, Y],
\quad (\text{relation}(P, 2), \text{atom_concat}(\text{no}_-, P, \text{NoP}),
\quad \text{call}(P, X, Y), \text{call}(\text{NoP}, X, Y)),
\quad L). \% \text{collect and return the invalid tuples.}
\]

For database systems, the essential test is a simple query that takes the intersection of tables that encode different states of the same relation. For example,

\[
\text{SELECT * FROM rel}_1
\text{INTERSECT}
\text{SELECT * FROM no}\_\text{rel}_1.
\]

A straightforward approach for setting up the required collection of queries is to build the query using another language such as PSP or Java. The query generating program must maintain a list of strings that encode the relation names. It builds the consistency checking queries by iterating over the relation names, substituting the next relation name into the query expressions, and appending the result to the final query.

\(^3\)If a tuple is not explicitly represented then it defaults to some state such as indefinite, thus there is no need to check whether the state for all tuples has been explicitly specified.
8.5 Resource Constraints

The QSTR application developer may need to be modified to accommodate resource constraints, such as limitations on the available memory for data storage or the requirement that reasoning executes within a specified amount of time. To accomplish this, the developer must reduce the amount of information used by the application model when describing a scene. Note that a single datum of information in a qualitative model expresses the state of a single relation for a single object, e.g. expressing that “the kitchen is near the bathroom” in software requires one datum.

The key to reducing the amount of data used in the model is to implicitly maintain certain relation states in cases when they can be inferred by other explicitly represented relation states. That is, the designer must avoid expressing redundant information in the model by forming and associating intermediate fragments. For example, the relation directed at specifies whether a light source is pointing towards a surface and occluded specifies whether a light beam is blocked from a surface. The designer can use these relations in combination for determining how bright a surface will appear. However, if the light source and the surface are in different rooms then, in the vast majority of scenarios, the value of occluded will not be applicable and directed at will be irrelevant and never used for reasoning. Thus, without interfering with the QSTR logic that performs the user’s task, the designer can introduce the relation in same room; if a light source and a surface are not together in a room then the value of occluded and directed at will not be explicitly represented in the model. This vastly reduces the amount of data needed to model the scenario, potentially by orders of magnitude.

Chapter 10 presents a metric called H-complexity which the developer can employ to assist in the modification of the QSTR application design to reduce the overall memory requirements. In particular, Section 10.3 presents principles for reducing the complexity of expressions by modifying parameters such as relation arity and the number of required query variables. In general, intermediate fragments introduce layers of qualitative information by grouping objects that share the values of particular relations. The result is that relation values for each object in the intermediate fragment do not need to be explicitly enumerated. In terms of implementation, this intermediate fragment approach is a form of query pre-processing, where the common aspects of query results are stored and retrieved rather than recomputed.\(^4\)

\(^4\)From a design perspective, producing intermediate fragments is identical to classing objects using relations and normal fragments. The difference is that the designer’s motivation for introducing a relation in the initial model design is to satisfy the functional requirements of the task (e.g. modelling the appropriate input and output information described in Chapter 4) whereas their motivation for intermediate fragments is to satisfy resource constraints.
8.6 Summary

In this chapter a number of strategies have been presented for implementing a QSTR application design in software. The approaches to implementation focus on the use of declarative languages and relational database systems for encoding relations and constraints, and implementing general reasoning procedures. QSTR calculi are combined with custom high level fragments using loose integration. A loose integration of QSTR calculi and custom fragments enables modular software development, and thus provides the most efficient approach with respect to the development resources required for application design and validation.

Constraints are implemented based on a general set theoretic form. The general form of constraints divides conditions into three categories based on whether they admit or reject tuples from a set; the conditions are referred to as triggers, overrides and independent. Although any logical form is satisfactory for implementing QSTR applications, this construct has a number of desirable design properties; the construct is modular, orthogonal, and syntactically similar to the portion of Structured Query Language (SQL) that is equivalent to first-order logic. It was observed that all QSTR tasks can be implemented as a sequence of queries and updates. These operations have been used to implement deductive closure and consistency check procedures.

Finally, principles for lowering the resources required to represent a given qualitative scenario have been presented. In particular, the developer can minimise the amount of redundant information represented in the model by employing intermediate fragments. The issue of scenario complexity is discussed further in Chapter 10 where a metric is presented that quantifies the expressiveness of a qualitative language.
Chapter 9

Test Classes and Test Quality Metrics for Validating QSTR Applications

9.1 Introduction

The aim of program validation in software engineering is to determine if the system is fit for its intended purpose [43] by explicitly evaluating the program in terms of its application context. Researchers in QSTR typically apply general first-order theorem provers (and higher) for system validation [169]. However, the use of theorem provers for application level validation is not practical in general. Firstly, applying theorem provers can be very manually intensive, and even expert logicians in the QSTR research field find the task non-trivial (e.g. refer to page 292 and Section 6.2 in [46]). Secondly, they require axioms for the logic which in many cases will not be available, making theorem provers impossible to use [131]. For example, particularly during the early stages of application design, software developers may need to rapidly encode informal qualitative domain knowledge with the intention of refining the logic later if necessary. Thus, a thorough axiomatisation would not be necessary or appropriate for many QSTR application development projects.

As with general knowledge-based systems [131], an effective QSTR application validation strategy will require a range of different techniques that includes both formal and empirical validation. The problem is that knowledge based and expert system validation is relatively immature compared to other stages of the design process such as knowledge acquisition and representation [128, 131], and thus validation is often an ad hoc process [81]. This is also the case in the QSTR research field, where the validity of reasoning in a qualitative calculus is typically established using informal visual representations of configurations [169].

This chapter presents a significantly different set of methodologies for QSTR application validation, inspired by research in software engineering and finite model theory. The following section presents an overview of the QSTR validation process. Sections 9.3 and 9.4 present
white-box testing approaches, and Sections 9.5 and 9.6 present approaches for measuring the efficacy of a given test suite.

**9.2 Overview of QSTR Validation Methodologies**

This section presents an overview of the QSTR validation process and QSTR validation methodologies. QSTR validation consists of the following three key steps, as illustrated in Figure 9.1.

1. Create a test suite consisting of black-box tests derived from the software requirements, and white-box tests using unit and integration testing approaches.

2. Run the test suite on the QSTR application and refine the application design in order to satisfy the tests.

3. Evaluate the efficacy of the test suite by measuring test coverage, mutation testing and H-complexity coverage. If the developer is not satisfied with the degree of confidence that the test suite can provide, then they must update the test suite and return to step 2. Otherwise, the validation process is complete.

Firstly the developer must create a test suite consisting of black-box and white-box tests. Black-box tests can be derived from the functional requirements of the application. For example, if the application is required to perform an envisioning task then the developer can test the application by ensuring that the envisioning use cases specified in the requirements are satisfied. The types of tasks that form the core of QSTR black-box tests are presented in Section 4.4. Other than the basic task types, QSTR black-box testing is similar to imperative black-box testing, and therefore will not be discussed in further detail.

White-box tests are derived from the structure and details of the QSTR application design. By incorporating white-box testing approaches, the validation methodology can be used iteratively during application development rather than being limited to a black-box post development validation tool. Sections 9.3 and 9.4 present a unit testing and an integration testing strategy adapted from standard software engineering practices. Unit testing exercises independent components of functionality by isolating constraint expressions. Once unit correctness has been established, integration tests ensure that the designer’s intention holds when the atomic components are combined and incorporated into a more complex model [43], specifically, by exercising the relationship between set expressions.

A critical step in the validation process is analysing the quality of the test suite to help the developer determine whether or not they believe that the application has been sufficiently exercised. Standard software engineering applies test coverage criteria and mutation testing for evaluating the efficacy of a set of tests. These approaches are adapted for QSTR applications in Sections 9.5 and 9.6. Furthermore, Chapter 10 presents novel test coverage metrics that
9.2 Overview of QSTR Validation Methodologies

Figure 9.1: Flow graph of the QSTR validation process.
have been developed specifically for relational software. If the developer deems the test suite coverage and mutation scores to be inadequate then further tests must be created and added to the test suite, and the developer must return to step 2 in the validation process.

9.2.1 Formal Definitions for QSTR Validation

This section presents a detailed formal definition of the central concepts in QSTR validation. The salient concepts are QSTR tests, QSTR test sets, QSTR testing approaches (or validation methodologies), running tests on applications, passing tests, and failing tests.

A QSTR test consists of

- a description of an initial scenario,
- a specification of a reasoning task (e.g. query, deductive closure, envision, and so on), and
- conditions that are expected to be true in the scenario produced by the reasoning task.

A test is run on a QSTR application by setting up the test’s initial scenario in the application, and then executing the test’s specified reasoning task. An application passes a test if all of the test’s conditions are satisfied after the test has been run, and fails otherwise.

Any test that does not contain tautological or contradictory expected results has the potential to detect design faults. That is, a test is potentially useful if there exists some scenarios that satisfy the expected results, and some scenarios that do not satisfy the expected results. These two groups of scenarios partition the design space into applications that produce either acceptable or unacceptable scenarios when performing the specified reasoning task on the test’s initial scenario.

Formally, test $t = (s_t, \omega_t, e_t)$ where $s_t$ is a premise scenario, $\omega_t$ is a reasoning task such as infer, and $e_t$ is an expected scenario. $T = \{t_1, \ldots\}$ is a set of tests. Given a QSTR application $\mathfrak{A}$, a testing approach $\mathfrak{T}$ will produce a set of tests $T, \mathfrak{T}(\mathfrak{A}) = T$. A test is run on an application, $t(\mathfrak{A})$, and the application either passes or fails the test, $t(\mathfrak{A}) \in \{\text{pass}, \text{fail}\}$. Specifically, the scenario is loaded into the application and the $\omega$ task is performed to produce a new scenario which is compared to the expected scenario,

$$
t(\mathfrak{A}) = \begin{cases}
\text{pass}, & \text{if } \mathfrak{A}^{\omega}(s_t) \supseteq e_t \\
\text{fail}, & \text{if } \mathfrak{A}^{\omega}(s_t) \nsubseteq e_t
\end{cases}
$$

Similarly, a test set $T$ is run on an application, $T(\mathfrak{A})$, by running each test $t \in T$ on $\mathfrak{A}$; an application passes a test set if it passes all tests in the test set,

$$
T(\mathfrak{A}) = \begin{cases}
\text{pass}, & \text{if } t(\mathfrak{A}) = \text{pass}, \text{ for all } t \in T \\
\text{fail}, & \text{if } t(\mathfrak{A}) = \text{fail}, \text{ for some } t \in T
\end{cases}
$$
9.3 Unit Testing

Unit testing aims to validate small components of a program by exercising isolated aspects of functionality in an independent way [43]. This makes large, complex and evolving software systems more practical and efficient to test, by decomposing functionality into small, manageable pieces.

Software engineering units typically focus on natural code modules. For example in OO software testing, units can be methods; this reflects the OO modelling practice of creating cohesive methods that reflect some modular aspect of functionality. It is not immediately clear what QSTR application component satisfies the properties of a unit of functionality, however in Section 5.3, the QSTR analogue to OO class methods (or operations) were defined as QSTR fragment constraints. Therefore, using the general constraint form presented in Section 8.4.1, QSTR application units are defined as the set expressions on the left hand and right hand sides of constraints.

A QSTR unit test is defined as a partial scenario representing premise information, and a set of expected inference results [146]. The domain of a unit test is the collection of relations and their particular tuple arguments in the unit set expression being tested. For example, given the constraint

$\{x_1 \mid (x_1, x_2) \in R_1^+ \land (x_3, x_2) \in R_2^-\} = \{x_1 \mid (x_2, x_1) \in R_3^+\}$,

one unit test for the LHS set expression is the partial scenario

$R_1^+ = \{(a, b) , (b, c)\}$

$R_2^- = \{(c, b) , (b, c)\}$

and the expected inference result (from executing the set expression on the LHS) is $\{a\}$, i.e. when $x_1 = a$, $x_2 = b$ and $x_3 = c$.

The unit testing methodology works by enumerating different conditions in a constraint; the enumeration of conditions is represented as a decision table. As presented in Section 8.4.1 each set expression consists of three types of conditions: triggers (exists clause), overrides (does not exist clause), and independents. Thus the conditions for tuple selection in a set expression can be listed in two decision tables, one for the exists clause that triggers the expression, and one for the does not exist clause that overrides the expression. Every scenario can be expressed as zero or more rows from each table, and conversely, an infinite number of scenarios exist for each combination of rows. Thus, decision tables are used to formally represent QSTR constraints by enumerating all logically distinct behaviours of set expressions on the lefthand and righthand side of constraints.

---

1The set theoretic constraint form is $X \delta Y$ where $X, Y$ are set expressions and $\delta$ is a set comparator.

2Note that in Chapter 3 an implicit condition was defined for all expressions that required variable values to be distinct, so that $x_1 \neq x_2$, $x_1 \neq x_3$, and $x_2 \neq x_3$. 

Chapter 9. Test Classes and Test Quality Metrics for Validating QSTR Applications

<table>
<thead>
<tr>
<th>(y \in \text{light})</th>
<th>((y, x) \in \text{in})</th>
<th>(y \in \text{warm})</th>
<th>\text{cond. class}</th>
</tr>
</thead>
<tbody>
<tr>
<td>holds</td>
<td>holds</td>
<td>holds</td>
<td>trigger</td>
</tr>
<tr>
<td>holds</td>
<td>holds</td>
<td>not holds</td>
<td>independent</td>
</tr>
<tr>
<td>not holds</td>
<td>not holds</td>
<td>not holds</td>
<td>independent</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(y' \in \text{light})</th>
<th>((y', x) \in \text{in})</th>
<th>(y' \in \text{warm})</th>
<th>\text{cond. class}</th>
</tr>
</thead>
<tbody>
<tr>
<td>holds</td>
<td>holds</td>
<td>holds</td>
<td>independent</td>
</tr>
<tr>
<td>holds</td>
<td>holds</td>
<td>not holds</td>
<td>override</td>
</tr>
<tr>
<td>not holds</td>
<td>not holds</td>
<td>not holds</td>
<td>independent</td>
</tr>
</tbody>
</table>

Table 9.1: Decision tables for conditions in the exists clause (upper) and not exists clause (lower) of the left hand side set expression in the example constraint.

Example 9.3.1. In the QtvLight application (refer to Section 2.8.1) the following constraint determines the apparent room colour temperature [148]: “If a room has at least one warm light, and does not have lights of any other temperature, then the room has a warm colour temperature”. The constraint is formalised as

\[
\{ x | \exists y \cdot \text{light}^+ (y) \land \text{in}^+ (y, x) \land \text{warm}^+ (y) \land \\
\neg \exists y' \cdot \text{light}^+ (y') \land \text{in}^+ (y', x) \land \text{warm}^- (y') \} \subseteq \text{warm}^+.
\]

Table 9.1 shows the decision tables for the left hand side (LHS) set expression that enumerate the different relation states and the condition class of each state combination.

Given a set expression, the QSTR unit validation methodology produces a collection of unit tests that sufficiently exercise the logic of the set expression. The methodology consists of the following activities.

1. Declare decision tables for representing scenario classes.
2. Merge decision tables to remove invalid scenario classes (Section 9.3.1).
3. Enumerate valid relation states.
4. Identify relevant scenarios for testing by employing equivalence classes (Sections 9.3.2 and 9.3.3).
5. Generate prototype scenario test instances.
6. Assign the expected test results.

Step 1 represents scenario classes using decision tables as a basis for defining unit tests as described above. Step 2 performs dimensionality reduction by eliminating contradictory scenarios, as presented in Section 9.3.1. The significance of this step is that the designer can apply local consistency criteria to merge particular decision tables and benefit from the reduced test
set size with no loss in validation quality. Step 3 is trivial. Step 4 is the most critical aspect of specifying a unit test plan, and is the main focus of a unit testing methodology; two equivalence classes are presented in Sections 9.3.2 and 9.3.3. Step 5 is trivial as it consists of defining sets for relations and populating them appropriately. Step 6 requires the designer to manually specify the correct test result. Steps 1 to 5 can be automated to reduce development time.

9.3.1 Merging Decision Tables

Rows between the decision tables may not be independent as some combinations of rows represent inconsistent or invalid scenarios. These invalid classes of scenarios can be excluded by merging the decision tables, which often dramatically reduces the size of a test set. Thus merging facilitates more practical and effective testing by limiting the focus to relevant scenarios. Three primary reasons for rejecting scenarios are as follows.

1. The principle of contradiction: for all propositions $P$, $\neg(P \land \neg P)$.

2. External constraints: a pair of propositions $P_1, P_2$ are restricted by constraint $c$.

3. A series of $k$ external constraints: a pair of propositions $P_1, P_2$ are ultimately restricted by constraints $c_1, \ldots, c_k$.

These three cases define classes of local consistency analogous to those in constraint satisfaction, namely node, arc and $k$-path consistency [116, 120]. For arc and $k$-path consistency to be valid, each constraint in the path must also be valid.\footnote{Indeed, merging can be automated by encoding the decision tables and constraints as a constraint satisfaction problem.}

**Example 9.3.2.** In Example 9.3.1, not all of rows between the decision tables are independent; if there exists $y$ such that $y \in \text{warm}^+$ then there must also exist $y'$ such that $y' \in \text{warm}^+$. Therefore the decision tables are merged so that all represented scenario are node consistent, based on the shared qualitative relations $\text{light}, \text{in}$ and $\text{warm}$. The dimensionality is now low enough to represent the table in matrix form as shown in Table 9.2. The variable $\hat{y}$ is used to represent both $y$ and $y'$. Every scenario can be expressed as some combination of zero or more cells in this matrix.

9.3.2 Equivalence Class: Decision Table Row Combinations

In general, testers aim to cover all system conditions by recognising that different cases fall into the same logically equivalent category, and that some categories are more important than others. Thus, designers achieve satisfactory unit coverage by exercising tests for at least one prototypical case from each category, for example, ensuring that the test set covers all decisions
Chapter 9. Test Classes and Test Quality Metrics for Validating QSTR Applications

and control paths in a unit of OO code. Adapting this to QSTR applications, this chapter establishes two equivalence classes for unit testing based on an analysis of set expressions. Deciding on the scenario equivalence classes is the most important step in producing a practical and effective unit test plan.

The first equivalence class states that scenarios resulting in the same combination of rows in the decision tables are equivalent, and thus at least one scenario for each row combination must be tested. This class distinguishes between a qualitative relation being empty, and having one or more objects, which is consistent with the intuitive view that qualitative concepts do not make numerical distinctions. This equivalence class prescribes at least one test for each combination of qualitative relation states in the conditions of a set expression, thus yielding a finite critical test set size of $2^n$, where $n$ is the number of rows.

Example 9.3.3. Referring to the decision table in Example 9.3.2, the critical boundary in this section prescribes at least one test for each combination of cells in the constraint decision table. Applying the equation $2^n$ for $n = 8$ gives a unit test set size of 256.

### 9.3.3 Equivalence Class: Interaction of Rows With Different Condition

The second equivalence class refines row combinations by distinguishing between the interaction of different condition types, where the effect of the interaction of rows is different to the effect of the rows taken individually. This equivalence class is based on the observation that, if a set of scenarios each result in a single row of the same condition type (such as trigger) then any scenario that results in some combination of these rows is equivalent to the set of individual scenarios. For example, after testing all trigger rows individually the designer can ignore pairwise (and higher) comparisons of trigger rows. This is because, if two conditions correctly select

<table>
<thead>
<tr>
<th>$\hat{y} \in \text{light: holds}$</th>
<th>$\hat{y} \in \text{warm}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>holds</td>
<td>holds</td>
</tr>
<tr>
<td>holds</td>
<td>trigger</td>
</tr>
<tr>
<td>not holds</td>
<td>override</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\hat{y} \in \text{light: not holds}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>holds</td>
</tr>
<tr>
<td>holds</td>
</tr>
<tr>
<td>not holds</td>
</tr>
</tbody>
</table>

Table 9.2: Two three-dimensional matrices representing the left hand side set expression of the architectural lighting constraint in Example 9.3.1.
an element when individually exercised, then there is no logical way that they can erroneously prevent that same element being selected when both conditions are true in the same scenario.\(^4\)

This important class of tests further reduces the critical test set size without decreasing the effective test coverage. The critical test set size is now

\[
1 + (t + o + i) + (t \cdot i + t \cdot o + o \cdot i) + (t \cdot o \cdot i),
\]

(9.1)

where \(t, i, o\) are the number of trigger, independent and override rows respectively, with an additional scenario that corresponds to zero rows (i.e. the ‘1’ in the formula). This defines complete boundary case unit testing for a set expression and thus provides an ideal target for logic validation.

**Example 9.3.4.** Referring to the decision table in Example 9.3.2, the critical boundary in this section only considers the interaction between different condition classes. Applying Equation 9.1 for \(t = 1, o = 1, i = 6\) gives a unit test set size of 28, which is an order of magnitude smaller than the 256 tests prescribed by the former equivalence class.

### 9.3.4 Refined Unit Testing

There are two problems with the given critical boundaries. Firstly, covering all of the resulting unit tests may not be practical as the number of possible tests explodes with an increase in the number of conditions. Let \(n\) be the number of decision table rows, \(n = t + o + i\). In the worst case\(^5\) \(t = o = i = \frac{n}{3}\), and the test set size from substituting into Equation 9.1 is in the order of \(n^3\),

\[
1 + n + \frac{n^2}{3} + \frac{n^3}{27}.
\]

Secondly, covering the huge number of unit tests with equal attention may obscure tests that are more relevant. A bias is required to determine which tests are more important than others.

To address this, the observation is made that set expression domains often have a number of qualitative relations that primarily support the interaction between constraints and other external modelling requirements, which is a property of integration testing rather than unit testing [43]. For example, the domain of the constraint in Example 9.3.1 contains the relations warm, light and in, giving 28 unit tests after merging. However, if this constraint was the only constraint in the model, then the whole system could collapse into a simple Boolean test,

\[
\text{roomIsWarm} = \exists y \cdot \text{warm}^+ (y) \land \neg \exists y' \cdot \text{warm}^- (y').
\]

---

4If two triggers are intended to be mutually exclusive then an override condition is used.

5Equation 9.1 is maximised when the product of the variables is maximised. The product of two numbers that have a difference of 2\(j\) is \((i + j)(i - j) = i^2 - j^2\). This is maximised when \(j=0\); it follows that the product of any numbers (that must sum to \(n\)) is maximised when they are equal.
Table 9.3: Decision tables for conditions in the exists clause (upper) and not exists clause (lower) of the relevant portions of the example set expression that separate triggers and overrides.

<table>
<thead>
<tr>
<th>y ∈ warm</th>
<th>cond. class</th>
</tr>
</thead>
<tbody>
<tr>
<td>holds</td>
<td>trigger</td>
</tr>
<tr>
<td>not holds</td>
<td>independent</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>y' ∈ warm</th>
<th>cond. class</th>
</tr>
</thead>
<tbody>
<tr>
<td>holds</td>
<td>override</td>
</tr>
<tr>
<td>not holds</td>
<td>independent</td>
</tr>
</tbody>
</table>

Table 9.4: One-dimensional matrix representing the example set expression after merging the decision tables in Table 9.3.

The collapsed system only models one room in a scenario, and every defined object is a light source located in that room. While this simplification might appear arbitrary, reasoning will produce the same result as the more complex system about whether a given room is warm. Any other simplification that only implicitly models warm will not always provide the same inference results.

Specifically, the test set can be refined by focusing on conditions that separate triggers and overrides. That is, when such a condition is true it plays a role in satisfying a trigger row in the decision table while having no effect on any override rows, and vice versa for conditions that support overrides. Conditions that support both triggers and overrides have no net effect on triggering or overriding a set expression and can be ignored in the refined test set.

**Example 9.3.5.** Referring to the QtvLight set expression in Example 9.3.1, the exists and not exists clauses share two relations, light^+ and in^+ (but they differ on warm^+ and warm^-). Therefore, if these relations are ignored then the set expression collapses into the predicate roomIsWarm as described above. Table 9.3 presents the decision table for the relevant portion of the constraint, and Table 9.4 merges the two tables to achieve node consistency. Before merging, this yields a refined unit test set size of $2^4 = 16$ using the first equivalence class where $n = 8$, and $1 + 4 + (1 + 2 + 2) + 2 = 12$ using the second equivalence class where $t = 1$, $o = 1$ and $i = 2$. After merging, the refined unit test set size is $2^2 = 4$ and $1 + 2 + (1 + 0 + 0) + 0 = 4$ using the first and second equivalence classes, respectively.

### 9.4 Integration Testing

Integration testing is used to validate the interaction between different program components [43]. An integration test for a QSTR application exercises pairs of expressions (from either one or two constraints), and consists of a set of inputs and a set of expected outputs, where
the domain is the collection of relations that appear in the set expressions being tested. QSTR integration testing is divided into two categories, firstly, exercising the interaction between two expressions in the same constraint, and secondly, exercising the interaction between two expressions in different constraints.

### 9.4.1 Interaction Between Expressions in The Same Constraint

The first type of integration testing uses scenarios that test different set comparators between the left hand and right hand side expressions in a constraint. There are eight basic relationships between sets, i.e. given two sets $X, Y$ then the relationship between $X$ and $Y$ can be any disjunctive combination of the following.

- $X = Y, X \subseteq Y, X \supseteq Y$ (interpreted in the usual set theoretic way).
- $X \cap \neq Y$ (partial overlap).
- $X \cap Y$ (X and Y are disjoint and non-empty).
- $X \subseteq Y$ (X is empty, Y is non-empty and thus $X$ is a degenerate subset of $Y$).
- $X \supseteq Y$ (X is non-empty, Y is empty and thus $X$ is a degenerate superset of $Y$).
- $X = \emptyset Y$ (both X and Y are empty and thus degenerately equal).

Hence, given a constraint that consists of two sets $X$ and $Y$ and a comparator $\delta$ then the designer must create at least eight scenarios, each satisfying one of the basic set relationship conditions above. The expected outcome of each test is that the constraint should be satisfied in exactly those scenarios where the set relationship is one of the disjunctive conditions in the constraint’s comparator $\delta$.

### 9.4.2 Interaction Between Expressions in Different Constraints

The second type of integration testing exercises the interaction between constraints. If two constraints share a relation in their domains then the two constraints can influence each other and this interaction needs to be exercised. Consider two expressions $e_1$ and $e_2$ that share a relation $R$, and assume that $R$ is a trigger condition for both expressions. Scenario conditions that are pertinent to their interaction are

- either $e_1$ or $e_2$ is empty,

- $e_1$ and $e_2$ are both non-empty, and rely on different tuples in $R$ in order to be non-empty, and

- $e_1$ and $e_2$ are both non-empty, and rely on the same tuples in $R$ in order to be non-empty.
The first class of scenarios exercises a lack of interaction between the two expressions due to one or both expressions being empty and thus incapable of interaction through $R$ by default.

Alternatively, the second class of scenarios exercises a lack of interaction due to each expression being satisfied by different tuples; that is, if tuple $t_1 \in R$ satisfies appropriate conditions in expression $e_1$ such that some tuple $t_x$ is permitted into the set described by $e_1$, then $t_1$ does not satisfy any combination of conditions in expression $e_2$ that allows some tuple $t_y$ into the set described by $e_2$.

Finally, the third class of scenarios is the critical case that captures the interaction between the expressions. This case applies when the same tuple actively influences both expressions causing those expressions to be non-empty. More formally, there is some tuple $t_1 \in R$ that satisfies appropriate conditions in expressions $e_1$ and $e_2$ such that some tuple $t_x$ is permitted into the set described by $e_1$, and some tuple $t_y$ is permitted into the set described by $e_2$. The relevance of the interaction is that if, for example, reasoning determines that in order to satisfy expression $e_1$ the relation $R$ must be changed, then this change may also impact expression $e_2$. This prescribed integration test exercises the critical point at which the two expressions interact.

The above integration tests apply when $R$ is a trigger condition for both $e_1$ and $e_2$. Conversely, $R$ may be an override rather than a trigger condition, in which case the scenario conditions that specify integration tests are

- either $e_1$ or $e_2$ is non-empty,
- $e_1$ and $e_2$ are both empty, and rely on different tuples in $R$ in order to be empty, and
- $e_1$ and $e_2$ are both empty, and rely on the same tuples in $R$ in order to be empty.

The final case is that $R$ is a trigger condition for $e_1$ and an override condition for $e_2$. The scenario conditions that specify integration tests are

- either $e_1$ is empty or $e_2$ is non-empty,
- $e_1$ is non-empty and $e_2$ is empty, and they both rely on different tuples in $R$ in order to meet these conditions, and
- $e_1$ is non-empty and $e_2$ is empty, and they both rely on the same tuples in $R$ in order to meet these conditions.

Using this integration testing strategy the developer can achieve a test suite that exercises key aspects of the interaction between constraints.

### 9.5 Test Coverage for Test Set Evaluation

Once the application is capable of passing all of the tests in the test suite, the developer’s next task in the validation process is evaluating the quality of the set of tests. The quality of a test
suite is the efficacy of the collection of tests at detecting software faults. In this chapter two widely employed software engineering approaches are adapted to the QSTR domain, namely test coverage and fault seeding.

The developer’s primary aim during validation is to produce a set of tests that can provide an adequate degree of confidence that the application is fit for purpose. In standard software engineering, the set of tests that can be executed on a typical software program (called the test space) is determined by the system inputs and outputs, and the system structure such as statements, decisions and control paths. Executing all possible tests is clearly impractical and thus software engineers employ methods that isolate critical subsets such as boundary checking, equivalence class partitioning, and cause-effect graphs [43].

One standard technique for identifying significant test classes is to measure the test coverage of a particular type of program component [173]. For example, the set of tests that execute every statement in a program at least once is typically considered to be a minimum coverage requirement for validation. Test coverage is also one of the most widely used validation approaches for developing general knowledge-based systems [81]. Therefore, this section adapts test coverage to QSTR applications based on a graph called the sentence interaction graph.

### 9.5.1 Sentence Interaction Graph

In standard software engineering, the formal behaviour of imperative programs is typically represented using the control flow graph (CFG) where vertices represent program statements and directed edges represent control flow [43]. The CFG is the basis of a number of software validation approaches, such as branch and statement coverage testing. A new graph called the Sentence Interaction Graph (SIG) is defined as the QSTR analogue to the CFG, and is used to formally visualise the interaction between application theories. The SIG provides a foundation for defining QSTR test coverage metrics that a developer can use to validate their application.

**Definition 9.5.1.** Let $G = (V, E)$ be the sentence interaction graph of a QSTR application $(R, \Theta)$, where each vertex $v \in V$ represents a sentence $t \in \Theta$. For each pair $t_1, t_2 \in \Theta$ there is an undirected edge $(t_1, t_2) \in E$ iff there exists some $R \in R$ such that there exists atomic expressions $e_1 \in t_1, e_2 \in t_2$ that both use $R$.

The edges in the SIG indicate necessarily mutual relation dependencies; if relation $R$ is modified in order to satisfy the sentence $t_1$ and $R$ also appears in $t_2$ then $t_2$ must be revisited.

**Example 9.5.2.** Following are a selection of qualitative rules for the subjective impressions elicited by particular lighting configurations (based on [61]):

1. If the room has cool light, bright even work-surface illumination and some perimeter emphasis then the room evokes clarity.
Figure 9.2: Sentence Interaction Graph (SIG) of a selection of domain rules for architectural lighting.

2. If the room has uniform perimeter emphasis and bright even work-surface illumination then the room is spacious.

3. If the room has bright, warm ambient illumination, and nonuniform perimeter emphasis then the room is relaxing.

4. If the room has low ambient lighting in the occupancy area and a bright perimeter then the room is intimate.

The SIG of a formalisation of the four sentences is illustrated in Figure 9.2.

9.5.2 Test Coverage Methodology Overview

This section presents an overview of the process for calculating QSTR test coverage. Figure 9.3 illustrates the process in a flow graph. The test coverage methodology initially requires the test suite and the QSTR application that are being analysed. The designer then selects which coverage metric $L$ to calculate; the following section presents six possible coverage metrics from which $L$ can be selected. Next, the SIG of the application is generated and used to determine the set $S$ of $L$-components that can be covered. For each test in the test suite, the developer must determine the test’s coverage of $L$-components (for example, by executing the test on the application) and then record the results in a variable $\text{covered}$. Finally, the $L$ coverage metric score is calculated as the proportion of covered $L$-components out of all $L$-components, $\frac{|\text{covered}|}{|S|}$.

9.5.3 Test Coverage Metrics

This section presents six test coverage metrics adapted from standard software engineering program coverage techniques. Analogous to statement execution and branch execution in a CFG, an SIG vertex is executed in qualitative scenario $s$ if the reasoning algorithm has modified $s$ in order to satisfy the vertex’s constraint. An edge between vertices $v_1$ and $v_2$ with relation type $R$ is executed if $v_1$ is executed with the modification of some relation of type $R$, causing $v_2$ to execute (at some future time). These definitions are now used to adapt standard coverage measures for the SIG.
Figure 9.3: Overview of the process of calculating test coverage.
Chapter 9. Test Classes and Test Quality Metrics for Validating QSTR Applications

Figure 9.4: Partial ordering of coverage criteria from weakest to strongest (if full coverage is achieved).

**Vertex (statement).** Proportion of SIG vertices executed in at least one test.

**Edge (branch).** Proportion of SIG edges executed in at least one test.

**k-path (path).** Proportion of SIG paths of length \( k \) executed in at least one test (for \( k > 2 \)).

**Condition (decision).** Proportion of modification combinations resulting in execution, exercised in at least one test (applied to either vertices, edges, or paths).

Condition coverage refers to the observation that when a constraint involves a number of relations there are multiple ways in which the vertex could be executed. Similarly, when a constraint involves multiple relations of the same type there exist multiple ways for its edges to execute. Figure 9.4 illustrates the partial ordering of the coverage measures in terms of strength.

**Example 9.5.3.** Let the domain being tested contain four binary relations \( R_1, \ldots, R_4 \) that can take three states \( A_R = \{ +, -, ? \} \), and three constraints between the relations,

\[
\begin{align*}
(A) \quad & R_2 \text{ is the transitive of } R_1: \{(x,z) \mid (x,y) \in R_1^+ \land (y,z) \in R_1^+ \} = R_2^+. \\
(B) \quad & R_3 \text{ is asymmetric: } \{(x,y) \mid (x,y) \in R_3^+ \land (y,x) \in R_3^- \} = \emptyset. \\
(C) \quad & R_4 \text{ is the transitive of } R_2 \text{ to } R_3: \{(x,z) \mid (x,y) \in R_2^+ \land (y,z) \in R_3^+ \} = R_4^+.
\end{align*}
\]

The SIG of this application is illustrated in Figure 9.5. An SIG vertex is executed in qualitative scenario \( s \) if the reasoning algorithm has modified \( s \) in order to satisfy the vertex’s constraint. For the first constraint this means that the premise scenario of some test has either (or both)

\[
\{(x,z) \mid (x,y) \in R_1^+ \land (y,z) \in R_1^+ \} \supset R_2^+, \text{ or}
\]

\[
\{(x,z) \mid (x,y) \in R_1^+ \land (y,z) \in R_1^+ \} \subset R_2^+.
\]

Table 9.5 presents the unique vertex, edge and path execution sequences with example scenarios that produce the required execution sequence when inference is applied; any unspecified object relations are *indefinite*, e.g. \((a,b) \in R_1^-\), by default.

A key observation about coverage metrics is that stronger coverage does not imply a superior testing strategy [82]. The more relevant information is the *density* of useful fault detecting tests.
### 3-path

<table>
<thead>
<tr>
<th>metric</th>
<th>constraint</th>
<th>term modified causing execution</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex</td>
<td>(A) $R_2$ transitive $R_1$</td>
<td>$(x,y) \in R_1^+ \rightarrow (x,y) \in R_1^+$</td>
<td>$(a,c)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(y,z) \in R_1^+ \rightarrow (x,y) \in R_1^+$</td>
<td>$(a,c)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(a,b)$</td>
<td>$(a,c)$</td>
</tr>
<tr>
<td></td>
<td>(B) $R_1$ asymmetric</td>
<td>$(x,y) \in R_1^+$</td>
<td>$(b,a)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(y,z) \in R_1^+$</td>
<td>$(a,b)$</td>
</tr>
<tr>
<td></td>
<td>(C) $R_4$ transitive $R_2$ to $R_3$</td>
<td>$(x,y) \in R_4^+$</td>
<td>$(b,c)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(y,z) \in R_4^+$</td>
<td>$(a,c)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(a,b)$</td>
<td>$(a,c)$</td>
</tr>
<tr>
<td>edge</td>
<td>(A) $\rightarrow (A)$</td>
<td>$(x,y) \in R_1^+ \rightarrow (x,y) \in R_1^+$</td>
<td>$(c,d)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(y,z) \in R_1^+ \rightarrow (y,z) \in R_1^+$</td>
<td>$(a,c)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(y,z) \in R_1^+ \rightarrow (y,z) \in R_1^+$</td>
<td>$(b,d)$</td>
</tr>
<tr>
<td></td>
<td>(A) $\rightarrow (C)$</td>
<td>$R_1^+ \rightarrow (y,z) \in R_1^+$</td>
<td>$(a,b)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(y,z) \in R_1^+ \rightarrow (y,z) \in R_1^+$</td>
<td>$(a,d)$</td>
</tr>
<tr>
<td></td>
<td>(C) $\rightarrow (A)$</td>
<td>$R_1^+ \rightarrow (y,z) \in R_1^+$</td>
<td>$(a,b)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(y,z) \in R_1^+ \rightarrow (y,z) \in R_1^+$</td>
<td>$(a,c)$</td>
</tr>
<tr>
<td></td>
<td>(B) $\rightarrow (C)$</td>
<td>$(x,y) \in R_1^+ \rightarrow (y,z) \in R_1^+$</td>
<td>$(b,a)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(x,y) \in R_1^+ \rightarrow (y,z) \in R_1^+$</td>
<td>$(c,b)$</td>
</tr>
</tbody>
</table>

### 4-path

<table>
<thead>
<tr>
<th>metric</th>
<th>constraint</th>
<th>term modified causing execution</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex</td>
<td>(A) $R_2$ transitive $R_1$</td>
<td>$(x,y) \in R_1^+ \rightarrow \ldots \rightarrow (x,y) \in R_1^+$</td>
<td>$(e,f)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(y,z) \in R_1^+ \rightarrow \ldots \rightarrow (y,z) \in R_1^+$</td>
<td>$(a,b)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(y,z) \in R_1^+ \rightarrow \ldots \rightarrow (y,z) \in R_1^+$</td>
<td>$(a,c)$</td>
</tr>
<tr>
<td></td>
<td>(A) $\rightarrow (A)$</td>
<td>$(x,y) \in R_1^+ \rightarrow \ldots \rightarrow (x,y) \in R_1^+$</td>
<td>$(c,d)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(x,y) \in R_1^+ \rightarrow \ldots \rightarrow (x,y) \in R_1^+$</td>
<td>$(c,e)$</td>
</tr>
<tr>
<td></td>
<td>(A) $\rightarrow (C)$</td>
<td>$(x,y) \in R_1^+ \rightarrow \ldots \rightarrow (x,y) \in R_1^+$</td>
<td>$(b,d)$</td>
</tr>
</tbody>
</table>

---

**Figure 9.5:** Sentence interaction graph of the example application.

**Table 9.5:** Adapted coverage criteria for the example application.
within a coverage class. This is because testing is conducted with limited resources, and hence the designer can only take a sample of tests from large or infinite test classes. Broadening a test class without proportionally increasing the number of sampled tests will reduce the chance of selecting a test from the original smaller class. If the initial smaller class has a higher density of useful tests compared to the class extension then the overall probability that a useful test is selected decreases. An investigation conducted in Chapter 11 suggests that condition testing coverage classes are less dense than the non-condition counter parts. Further study is thus required by the QSTR community to determine the types of faults QSTR application designers are more likely to experience. Testing can then focus on sampling from coverage classes that are shown to have a high density of fault detecting tests. This issue is addressed further in Chapter 11.

9.6 Fault Seeding for Test Set Evaluation

Fault seeding is the validation approach of intentionally inserting faults into a piece of software (either manually or automatically, referred to as mutation testing [21, 42, 54, 83]) and executing a test set over the erroneous software [43]. The number of seeded faults that were caught by the test set can then be used to evaluate the quality of the test set, and gives the developer a measure of confidence that their test set is sufficiently effective at detecting faults. In imperative software, fault seeding and, in particular, mutation testing has been shown to be an effective validation technique compared to other data-flow test coverage strategies [21, 25, 42, 163, 171]. Because mutation testing has been shown to be a very useful validation approach for imperative software, it will now be adapted for QSTR applications. Furthermore, adapting fault seeding to QSTR applications provides a statistical validation approach that complements the other white-box test class techniques presented in this chapter.

The following section presents the mutation testing methodology for QSTR applications. Section 9.6.2 presents the methodology for selecting the types of faults that the application will be seeded with, called mutation operators. The final sections present six specific QSTR mutation operators adapted from research in imperative software engineering.

9.6.1 QSTR Mutation Testing Methodology

This section presents the process that application developers must perform for QSTR mutation testing, as illustrated in Figure 9.6. Let \( A \) be the QSTR application being tested, and let \( T \) be a set of tests used to validate \( A \). Each test in \( T \) either passes, indicating that no fault was identified, or fails, indicating that a fault was identified. Assume that all tests in \( T \) pass when executed on \( A \), that is, \( T(A) = \text{pass} \). Mutation testing is used to determine the effectiveness of \( T \) at identifying faults in \( A \).
Having selected mutation operators $O$, the application developer conducts mutation testing with the following steps.

1. Apply mutation operators $O$ on $\mathfrak{A}$ to generate a set $M$ of faulty versions of $\mathfrak{A}$, called \textit{mutants}, written $O(\mathfrak{A}) = M = \{m_1, \ldots\}$.

2. Execute test case $T$ on each mutant $m$ in $M$; if any of the tests fail then the mutant’s fault has been detected (referred to as \textit{killing} the mutant).

3. Use the results from Step 2 to calculate the \textit{mutation score}; this measures the effectiveness of $T$ at detecting faults in $\mathfrak{A}$.

In step 1 a set of faulty versions of $\mathfrak{A}$ are automatically generated. Each faulty version is called a \textit{mutant}, and $M$ is the set of mutants. A mutant is generated by applying a \textit{mutation operator} on $P$, where each operator makes a single change that produces one mutant version of $P$. For example, in imperative software the mutation operator $\text{abs} ([53], \text{refer to Table 1, page 148})$ generates mutants by changing an expression $x$ to $\text{abs}(x)$, $-\text{abs}(x)$ and $z\text{push}(x)$ (which throws an exception when $x = 0$);\textsuperscript{6} thus $\text{abs}$ generates three mutants whenever an expression appears in a statement.

Next, the test set $T$ is executed on each mutant in $M$. If a test set contains at least one test that fails on a mutant, then that mutant is referred to as being \textit{killed} by the test set; this means that the test set was capable of identifying the seeded fault in $P$. The proportion of killed mutants, called the \textit{mutation score}, is a measure of the effectiveness of $T$ at detecting faults in $P$,

\[
\text{mutation score}(T) = \frac{\text{number of mutants killed by } T}{\text{number of (unique, non-equivalent) mutants, } |M|}.
\]

Note that every mutant in $M$ must not be equivalent to $P$ and must be unique (i.e. no two mutants are equivalent). Thus, once a mutation operator has been applied, the modified program must be compared to $P$ to ensure that they are not equivalent, and must be compared to previously generated mutants in $M$ to ensure that it is unique.

\subsection*{9.6.2 Selecting and Evaluating QSTR Mutation Operators}

A key issue when conducting mutation testing is determining whether the particular set of mutation operators reflect realistic fault types in QSTR applications. Note that this question has only recently been adequately addressed in the OO and imperative research literature [25]. Assessing whether mutation operators reflect realistic faults requires a set of subject program pairs, one version with real known faults and another clean version with the faults removed. No such repository of QSTR applications exists, and thus directly assessing the efficacy of a collection of QSTR mutation operators is not possible at this time.

\textsuperscript{6}The purpose of $z\text{push}(x)$ is to check if a testset contains a test where $x = 0$. When $x = 0$, $z\text{push}(0)$ throws an exception causing the mutant program to fail the test.
Chapter 9. Test Classes and Test Quality Metrics for Validating QSTR Applications

Figure 9.6: Overview of the process of calculating mutation score.
Although the QSTR mutation operators can not currently be directly evaluated, the results of the studies on the realism of imperative mutation operators are promising [25]. It was shown that mutation testing reflects realistic faults very well, and indeed better than manually seeded faults where humans tend to seed extremely difficult faults that are too complex and subtle to be realistic. This suggests that adapting the imperative mutation operators is the most reasonable approach until further information about QSTR applications is available.

Another key issue is the standard mutation testing assumption that if a test suite can detect simple errors then it is also capable of detecting complex errors, known as the coupling effect hypothesis [54]. Empirical evidence has been presented that supports the coupling effect hypothesis between simple and complex errors in imperative software [129]. The critical question for the QSTR community is whether the coupling effect also occurs in QSTR application software, however, without a repository of QSTR applications annotated with real design faults this question cannot be answered.

Despite this, it is feasible that the logic underlying mutation testing also holds for QSTR applications. For example, another perspective on mutation testing is that software designers typically develop programs that are *almost* correct, and thus the required test suite must be refined enough to detect small discrepancies between an erroneous program and the ideal program [54], which is not unreasonable to accept for QSTR application designers until further information is available. Yet another way to describe the logic behind mutation testing is that by testing for simple faults explicitly, an implicit test for more complex faults is also occurring [129]; thus a collection of simple detected faults provides a designer with “hints” about more complex faults [54]. Once again it is not unreasonable to assume for the present that any complex QSTR application faults should manifest at some point in the design as more simple faults.

Table 9.6 summarises the adaptations of imperative mutation operators for QSTR applications. In [171] mutation testing with *abs* and *ror* operators were shown to be superior to data flow coverage metric *all-uses*. In [95] Kim *et al.* presented OO specific mutation operators that will also be adapted to QSTR applications. The following sections present the QSTR mutation operators in detail.

### 9.6.3 *abs* Mutation Operator

The *abs* operator forces the sign of a numeric expression to be either positive or negative, or causes the mutant program to throw an exception when the expression’s value is 0 [53, 171]. The aim is to ensure that different regions of the input domain are sufficiently exercised. The equivalent QSTR mutation operator replaces the state of a relation, for example, from *holds* (+) to *does not hold* (−) or *not applicable* ( ). This exercises the QSTR input domain in an
Table 9.6: Adapting mutation operators for imperative and OO software to QSTR applications.

### Imperative Mutation Operator

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs</code></td>
<td>Force the sign of a numeric variable to be either positive or negative. E.g. <code>x</code> becomes <code>abs(x)</code> or <code>−abs(x)</code>.</td>
</tr>
<tr>
<td><code>ror</code></td>
<td>Replace numerical relational operators, E.g. <code>x &lt; y</code> becomes <code>x ≤ y</code> or <code>x ≡ y</code> or <code>x ≠ y</code>, etc.</td>
</tr>
<tr>
<td><code>CRT</code> and <code>ICE</code></td>
<td>Replace object declared types and instantiated types with compatible classes. E.g. <code>S s = new S();</code> becomes <code>T s = new S();</code> or <code>S s = new U();</code></td>
</tr>
</tbody>
</table>

### QSTR Mutation Operator

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs</code></td>
<td>Replace the sign of a relation. E.g. <code>near⁺</code> becomes <code>near⁻</code> or <code>near⁻</code>.</td>
</tr>
<tr>
<td><code>ror</code></td>
<td>Replace the constraint set expression comparator. E.g. <code>⊂</code> becomes <code>⊆</code> or <code>=</code>, etc.</td>
</tr>
<tr>
<td><code>CRT</code> and <code>ICE</code></td>
<td>Replace relations with their generalisations and specialisations. E.g. <code>before⁺</code> becomes <code>older than⁺</code>.</td>
</tr>
</tbody>
</table>

### Method overloading

<table>
<thead>
<tr>
<th>Method overloading</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manipulate parameters in overloaded method declarations and arguments in their invocations.</td>
<td>a) Manipulate edges of neighbourhood graphs. E.g. swap neighbours of <code>before⁺</code> and <code>meets⁺</code>. b) Replace relations with neighbours. E.g. <code>before⁺</code> becomes <code>meets⁺</code>.</td>
</tr>
</tbody>
</table>

### Explanation

Exercise different regions of the input domain.

Exercise the boundary between valid and invalid input.

Exercise the boundary between valid types and compatible but invalid types.

Exercise transitions between similar relations.

Table 9.6: Adapting mutation operators for imperative and OO software to QSTR applications.

An equivalent manner to a numerical input domain. If the state of every relation that appears in a set expression is replaced by the remaining two states then the number of mutants generated is

\[ 2 \times \text{number of specified relations}. \]

#### 9.6.4 ror Mutation Operator

The `ror` operator replaces numerical relational operators, such as `≤`, with other relational operators [53, 171]. The aim is to ensure that the boundary between acceptable input and unacceptable input is exercised [171]. The equivalent QSTR mutation operator replaces the set comparator in constraints with an alternative set comparator.

Given that there are \(2^8 - 1 = 255\) different set comparators, replacing the set comparator with every other possible comparator in every constraint will lead to an intractable number of mutants to process, i.e. \(254 \times \text{number of constraints}\). This exhaustive approach is unnecessary, and instead the designer can apply a smaller, more focused collection of mutants. To identify the most critical collection of set comparators to test, the following two factors for relation operators are considered; the designer must test for both inclusive correctness, i.e. ensuring that valid input is accepted, and exclusive correctness, i.e. ensuring that invalid input is rejected.

Firstly, inclusive correctness is established by testing comparators that omit the correct comparator. If the set comparator is a singleton, then one mutant is generated by replacing the set comparator with another randomly chosen singleton. If the set comparator consists of a disjunction of \(n\) basic comparators (e.g. `⊂` is equivalent to `⊂ ∨ =`, where `n = 2`) then \(n\) mutants
are generated, each with a set comparator that omits one of the disjoined comparators (e.g., in
the previous example two mutants will be generated, one with $\subseteq$ and the other with $\subset$). If an
inclusive test fails, then it means that valid input was erroneously rejected; these mutants should
cause inclusive tests to fail. If the test set cannot detect these mutants then the test set has given
a false positive for inclusive correctness. Thus, these mutants ensure that each valid component
of the comparator is exercised by the test set.

Secondly, exclusive correctness is established by testing comparators that contain the correct
comparator along with other incorrect comparators. If the set comparator consists of a disjunc-
tion of $n$ basic comparators then $8 - n$ mutants should be generated, each with a set comparator
that includes one additional erroneous comparator (e.g., the comparator $\subseteq$ will yield 6 mutants,
each with one of $\subseteq \lor \supset, \subseteq \lor \subset, \subseteq \lor \cap, \subseteq \lor \cup, \subseteq \lor \cap \neq$). If an exclusive test
fails, then it means that invalid input was erroneously accepted; these mutants should cause ex-
clusive tests to fail. If the test set cannot detect these mutants then the test set has given a false
positive for exclusive correctness. Thus, these mutants ensure that each invalid component of
the comparator is exercised by the test set.

This approach yields a very practical number of mutants to process per constraint. Let $n$ be
the number of disjoined set comparators for a constraint $c$. Inclusive correctness requires $n$ mu-
tants to be generated for the constraint, and exclusive correctness requires $8 - n$ mutants. Thus,
each constraint requires $n + 8 - n = 8$ mutants, an order of magnitude smaller than exhaustive
testing with 254 mutants.

### 9.6.5 CRT and ICE Mutation Operators

The compatible reference type (CRT) replaces an object’s declared type with a compatible class
type according to class inheritance hierarchies [95], for example

```java
S s = new S();
```

is replaced with

```java
T s = new S();
```

where $T$ is a super class of $S$, or an interface implemented by $S$. The instance creation expression
(ICE) replaces an object’s instantiated class type [95], for example

```java
S s = new S();
```

is replaced with

```java
S s = new U();
```

where $U$ is a subclass of $S$. The aim of CRT and ICE is to ensure that the subtle differences
between compatible types are exercised [95]. The equivalent QSTR mutation operator replaces
relations with generalisations and specialisations, respectively (QSTR generalisations and specialisations are defined in Sections 5.3.2 and 5.3.1). This exercises the distinction between relations that represent a similar concept but at different levels of granularity. The number of mutants generated is

\[ \sum_{R \in R} (|R\text{ generalisations}| + |R\text{ specialisations}|) \times \text{[number of times } R\text{ specified].} \]

9.6.6 Method Overloading: NEM and NR Mutation Operators

Four mutation operators are presented in [95] for methods that share the same name (i.e. overloaded methods). The four operators eliminate methods, manipulate the parameters in method declarations, and manipulate arguments in method invocations. The aim is to ensure that methods with similar properties to the intended method (such as method name and parameter types) are not being erroneously called. Two equivalent QSTR mutation operators are defined that use neighbourhood graphs to determine the similarity between relations.

The first QSTR mutation operator manipulates the edges of neighbourhood graphs by either swapping the edges of relations in the graph (e.g. exchanging the neighbours of before with the neighbours of meets), or adding and removing edges (e.g. assigning before and overlaps as neighbours). This operator is called \textit{neighbourhood edge manipulation} (NEM). \textit{NEM} exercises the transition between relations, corresponding to a manipulation of method declarations (parameter order change and overloading method removal in [95]).

The number of mutants generated using \textit{NEM} is calculated as follows. Given a neighbourhood graph, the number of possible ways to manipulate the graph is equal to the number of possible variations of the edges. If there are \( n \) vertices then the maximum number of undirected edges is \( \frac{n(n-1)}{2} \) and hence the total number of graph variations is the total number of neighbour combinations, \( 2^{\frac{n(n-1)}{2}} \). This yields an intractable number of mutants, even for one graph (i.e. a single application fragment); for example, Allen’s interval calculus has 13 relations giving \( 2^{78} \) possible graph variants. To reduce the number of mutants, the following two refinements that focus on exercising pairwise transitions are employed.

- Swap the edges of neighbours only; the number of mutants is less than\(^7\) or equal to the number of edges, \( n \).

- For each pair of relations, invert neighbour status (one inversion per mutant); the number of mutants is equal to the number of pairwise comparisons, \( \frac{n(n-1)}{2} \).

The mutants generated from these two approaches are distinct from each other (i.e. the graph variants do not overlap between the two approaches). To prove this, observe that after swapping two vertices in a graph the number of edges has not changed. Alternatively, inverting neighbour

\(^7\)If two vertices have the same set of edges then swapping the edges will not produce a new graph, thus the number of mutants may be less than \( n \).
status means that either one new edge has been added, or one existing edge has been removed. Thus, the two approaches produce distinct mutants from each other with unique graph variants. A tractable number of mutants is generated per neighbourhood graph, i.e. at most $n + \frac{n(n-1)}{2}$.

The second QSTR mutation operator replaces relations that appear in constraints with their conceptual neighbours. This operator is called neighbour replace (NR). $NR$ exercises the erroneous invocation of relations that are similar to the intended relation, corresponding to a manipulation of method invocations (argument order change and argument number decrease in [95]). If every relation that appears in a set expression is replaced by all of its immediate neighbours (one neighbour per mutant), then the number of mutants generated is

$$\text{[average number of neighbours per specified relation]} \times \text{[number of specified relations]}$$

where the average number of neighbours per relation is

$$\frac{\text{[total number of edges of specified relations]}}{\text{[total number of vertices of specified relations]}}.$$ 

9.6.7 Remove Relation

A particularly difficult and insidious fault for white-box based testing strategies to detect is the complete elimination of a disjunct component in an expression. The problem is that the expression can be satisfied without referring to the eliminated relation. For example, given the expression

$$\{x \mid x \in R_1 \lor x \in R_2\},$$

let the relation replace operator mutate the expression into

$$\{x \mid x \in R_1\}.$$ 

A white-box testing strategy will develop tests by referring to the contents of the expression. For example, a testing strategy may produce the following two tests.

- test 1: $R_1 = \{a\}$, expected expression result: $\{a\}$
- test 2: $R_1 = \{\}$, expected expression result: $\{\}$. 

The problem is that a testing oracle would not disagree with the expected results of the test. Indeed, the only test that is capable of detecting the fault is one that manages to identify and isolate the eliminated relation,

- test: $R_2 = \{a\}$, expected expression result: $\{\}$. 

The testing oracle will disagree with the expected result of the test and the fault will be detected.

The remove relation mutation operator eliminates one relation from each expression per mutant. The number of mutants generated is

$$\text{[number of specified relations]}.$$
9.7 Summary

This chapter has presented the QSTR validation process methodology. The process consists of creating a test suite consisting of black-box and white-box tests, using the test suite to refine the application design, and then quantifying the efficacy of the test suite to determine when to terminate the testing process.

In particular, two well known white-box testing methodologies have been adapted from software engineering, namely unit testing and integration testing, that support a developer in creating an appropriate test suite. QSTR units are defined as expressions on the left-hand and right-hand side of constraints as these have been shown to be natural atomic QSTR design modules that allow the testing of isolated aspects of QSTR application functionality in an independent way. The most crucial components of the QSTR unit testing methodology are two scenario equivalence classes defined based on a decision table representation of scenarios. QSTR integration tests exercise the interaction between pairs of expressions. Two main types of integration tests have been defined, firstly, tests that exercise the interaction between two expressions in the same constraint, and secondly, tests that exercise the interaction between two expressions in different constraints.

Furthermore, two methodologies for analysing the efficacy of a test suite at detecting faults have been adapted from software engineering. Six test coverage metrics have been defined using an analogue to the control flow graph called the sentence interaction graph (SIG). SIG vertices represent application constraints and undirected edges represent shared relations that occur in the domains of two constraints. Analogous to statement and branch execution in imperative software, the key concepts of expression execution (vertex) and sequences of expression executions (edge and $k$-path) have been defined as a basis for the coverage metrics. Mutation testing has also been adapted for QSTR applications. The key component of QSTR mutation testing is a set of six mutation operators based on analogous mutation operators applied to imperative and OO software. The mutation operators manipulate expressions in minor ways, such as changing relation states and set comparators in constraints. Importantly, the distinction between neighbouring relations is exercised by replacing relations with their neighbours in expressions, and by manipulating the edges in neighbourhood graphs; hierarchical distinctions are exercised by replacing relations with their specialisations and generalisations.

The following chapter presents a measure of expressiveness called H-complexity, which is used to define further coverage metrics for assessing test suite efficacy. Subsequently, Chapter 11 presents a methodology for evaluating the efficacy of competing testing strategies.
Chapter 10

H-Complexity for Analysing and Validating QSTR Applications

10.1 Introduction

In the previous chapter the main QSTR validation process was presented. In particular, two testing methodologies and two test suite analysis methodologies were adapted from well known methods in software engineering to make QSTR application validation accessible to typical software programmers.

In this chapter a novel measure of language complexity called H-complexity is presented and used to develop further QSTR specific validation tools. By employing research in finite model theory, H-complexity is directly derived from the relational theoretic model that underlies QSTR applications. The significance is that the complexity of the language required to encode a constraint expression directly influences the resources required for testing that constraint. Thereby, H-complexity is used to develop validation tools to support QSTR application developers. Firstly, the formulation of complexity provides the developer with key insights about the interaction between salient expression properties and the complexity of the language required to encode the expression. The developer can use these basic properties of complexity to manage development resources and complexity when modifying their design. Secondly, H-complexity provides natural test coverage classes that the developer can use to analyse and assess the efficacy of a test suite.

This chapter is structured as follows. Section 10.2 presents the theoretical foundations of H-complexity. Section 10.3 uses the formulation of complexity to highlight key properties of expressiveness such as the relative influence of particular components in an expression, and presents principles for managing the complexity of an expression. Section 10.4 uses H-complexity to define four coverage metrics, and presents guidelines on utilising these coverage metrics in conjunction with the adapted coverage metrics in the previous chapter.
10.2 H-Complexity Theory

This section presents the formal foundations of H-complexity. Firstly, the novel concept of homogeneous sets is defined as a special class of definable sets in the model theoretic sense. Homogeneous sets are then used to define the H-complexity metric by counting the number of queries that can be expressed using a collection of homogeneous sets, or equivalently, counting the number of expressible scenarios. The central formula for calculating the H-complexity of a given QSTR language is then derived. Finally, the semantics of partial scenario descriptions are defined as they are required for calculating H-complexity test coverage metrics.

10.2.1 Homogeneous Sets

In model theory [88, 118], a set $X$ is definable (in model $M$) if there is some query in first order logic that can distinguish precisely this set of objects (that is, a formula $\phi$ exists such that $X = \{(v_1, \ldots, v_n) \in U^n | M \models \phi(v_1, \ldots, v_n)\}$, where $U$ is the universe of objects and entails $\models$ means that the formula is true in $M$).

Homogeneous sets (or H sets) are defined in this thesis as a special class of definable sets [144]. H sets are atomic definable sets, that is, no query exists that can separate two objects within the same H set, thus objects within an H set are equivalent and indistinguishable. Let $H = \{h_1, \ldots, h_n\}$ be a set of homogeneous sets, where each $h_i \subseteq U$. By definition, $h_1, \ldots, h_n$ partition $U$. H-complexity of a language is defined as $|H|$.

10.2.2 H-complexity: Measuring Complexity With H Sets

Complexity of a QSTR application language can be considered as either the number of distinct queries that can be expressed, or (equivalently) the number of distinct scenarios that can be encoded.

A query is used to access a subset of objects in a scenario, and query complexity of a language is defined as the maximum number of unique non-empty subsets that can be accessed by some query. H sets are indivisible and mutually exclusive (by definition), so the query that defines an H set must also be the query that returns the smallest non-empty subset of those objects. The smallest subset containing objects from two different H sets $h_1, h_2$ must be the union of the queries that define those two H sets, $h_1 \cup h_2$. It follows that any accessible subset of objects must be the union of some combination of H sets, and thus query complexity is equal to the number of different combinations of H sets, $2^{|H|}$.

Scenario complexity will now be defined. Intuitively, qualitative models do not distinguish between numerical quantities, unlike metric systems. If two objects in a scenario can not be separated by a query, then the objects are considered equivalent and indistinguishable, that is, the objects must be in the same H set. Accordingly, if the only difference between two
scenarios is the number of indistinguishable objects in each non-empty H set then the scenarios are considered equivalent. Thus, a scenario equivalence class is defined by the combination of H sets from which objects are selected, and the number of such scenario classes, or the scenario complexity of the language, is $2^{|H|}$.

### 10.2.3 Calculating H-Complexity

This section derives the formula for calculating H-complexity of a language by counting the number of H sets, $|H|$. The process of deriving the formula is summarised in the following steps.

- Firstly, the number of H sets permitted by a single unary relation is derived; this is then extended to allow a single relation of arbitrary arity.

- It is then observed that binary relations (and higher) admit an infinite number of H sets, making H-complexity unusable.

- To overcome this, restrictions are imposed on the query language to calculate the number of basic queries that can be expressed for a set of relations.

- It is then shown that basic queries are not jointly exhaustive and pairwise disjoint (JEPD) and so do not correspond to H sets.

- Thus, H-complexity is finally derived as the smallest number of unique JEPD queries that can be expressed using combinations of basic queries.

Initially, assume that all relations have an arity of 1 (i.e. they represent qualitative properties such as round or large). Tuples can take one of $|A_R|$ states for a relation $R$ (such as holds), hence the complexity of a unary relation is $|H_R| = |A_R|$. However, once relations have an arity greater than 1 there are an infinite number of potential H sets, because a binary relation constitutes a total order.

To proceed in this analysis, graphs are used to represent a scenario that consists of a single binary relation $R_i$, where objects represent vertices and directed edges represent tuples, as illustrated in Figure 10.1. A set theoretic query describes the structure of a graph and specifies the vertex to be selected with $v$ bound variables (universally or existentially quantified), e.g.
\[\forall x_1 \ldots \exists x_v . x_1 \neq x_2 \land \ldots \land x_1 \neq x_v \land \ldots \land x_{v-1} \neq x_v.\]

For brevity, these quantifications and conditions will not be explicitly stated in further queries, and for simplicity the variables in further examples will only be existentially quantified. For example, the query \(\{x_2 \mid (x_1, x_2) \in R_1 \land (x_3, x_2) \in R_1\}\) will access \(b\) from the graph of \(R_1\) in Figure 10.1.

While there are an infinite number of potential graphs and unique accessible subsets, homogeneous sets still exist that contain indistinguishable objects. Indeed, homogeneous sets correspond to graph symmetries.

**Example 10.2.1.** Regarding the graph of \(R_1\) in Figure 10.1, no query exists that can separate objects \(a\) and \(c\) (without directly referring to those objects),

\[
\{a, c\} = \{x_1 \mid (x_1, x_2) \in R_1 \land (x_3, x_2) \in R_1\} \\
= \{x_3 \mid (x_1, x_2) \in R_1 \land (x_3, x_2) \in R_1\},
\]

and the graph of \(R_2\) has three \(H\) sets, accessed by the query

\[
\{x_i \mid (x_1, x_1) \in R_2 \land (x_1, x_2) \in R_2 \land \\
(x_2, x_3) \in R_2 \land (x_4, x_4) \in R_2 \land \\
(x_4, x_5) \in R_2 \land (x_5, x_3) \in R_2\},
\]

namely \(\{a, d\}\) when \(i = 1\) or \(4\), \(\{b, e\}\) when \(i = 2\) or \(5\) and \(\{c\}\) when \(i = 3\).

Given a graph of a scenario, the number of \(H\) sets is the number of vertices minus the number of symmetries. However, to make \(|H_R|\) a function of the entire QSTR application language, rather than just isolated scenarios (i.e. rather than particular graphs), the concept of restricted query languages is employed from finite model theory (normally used for studying descriptive complexity [118]). If the restricted query language only recognises a finite number of graphs, it will admit a finite number of \(H\) sets. It is then possible to quantify the complexity of a relation independent of a particular scenario, and measure the relative difference in expressiveness between two languages.

One common query restriction is to limit the number of variables (i.e. the number of unique vertices accessible in a graph). Previously, queries have referred to variables \(x_i\) where \(i\) can be any positive integer. For example, if \(i \leq 2\) then the allowable tuples are \((x_1, x_1), (x_1, x_2), (x_2, x_1),\) and \((x_2, x_2)\). If \(v\) is the number of variables allowed in a query, and \(a_R\) is the arity of relation \(R\) (i.e. the size of the tuples) then for each query the number of tuples is \(v^{a_R}\). These queries are called basic queries. For example, if \(v = 2\) then one basic query on a binary relation \(R\) is \(\{x_1 \mid (x_1, x_1) \in R^- \land (x_1, x_2) \in R^+ \land (x_2, x_1) \in R^- \land (x_2, x_2) \in R^-\}\). Each tuple can be assigned to one of \(|A_R|\) relation states, thus, the number of unique basic queries for relation \(R\) is \(|A_R|^{\text{number of tuples}}\), where \(\text{number of tuples} = v^{a_R}\).
Previously only one relation was referred to within a query. Given \( v \) bound variables, queries will now take the form,

\[
\{ x_1 | \text{query } R_1, \text{ query } R_2, \ldots, \text{query } R_n \},
\]

where query \( R_i \) is one of the unique basic queries for relation \( R_i \). Hence, the total number of queries permitted over \( n \) relations is \(|\text{basic } R_1 \text{ queries}| \times \ldots \times |\text{basic } R_n \text{ queries}|\). Moreover, each query variable can be either existentially or universally quantified, i.e. the query can use any one of the combinations from \( \exists x_1 \exists x_2 \ldots \exists x_v \) to \( \forall x_1 \forall x_2 \ldots \forall x_v \). In general, the number of allowable variable quantifications \( q \) is \( 2^v \) if all combinations are acceptable. Thus, the number of unique basic relations including the acceptable variable quantifications is \( q \times |\text{basic } R_1 \text{ queries}| \times \ldots \times |\text{basic } R_n \text{ queries}|\).

For \( H \) sets to truly represent the maximum refinement possible, they must be JEPD so that every object in a scenario will appear in exactly one \( H \) set. This property is critical; if it did not hold then further refinements could be achieved by taking \( H \) set intersections and differences. Basic queries are not necessarily JEPD (specifically, when their corresponding graphs are overlapping induced subgraphs of the full scenario graph) and so they do not specify \( H \) sets.

**Example 10.2.2.** Consider the scenario graph in Figure 10.2. If \( v = 2 \) then two basic queries are:

\[
\{ x_1 | (x_1, x_1) \in R^- \land (x_1, x_2) \in R^+ \land (x_2, x_1) \in R^- \land (x_2, x_2) \in R^- \} = \{ a, e \},
\]

\[
\{ x_1 | (x_1, x_1) \in R^- \land (x_1, x_2) \in R^+ \land (x_2, x_1) \in R^- \land (x_2, x_2) \in R^+ \} = \{ c, e \}.
\]

Vertex \( e \) appears in both results, and therefore the basic queries are not JEPD.

To calculate \(|H|\) it is necessary to determine the smallest JEPD queries that contain the basic queries. This is achieved by taking all combinations of basic queries by intersection and difference, hence \(|H| = 2^{\text{number of unique basic queries}} - 1\) (the trailing \(-1\) is insignificant and can be ignored).

To summarise,

- the language being measured has a set of relations \( R \),
- \( A_R \) is the number of relation states allowed for relation \( R \) (such as holds and does not hold),
Table 10.1: Basic queries are derived from different combinations of relation state tuples.

<table>
<thead>
<tr>
<th>H set</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_1$</td>
<td>$b_1 \setminus \left( b_2 \cup b_1 \cup \ldots \cup b_m \right)$</td>
</tr>
<tr>
<td>$H_2$</td>
<td>$b_2 \setminus \left( b_1 \cup (b_1 \cup \ldots \cup b_m) \right)$</td>
</tr>
<tr>
<td>$H_3$</td>
<td>$b_3 \setminus \left( (b_1 \cup b_2) \cup (b_1 \cup \ldots \cup b_m) \right)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$H_{2m}$</td>
<td>$b_1 \cup \ldots \cup b_m$</td>
</tr>
</tbody>
</table>

Table 10.2: Homogeneous (H) sets are derived from different combinations of basic queries.

- $a_R$ is the arity of relation $R$ (e.g. binary relations have arity 2),
- $v$ is the number of variables allowed in a query ($1 \leq v$),
- $q$ is the number of variable quantifications ($1 \leq q \leq 2^v$),
- $H$ is the set of H sets, and $|H|$ is the H complexity of the language.

The formula for calculating H-complexity is

$$|H| = 2 \uparrow \left( q \times |\text{basic } R_1 \text{ queries}| \right) \times \ldots \times \left( q \times |\text{basic } R_n \text{ queries}| \right)$$

$$= 2 \uparrow \left( q \times |A_{R_1}|^{a_{R_1} v} \times \ldots \times |A_{R_n}|^{a_{R_n} v} \right)$$

$$= 2 \uparrow \left( q \prod_{R \in R} \left( |A_R|^{a_R v} \right) \right).$$

Tables 10.1, 10.2 and 10.3 illustrate the relationship between the different components of H-complexity. Given $n$ relations, Figure 10.1 illustrates the number of basic queries where $m = 2^n$; note that the relations shown are binary, but any arity is acceptable. Homogeneous sets are combinations of basic queries, as illustrated in Table 10.2. Finally, scenarios consist of different combinations of non-empty H sets, as illustrated in Table 10.3. Hence, scenarios can be described as combinations of combinations of basic queries.
10.2.4 Partial Basic Queries, Partial H Sets and Partial Scenarios

This section defines the semantics of the useful concept of partial scenario descriptions; for example, partial scenarios are required in Section 10.4.6 for analysing H-complexity test coverage metrics. A (complete) basic query specifies some state for all expressable tuples for all relations,

\[ \{ (x,z) \mid (x,x) \in R_1^+ \wedge (x,y) \in R_1^- \wedge (x,z) \in R_1^+ \wedge \ldots \wedge (z,z) \in R_4^2 \} \]

A partial basic query is an incomplete description of a basic query, e.g.

\[ \{ (x,z) \mid (x,y) \in R_1^+ \wedge (y,z) \in R_1^+ \} , \]

that is, there is some expressible tuple for which the state has not been specified. A partial basic query is equivalent to the union of a particular set of complete basic queries (i.e. they will always return the same set of objects for any given scenario). For example, the partial basic query above is equivalent to

\[ \{ (x,z) \mid (x,x) \in R_1^+ \wedge (x,y) \in R_1^- \wedge \ldots \wedge (y,z) \in R_1^+ \wedge \ldots \wedge (z,z) \in R_4^+ \} \]

\[ \cup \quad \{ (x,z) \mid (x,x) \in R_1^- \wedge (x,y) \in R_1^+ \wedge \ldots \wedge (y,z) \in R_1^+ \wedge \ldots \wedge (z,z) \in R_4^+ \} \]

\[ \cup \quad \ldots \]

\[ \cup \quad \{ (x,z) \mid (x,x) \in R_1^+ \wedge (x,y) \in R_1^+ \wedge \ldots \wedge (y,z) \in R_1^+ \wedge \ldots \wedge (z,z) \in R_4^+ \} \].

That is, the given partial basic query is equivalent to the union of all (complete) basic queries that contain the tuple states \((x, y) \in R_1^+\) and \((y, z) \in R_1^+\).

A partial H set is constructed from partial basic queries. For example, a partial H set is

\[ \{ (x,z) \mid (x,y) \in R_1^+ \wedge (y,z) \in R_1^+ \} \cap \{ (x,y) \mid (x,y) \in R_2^+ \} \].

Partial H sets are equivalent to the union of a particular collection of H sets. The H sets must consist of one complete basic query for each partial basic query in the partial H set. One (complete) H set from the above example is

\[ \{ (x,z) \mid (x,x) \in R_1^+ \wedge (x,y) \in R_1^+ \wedge \ldots \wedge (y,z) \in R_1^+ \wedge \ldots \wedge (z,z) \in R_4^+ \} \]

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₁</td>
<td>( H_1 \setminus (H_2 \cup H_3 \cup \ldots \cup H_m) )</td>
</tr>
<tr>
<td>s₂</td>
<td>( H_2 \setminus {(H_1) \cup (H_3 \cup \ldots \cup H_m)} )</td>
</tr>
<tr>
<td>s₃</td>
<td>( H_3 \setminus {(H_1 \cup H_2) \cup (H_4 \cup \ldots \cup H_m)} )</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>sₙm</td>
<td>( H \cup \ldots \cup H_m )</td>
</tr>
</tbody>
</table>

Table 10.3: Scenarios are derived from different combinations of H sets being non-empty.
Similarly, partial scenarios are constructed from partial H sets.

### 10.3 Using H-complexity to Analyse QSTR Applications

H-complexity can be used by a developer to analyse components of their application because it quantifies the complexity of the language that permits a given set expression. Given a set expression from a constraint, or a use case query that a potential user might run on the application, such as

\[
\{ x | (x, y) \in \text{near}^+ \land y \in \text{hospital}^+ \},
\]

the developer can use H-complexity to measure the resources required to sufficiently test the expression. Furthermore, the formulation presented in this chapter highlights some basic properties of QSTR language expressiveness that a developer can use to manage development resources and complexity when modifying their design. For example, the formulation clearly identifies which variables have the greatest impact on complexity, and thus which variables should be manipulated by the developer in order to reduce the expression’s complexity and to make testing more practical.

Particular properties of QSTR language expressiveness will now be analysed. In general, the formulation shows that H-complexity is a function of the number of atomic queries that the QSTR language allows. Restricting the number of variables in a query to \( v \) makes complexity a function of the number of relation states \( |A_R| \), the number of variables \( v \), and the relation arity \( a_R \). Moreover, Equation 10.1 specifies the relative influence that each component has on complexity; relation arity \( a_R \) has the most influence, followed by the number of variables \( v \) having exponentially less influence, and finally the number of relation states \( |A_R| \) having exponentially less influence again. These properties help to inform the developer about how changes to each component influence expressiveness, and the relative difference in expressiveness between two QSTR systems.

The following sections analyse the influence that relation arities, the number of query variables and the number of relation states have on complexity.

#### 10.3.1 Relation Arity

To reduce the complexity of a set expression the developer should first consider modifying the arity of relations in the expression, as this variable has the most impact on expressiveness.
Example 10.3.1. The set expression

\[ \{ x \mid (x, x) \in \text{watching}^+ \land (y, y) \in R_2^+ \land (x, y) \in R_3^+ \}, \]

has two existentially quantified query variables \( v = 2 \), and \( q = 1 \) (i.e. the variables are quantified in exactly one way, \( \exists x \exists y \)). Assume that each relation can only take one of two states, \( \text{holds} \) and \( \text{not holds} \) so that \( |A_R| = 2 \). Then

\[ |H| = 2 \uparrow \left( q \prod_{R \in \{\text{watching}, R_2, R_3\}} (|A_R|^{v_R}) \right) = 2 \uparrow \left( 2^{2^2 \cdot 2^2 \cdot 2^2} \right) = 2 \uparrow 4096. \]

To reduce this complexity the developer can replace the first relation \( \text{watching} \) with a new, lower arity relation \( \text{watchingSelf} \), hence converting the expression to

\[ \{ x \mid x \in \text{watchingSelf}^+ \land (y, y) \in R_2^+ \land (x, y) \in R_3^+ \}. \]

This new expression has a vastly reduced complexity,

\[ |H| = 2 \uparrow \left( q \prod_{R \in \{\text{watchingSelf}, R_2, R_3\}} (|A_R|^{v_R}) \right) = 2 \uparrow \left( 2^{2^1 \cdot 2^2 \cdot 2^2} \right) = 2 \uparrow 1024. \]

Note that Section 10.4 presents a practical methodology for managing this enormous complexity when conducting testing; the point being illustrated in this section is that an understanding of the influence that each variable has on complexity can be extremely useful for informing the developer’s modelling decisions.

Replacing a relation with an equivalent lower arity relation is appropriate in applications where specific objects are always small in number (less than 10 or 20) and almost always present in a scenario.

Example 10.3.2. In QtvLight (refer to Section 2.8.1) the types of rooms in a domestic house are fairly standard; if each scenario always consists of one household, then it is reasonable to assume that the house will contain a relatively small number of bathrooms, bedrooms, one kitchen, and so on, giving a total of less than approximately 20 rooms. In this case, expressions that have exploding complexity can be replaced by a small number of tailored expressions that have relations with reduced arities such as \( \text{nearBedroomA} \) and \( \text{adjacentToKitchen} \).

Example 10.3.3. TreeSap (refer to Section 2.8.2) may be configured to always load the same specific map. The region in the map may have a relatively small number of specific features such as schools, landmarks, hospitals, libraries, and so on. Reducing the arity of key relations will again help to reduce the overall complexity of the application, for example replacing \( \text{near} \) with \( \text{nearMaungawhauPrimarySchool} \) in particular expressions.
Replacing relations with lower arity relations can also have an additional influence in reducing the number of unique variables required in the expression.

**Example 10.3.4.** The expression

\[ \{ x \mid (x, y) \in \text{near}^+ \land y \in \text{hospital}^+ \} \]

has two existentially quantified query variables \( v = 2 \), \( q = 1 \). Assume that each relation can take one of three states, \textit{holds}, \textit{not holds}, and \textit{indefinite} so that \( |A_R| = 3 \). Then

\[ |H| = 2 \uparrow \left( q \prod_{R \in \{\text{near}, \text{hospital}\}} (|A_R|^{v^R}) \right) = 2 \uparrow \left( 3^{2^2} \cdot 3^{2^1} \right) = 2 \uparrow 729. \]

To reduce this complexity the developer replaces the first relation \textit{near} with a unary relation \textit{nearHospital},

\[ \{ x \mid x \in \text{nearHospital}^+ \land y \in \text{hospital}^+ \}, \]

thus giving a reduced complexity of

\[ |H| = 2 \uparrow \left( q \prod_{R \in \{\text{nearHospital}, \text{hospital}\}} (|A_R|^{v^R}) \right) = 2 \uparrow \left( 3^{2^1} \cdot 3^{2^1} \right) = 2 \uparrow 81. \]

However, the second term is no longer required because the developer can assume that if \( x \in \text{nearHospital}^+ \) then clearly there must exist some \( y \) that is a \textit{hospital},

\[ \{ x \mid x \in \text{nearHospital}^+ \}. \]

Thus, reducing the arity of one of the relations has had the additional effect of eliminating another relation, and in turn lowering the number of unique query variables required for the expression, \( v = 1 \). The new complexity is

\[ |H| = 2 \uparrow \left( q \prod_{R \in \{\text{nearHospital}\}} (|A_R|^{v^R}) \right) = 2 \uparrow \left( 3^{1^1} \right) = 2 \uparrow 3. \]

### 10.3.2 Number of Query Variables

If reducing the arity of relations in an expression is not a suitable solution then the developer can attempt to reduce the number of unique variables permitted. One method of doing this is to split long expressions that require many unique variables into a number of smaller expressions that each require fewer variables.

**Example 10.3.5.** In the Image Retrieval application (refer to Section 2.8.4), the following set expression describes a particular type of image

\[ \{ x \mid x \in \text{image}^+ \land \]

\[ (y, x) \in \text{partOf}^+ \land y \in \text{mountain}^+ \land \]

\[ \} \]
\[ (z, x) \in \text{partOf}^+ \land z \in \text{water}^+ \land \]
\[ (w, x) \in \text{partOf}^+ \land w \in \text{mountain}^+ \land \]
\[ (y, z) \in \text{above}^+ \land (z, w) \in \text{above}^+ \}\]

It requires four unique query variables \( v = 4 \), \( q = 1 \). The complexity is
\[
|H| = 2 \uparrow \left( q \prod_{R \in \{\text{image, partOf, mountain, water, above}\}} (|A_R|^{v_R}) \right)
\]
\[
= 2 \uparrow \left( 2^{4^1} \cdot 2^{4^2} \cdot 2^{4^1} \cdot 2^{4^2} \right)
\]
\[
= 2 \uparrow (2^{44}).
\]

The developer can reduce the number of unique variables by splitting the expression into separate expressions.

\[
\{ x | x \in \text{image}^+ \land \]
\[
(y, x) \in \text{partOf}^+ \land y \in \text{mountain}^+ \land \]
\[
(z, x) \in \text{partOf}^+ \land z \in \text{water}^+ \land \]
\[
(y, z) \in \text{above}^+ \}\]
\[
\cap
\]
\[
\{ x | x \in \text{image}^+ \land \]
\[
(z', x) \in \text{partOf}^+ \land z' \in \text{water}^+ \land \]
\[
(w, x) \in \text{partOf}^+ \land w \in \text{mountain}^+ \land \]
\[
(z', w) \in \text{above}^+ \}\].

Each expression now only requires three variables, \( v = 3 \). The complexity of each expression becomes
\[
|H| = 2 \uparrow \left( q \prod_{R \in \{\text{image, partOf, mountain, water, above}\}} (|A_R|^{v_R}) \right)
\]
\[
= 2 \uparrow \left( 2^{3^1} \cdot 2^{3^2} \cdot 2^{3^1} \cdot 2^{3^2} \right)
\]
\[
= 2 \uparrow (2^{27}).
\]
The total complexity of the expressions is thus $2 \uparrow (2^{27}) + 2 \uparrow (2^{27}) = 2 \times 2 \uparrow (2^{27}) = 2 \uparrow (2^{27} + 1)$ which is a vast reduction from $2 \uparrow (2^{44})$.

Dividing large expressions into smaller expressions reduces complexity by implicitly utilising properties of relations. This particular example works because above is transitive. That is, the original expression required that the same region of water $z \in \text{water}^+$ be compared to both mountain regions, $(y, z) \in \text{above}^+$ and $(z, w) \in \text{above}^+$. In the case of the two smaller expressions, the water region is not necessarily the same object that satisfies both expressions, $z \neq z'$; however, because above is transitive then if two different objects satisfy the expressions, i.e. $z \neq z'$, $(y, z) \in \text{above}^+$ and $(z', w) \in \text{above}^+$ then either $(z, z') \in \text{above}^+$ or $(z', z) \in \text{above}^+$. In other words, if the expressions are satisfied when $z \neq z'$ then they must necessarily also be satisfied by some $z, z'$ when $z = z'$. When dividing expressions, the developer must identify such properties to ensure that the original expression and the collection of smaller expressions are existentially equivalent (that is, they must produce the same set of objects, but the variable bindings may be different).

One simple property that developers should analyse is the scope of relations in a given expression. The developer may find that certain relations are redundant because they are subsumed by other relations, and can thus be removed from the expression. This can be viewed as directly reducing the required expressiveness by constraining the structure of relations.

**Example 10.3.6.** In the Image Retrieval application, it may be reasonable to assume that objects can only be $\text{partOf}$ image objects. Thus the expression

$$\{x \mid x \in \text{image}^+ \land (z', x) \in \text{partOf}^+ \land z' \in \text{water}^+ \land (w, x) \in \text{partOf}^+ \land w \in \text{mountain}^+ \land (z', w) \in \text{above}^+\}.$$  

can be reduced to

$$\{x \mid (z', x) \in \text{partOf}^+ \land z' \in \text{water}^+ \land (w, x) \in \text{partOf}^+ \land w \in \text{mountain}^+ \land (z', w) \in \text{above}^+\}.$$  

To analyse QSTR expressions, developers can employ some of the many existing reasoning tools used for analysing ontologies (for example, see [89, 127, 164]).
10.3.3 Allowable Relation States

Finally, the developer can also attempt to reduce the number of allowable relation states. For example, the expression

\[
\{ x | (x, y) \in \text{near}^+ \land y \in \text{hospital}^+ \}
\]

has two existentially quantified query variables \( v = 2, q = 1 \). Assume that each relation can take one of three states, \textit{holds}, \textit{not holds}, and \textit{indefinite} so that \( |A_R| = 3 \). Then, as given previously,

\[
|H| = 2 \uparrow \left( 3^{2^2} \cdot 3^{2^1} \right) = 2 \uparrow 729.
\]

If the developer reduces the number of allowable states to \textit{not holds} and \textit{indefinite} (as used in all Allen based QSTR formalisms; refer to Chapter 3) then the complexity decreases to

\[
|H| = 2 \uparrow \left( 2^{2^2} \cdot 2^{2^1} \right) = 2 \uparrow 64.
\]

10.4 Using H-complexity to Quantify Test Coverage

As presented in Section 9.5, Chapter 9, measuring the test coverage of particular program components is a standard software engineering technique for quantifying the efficacy of a test suite [173]. H sets are a natural option for analysing test classes because, on one hand, they specify the absolute limit for distinguishing between objects, and on the other hand, they can be used to describe any possible distinct set of objects. This section presents the methodology for applying H-complexity to measure test coverage.

According to the validation methodology overview in Section 9.2, when quantifying test coverage the developer has a set of QSTR application components that are currently being tested, and the set of tests. To employ H-complexity for measuring test coverage the developer must undertake the following five activities, as illustrated in Figure 10.3.

1. Identify the domain of the components being tested.
2. Refine the test space by specifying conditions that are not appropriate for exhaustive testing.
3. Calculate the complexity of the original test space and the refined test space.
4. Determine the class of each test in the test suite.
5. Calculate H-complexity test coverage results.

The following sections present the details of each activity.
Figure 10.3: Overview of the process of calculating H-complexity coverage scores (regarding the input “get app design”, note that the developer may only be analysing a portion of the full QSTR application).
10.4 Using H-complexity to Quantify Test Coverage

10.4.1 Activity 1: Identify Component Domains

Firstly the developer must identify the domain (a set of relations) of the components being tested. For a unit test, the domain contains the relations in the set expression, e.g. the domain of the set expression \( \{ x_1 \mid (x_2, x_2) \in R_1^+ \land (x_3, x_2) \in R_2^- \} \) is \( \{ R_1, R_2 \} \). For an integration test, the domain contains the relations that appear in the subset of constraints being tested, e.g. the domain of the constraints

\[
\{ x_1 \mid (x_1, x_2) \in R_1^+ \land (x_3, x_2) \in R_2^- \} = \{ x_1 \mid (x_2, x_1) \in R_3^+ \},
\]

\[
\{ x_1 \mid (x_1, x_2) \in R_1^+ \land (x_3, x_2) \in R_2^- \} \subseteq \{ x_1 \mid x_1 \in R_4^+ \}
\]

is \( \{ R_1, R_2, R_3, R_4 \} \).

10.4.2 Activity 2: Specify Conditions to Refine the Test Space

H-complexity is calculated as all possible combinations of H sets. When considered as a test space, each H set is being exercised in conjunction with every other combination of H sets. However, many H sets represent conditions that may not require this exhaustive testing. By isolating such conditions and testing them independently, the developer can achieve a smaller, more focused and hence more practical and effective test space. Table 10.4 presents guidelines for common conditions that can be used to refine test spaces.

For example, the relation \( \text{in} \) is not (usually) symmetric, that is, if \( x \) is \( \text{in} \) \( y \), then \( y \) cannot also be \( \text{in} \) \( x \). If this condition is violated then the scenario is clearly inconsistent with the QSTR application, regardless of the other remaining components of the scenario. Rather than exhaustively testing \( \text{in} \) for every unit that it is used, the developer can isolate the erroneous symmetric condition and test it once. They can then assume that every time \( \text{in} \) is used the application will respond correctly regarding symmetry. Another example is that, in many spatial models of indoor environments, no objects (such as chairs and walls) are ever \( \text{inside} \) light bulb objects.

10.4.3 Activity 3: Calculate H-Complexity for Refined Test Spaces

The complexity of a language can be calculated simply by applying the formula in Section 10.2.3. However, once the developer has specified conditions for refining the test space, the formula can no longer be used. This section presents a method for calculating the refined test space complexity by encoding it as a constraint satisfaction problem (CSP). The developer can then use any standard CSP solver, such as JaCoP [97], to calculate the complexity.

A CSP consists of a finite number of variables (where each variable has a finite domain), and a set of constraints between variables. Given a domain of relations \( R \) and the number of allowable query variables \( v \), the CSP solver will return the number of basic queries permitted,
<table>
<thead>
<tr>
<th>Condition</th>
<th>Mathematical Property</th>
<th>Example (application specific)</th>
<th>Formal description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Using Object Types (special unary relations)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>During testing, assume certain objects are certain types.</td>
<td>n/a</td>
<td>Object x is always a room, and object y is always a light.</td>
<td>given $x, \ldots, z$ in $U$, $R_1^+(x)$ and … and $R_n^+(z)$ used in conjunction with other constraints</td>
</tr>
<tr>
<td>Semantically, certain types of objects can never have certain relations.</td>
<td>n/a</td>
<td>No objects are ever inside a light bulb object.</td>
<td>for all $x, \ldots, z$ in $U$, $R_1^+(x) \rightarrow R_2^+(x, \ldots, z)$</td>
</tr>
<tr>
<td><strong>One Binary Relation</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>An object is necessarily related to itself with respect to this relation.</td>
<td>Reflexive</td>
<td>Objects are always near themselves.</td>
<td>for all $x$ in $U$, $R^+(x,x)$</td>
</tr>
<tr>
<td>An object can never be related to itself with respect to this relation.</td>
<td>Irreflexive</td>
<td>No object is inside itself.</td>
<td>for all $x$ in $U$, $R^-(x,x)$</td>
</tr>
<tr>
<td>If the relation holds from one object to another object, then it must also hold in the other direction (i.e. from the latter object to the former object).</td>
<td>Symmetric</td>
<td>If an object $x$ is near another object $y$, then object $y$ must also be near object $x$.</td>
<td>for all $x,y$ in $U$, $R^+(x,y) \rightarrow R^+(y,x)$</td>
</tr>
<tr>
<td>If the relation holds from one object to another object, then it can never hold in the other direction.</td>
<td>Asymmetric</td>
<td>If an object $x$ is inside object $y$, then object $y$ can never be inside object $x$.</td>
<td>for all $x,y$ in $U$, $R^-(x,y) \rightarrow R^-(y,x)$</td>
</tr>
<tr>
<td><strong>Two Binary Relations</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If relation $R_1$ holds from one object to another object, then relation $R_2$ must also hold in the other direction (i.e. from the latter object to the former object).</td>
<td>Inverse (or converse)</td>
<td>If time interval $x$ is before time interval $y$, then interval $y$ must be after interval $x$.</td>
<td>for all $x,y$ in $U$, $R_1^+(x,y) \rightarrow R_2^+(y,x)$</td>
</tr>
<tr>
<td>If relation $R_1$ holds from one object to another object, then relation $R_2$ can never hold in the other direction.</td>
<td>n/a</td>
<td>If object $x$ contains object $y$, then $y$ can never be larger than $x$.</td>
<td>for all $x,y$ in $U$, $R_1^+(x,y) \rightarrow R_2^-(y,x)$</td>
</tr>
<tr>
<td><strong>Pairs of Relations, Arbitrary Arity</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One relation trivially implies another relation.</td>
<td>Implication</td>
<td>If an object $x$ is accelerating then it must also be moving.</td>
<td>for all $x, \ldots, z$ in $U$, $R_1^+(x, \ldots, z) \Rightarrow R_2^+(x, \ldots, z)$</td>
</tr>
<tr>
<td>If one relation does not hold, then it trivially implies that another particular relation must hold.</td>
<td>Complement</td>
<td>If an object $x$ is not stationary, then it must be moving.</td>
<td>for all $x, \ldots, z$ in $U$, $R_1^-(x, \ldots, z) \Rightarrow R_2^+(x, \ldots, z)$</td>
</tr>
<tr>
<td><strong>Groups of Relations, Arbitrary Arity</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No two relations (out of a set of relations $R$) can both hold for a given set of objects at the same time.</td>
<td>Pairwise mutually exclusive</td>
<td>No pair of the relations near, moderately near, and far can ever hold at the same time.</td>
<td>for all $x, \ldots, z$ in $U$, and for all relations $R_1, R_2$ in $R$, not $(R_1^+(x, \ldots, z)$ and $R_2^+(x, \ldots, z))$</td>
</tr>
<tr>
<td>At least one relation (out of a set of relation $R$) must hold for a given set of objects at any time.</td>
<td>Jointly exhaustive</td>
<td>All objects are either moving or stationary at any time.</td>
<td>for all $x, \ldots, z$ in $U$, there exists relation $R$ in $R$, $R^+(x, \ldots, z)$</td>
</tr>
</tbody>
</table>

Table 10.4: Common conditions for refining test spaces.
The CSP encoding will now be presented. Each variable represents a tuple from a basic query. From Section 10.2.3, the number of tuples for relation \( R \) is \( v_{RA} \). Let integers 1, 2, 3, 4 represent holds, does not hold, not applicable, and indefinite respectively. For each relation \( R \), declare \( v_{RA} \) variables with domain \( A_R \), encoded as the appropriate subset of \( \{1, 2, 3, 4\} \). Next, specify the refinement conditions from 10.4.2. Finally, execute the solver and have it return the number of solutions (there is no need to record the solutions).

If computation is not too time consuming, the solver can be executed multiple times to determine the impact of each constraint. This is accomplished by executing the solver with only one constraint at a time, and recording the different complexities. Alternatively, the solver can be executed with all but one constraint (for each constraint) to quickly determine whether any constraints are redundant.

For example, given the domain \( R = \{\text{light, room, warm, in}\} \), \( v = 2 \) allowable query variables \( x \) and \( y \), and for every relation \( A_R = \{+, -, \sim, ?\} \), \( q = 1 \) variable quantification (both variables existentially quantified), the number of basic queries is

\[
q \prod_{R \in R} |(A_R | v_{RA})|
\]

\[
= 4^{2^1} \cdot 4^{2^1} \cdot 4^{2^1} \cdot 4^{2^2}
\]

\[
= 16 \cdot 16 \cdot 16 \cdot 256
\]

\[
= 1,048,776,
\]

yielding a completely intractable test space, with complexity \( |H| = 2^{1,048,776} \). A developer then refines this test space by specifying the following conditions:

- only test when \( x \) is a light and \( y \) is a room
- because both object types are assumed, ignore the not applicable relation state (all relations apply for lights and rooms)
- the types light and room are mutually exclusive
- \( in \) is not reflexive
- \( in \) is not symmetric
- nothing is ever in a light

The developer then encodes the CSP problem using JaCoP [97]. Firstly the variables are declared, using a restricted domain that excludes not applicable. Next, the developer runs the
solver which returns 27 solutions. The test space of size $2^{27}$ is now practical for certain important coverage metrics, although further refinements can be made, for example, choosing to only test one room at a time [148]. The developer then executes the solver multiple times, each run using only one of the constraints, and determines that constraints 1 and 4 have the most impact. Once again the developer executes the solver multiple times, each run using all but one constraint (for each constraint) and determines that some constraints are redundant and can be removed. Note that, after removing one constraint, the process should be repeated rather than removing multiple constraints between runs.

10.4.4 Activity 4: Calculate the Class of a Given Test Instance

To determine the test coverage of a given set of tests, the developer must compare the tests to the test space in terms of H-complexity. That is, the developer needs to determine which combination of H sets are exercised in a given test. A test is a set of input premises and expected outputs. The input premise information is a set of relations that contain object tuples, such as $\text{within}^+ = \{(a,b),(b,c)\}$, $\text{school}^+ = \{a\}$, $\text{Downtown}^+ = \{b\}$, $\text{Auckland}^+ = \{c\}$, $\text{near}^- = \{(c,d)\}$, $\text{within}^- = \{(a,c)\}$. By convention, for each relation, any unspecified tuples can be assumed to be in the indefinite relation state (e.g. $\text{Auckland}^- = \{a,b,d\}$). The expected output is again a set of relations that contain object tuples, such as $\text{within}^+ = \{(a,b),(b,c),(a,c)\}$. A test is satisfied by a QSTR application if, given the premises, reasoning produces a scenario that satisfies the expected outputs.

Given a domain being tested and a test instance, the developer needs to determine which test class the given test is in with respect to the domain’s test space. Using H sets, a test class is defined as the premise scenario specified in the test (that is, the expected output is ignored).

A CSP encoding will now be used to calculate the class of a given test. Firstly, the input scenario is encoded as a CSP. To accomplish this, for each non-empty relation state set in the input scenario, a variable is created (e.g. $\text{within}^+$ can be represented by a variable $\text{withinHolds}$) with a domain of values representing object tuples in the relation state (e.g. $\{(a,b),(b,c)\}$). For each allowable query variable, one CSP variable is created (e.g. for $v = 2$ query variables, create two CSP variables $x,y$), with domains of values representing every object in the scenario (e.g. $\{a,b,c,d\}$), and a constraint is imposed which forces the variables to be non-equal to ensure that they represent different objects in each solution.

Secondly, the basic queries are encoded in terms of H sets using the method from the previous section. That is, for each tuple from a basic query of relation $R$, a variable is created with a domain that represents the allowable tuple states $A_R$, i.e. some subset of $\{1,2,3,4\}$ where integers 1,2,3,4 represent holds, does not hold, not applicable, and indefinite respectively. The test space refinement constraints must not be encoded.
Finally, the scenario encoding is linked to the basic query encoding. For each relation state set variable from the scenario, a constraint is created that associates it to the equivalent basic query tuple. For example, given variables representing scenario objects \( x, y \), relation state \( \text{withinHolds} \), and the variable representing the tuple from the basic query \( \text{withinXY} \), impose the following constraint to associate the two encodings:

\[
(\text{withinHolds} = (x, y)) \iff (\text{withinXY} = 1)
\]

The solver is executed to get all of the solutions. The class of a test is determined by the set of solutions for the basic query encoding variables and the value of the object variables that satisfy those basic queries. Once the developer knows the class that each test is in (that is, the combination of H sets from which objects are specified), they can run the test coverage metrics presented in the following section.

### 10.4.5 Activity 5: H-Complexity Coverage Metrics

This section presents four test coverage metrics based on H-complexity. The four test coverage metrics, strictly ordered in terms of coverage strength (from weakest to strongest) are

- tuple state coverage (TS),
- basic query coverage (BQ),
- H set coverage (H), and
- scenario coverage (S).

The following example will be referred to throughout this section to illustrate the parts of an application that the metrics quantify.

**Example 10.4.1.** Let the domain being tested contain one binary relation \( R \) that can take two states \( A_R = \{+, -\} \). Two query variables are allowed, \( v = 2 \), and one variable quantification, \( q = 1 \). The number of query tuples is \( v^{\sigma_R} = 2^2 = 4 \), which are

\[
(x_1, x_1) \in R, (x_1, x_2) \in R, (x_2, x_1) \in R, \text{ and } (x_2, x_2) \in R.
\]

The number of basic queries is \( q \prod_{R \in \mathcal{R}} (|A_R|^{\sigma_R}) = 1 \cdot 2^4 = 16 \), which are

\[
\begin{align*}
\mathcal{b}_1 &= \{x_1 | (x_1, x_1) \in R^+ \land (x_1, x_2) \in R^+ \land (x_2, x_1) \in R^+ \land (x_2, x_2) \in R^+ \}, \\
\mathcal{b}_2 &= \{x_1 | (x_1, x_1) \in R^+ \land (x_1, x_2) \in R^+ \land (x_2, x_1) \in R^+ \land (x_2, x_2) \in R^- \}, \\
\mathcal{b}_3 &= \{x_1 | (x_1, x_1) \in R^+ \land (x_1, x_2) \in R^+ \land (x_2, x_1) \in R^- \land (x_2, x_2) \in R^+ \}, \\
\vdots
\end{align*}
\]
\[b_{16} = \{x_1 | (x_1, x_1) \in R^- \land (x_1, x_2) \in R^- \land (x_2, x_1) \in R^- \land (x_2, x_2) \in R^- \}.\]

The number of H sets is \(|H| = 2 \uparrow |\text{basic queries}| = 2^{16}. The number of scenario classes is \(2^{|H|} = 2^{2^{16}}\). Let the example test set consist of two tests with the following H sets

- test \(t_1\): \(\{b_1, b_2\}\)
- test \(t_2\): \(\{b_1\}, \{b_2, b_3\}\)

Tuple state (TS) coverage measures the number of query tuples that have taken a particular state in at least one test. Full TS coverage means that every query tuple has been assigned to every allowable relation state in at least one test. This should be viewed as an absolute minimum coverage requirement that all QSTR application test sets must satisfy. The total number of tuples with states that a language admits is \(\sum_{R \in R} |A_R| \cdot |v^{\mu R}|\). The minimum number of queries required for full TS coverage is \(\max_{R \in R} \{|A_R|\}\), i.e. one query where all tuples take the first state in \(A_R\), one query where all tuples take the second state in \(A_R\), and so on.

**Example 10.4.2.** In the running example there are 4 query tuples, and each tuple can take 2 states, giving \(2 \cdot 4 = 8\) possible tuples with states, namely

\[(x_1, x_1) \in R^+, (x_1, x_2) \in R^+, (x_2, x_1) \in R^+, (x_2, x_2) \in R^+,\]
\[(x_1, x_1) \in R^-, (x_1, x_2) \in R^-, (x_2, x_1) \in R^-, (x_2, x_2) \in R^- .\]

The example test set contains the tuples \((x_1, x_1) \in R^+, (x_1, x_2) \in R^+, (x_2, x_1) \in R^+, (x_2, x_2) \in R^+, (x_2, x_1) \in R^-, (x_2, x_2) \in R^-.\) Hence, percent TS coverage is \(5/8 = 62.5\%\). The running example can achieve full TS coverage with one test, \(\{b_1, b_{16}\}\).

Basic query (BQ) coverage measures the number of basic queries that have appeared in at least one test. Full BQ coverage means that every basic query has been used to describe some test scenario. While stronger than full TS coverage, full BQ coverage should also be viewed as a minimum coverage requirement for application validation.

**Example 10.4.3.** In the running example, the test set contains 3 basic queries \((b_1, b_2, b_3)\), giving a percent BQ coverage of \(3/16 = 18.75\%\).

H set (H) coverage measures the number of H sets that have been used to specify scenarios in at least one test. In practice, full H coverage is often very difficult to achieve, as it constitutes a vast class of tests. Instead the developer should focus on satisfying important subclasses within full H coverage, discussed below.

**Example 10.4.4.** The running example test set has 3 H sets, namely \(\{b_1\}, \{b_1, b_2\}\) and \(\{b_2, b_3\}\) giving a percent H coverage of \(3/(2^{16}) \approx 0\%\).
Scenario (S) coverage measures the number of scenario classes exercised in at least one test, where a scenario class is some unique combination of H sets from which objects in the class of scenarios are drawn. In practice, full S coverage is impossible to achieve, except for trivially small domains. However, after test space refinement S coverage can be a useful measure.

Example 10.4.5. In the running example two scenario classes are exercised, namely \{\{b_1, b_2\}\} and \{\{b_1\}, \{b_2, b_3\}\}. This gives a percent S coverage of \(\frac{2}{2^{16}} \approx 0\%\).

Full BQ coverage is trivially easy to achieve; for example, full BQ coverage is satisfied by one test where the scenario returns objects from all 16 basic queries. On the other hand, achieving full H coverage is often difficult in practice, and achieving full S coverage is, in almost all cases, impossible. An interesting topic for further research is identifying valuable classes within this test space in terms of H and S coverage. For example, three potentially significant H coverage criteria are

- all H sets that consist of exactly one basic query,
- all H sets that consist of exactly one basic query, selected from a special set of interesting basic queries, and
- all H sets that consist of exactly two basic queries

The first class of tests will ensure that all basic queries have been exercised in isolation (giving full TS and BQ coverage). The second class of tests applies a relevance bias to particular basic queries. For example, reflexive basic queries can be defined as any basic query for which all reflexive tuples hold i.e. \((x, \ldots, x) \in R^+\) and all other tuples take some other state. Similarly, transitive basic queries can be defined as basic queries for which transitive tuples hold \((x, y) \in R^+\) and all other tuples take some other state. It may be the case that these classes of expressions have a higher fault density. Finally, the third class ensures that the interactions between all pairs of basic queries have been exercised. All three of these test classes are relatively small and often practical to achieve. If the number of basic queries is \(b\) then the test class sizes are \(b\) for the first class, less than \(b\) for the second class, and \(b(b - 1)\) for the third class, where the maximum size of \(b\) is \(q \prod_{R \in R} (|A_{R}|)^{v_{R}}\).

10.4.6 Comparison Between Adapted and H-Complexity Test Coverage Metrics

This section compares the adapted coverage metrics presented in Section 9.5, Chapter 9, to the H-complexity coverage metrics. The correspondance between adapted and H-complexity coverage is analysed by determining the minimal language (in terms of H-complexity metrics) required to express the scenario that will cause reasoning to replicate a particular sequence
of executions (in terms of adapted coverage metrics). To summarise, vertex, edge and $k$-path coverage correspond to H and S coverage as follows.

- Each vertex condition corresponds to a partial H set.
- Each edge and $k$-path condition corresponds to a partial H set that may require a more complex language.

In subsections 10.4.6.1 and 10.4.6.2 these correspondances will be analysed in detail. The following example (also used in Section 9.5, Chapter 9 to explain the adapted coverage metrics) will be used to illustrate the relationship between the coverage approaches.

**Example 10.4.6.** Let the domain being tested contain four binary relations $R_1, \ldots, R_4$ that can take three states $A_R = \{+,-,\?\}$, and three constraints between the relations,

(A) $R_2$ is the transitive of $R_1$: \[ \{ (x,z) | (x,y) \in R_1^+ \land (y,z) \in R_1^+ \} = R_2^+. \]

(B) $R_3$ is asymmetric: \[ \{ (x,y) | (x,y) \in R_3^+ \land (y,x) \in R_3^- \} = \emptyset. \]

(C) $R_4$ is the transitive of $R_2$ to $R_3$: \[ \{ (x,z) | (x,y) \in R_2^+ \land (y,z) \in R_3^+ \} = R_4^+. \]

### 10.4.6.1 Correspondance Between Vertex Conditions and Partial H Sets

Each vertex condition corresponds to a partial H set. All scenarios in which reasoning causes the vertex to execute in the required way must have objects in any of the complete H sets from the partial H set.

**Example 10.4.7.** Given the vertex representing constraint (A), the first vertex condition requires that the initial scenario has the condition

\[ \{ (x,z) | (x,y) \in R_1^+ \land (y,z) \in R_1^+ \} \subset R_2^+ \]

due to the term $(x,y) \in R_1^+$ failing (reasoning must then be able to modify the scenario to satisfy the constraint). The vertex condition is equivalent to the partial H set

\[ \{ (x,z) | (x,y) \in R_1^+ \land (y,z) \in R_1^+ \} \cap \{ (x,y) | (x,y) \in R_2^+ \}, \]

where the term that causes the constraint to fail has the *indefinite* state for the correct tuple. All scenarios that execute the vertex in the required way must have objects in this partial H set; reasoning will rectify such a scenario by moving the *indefinite* tuple into the *definite* state $(x,y) \in R_1^+$ thereby satisfying the constraint and executing the vertex. Note that in this case the vertex condition is equivalent to the partial basic query

\[ \{ (x,z) | (x,y) \in R_1^+ \land (y,z) \in R_1^+ \land (x,z) \in R_2^+ \}. \]
This is because the righthand side expression, $R_2^+$, does not introduce any new variables. If the righthand side expression was a more complicated expression that contained new unbound variables then the partial basic query would require more than the available $v = 3$ variables to express the condition.

In general, given a vertex condition, the corresponding partial H set is formed by taking the intersection of the left hand and right hand expressions in the vertex’s constraint. Some partial basic queries that are capable of encoding the left hand and right hand expressions must exist by definition, because the complexity of the language used to specify the basic queries (e.g. the number of query variables and so on) is established in order to express the most complex application constraint expression. The partial H set is then the intersection of the two appropriate basic queries. Furthermore, the variables in the left hand and right hand expressions are independent, and therefore can always be individually encoded as basic queries (otherwise a more complex language would be required that used more query variables).

### 10.4.6.2 Correspondance Between Edge Conditions, $k$-Path Conditions and Partial Scenarios

Each edge and $k$-path condition corresponds to a partial H set that may require a more complex language to express.

**Example 10.4.8.** Consider an edge between vertices $(A)$ and $(C)$ that firstly requires vertex $(A)$ to execute by updating $R_2^+$ which then causes vertex $(C)$ to execute by updating $R_4^+$. To make vertex $(A)$ execute in the appropriate way, the scenario must have the condition

$$\{ (x,z) \mid (x,y) \in R_1^+ \land (y,z) \in R_1^+ \} \supset R_2^+.$$

In order to ensure that vertex $(C)$ executes after vertex $(A)$ the scenario must initially have the condition

$$\{ (x,z) \mid (x,y) \in R_2^+ \land (y,z) \in R_3^+ \} = R_4^+,$$

and then after vertex $(A)$ has executed the condition must be transformed into

$$\{ (x,z) \mid (x,y) \in R_2^+ \land (y,z) \in R_3^+ \} \supset R_4^+.$$

Importantly, to ensure that the sequence of executions is correct, the objects added to $R_2^+$ must be the same objects that satisfy the subsequent condition for vertex $(C)$. This requires an H set that uses more variables than the language allows. To illustrate the limitation, consider a naive approach that represents each execution independently with two partial H sets,

$$\{ (x,z) \mid (x,y) \in R_1^+ \land (y,z) \in R_1^+ \} \cap \{ (x,y) \mid (x,y) \in R_2^+ \} \ (executing \ vertex \ (A)).$$
\[
\{(x, z) \mid (x, y) \in R_2^3 \land (y, z) \in R_3^4\} \cap \{(x, y) \mid (x, y) \in R_4^1\} \text{ (executing vertex } C)\).
\]

The scenario description does not associate the objects that will be added to the \(R_2^3\) relation (when vertex \(A\) executes) with the objects needed in the \(R_3^4\) relation to cause vertex \(C\) to execute. Hence, this scenario description is incorrect and will not necessarily produce the required execution path. The correct partial H set is

\[
\{(x, w) \mid (x, y) \in R_1^+ \land (y, z) \in R_1^+ \land (x, z) \in R_2^+ \land (executing vertex } A))
\]

\[(x, z) \in R_2^3 \land (z, w) \in R_3^4\} \cap \{(x, y) \mid (x, y) \in R_4^1\} \text{ (executing vertex } C)\)

which requires 4 variables.

In general, \(k\)-path conditions will require an expression that combines \(2k - 1\) expressions (where each of the \(k\) constraints in the path has 2 expressions, i.e. the left hand and right hand sides), which is then intersected with the right hand side expression of the \(k^{th}\) constraint. If the variables in any pair of the \(2k - 1\) expressions refer to different objects then the combined expression will have more variables than each expression individually. Because the number of variables permitted in the language is defined as the number required by the most complex expression, it is possible that the combined expression will require more variables than the given language permits.

**Example 10.4.9.** Given the constraint

\[
\{(x, z) \mid (x, y) \in R_1^+ \land (y, z) \in R_1^+\} = R_2^+,
\]

in order to create a scenario that produces the execution path

\[(x, y) \in R_1^+ \rightarrow (x, y) \in R_1^+ \rightarrow (x, y) \in R_1^+
\]

an expression with 5 variables is required,

\[
\{(x_1, x_2) \in R_1^2 \land (x_2, x_3) \in R_1^2 \land (x_1, x_3) \in R_2^+ \land
\]

\[\text{ (updating } (x, y) \in R_1^+ \text{ where } x = x_1 \text{ and } y = x_2)\]

\[(x_4, x_1) \in R_1^2 \land (x_1, x_2) \in R_1^2 \land (x_4, x_2) \in R_2^+ \land
\]

\[\text{ (updating } (x, y) \in R_1^+ \text{ where } x = x_4 \text{ and } y = x_1)\]

\[(x_5, x_4) \in R_1^2 \land (x_4, x_1) \in R_1^2 \land (x_5, x_1) \in R_2^+\}\}

\[\text{ (updating } (x, y) \in R_1^+ \text{ where } x = x_5 \text{ and } y = x_4),\]

because each additional execution requires one new variable. Alternatively, the language used to express the constraint only requires 3 variables.
10.4.7 Utilising Adapted and H-Complexity Test Coverage Metrics

The analysis from the previous section shows that the two coverage approaches complement each other. That is, the adapted coverage metrics provide a focused test set derived from the content of the design (specifically, the conditions required to execute constraints) and the H-complexity metrics provide a broader test set derived from the language.

The adapted coverage criteria derive tests with respect to the *expressions* in the application, by focusing on tests that cause reasoning to modify sequences of relations in constraints. The advantage is that reasoning is directly tested because the content of the design is used to drive testing. The disadvantage is that the expressions in the design are not thoroughly exercised. In particular, testing criteria are satisfied if a constraint is executed, but no tests explore the different ways that a constraint is *not* executed. For example, the adapted metrics will not be able to detect if a relation has been erroneously removed from a constraint.

On the other hand, H-complexity coverage tests with respect to the *language*. For instance, basic query coverage checks how many unique expressions have been exercised out of the set of all possible unique expressions. The advantage is that H-complexity provides a framework for comparing other coverage metrics, as it specifies groups of objects based on the finest possible distinctions that can be made. Furthermore, some of the weaker and more tractable H-complexity coverage metrics test a range of language features regardless of whether they appear in expressions. Thus, they are capable of detecting omissions from constraints unlike the adapted coverage metrics. The disadvantage is that the test class is vast and lacks focus, thus reducing the developer’s chance of selecting effective fault detecting tests.

Another perspective on the relationship between adapted coverage and H-complexity coverage is that of test depth and test breadth. Adapted coverage focuses on execution paths; the language needed to express scenarios that produce the desired execution paths may require more variables than permitted by the language used to express the constraints. Thus, when a developer specifies an edge or a path coverage metric they are potentially employing a more complex language in order to specify a particular test. This can be viewed as a narrow but deep test (in the sense of requiring a complex language) in the test space. Alternatively, H-complexity coverage takes the full test space admitted by the given parameters of the language into account. If the developer aims to achieve coverage greater than 1% with H-complexity metrics then they must develop a large number of tests with subtle variations. This is a broad but shallow (in the sense of the required language expressiveness) set of tests in the test space.

The developer can simultaneously utilise adapted and H-complexity coverage approaches in the following two ways.

1. Employ the two coverage approaches as independent, complementary strategies.
Chapter 10. H-Complexity for Analysing and Validating QSTR Applications

The developer can apply the adapted coverage metrics to produce focused, specific tests that target reasoning paths and then apply the H-complexity coverage metrics to produce tests that thoroughly exercise the essential components of the language.

2. Combine the two coverage approaches into a single testing strategy.

Alternatively, the developer can use the adapted coverage metrics to establish a core test set, and then use concepts from H-complexity to exercise subtle variations on the core tests. Chapter 11 presents a methodology for evaluating the efficacy of competing testing strategies.

10.5 Summary

In this chapter, H-complexity has been defined to measure the complexity of the language required to encode a given expression. The significance for QSTR application development is that the complexity of the language used to encode expressions directly influences the resources required to validate those expressions, and thus by understanding and managing complexity a developer can employ a more efficient and effective validation plan.

H-complexity is derived from the novel concept of homogeneous (H) sets, defined as atomic definable sets. That is, no query exists that can separate two objects within the same H set, thus objects within an H set are equivalent and indistinguishable. H sets can then be used as a measure of the number of expressible queries (or equivalently, the number of distinguishable scenarios) because any accessible subset of objects must be the union of some combination of H sets. Thus, complexity is equal to the number of different combinations of H sets, $2^{|H|}$. A formula for calculating the number of H sets $|H|$ for a given language has been derived.

The formulation of H-complexity highlights properties of language expressiveness that a developer can use to manage complexity when modifying a QSTR application. In particular, the important parameters that determine the complexity of a language, in decreasing order of influence, are relation arity, the number of required query variables, and allowable relation states. Guidelines have been presented for effectively manipulating these parameters to manage complexity.

H-complexity has also been used to define four novel test coverage metrics. A practical methodology has been developed for calculating the H-complexity of a collection of expressions, which can then be used to determine the degree of coverage that a test suite provides. Because H-complexity takes into account the finest degree of distinction that first-order logic is capable of expressing, the resulting test space is intractable. Hence, a crucial component of the H-complexity coverage methodology is a set of guidelines that enable a developer to define a focused, refined test space. A comparative analysis of H-complexity test coverage classes and the test coverage classes adapted from software engineering has shown that the two coverage approaches complement each other; the adapted coverage metrics provide a focused test set de-
rived from the content of the design (specifically, the conditions required to execute constraints) and the H-complexity metrics provide a broader test set derived from the language.

The QSTR validation process methodology presented in this chapter and the previous chapter contains a collection of validation tools and strategies that QSTR application developers can utilise. The next challenge for the developer is deciding which combination of validation approaches should be employed. To address this, the following chapter presents a methodology for evaluating the efficacy of a particular validation technique, thereby allowing the QSTR research community to determine which combination of validation approaches is the most effective for different types of QSTR applications.
Chapter 11

Evaluating Validation Techniques

11.1 Introduction

In Chapters 9 and 10 a collection of validation methodologies were presented that support the developer in determining whether their QSTR application is fit for its intended purpose. In practice, a software project will only be assigned a finite number of testing staff and a limited amount of testing time. The developer is therefore required to formulate a testing plan that balances the tradeoff between testing cost and the achievable degree of confidence that the application meets the formal requirements. Specifically, the developer must produce a testing plan that is likely to reveal the most faults given the available testing resources. It is therefore crucial to establish a meta-validation methodology that QSTR application developers, and the QSTR research community in general, can use to evaluate the relative efficacy of different validation approaches for different types of QSTR applications.

In Chapter 9 fault seeding was used to assess the quality of a given test set. Fault seeding can also be used to evaluate the effectiveness of other validation techniques if it can be shown to reflect real software faults (for example, refer to [25, 94, 156, 172]). In [25] Andrews et al. evaluate manual fault seeding and mutation testing with respect to real faults in imperative software, and conclude that generated mutants satisfactorily approximate real faults. Hence this chapter establishes a QSTR meta-validation methodology by adapting mutation testing approaches that are typically used for evaluating imperative validation methods.

The QSTR evaluation validation methodology enables a developer to conduct experiments that empirically investigate whether a given set of QSTR validation techniques are effective at detecting faults in QSTR application designs. Specifically, the results of the meta-validation experiments identify the types of faults that each validation methodology is likely or unlikely to detect. Furthermore, the experiment results measure the overall efficacy of the validation techniques relative to each other, and relative to a default baseline validation technique such as random testing.
This chapter is organised as follows. In the following section an overview of the evaluation methodology is presented. A key component of the methodology is a repository of QSTR applications; as an alternative to using existing QSTR applications, Section 11.3 presents an approach for generating random QSTR applications that can be used as subject applications for conducting meta-validation experiments. In Section 11.4 the evaluation methodology is used to conduct experiments that analyse the relative efficacy of the validation techniques presented in Chapters 9 and 10. Finally, Section 12.3 discusses the current limitations of the evaluation methodology and highlights the need for a uniform repository of real QSTR applications.

### 11.2 Evaluation Methodology

This section presents an overview of the methodology for evaluating QSTR validation techniques. The evaluation methodology is based on software engineering approaches for investigating the efficacy of imperative software validation techniques, such as [25, 94, 156, 172]. A QSTR application developer can use the methodology to conduct experiments that assess the relative efficacy of a set of validation techniques at detecting design faults.

Automated fault seeding (i.e. mutation testing) is the preferred fault seeding approach as it provides three key advantages over manual fault seeding for conducting experiments [25]. Firstly, mutation operators must be defined in a formal manner, thus facilitating experiment repeatability, as opposed to manual fault seeding which is subjective and inexact. Secondly, the process of generating mutants is completely automatable, hence making it practical to generate large enough quantities of data to obtain statistically significant experiment results, whereas manual fault seeding is a relatively labour intensive and time consuming process. Finally, studies such as [25] have shown that mutation operators tend to produce more realistic faults in imperative software compared to manual seeding approaches as humans tend to seed very obscure faults that are more subtle and difficult to detect than actual software errors.

To conduct meta-validation experiments the developer must provide a repository of clean, fault-free QSTR applications, $A_1, \ldots, A_n$, and a collection of validation techniques, $T_1, \ldots, T_m$, that produce test suites from a QSTR application. For each clean QSTR application $A_i$, and for each validation technique $T_j$, the methodology consists of the following steps as illustrated in Figure 11.1.

1. Apply mutation operators $O$ on $A_i$ to generate a set $M$ of faulty versions of $A_i$; each mutant contains a single fault, inserted using a single mutation operator.

2. For each mutant $m \in M$, create a test suite $T$ using the current validation technique being analysed, $T_j(m) = T$.

---

1The issue of obtaining fault-free applications is discussed further in Section 11.3.
3. Execute each test case \( t \in T \) on the current mutant and the clean application; the scenario produced from the clean application is used as the expected results for the test. Thus, the mutant is killed by \( \mathcal{T}_j \) (i.e. the fault is detected) if \( m(t) \neq \mathcal{A}_i(t) \), for any of the tests in \( T \).

4. Use the results from Step 3 to calculate the **mutation score**; this measures the effectiveness of the validation technique at producing a fault detecting test suite for the current clean QSTR application.

The mutation score for \( \mathcal{T} \) on \( M \) is \( \frac{|\text{killed}|}{|M|} \), where killed is the set of mutants killed by \( \mathcal{T} \). For brevity it is stated that \( \mathcal{T} \) achieves a mutation score on application \( \mathcal{A} \) (rather than \( M \)), meaning that \( \mathcal{T} \) achieves a mutation score on \( M \) for mutation operator \( O \), where \( M = O(\mathcal{A}) \).

The validation technique is used to produce a set of test cases from the faulty version of the application. This reflects the real world situation where a validation technique is used on the current, potentially faulty QSTR application thereby producing a test suite that exercises the design. Once the test suite has been generated, the clean application is employed as an oracle to determine each test’s expected result. If the validation technique generates a test that does not produce the expected results when executed on the mutant, then the test was able to distinguish the faulty application from the correct fault-free application. The mutation scores of different validation techniques for each application can be compared to give the developer a quantitative measure of their relative efficacy at detecting faults. Furthermore, the developer can create a test suite that consists of random tests to function as a control for their experiments.

### 11.2.1 Metrics for Assessing Test Approach Efficacy

A testing approach is effective if it defines a class of tests that, on average, contains a high density of fault detecting tests. That is, for a fixed testset size, a set of tests from a class that contains a higher density of fault detecting tests is more likely to detect a higher number of faults in a given application. Thus, the amount of resources required to employ a testing approach, such as the size of a testset, is also an important factor in determining the most effective testing strategy, in conjunction with the average mutation score. Given application \( \mathcal{A} \), the average testset size \( \bar{N}_T \) using testing approach \( \mathcal{T} \) is

\[
\bar{N}_T = \frac{1}{|M|} \sum_{m \in M} |\mathcal{T}(m)|, \text{ where } M = O(\mathcal{A}).
\]

Thus, a novel metric called functional efficacy \( E_F \) of a testing approach is defined as a measure of the mutation score per test (on average),

\[
E_F = \frac{|\text{killed}|}{|M|}/\bar{N}_T.
\]

Intuitively, this measures the proportion of fault-detecting 'work' that each test performs on average.
Figure 11.1: Overview of evaluation methodology for assessing QSTR validation techniques.
The cognitive burden required to determine an expected test result for a testing approach must also be taken into account when assessing the efficacy of a testing approach. For instance, a testing approach that generates a large number of small, simple scenarios may require less cognitive effort for a human oracle to process compared to a testing approach that generates a small number of large and complex scenarios. Cognitive efficacy is investigated by measuring the average size of the test scenarios specified, the average number of relevant constraints that the oracle must analyse to determine whether they are executed by the initial scenario configuration, and the average number of inference steps that must be performed to determine the final expected scenario.

Three novel metrics will now be defined that enable a developer to investigate the cognitive efficacy of a QSTR application testing approach. The three metrics are

- the average number of objects and assertions per test scenario,
- the average number of salient constraints, and
- the average number of inference steps required to achieve deductive closure of the test scenario.

Firstly a heuristic measure of the complexity of a test scenario is defined as the number of objects and assertions. Specifically, each test consists of a number of assertions about tuples of objects that express a scenario configuration. An assertion simply declares the state of a tuple for a relation, such as \((a, b) \in R^+_{1}.\)\(^2\) The average number of objects per test scenario and the average number of explicit assertions per test are fine-grained measures of the size of a testset generated by a testing approach. If it is assumed that tests with a greater number of objects and assertions are more difficult to process then these measures can be used to provide information on the cognitive effort required to determine the expected test result.

Secondly a measure of the effort required by a human oracle is defined as the number of constraints that a typical human oracle will need to consider when determining the expected test result. That is, when presented with a test scenario the oracle must reason about the scenario to determine the final expected test result. The initial step in the inference process is searching for inference rules that might apply to the scenario. An efficient oracle will not exhaustively consider all rules that exist in the domain. Instead, they will ignore rules that clearly do not apply, leaving a subset of rules that are potentially relevant to a given scenario. It is assumed that a testing approach is more cognitively adequate if the oracle is required to consider fewer constraints on average to determine the expected test result. Based on this assumption, the following measure is defined to estimate the effort required by the oracle to search for applicable constraints: a constraint \(c\) can potentially execute (and thus requires further investigation by the

\(^2\)Theoretically all tuples must take exactly one state for each relation. However in practice the developer can assign one of the states to be the default state, thereby avoiding the burden of explicitly specifying all states for every tuple.
oracle) if the domain of the constraint $\sigma(c)$ contains relations that are non-empty in the scenario $s_t$, that is, $\sigma(c) \cap \sigma(s_t) \neq \emptyset$. That is, the measure is based on the assumption that a constraint requires negligible cognitive effort to discard if the relations in the domain of the constraint are empty in the scenario.

Thirdly the total number of constraint executions required to achieve deductive closure is defined as a measure of the effort required by the human oracle. This measure is based on the observation that in order to determine the scenario that is expected from reasoning over a test’s premise scenario, the oracle must repeatedly execute applicable constraints until no further inferences can be made. Thus, the number of constraint execution iterations is assumed to be proportional to the required cognitive effort.

### 11.2.2 Discrete Sampling Theory

The data obtained from conducting a series of meta-validation experiments measure the efficacy of a testing approach at detecting faults for the given set of applications. That is, the repository of subject applications is only a sample of the complete set of QSTR applications. The developer needs a methodology that uses the experiment results to predict how the testing approach performs on applications that were not present in the repository. This section formalises the evaluation methodology as a sampling process that provides an estimate of the actual average mutation score achieved by a testing approach [115].

Sampling methodologies must formally specify the objects being sampled, the object feature being measured, the population for which the average feature value is being estimated and the subset of the population from which samples are drawn ([115] Chapter 1). The following definitions are used in all experiment procedures that apply this meta-validation methodology.

- The **observation unit** is a QSTR application, $\mathcal{A}$.

- The **feature** being studied, $y$, is the mutation score that $\mathcal{T}$ achieves on $\mathcal{A}$. Let $y_i$ be the mutation score that $\mathcal{T}$ achieves for application $\mathcal{A}_i$.

The following parameters are more specific to a given experiment procedure and may change depending on the objectives and resources available to the group conducting the experiments. The QSTR application developer that is developing an experiment procedure must formally define the following parameters.

- The **target population** is either the set of all QSTR applications or a class of QSTR applications, represented by the universe, $U = \{1, ..., N\}$. 
• The \textit{sampled population} is the set of QSTR applications that could potentially appear in the repository. This may differ from the target population depending on how the repository is compiled.\footnote{For example, if the repository is a collection of existing QSTR applications then applications that have not yet been developed, or applications that the person compiling the repository is not aware of, are excluded from the sampled population but are still in the target population.}

• The \textit{sampling units} are ideally QSTR applications, \( A \), however this may not always be possible.\footnote{For example, although the developer may want to analyse individual QSTR applications, they may not have a list that exhaustively enumerates all of the (real) QSTR applications that exist. Alternatively, the developer can use relevant organisations (such as research teams) as the sampling units, and the observation units are the applications that have been developed by the organisations.}

• The \textit{sampling frame} is the list of sampling units.

A series of \textit{experiments} are conducted by following an experiment procedure. For a given experiment procedure, the parameter that will vary between experiments is the \textit{sample}. The sample is the actual set of QSTR applications, drawn from the sampled population, that are used in a given experiment.

The aim of conducting a series of meta-validation experiments is to determine the population mean \( \bar{y}_U \) and variance \( S^2 \) of mutation scores that \( \Sigma \) achieves on the target population of QSTR applications. The sampling methodology employed is Simple Random Sampling With Replacement (SRSWR).\footnote{Other sampling approaches are discussed in Section 12.3.} Assuming that each QSTR application is equally likely to be selected, and assuming a normal distribution for the mutation scores achieved,\footnote{The central limit theorem is invoked to assume a normal distribution. The central limit theorem requires that the mutation scores achieved for each application are independent of other mutation scores, and that the mutation scores have identical probability distributions. The first criterion is satisfied because test set generation depends only on the given application and is therefore independent of other mutation scores. The second criterion is satisfied if application selection is uniformly distributed. This issue is discussed further in Section 11.3.} then (\cite{115} Chapter 2)

\[
\bar{y}_U = \frac{1}{N} \sum_{i \in U} y_i, \text{ and} \\
S^2 = \frac{1}{N} \sum_{i \in U} (y_i - \bar{y}_U)^2.
\]

A sample \( S \subseteq U \) of the population is used to estimate the population mean and variance. Let \( n = |S| \) be the sample size. The mean \( \bar{y}_S \) and variance \( s^2 \) of the sample is used to estimate (written \( \rightsquigarrow \)) the mean and variance of the population, respectively. Specifically \cite{115}

\[
\bar{y}_S = \frac{1}{n} \sum_{i \in S} y_i, \text{ where } \bar{y}_S \rightsquigarrow \bar{y}_U, \text{ and} \\
s^2 = \frac{1}{n-1} \sum_{i \in S} (y_i - \bar{y}_S)^2, \text{ where } s^2 \rightsquigarrow S^2 (s^2 \text{ is an unbiased estimator}).
\]
\[ \bar{y}_S \pm e, \]

where \( e \) is the margin or error of the estimated mean mutation score \( \bar{y}_S \). The error margin is given by

\[ e = z_{\frac{\alpha}{2}} \frac{S}{\sqrt{n}}, \]

where \( \alpha \) is the confidence level and \( z_{\frac{\alpha}{2}} \) is the \( (1 - \frac{\alpha}{2}) \)th percentile of the normal distribution. Finally, the accuracy of the estimates depends on the size of the sample taken, derived from the confidence interval equations. The precision \( 1 - \alpha \) of the estimate \( \bar{y}_S \) is the probability that the estimate is within a given margin of error \( e \) of the true mean \( \bar{y}_U \),

\[ P(|\bar{y}_S - \bar{y}_U| \leq e) = 1 - \alpha. \]

Rearranging the equation for the margin of error gives the equation for the sample size required to achieve the desired precision,

\[ n = \frac{z_{\frac{\alpha}{2}}^2 S^2}{e^2}. \]

Typically \( \alpha = 0.05 \) and \( e = 0.03 \), meaning that there is a 95\% probability that the given confidence interval contains the true mean \( \bar{y}_U \). The \( (1 - \frac{0.05}{2}) \)th percentile of the standard normal distribution is \( z_{0.05} = 1.96 \).

### 11.3 Generating Random Subject QSTR Applications

A crucial property of the subject repository is that the applications must be uniformly specified for the evaluation experiments to provide useful information. This is because each mutation operator must have exactly one formal definition to ensure that faults are seeded consistently across all applications (as opposed to a mutation operator having multiple definitions to accommodate alternative styles of application implementation). A uniform repository is also necessary to ensure that the developer can consistently analyse the performance of validation techniques across the different applications. A further desirable property is that the QSTR applications are approximately fault-free so that they can function as oracles for determining the expected test results. In particular, the repository must not contain applications with technical faults such as cyclical definitions or contradictory constraints.\(^7\) Design faults, such as the relation in being erroneously defined as symmetric, are difficult to generalise and less critical to the quality of the repository; an exception is when the repository must strongly reflect a realistic distribution of QSTR applications.\(^8\)

---

\(^7\)Many verification tools already exist for analysing knowledge based systems that are effective at detecting technical faults; refer to [81, 128, 131].

\(^8\)Applications with design faults can still accurately function as oracles when analysing seeded faults. However, if the characteristics of relations and constraints are intended to reflect the distribution of realistic QSTR applications, then design faults will skew the distribution. For example, certain properties such as jointly exhaustive, pairwise disjoint (JEPD) may be more likely to hold over low level relations; design faults that interfere with such properties will therefore also skew the distribution of applications in the repository.
As discussed previously in Section 9.6.2, no such gold standard repository of QSTR applications currently exists that can be used to perform experiments using the evaluation methodology. Hence, the developer must compile a custom repository by either selecting existing QSTR applications or by generating QSTR applications. It is difficult to selectively populate a repository as the small number of QSTR applications that do exist have been designed and implemented in different ways (for example, every QSTR application case study presented in Section 2.8 employs a unique implementation platform and style). Furthermore, the existing QSTR applications are typically research prototypes, and hence only informal, ad hoc approaches to validation have been employed to achieve a proof-of-concept rather than to ensure that the applications are robust and fault-free. An alternative generative approach is to populate the repository with randomly created QSTR applications. Generating random data is a common method for conducting experiments in software engineering and artificial intelligence. For instance, in constraint satisfaction programming [71] and the QSTR community (Section 6.1 in [137]), researchers routinely measure the efficacy of software tools on randomly created problems. In particular, one area of AI research focuses specifically on generating random ontologies (for example, [155] and Section 6.3 in [86]), which is analogous to creating random QSTR applications.

There are a number of important advantages to generating random applications. Firstly, the developer has access to all QSTR applications that are defined by the application model, and thus the sampled population can, in theory, be equal to the target population; this reduces the probability of selection bias. Secondly, the developer can directly select applications from the sampled population and thus further reduce the probability of selection bias, that is, the sampling unit is the observation unit (refer to Section 11.2.2). Thirdly, the developer can quickly and easily generate samples of applications that are large enough to be statistically significant. Finally, applications are generated in a uniform way, thus satisfying a critical requirement of the repository.

Generating random applications firstly requires a model that describes QSTR applications. An appropriate model of QSTR applications has been developed in Chapters 3 and 5 from an analysis of real QSTR applications. In Section 3.2 QSTR applications were defined as a set of relations and a set of constraints between those relations. Section 5.2 then established a basic structure to QSTR applications by grouping relations into fragments. The model was then further developed in Section 5.3 by establishing basic types of constraints that typically

---

9 Another selective approach that was explored was to use a related type of software application that approximated QSTR applications, namely knowledge based and expert software systems. For example, the Protege ontology development community maintains an open repository of ontologies [127]. The intention was that the results of the experiments would then partially support the claim that the validation techniques are effective for the related portion of QSTR applications. However, to evaluate the soundness of a mapping between domains itself requires a repository of realistic QSTR applications. Therefore, it was found that this approach only pushes the problem back one step.

10 Other generative approaches include mutating and generalising instances of real applications using research in genetic programming, case based reasoning, and so on [142].
occur within fragments and between fragments. In this section salient parameters are defined that describe QSTR applications based on the model of QSTR applications established in this thesis. Probability mass functions are then defined over the parameter values to specify the distribution of QSTR applications that are to be randomly generated.

11.3.1 Random Variables for Representing the Distribution of QSTR Applications

QSTR applications fundamentally consist of a set of relations and a set of constraints. In this section, these two parameters are used to establish random variables that specify distributions of QSTR applications. As defined in Section 3.2, each relation has an arity $a_R$ and a set of permitted tuple states $A_R$. For simplicity, it is assumed that all relations have three allowable states, $A_R = \{+, -, ?\}$. Two random variables will now be defined that directly refer to the set of relations and the probable arity of a given relation $R$. Let random variable $X_{|R|}$ represent the number of relations in an application, $|R|$, with a maximum possible size of $N_R$,

$$X_{|R|} = \begin{cases} 1, & \text{if } |R| = 1, \\ 2, & \text{if } |R| = 2, \\ \vdots, & \\ N_R, & \text{if } |R| = N_R. \end{cases}$$

Let random variable $X^{a_R}$ represent the arity of a given relation $R$, with a maximum possible arity of $N_{a_R}$,

$$X^{a_R} = \begin{cases} 0, & \text{if } a_R = 0, \\ 1, & \text{if } a_R = 1, \\ \vdots & \\ N_{a_R}, & \text{if } a_R = N_{a_R}. \end{cases}$$

Probability mass functions $f_{X_{|R|}}$ and $f_{X^{a_R}}$ will now be defined that allow developers to specify different distributions of QSTR applications according to the characteristics of relations. Let $p^{|R|}_i$ be the probability that an application has $i$ relations, and let $p^{a_R}_j$ be the probability that a relation has arity $j$. Then,

$$f_{X_{|R|}}(x) = \begin{cases} p^{|R|}_1, & x = 1, \\ p^{|R|}_2, & x = 2, \\ \vdots & \\ p^{|R|}_{N_R}, & x = N_R, \\ 0, & x \notin \{1, \ldots, N_R\}, \end{cases}$$
11.3 Generating Random Subject QSTR Applications

The developer can configure the distribution of generated QSTR applications according to their objectives for conducting meta-validation experiments by selecting appropriate values for $p_i^R$ and $p_j^{AR}$. Without further information, the developer can employ default probability mass functions that uniformly distribute values within an appropriate range. That is, the probability $p_k$ of a random variable taking the value $k$ is $p_k = \frac{1}{b-a}$, where $k = a \ldots b$ for some $a, b$.

In Section 10.4.6 it was shown that constraints are equivalent to pairs of expressions (as defined by H-complexity in Section 10.2.3). Random variables that represent arbitrary constraints will now be defined for the purpose of explicitly formalising the foundation of randomly generated applications. It must be noted, however, that it is considerably more practical and convenient to specify random variables that represent different classes of constraints, as presented in the following sections. As shown in Section 10.2.3, there are an infinite number of possible constraints that can be specified for any non-empty set of relations $R$, because there is no theoretical limit on the number of variables that can be used to specify an expression. A common approach used in finite model theory for controlling the cardinality of expressions is to limit the number of allowable query variables to $v$. Each value for $v$ defines a unique, finite set of expressible constraints, referred to as the $v$th constraint space. The size of the $v$th constraint space, denoted $c_v$ can be calculated using H-complexity theory as follows. Constraints are either complete basic queries or partial basic queries (i.e. some union of complete basic queries). Therefore, the total number of constraints is equal to the number of combinations of basic queries. That is,

$$c_v = |H| = 2 \uparrow \left( q \prod_{R \in \mathbb{R}} (|A_R|)^{v_R} \right).$$

Let $g_v$ be an integer encoding that maps each integer from 1 to $c_v$ to a unique constraint that uses $v$ query variables. Let random variable $X^C_{v,i}$ represent the inclusion of constraint $g_v(i)$ in an application, for $v = 1 \ldots N_v$,

$$X^C_{v,i} = \begin{cases} 
1, & \text{if constraint } g_1(i) \text{ is in the application}, \\
0, & \text{if } g_1(i) \text{ is not in the application}, \\
& \text{for } i = 1 \ldots c_1, \\
& \ldots, \\
1, & \text{if constraint } g_N_v(i) \text{ is in the application}, \\
0, & \text{if } g_N_v(i) \text{ is not in the application}, \\
& \text{for } i = 1 \ldots c_{N_v}. 
\end{cases}$$
Probability mass functions $f_{X^C}$ for each random variable $X^C_{vi}$ will now be defined that allow developers to specify different distributions of QSTR applications according to the characteristics of constraints. Let $p^C_{vi}$ be the probability that constraint $g_v(i)$ is in a randomly generated application. Then,

$$f_{X^C_{vi}}(x) = \begin{cases} 
  p^C_{vi}, & x = 1, \\
  1 - p^C_{vi}, & x = 0, \\
  0, & x \not\in \{0,1\}.
\end{cases}$$

To specify a uniform distribution for all constraints with variables $v = 1 \ldots N_v$, the developer must specify the mean number of constraints per application $\bar{N}_c$. The total number of constraints with variables $v = 1 \ldots N_v$ is $\sum_{v=1}^{N_v} c_v$, therefore the probability $p^C$ that a given constraint is selected according to a uniform distribution is $p^C = \frac{\bar{N}_c}{\sum_{v=1}^{N_v} c_v}$.

In some cases the distribution of constraints is determined by the application domain, or other factors specified in the formal requirements of a development project. A practical approach for controlling the distribution of constraints is to define probability mass functions over classes of constraints. Some basic classes can be derived directly from the variables that define constraints; for example, one class distinction is the size of the subset of relations used in constraints, i.e. the size of the constraint domain, $|\sigma(c)|$. The following sections specify important higher-level classes of constraints based on the QSTR application model refinements that have been presented in Chapter 5.

### 11.3.2 Parameters for the Structured Model of QSTR Applications

QSTR fragments partition relations and constraints into cohesive groups (refer to Chapter 5). By incorporating fragments into the model used to generate random applications, the developer is able to specify classes of constraints (for example, constraints that define neighbourhoods, transitivity, and so on) and then specify probability mass functions for each constraint type. Specifying probability mass functions for higher-level modelling concepts facilitates the generation of more realistic QSTR applications.

Given a set of relations $R$, the developer needs to specify how the relations will be partitioned into fragments. A partition $P$ of $R$ is a set of jointly exhaustive, non-overlapping subsets of relations in $R$, i.e., $\bigtriangleup S = R$, where $\bigtriangleup$ is the symmetric difference operator. The number of possible partitions of $R$ is the $n$th Bell number,

$$B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k,$$

where $n = |R|$.

Given an integer encoding that maps each partition to a unique integer in $\{0, \ldots, B_n\}$, partitioning the set of relations into fragments can be represented by the random variable,
11.3 Generating Random Subject QSTR Applications

\[ X_n^F = \begin{cases} 
0, & \text{if partition selected is encoded as 1}, \\
\ldots, \\
B_n, & \text{if partition selected is encoded as } B_n.
\end{cases} \]

The simplest approach is to assume a uniform distribution over the possible partitions, that is, each partition of the set of relations into fragments is equally likely. To improve this model, the developer can rule out partitions that typically never occur in real QSTR applications, such as fragments that contain relations with different arities.

Alternatively, it may be more convenient to define random variables that directly refer to the set of fragments and the size of each fragment in an application. Let \( F \) be the set of fragments in an application. Let random variable \( X_{n|F} \) represent the number of fragments,

\[ X_{n|F} = \begin{cases} 
1, & \text{if } |F| = 1, \\
\ldots, \\
n, & \text{if } |F| = n,
\end{cases} \]

and let \( X_{|F|}^{m,m'} \) represent the number of relations in a given fragment \( F \),

\[ X_{|F|}^{m,m'} = \begin{cases} 
m, & \text{if } |F| = m, \\
\ldots, \\
m', & \text{if } |F| = m',
\end{cases} \]

where \( m \) and \( m' \) are the minimum and maximum number of allowable relations in \( F \) respectively. This approach is particularly useful if the set of relations is generated incrementally in conjunction with each fragment (as presented in Section 11.4).

11.3.3 Parameters for Constraints within a Fragment

As presented in Section 5.2, particular types of constraints frequently apply to relations that are grouped together in fragments, such as reflexivity and conceptual neighbours. To incorporate constraints that apply exclusively to relations within a fragment, the developer needs to specify probability mass functions for random variables that represent different constraint types. Let \( C_a \) represent a constraint type that applies to tuples with arity \( a \). Constraints for relations within a fragment are

\[ C_1 \in \{ \text{reflexive, irreflexive, symmetric, asymmetric} \}, \]

\[ C_2 \in \{ \text{inverse, conceptual neighbours} \}, \text{ and} \]

\[ C_i \in \{ \text{transitive} \}, \text{ for } i \geq 3. \]
Let $C \sim$ refer to fragment constraints that apply to unordered relation subsets of arbitrary size rather than tuples. Then

$$C \sim \in \{\text{disjoint, jointly exhaustive}\}.$$  

The specification of a fragment constraint holding for a given $a$-tuple of relations is represented by the random variable $X_{C_a}$,

$$X_{C_1} = \begin{cases} 1, & \text{constraint holds for given relation}, \\ 0, & \text{constraint does not hold for given relation}, \end{cases}$$

$$X_{C_2} = \begin{cases} 1, & \text{constraint holds for given relation pair}, \\ 0, & \text{constraint does not hold for given relation pair}, \end{cases}$$

... 

The specification of a constraint holding for a given unordered subset of relations in a fragment is given by the random variable $X_{C \sim}$,

$$X_{C \sim} = \begin{cases} 1, & \text{constraint holds for given relation subset}, \\ 0, & \text{constraint does not hold for given relation subset}. \end{cases}$$  

The developer can now specify the distribution of higher-level constraints by using the probability mass function $f_{X_{C\sim}}$ for each random variable $X_{C_a}$. Let $p_{C_a}$ be the probability that constraint type $C$ holds for a given $a$-tuple of relations (or a given relation subset where $a = \sim$),

$$f_{X_{C_a}}(x) = \begin{cases} p_{C_a}, & x = 1, \\ 1 - p_{C_a}, & x = 0, \\ 0, & x \notin \{0, 1\}. \end{cases}$$

### 11.3.4 Parameters for Constraints between Fragments

Section 5.3 defined two common constraints that apply to relations between fragments, namely definitions and generalisations. Both of these constraints form hierarchies, that is, given fragments $F_1$, $F_2$ and $F_3$, if $F_1$ is defined by $F_2$, and $F_2$ is defined by $F_3$ then $F_3$ cannot be defined by $F_1$. Thus, a valid set of definitions (or generalisations) can be represented as a directed graph with no cycles, where graph vertices represent fragments, the vertex at the head of an edge represents a defined fragment and the vertex at the tail of an edge represents a defining fragment. Given a set of fragments, the set of possible definitions (or generalisations) is equivalent to the set of directed graphs with no cycles, where each graph vertex corresponds to a fragment. The size of the set of possible definitions, $N$, is the number of partially ordered sets with $n$ labelled elements [151], where $n$ is equal to the number of fragments. Given an integer encoding that maps each directed acyclic graph (DAG) to a unique integer in $\{1, \ldots, N\}$, directed acyclic graphs can be represented by the random variable,
11.4 An Experiment for Evaluating the Presented QSTR Validation Techniques

A series of meta-validation experiments have been conducted to analyse the relative efficacy of the QSTR validation techniques presented in Chapter 9. This section presents the experiment procedure, the results of the experiments, and an analysis of the validation approaches based on the experiment results.

11.4.1 Experiment Procedure

This section presents the details of the experiment procedure used to conduct the meta-validation experiments. The experiment procedure employs the following sampling parameters (refer to Section 11.2.2 for definitions of the sampling terms).

- The target population is the set of all QSTR applications.
- The sampled population is restricted to QSTR applications where
  - constraints take the form $LHS \subseteq RHS$ where $RHS$ is a singleton,
  - accepted constraint types are jointly exhaustive, disjoint, neighbour, and definition,
  - one scenario is modelled at a time (e.g. dynamic scenarios are not modelled),
  - the number of fragments per application is between 1 and 20,
  - the number of relations in each fragment is between 1 and 10, and
  - all relations have three states: holds (+), does not hold (−), and indefinite, (?).
- The sampling units are QSTR applications, $\mathfrak{A}$.
- The sampling frame is the QSTR application space defined by the above restrictions; applications are generated randomly from this space.

$$X_n^D = \begin{cases} 1, & \text{if } DAG \text{ selected is encoded as } 1, \\ \ldots, \\ N, & \text{if } DAG \text{ selected is encoded as } N. \end{cases}$$

To incorporate these fragment constraints the developer must specify a probability mass function for the set of corresponding directed acyclic graphs. For example, a uniform probability distribution is specified by the mass function

$$f_{X_n^D} (x) = \begin{cases} \frac{1}{N}, & x \in \{1, \ldots, N\}, \\ 0, & x \notin \{1, \ldots, N\}. \end{cases}$$
As previously mentioned, the specified sampled population was selected to reflect the characteristics of the case studies presented in Section 2.8. Because most of the case studies are small research prototypes (such as Image Retrieval; refer to Section 2.8.4), it is intended that the sampled population has a slight bias towards more complex applications such as QtvLight (refer to Section 2.8.1) as opposed to taking the mean value for any given feature of the case studies. Furthermore, the majority of the case studies modelled static scenarios, and thus, applications that employ dynamic models were not included in the sampled population.

The experiments are automated using a script that performs the following activities based on the evaluation methodology presented in Section 11.2. Appendix A presents excerpts of the Prolog test scripts and test result files employed in these experiments.

1. A random QSTR application is created and saved to a file.

2. The mutation operators are applied to the application to generate faulty mutant versions, and each mutant is saved to a file.\(^{11}\)

3. For each validation methodology, for each mutant, a test suite is created and stored as a Prolog script; the expected test results are obtained by referring to the clean QSTR application.

4. The test suite is then executed on the mutant by running the test script, and the results are used to determine if the test suite detected the fault. The results are recorded in a text file, and eventually compiled to calculate the mutation scores.

Two types of tests are employed: expression-based tests and inference-based tests. Expression-based tests isolate expressions and exercise them independently. Each expression-based test consists of

- a name (predicate \textit{testname\_id}),
- a test expression (predicate \textit{test\_id}) derived from the faulty application that executes a query and collects the results,
- the expected expression (predicate \textit{expected\_id}) derived from the clean application,
- a collection of scenarios (predicate \textit{scenario\_id}) derived from the faulty application that populate relations with objects, and
- a script (predicate \textit{passes\_test\_id} and \textit{passes\_all\_tests\_id}) for setting up and executing the tests.

\(^{11}\)For this experiment procedure all types of faults are applied to all applications. That is, no distinction is made between different types of QSTR applications.
Each scenario represents a test, and each expression represents a set of tests (e.g. a unit test set that is exercising a particular constraint expression). A test scenario passes if the set of objects returned when the test expression is executed is the same as the set of objects returned when the expected expression is executed.

Inference-based tests exercise runtime inference, for example, to test sequences of vertex executions. Each inference-based test consists of

- a name (predicate testname_[id]),
- a single scenario (predicate scenario_[id]) derived from the faulty application,
- a test (predicate passes_test_[id]) that simultaneously loads the scenario into both the faulty and clean versions, runs the inference task on both applications, and compares the final set of object tuples in each relation between the faulty and clean versions, and
- a script (predicate passes_all_tests_[id]) for setting up and executing the tests.

Each scenario represents a single test. A test scenario passes if, after running the inference process on both the faulty and clean versions, the set of object tuples present in each relation of the faulty application is the same as the set of object tuples in the corresponding relation in the clean application.

One test script is generated for each clean application. The script executes each test and records the results. If the faulty application passes all the scenario tests for a given test set, then the test suite was not able to detect the mutant’s fault. Alternatively, if at least one test failed then the fault was detected and the mutant was killed. For each mutant, if at least one test set was able to detect the fault then the mutant was killed. The mutation score of the validation technique for the given clean application is the proportion of killed mutants. The mutation scores across all clean applications can be plotted in a histogram or aggregated to determine an average mutation score.

### 11.4.2 Experiment Configuration and Parameter Values for Generating Random QSTR Applications

The key parameters of the experiment are the mutation operators that were used to seed faults, the testing approaches that were being evaluated, and specific details about the sampled population of QSTR applications. Four of the mutation operators defined in Section 9.6 were used to seed faults in the generated applications, namely Neighbour Replace (NR), Instance Creation Expression (ICE), abs, and Remove Relation. The mutation operator ror was excluded because the sampled population required that all constraints use the set operator $\subseteq$. The neighbourhood edge manipulation mutation operator was excluded because the sampled applications only modelled static scenarios that did not directly employ neighbourhoods for reasoning.
Three validation techniques from Chapter 9 were used in the experiments: unit testing (Section 9.3), vertex testing (Section 9.5) and vertex condition testing (Section 9.5). These were selected to represent a combination of contrasting testing approaches. Specifically, unit testing is an expression-based approach that thoroughly exercises all combinations of relation states. In contrast, vertex testing is an inference-based approach that is more selective about the combinations of relation states that are tested. Vertex condition testing is an exhaustive version of vertex testing, and therefore was included to investigate the properties of test approach subsumption. Random scenario testing (inference-based) and random unit testing (expression-based) were used as controls.

The generator that was developed to produce random QSTR applications operates by incrementally sampling random variables and then using the value to construct the associated QSTR application component. Firstly, the generator creates a randomly chosen number of empty fragments between 1 and 20. It then populates each fragment with a randomly chosen number of generated relations between 1 and 10, with the restriction that all relations in the same fragment must have the same arity. The number of fragments and the number of relations in a given fragment are uniformly distributed, that is,

\[
f_{X_{|F}^{20}}(x) = \begin{cases} \frac{1}{20}, & x \in \{1, \ldots, 20\}, \\ 0, & x \notin \{1, \ldots, 20\}, \end{cases}
\]

\[
f_{X_{|F}^{10}}(x) = \begin{cases} \frac{1}{10}, & x \in \{1, \ldots, 10\}, \\ 0, & x \notin \{1, \ldots, 10\}. \end{cases}
\]

It follows that the generator can produce applications that have between 1 and 200 relations, however, the distribution of the total number of relations is not uniform. Figure 11.2 presents an estimation of the probability mass function, \( f_{X_{|R}} \), using \( 1 \times 10^9 \) data points. The mean is estimated to be 57.74 (2 d.p.) and the standard deviation is estimated to be 33.05 (2 d.p.). The distribution shows that this approach is strongly biased against generating QSTR applications that have above approximately 140 relations; the probability of an application being generated that has the maximum number of 200 relations is extremely low, \( \Pr(X_{|R} = 200) = \frac{1}{20} \cdot \left(\frac{1}{10^{10}}\right) \). It is important to note however that the estimated distribution is consistent with the mean number of relations per QSTR application in the case studies, 31 (standard deviation 30; refer to Table 5.3, Section 5.2), with a slight bias towards larger and more complex QSTR applications as intended. Thus, despite its non-uniformity, the distribution is ideal for the current experiment procedure.

The relations have uniformly distributed arities between 1 and 3,

\(^{12}\)It turns out that the constraint types used in the experiments only have trigger and independent conditions as defined in ??, and thus unit testing did not exercise interactions between condition classes. That is, each unit test set consisted of \( t + i \) tests as defined in 9.3.3.
11.4 An Experiment for Evaluating the Presented QSTR Validation Techniques

Figure 11.2: The estimated probability mass function, \( f_{X|R} \), of the total number of relations in a QSTR application, generated by first specifying the number of fragments, and then incrementally populating each fragment with a random number of relations, uniformly distributed between 1 and 10.

\[
f_{X|R}(x) = \begin{cases} 
\frac{1}{3}, & x \in \{1, 2, 3\}, \\
0, & x \notin \{1, 2, 3\}.
\end{cases}
\]

For each fragment, the generator specifies exactly one random subset of relations as disjoint, and exactly one random subset as jointly exhaustive,

\[
f_{X^-}(x) = \begin{cases} 
\frac{1}{2^{\pi}}, & x = 1, \\
1 - \frac{1}{2^\pi}, & x = 0, \\
0, & x \notin \{0, 1\}.
\end{cases}
\]

A simple and practical method was used for generating constraints between fragments by defining hierarchy tiers that consist of a set of fragments.\(^{13}\) A fragment can only be defined by other fragments in the same or lower tiers thereby preventing cycles. This also allows the developer to control other parameters such as the distribution of the lengths of definition chains.

11.4.3 Pilot Study for Estimating the Standard Deviation of Mutation Scores

Achieving the desired precision requires selecting a large enough sample size \( n \). As shown in Section 11.2.2, the sample size required to achieve a given degree of precision depends on the standard deviation of the target population,

\(^{13}\)A similar approach was used in [86], refer to Section 6.3.
Chapter 11. Evaluating Validation Techniques

<table>
<thead>
<tr>
<th>Neighbour Replace:</th>
<th>Absolute:</th>
<th>Instance Creation Expression:</th>
<th>Remove Relation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean (St’d Dev.)</td>
<td>Mean (St’d Dev.)</td>
<td>Mean (St’d Dev.)</td>
<td>Mean (St’d Dev.)</td>
</tr>
<tr>
<td>Unit Testing</td>
<td>1.0000 (0.0000)</td>
<td>1.0000 (0.0000)</td>
<td>1.0000 (0.0000)</td>
</tr>
<tr>
<td>Vertex Testing</td>
<td>0.9417 (0.0706)</td>
<td>0.9647 (0.0676)</td>
<td>0.9726 (0.1038)</td>
</tr>
<tr>
<td>Vertex Condition Testing</td>
<td>0.9787 (0.0466)</td>
<td>0.9997 (0.0017)</td>
<td>0.9978 (0.0084)</td>
</tr>
<tr>
<td>Random Testing</td>
<td>0.4797 (0.1525)</td>
<td>0.4861 (0.2037)</td>
<td>0.6238 (0.3550)</td>
</tr>
<tr>
<td>Random Unit Testing</td>
<td>0.4279 (0.1643)</td>
<td>0.4085 (0.1817)</td>
<td>0.6078 (0.3488)</td>
</tr>
</tbody>
</table>

Table 11.1: Mean and standard deviation of mutation scores (to 4 decimal places) for five testing approaches resulting from the pilot study with sample size $n_p = 30$. The cell with the highest standard deviation (Random Testing, Instance Creation Expression) is highlighted.

\[
n = \frac{z^2 \sigma^2}{e^2}.
\]

A small pilot study was conducted to estimate the standard deviation of the target population \( s_p \leadsto S \). The estimate, \( s_p \), was then used to determine a feasible sample size \( n \) for the main experiments. The size \( n_p \) of the pilot study sample \( S_p \) was selected to be 30. After the experiments were conducted, the pilot study sample mean \( \bar{y}_{S_p} \) and variance \( s_{p}^2 \) were calculated,

\[
\bar{y}_{S_p} = \frac{1}{n_p} \sum_{i \in S_p} y_i,
\]

\[
s_{p}^2 = \frac{1}{n_p-1} \sum_{i \in S_p} (y_i - \bar{y}_{S_p})^2.
\]

The mean and standard deviation of the mutation scores for each testing approach resulting from the pilot study are presented in Table 11.1. The results show a range of standard deviations from 0 to 0.3550; the largest standard deviation was taken as the estimate of the target population standard deviation, 0.3550 \( \leadsto S \), as this value will yield the minimum sample size required to achieve the necessary precision for all testing approaches.

Table 11.2 presents the mean and standard deviation of the number of relations, fragments, conceptual neighbours, and constraints per application. The mean number of relations and fragments per application, approximately 45 (standard deviation 25) and 10 (standard deviation 5) respectively, is consistent with the results from the case studies presented in Section 2.8 where the mean number of relations and fragments were shown in Section 5.2, Table 5.3 to be 31 (standard deviation of 30) and 6 (standard deviation 5), respectively; as intended, there is a slight bias towards larger and more complex applications such as QtvLight. Hence, the repository of generated applications is ideal for conducting the pilot study.
An Experiment for Evaluating the Presented QSTR Validation Techniques

Table 11.2: Mean and standard deviation of the number of components per QSTR application (to 4 decimal places).

<table>
<thead>
<tr>
<th></th>
<th>Mean number per application</th>
<th>St’d Dev. per application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relations</td>
<td>45.2</td>
<td>24.6</td>
</tr>
<tr>
<td>Fragments</td>
<td>9.6</td>
<td>5.2</td>
</tr>
<tr>
<td>Conceptual Neighbours</td>
<td>35.6</td>
<td>19.6</td>
</tr>
<tr>
<td>Constraints</td>
<td>20.8</td>
<td>9.9</td>
</tr>
<tr>
<td>Definitions</td>
<td>8.3</td>
<td>5.2</td>
</tr>
<tr>
<td>(Definition size)</td>
<td>1.1</td>
<td>0.4</td>
</tr>
<tr>
<td>Disjoint</td>
<td>6.5</td>
<td>4.1</td>
</tr>
<tr>
<td>Jointly Exhaustive</td>
<td>6.0</td>
<td>3.5</td>
</tr>
</tbody>
</table>

11.4.4 Experiment Results and Analysis

The central objective for conducting the experiments is to calculate the confidence interval, $\bar{y} \pm e$. This interval provides an estimate of the means of the mutation scores for the five testing approaches, with a 5% error margin and 95% confidence. This estimate is used as a measure of the relative efficacy of each testing approach at detecting faults in QSTR applications.

The sample size for the experiments, $n_0$, is determined by the estimate of the standard deviation $s_p$ of mutation scores on the target population. The estimate from the pilot study was $s_p = 0.3550$. Therefore, rounding up to the nearest integer gives

$$n_0 = \left\lceil \frac{1.96^2 s_p^2}{0.05^2} \right\rceil = 194.$$ 

A small tolerance is added by rounding up to the nearest ten, giving $n_0 = 200$. Thus, a sample $S$ was taken by generating 200 random QSTR applications using the probability mass functions presented in the previous sections.

11.4.4.1 Characteristics of the Generated QSTR Applications and Seeded Faults

This section presents a summary of the properties of the generated QSTR applications and the faults that were seeded. The characteristics of the generated applications are consistent with the case studies, and thus the sample of applications in the repository is ideal for conducting experiments. Table 11.3 presents the mean and standard deviation of the number of relations, fragments, conceptual neighbours, constraints, implemented code features, and the average number of assertions in each implemented code feature per application. The results show that the number of relations and fragments per generated application, approximately 49 (standard deviation 28) and 10 (standard deviation 5) respectively, is similar to the case studies in Section 2.8 where the mean number of relations and fragments were shown to be 31 (standard deviation 30) and 6 (standard deviation 5), respectively as presented in Section 5.2, Table 5.3.

A distinction is made between the number of constraints specified in the design of a QSTR application and the number of actual code features that is used to implement a constraint in
### Table 11.3: Mean and standard deviation of the number of components per QSTR application to 4 decimal places.

<table>
<thead>
<tr>
<th>Component</th>
<th>Mean number per application</th>
<th>St’d Dev. per application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relations</td>
<td>49.4</td>
<td>28.2</td>
</tr>
<tr>
<td>Fragments</td>
<td>10.0</td>
<td>5.4</td>
</tr>
<tr>
<td>Conceptual Neighbours</td>
<td>39.4</td>
<td>23.0</td>
</tr>
<tr>
<td>Constraints</td>
<td>22.2</td>
<td>11.3</td>
</tr>
<tr>
<td>Definitions</td>
<td>8.7</td>
<td>6.2</td>
</tr>
<tr>
<td>(Definition size)</td>
<td>1.2</td>
<td>0.5</td>
</tr>
<tr>
<td>Disjoint</td>
<td>6.8</td>
<td>3.9</td>
</tr>
<tr>
<td>Jointly Exhaustive</td>
<td>6.8</td>
<td>4.0</td>
</tr>
<tr>
<td>Implemented Code Features</td>
<td>61.3</td>
<td>34.3</td>
</tr>
<tr>
<td>Average Assertions per Implemented Feature</td>
<td>4.4</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Table 11.3: Mean and standard deviation of the number of components per QSTR application to 4 decimal places.

A particular programming language, such as Prolog predicates or SQL queries. For example, specifying a particular subset of relations as disjoint is counted as one constraint, whereas it may take a number Prolog predicates to implement it in software. The number of code features used is specific to the programming language and is therefore not an inherent property of the application design. However, the implemented code features are the components that are subject to fault seeding and mutation, and thus the number of such features is relevant to the results of these experiments.

Table 11.4 presents the mean and standard deviation of the number of mutants generated (equivalently, the number of seeded faults) per application. The maximum number of mutants was generated for each fault type; for example, the absolute fault produced one mutant for every assertion (i.e. the occurrence of a relation in a constraint). A tractable number of mutants was generated per application, thereby making it practical to conduct the experiments using a statistically significant sample size.

The number of seeded ICE faults is significantly lower compared to the other types of seeded faults, despite the mean number of definitions per application being higher than the means of the other types of constraints. This is because ICE only replaces a higher level relation with reference relations that have the same arity. In the generated applications for this experiment procedure, relation arities are uniformly distributed between 1 and 3. Moreover, fragments can only contain relations with the same arity, but a relation can only be defined by reference relations in different fragments. Hence, a number of relations with the correct arity can never
function as reference relations because they are in the same fragment as the high level relation. Thus, less than one third of the application relations qualify as valid replacements for ICE faults, thereby explaining the comparatively low number of ICE faults.

The most influential property of applications on the number of mutants generated \(|M|\) is the number of implemented code features. Pearson’s product-moment correlation coefficient \(\rho_{|M|,X}\) is used to measure the relative influence that an application parameter \(X\) has on the number of mutants generated \(|M|\), with expected values \(\mu_{|M|}\) and \(\mu_X\), and standard deviations \(\sigma_{|M|}\) and \(\sigma_X\), [59]

\[
\rho_{|M|,X} = \frac{E[(|M| - \mu_{|M|})(X - \mu_X)]}{\sigma_{|M|}\sigma_X},
\]

where \(E\) is the standard expected value operator. Table 11.5 presents the linear correlation coefficients for the salient QSTR application parameters. The results show that the number of implemented code features has the largest influence on \(|M|\), whereas the size of the implemented features, measured as the average number of assertions per implemented feature, is not strongly related to \(|M|\). Interestingly, the number of conceptual neighbours and the number of relations in an application have as strong of an influence on \(|M|\) as the number of design constraints. The number of definitions is least indicative of the number of generated mutants because of the relatively small number of ICE faults.

### 11.4.4.2 Mutation Scores

The mean and standard deviation of the mutation scores for each testing approach are presented in Table 11.6. The results show that unit testing is the only testing approach that detected all neighbour replace faults, although the mutation scores for vertex testing, 0.9447, and vertex condition testing 0.9813 were also very high. Both unit testing and vertex condition testing detected almost all absolute and ICE faults, with vertex testing once again achieving very high mutation scores, 0.9643 and 0.9923, for absolute and ICE faults respectively. Vertex condition
testing achieved the highest mutation score of 0.9702 for the remove relation faults, followed by vertex testing, 0.8490, and unit testing, 0.5776. Random unit testing performed the worst overall, with mutation scores of approximately 0.46 for neighbour replace, absolute and remove relation.

The results also show that the performance of a testing approach depends strongly (or entirely, if the testing approach is deterministic) on the properties of the application and the seeded faults being tested. For example, unit testing can detect all neighbour replace, absolute and instance creation expression (ICE) faults, however it performs extremely poorly on applications that have constraints with disjoined terms and remove relation faults. As discussed in Section 9.6.7, the problem is that the faulty expression can be satisfied without referring to the erroneously removed relations. For example, if relation $R_2$ is removed from the following expression,$\{x| x \in R_1 \lor x \in R_2\}$, then the resulting faulty expression that is used to generate unit tests is $\{x| x \in R_1\}$.

Unit testing (without interaction testing between condition classes, refer to 11.4.2) will create two tests,

\[
test 1: R_1^+ = \{a\}, \text{ expected expression result: } \{a\}
\]
Table 11.6: Estimated means (\( \bar{y}_S \sim \bar{y}_U \)) and standard deviations (\( s \sim S \)) of mutation scores for the five test approaches for four different fault types. For each fault type, highlighted cells contain the highest mean mutation score.

<table>
<thead>
<tr>
<th>Neighbour Replace:</th>
<th>Absolute:</th>
<th>Instance Creation Expression:</th>
<th>Remove Relation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean ± Error (St'd Dev.)</td>
<td>Mean ± Error (St'd Dev.)</td>
<td>Mean ± Error (St'd Dev.)</td>
<td>Mean ± Error (St'd Dev.)</td>
</tr>
<tr>
<td>Unit Testing</td>
<td>1.0000 ± 0.0000 (0.0000)</td>
<td>1.0000 ± 0.0000 (0.0000)</td>
<td>1.0000 ± 0.0000 (0.0000)</td>
</tr>
<tr>
<td>Vertex Testing</td>
<td>0.9447 ± 0.0072 (0.0519)</td>
<td>0.9643 ± 0.0067 (0.0485)</td>
<td>0.9923 ± 0.0051 (0.0366)</td>
</tr>
<tr>
<td>Vertex Condition Testing</td>
<td>0.9813 ± 0.0036 (0.026)</td>
<td>1.0000 ± 0.0000 (0.0003)</td>
<td>0.9999 ± 0.0001 (0.0009)</td>
</tr>
<tr>
<td>Random Testing</td>
<td>0.5065 ± 0.0251 (0.1815)</td>
<td>0.5542 ± 0.0282 (0.2036)</td>
<td>0.6366 ± 0.0485 (0.3502)</td>
</tr>
<tr>
<td>Random Unit Testing</td>
<td>0.4406 ± 0.0236 (0.1706)</td>
<td>0.4676 ± 0.0244 (0.1759)</td>
<td>0.6121 ± 0.0474 (0.342)</td>
</tr>
</tbody>
</table>

The clean and faulty versions of the expression agree on the expected results of these tests, and thus unit testing is unable to detect that an error has occurred.

Interestingly, vertex testing achieves an extremely high mutation score for the remove relation fault despite the remove relation fault being notoriously difficult to detect. This is because each vertex test is in fact testing the entire application through the use of the inference task, which interacts with all application constraints. Thus, a vertex test will be able to isolate a removed disjoint relation \( R \) in constraint \( c \) if \( R \) is required to execute some other constraint \( c' \) without the other relations in \( c \). For example, the removed relation \( R_2 \) from above can be detected by vertex testing if some other constraint can be executed using \( R_2 \) without \( R_1 \), such as

\[
\{ x | x \in R_2 \land x \in R_3 \land x \in R_4 \}.
\]

To detect a removed relation fault, vertex testing requires just one constraint \( c' \) in which the non-removed relations are not required for execution when the removed relation is required. The results suggest that, statistically, it is highly probably that such a constraint \( c' \) exists for each relation in the domain of every constraint with disjoined terms.

As expected, vertex condition testing outperforms vertex testing. This is because vertex condition testing subsumes vertex testing, and hence contains every vertex test. However, the results show that the improvement is only marginal. Moreover, the improvement is within
Chapter 11. Evaluating Validation Techniques

the standard deviation of vertex testing for neighbour replace (0.0519), absolute (0.0485) and ICE (0.0366) faults, where the differences in the mean mutation scores are 0.0366, 0.0357 and 0.0076, respectively. The difference between the mutation scores for the remove relation fault, 0.1095, is slightly greater than the standard deviation of vertex testing, 0.1212.

The results show that random testing is capable of detecting approximately half of the faults on average. This is particularly interesting because random testing has a key advantage over other testing approaches in terms of design modifiability and extensibility. Unlike other testing approaches, a random testset is generated by referring solely to the set of relations, and is thus independent of the application constraints. If a constraint is modified then the set of random tests is still a valid random testset, while a new set of vertex and unit tests would need to be created. The problem with creating new tests is that an expensive oracle, typically a human with expertise in the application domain, is required to manually process the tests and specify the expected test results. Thus, random testing reduces the number of oracle requests because the developer can use the same random testset to detect faults in an application for which the constraints are being modified. This suggests that random testing may be an ideal testing approach for more agile software development processes or during the prototyping stage of application development.

11.4.4.3 Functional Efficacy of Testing Approaches

This section presents results that provide information on the testing resources required by each testing approach. Table 11.7 presents the lower quartile, median, median absolute deviation, and upper quartile of the average number of tests in a testset used to exercise an application; medians were used to reduce the impact of outliers. The results show that unit testing has the largest median testset size of approximately 1385 which is an order of magnitude greater than vertex testing, approximately 66, and vertex condition testing, approximately 140. Moreover, unit testing has the most variable testset sizes, where the deviation is approximately 86% of the median testset size compared to 41% for vertex testing and 49% for vertex condition testing. Both of the random testing controls were configured to always generate 20 test scenarios.

As defined in Section 11.2.1, the functional efficacy of a testing approach is measured as the average mutation score achieved by a randomly selected test. Table 11.8 presents the functional efficacy results for the five test approaches. Interestingly, both random testing controls significantly outperform the other three validation approaches. This is due to the relatively small number of random scenarios employed per testset; furthermore, as shown in the following section, the random scenarios are significantly more complex than the unit, vertex, or vertex condition scenarios leading to a high functional efficacy. This indicates that a small number of complex random scenarios may be a highly effective utilisation of testing resources. The

14 Although, if only a few constraints are modified then only a few new unit and vertex tests will need to be created and manually processed by the human oracle.
11.4 An Experiment for Evaluating the Presented QSTR Validation Techniques

<table>
<thead>
<tr>
<th></th>
<th>Average Tests per Testset</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lower Quartile</td>
<td>Median</td>
<td>Upper Quartile</td>
<td></td>
</tr>
<tr>
<td><strong>Unit Testing</strong></td>
<td>499.4</td>
<td>1384.8</td>
<td>1191.8</td>
<td>3107.7</td>
</tr>
<tr>
<td><strong>Vertex Testing</strong></td>
<td>32.5</td>
<td>65.5</td>
<td>26.5</td>
<td>85.0</td>
</tr>
<tr>
<td><strong>Vertex Condition Testing</strong></td>
<td>59.9</td>
<td>139.9</td>
<td>69.0</td>
<td>195.4</td>
</tr>
<tr>
<td><strong>Random Testing</strong></td>
<td>20.0</td>
<td>20.0</td>
<td>0.0</td>
<td>20.0</td>
</tr>
<tr>
<td><strong>Random Unit Testing</strong></td>
<td>20.0</td>
<td>20.0</td>
<td>0.0</td>
<td>20.0</td>
</tr>
</tbody>
</table>

Table 11.7: Minimum, mean, standard deviation of, and maximum number of tests in a testset for each of the five test approaches, where one testset is used to test one application.

<table>
<thead>
<tr>
<th></th>
<th>Neighbour Replace:</th>
<th>Absolute:</th>
<th>Instance Creation Expression:</th>
<th>Remove Relation:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean (St'd Dev.)</td>
<td>Mean (St'd Dev.)</td>
<td>Mean (St'd Dev.)</td>
<td>Mean (St'd Dev.)</td>
</tr>
<tr>
<td><strong>Unit Testing</strong></td>
<td>0.0723% (0.0480%)</td>
<td>0.0723% (0.0480%)</td>
<td>0.0723% (0.0480%)</td>
<td>0.0313% (0.0220%)</td>
</tr>
<tr>
<td><strong>Vertex Testing</strong></td>
<td>1.4515% (0.5159%)</td>
<td>1.4743% (0.5183%)</td>
<td>1.5152% (0.5204%)</td>
<td>1.3351% (0.5276%)</td>
</tr>
<tr>
<td><strong>Vertex Condition Testing</strong></td>
<td>0.6992% (0.3013%)</td>
<td>0.7149% (0.3063%)</td>
<td>0.7149% (0.3063%)</td>
<td>0.6986% (0.2985%)</td>
</tr>
<tr>
<td><strong>Random Testing</strong></td>
<td>2.5000% (0.4407%)</td>
<td>2.7251% (0.6446%)</td>
<td>3.4891% (1.5109%)</td>
<td>2.5202% (0.6196%)</td>
</tr>
<tr>
<td><strong>Random Unit Testing</strong></td>
<td>2.1643% (0.3838%)</td>
<td>2.2926% (0.4918%)</td>
<td>2.9474% (1.8012%)</td>
<td>2.2461% (0.4925%)</td>
</tr>
</tbody>
</table>

Table 11.8: Mean and standard deviation of the functional efficacy percentage scores for each test approach with respect to different fault types. The highlighted cells indicate the highest functional efficacy scores.

Results also show that vertex testing makes an extremely good tradeoff between mutation scores achieved and the testing resources required compared to unit and vertex condition testing.

11.4.4.4 Cognitive Efficacy of Testing Approaches

This section investigates the relative effort required by a human oracle to determine the expected test results for each of the five test approaches. Table 11.9 presents the mean and standard deviation of the average number of assertions in a testset used to exercise an application. Unit, vertex, and vertex condition testing all employ extremely small scenarios, consisting of only \( \sim 2 \) objects on average. The random controls were configured to employ more complex scenarios consisting of \( \sim 10 \) objects on average. The two vertex-based test methods also employed an extremely small number of assertions per test on average, \( \sim 2 \); unit testing employed an intermediate number of assertions per test, \( \sim 5 \) and the random controls employed the most assertions per test, \( \sim 11 \).

As presented in Section 11.2.1, at each inference step the human oracle must consider all constraints that could potentially apply when determining the expected result of a given test.
### Table 11.9: Minimum, mean, standard deviation of, and maximum number of assertions in a testset for each of the five test approaches.

<table>
<thead>
<tr>
<th>Test Method</th>
<th>Average Scenario Objects per Test</th>
<th>Average Assertions per Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean (St’d Dev.)</td>
<td>Mean (St’d Dev.)</td>
</tr>
<tr>
<td><strong>Unit Testing</strong></td>
<td>2.3 (0.6)</td>
<td>5.1 (1.7)</td>
</tr>
<tr>
<td><strong>Vertex Testing</strong></td>
<td>2.3 (0.5)</td>
<td>2.3 (0.6)</td>
</tr>
<tr>
<td><strong>Vertex Condition Testing</strong></td>
<td>2.1 (0.4)</td>
<td>1.7 (0.4)</td>
</tr>
<tr>
<td><strong>Random Testing</strong></td>
<td>10.0 (3.6)</td>
<td>11.3 (6.5)</td>
</tr>
<tr>
<td><strong>Random Unit Testing</strong></td>
<td>10.0 (3.6)</td>
<td>11.3 (6.5)</td>
</tr>
</tbody>
</table>

Table 11.10 presents the mean and standard deviation of the number of inference steps per test and the number of initially relevant constraints per test, as defined in Section 11.2.1; it must be noted that the initial relevant constraints are the implemented code features of an application as opposed to high level constraints. The results highlight the power of unit testing approaches where small, cohesive aspects of functionality are exercised in isolation from the majority of the application. Each unit test applies to exactly one specified constraint and thus minimises the cognitive effort required to determine the expected result. The vertex tests (and in general any inference-based testing method) has the property that every test can potentially trigger any constraint in the application. While this provides a highly effective mechanism for detecting faults as presented in the previous two sections, it also places a greater cognitive burden on the human oracle when determining the expected test result. Each vertex and vertex condition test only required a small number of inference steps, ~ 4, although the number of potentially relevant constraints is comparatively high, ~ 11; this constitutes 17.7% of the total number of implemented code features in an application on average. Inference-based random testing invoked a significantly higher number of inference steps, ~ 11, and initial relevant constraints, ~ 38, compared to all other validation methods.

#### 11.4.4.5 Experiment Runtime and Memory Usage

This section presents technical details about the hardware, software, and other resources that were used to run the experiments. The experiments were conducted on a PC with an Intel(R) Core(TM)2 Duo CPU, 2.66GHz processor, with 4GB RAM, running Windows Vista. The random QSTR application generator was built in Java, and Prolog was used to run the test scripts. Table 11.11 presents the mean and standard deviation of the time taken to execute a testset on all mutants of a given application for each validation method. That is, given that the experiment
### Table 11.10: Mean and standard deviation of the number of inference steps required to determine an expected test result, and the mean and standard deviation of the number of constraints with non-empty domains in the initial premise scenario of a test (thus requiring further inspection by a human oracle).

<table>
<thead>
<tr>
<th>Method</th>
<th>Mean (Std Dev.)</th>
<th>Mean (Std Dev.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inference steps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit Testing</td>
<td>1.0 (0.0)</td>
<td>1.0 (0.0)</td>
</tr>
<tr>
<td>Vertex Testing</td>
<td>4.0 (2.0)</td>
<td>11.3 (3.9)</td>
</tr>
<tr>
<td>Vertex Condition Testing</td>
<td>4.6 (1.8)</td>
<td>10.3 (3.1)</td>
</tr>
<tr>
<td>Random Testing</td>
<td>10.8 (7.4)</td>
<td>37.6 (20.4)</td>
</tr>
<tr>
<td>Random Unit Testing</td>
<td>1.0 (0.0)</td>
<td>1.0 (0.0)</td>
</tr>
</tbody>
</table>

### Table 11.11: Mean and standard deviation of the processing time in minutes per application (including all mutants) required by each validation method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Mean (Std Dev.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Testing</td>
<td>10.2 (14.2)</td>
</tr>
<tr>
<td>Vertex Testing</td>
<td>7.6 (9.6)</td>
</tr>
<tr>
<td>Vertex Condition Testing</td>
<td>19.3 (25.5)</td>
</tr>
<tr>
<td>Random Testing</td>
<td>7.0 (7.5)</td>
</tr>
<tr>
<td>Random Unit Testing</td>
<td>6.2 (6.6)</td>
</tr>
</tbody>
</table>

Sample contained 200 applications, unit testing required approximately $10.1524 \times 200 = 2030$ minutes, or 34 hours. Vertex condition testing requires a significantly greater amount of processing time than all other validation methods. It must be noted that each application is processed independently, and thus the total experiment time can be greatly reduced by processing applications in parallel.

### 11.5 Summary

This chapter has established an evaluation methodology that developers, and the QSTR research community in general, can employ to investigate the relative efficacy of alternative QSTR validation methodologies. The evaluation methodology is formalised as a sampling process that selects from a repository of QSTR applications. In lieu of any repository of real QSTR applications from which samples can be selected, a method for randomly generating QSTR applications was presented. Random variables were defined that represent the model of QSTR applications...
developed in Chapters 3 and 5. Test approach efficacy was evaluated in two ways. Firstly test methods were assessed according to the proportion of faults detected per test on average, referred to as functional efficacy. Secondly test methods were assessed according to the predicted cognitive burden required by a human oracle to determine the expected result of a given test.

Experiments were conducted to evaluate the efficacy of the validation methodologies presented in Chapter 9. The test methods selected for the experiments were unit testing, vertex testing and vertex condition testing; random unit testing and random inference-based testing were also employed as controls. The results of the experiments show that vertex testing and vertex condition testing perform extremely well for all fault types, achieving mutation scores above 94%. Moreover, the vertex and vertex condition methods employ a relatively small number of tests, on average using 65 and 140 respectively, thus achieving a reasonable functional efficacy scores of 1.4% and 0.7% respectively. Unit testing was able to detect all neighbour replace, absolute and ICE fault types. However unit testing performed poorly on the remove relation fault, detecting approximately 58%, that is, approximately 10% higher than the random controls. Furthermore, unit testing required testsets that were an order of magnitude larger than all other validation approaches, and thus the overall functional efficacy is low. The random controls detected approximately half of the faults on average, with the exception of ICE faults for which both random controls detected approximately 60%.

The results of the experiments have been used to identify the most effective utilisation of the respective validation methodologies according to the software development process being employed. It is observed that random testsets are more robust to changes in an application design than other validation methods. Specifically, if a constraint is modified then the set of random tests is still a valid random testset, while a new set of vertex and unit tests would need to be created. Based on the reasonable mutation scores achieved by random inference-based testing, random testing is suggested as an ideal testing approach for agile software development processes or during the early, prototyping stage of QSTR application development. If the developer requires a validation methodology that is likely to detect a much greater proportion of defects then vertex testing should be employed. In general, vertex testing is the most effective use of testing resources as it achieves high mutation scores with relatively small testsets; in particular, vertex condition testing achieves only a marginally higher mutation score on average while requiring over twice the number of tests. Finally, the results of the experiments show that unit testing is not feasible for exercising a large number of complex constraints. However, unit testing is the most effective validation approach for thoroughly exercising isolated aspects of functionality, and thus it can be employed to validate small, focused aspects of safety critical components.

The chapter concluded by highlighting a number of pertinent research directions in QSTR validation evaluation methodologies. The most crucial area of future research is in the compilation of a repository of realistic QSTR applications, annotated with actual design faults. This
would greatly facilitate the identification of test coverage classes that have high densities of fault-detecting tests.
Chapter 12

Conclusions

This chapter presents the main conclusions of this thesis. The primary objective of this thesis has been to facilitate the development of applications that utilise qualitative spatial and temporal reasoning (QSTR) calculi. The four main areas of QSTR application development that have been addressed in this thesis are: defining QSTR applications; facilitating QSTR application requirements specification; facilitating QSTR application design; facilitating QSTR application validation. Five QSTR application case studies have been presented that address a range of disciplines including architecture (QtvLight), geographic information systems (TreeSap) and image retrieval; these case studies will be referred to throughout this chapter. The chapter concludes by presenting a number of directions for future research.

Establishing a comprehensive definition of QSTR applications

QSTR applications are very different types of logical systems compared to QSTR calculi, and the motivation for the development of QSTR applications differs significantly to the objectives of developing QSTR calculi. Two definitions of QSTR applications have been established. Firstly, the underlying theoretical foundations of QSTR applications have been presented to provide a formal basis for establishing application development methodologies. Secondly, a higher-level definition has been established to assist developers in the process of organising and designing QSTR application components.

Theoretical foundations for QSTR applications

QSTR applications have been formalised as logical systems that respond to specific patterns in scenarios that are described using pertinent QSTR calculi. Formally, a QSTR application consists of a set of relations and a first-order theory for describing the interaction between relations. This theoretical foundation has been used to formally define the roles of QSTR application developers and users. Developers are responsible for selecting suitable QSTR calculi, selecting a
set of custom relation symbols and encoding a suitable set of constraints. Users specify scenarios in the language of the QSTR application and request that certain tasks be performed on the scenarios by the application.

The theoretical foundation has also been used to identify a fundamental set of basic QSTR application operations that can be performed on scenarios. The basic set of tasks consists of selecting objects from scenarios, refining scenarios by reducing the ambiguity of relations, and modifying the set of scenario objects. These basic tasks are used to enumerate a complete set of purely qualitative application tasks for assisting in the specification of QSTR application functional requirements.

This formal definition has been used in conjunction with an analysis of the case studies to identify four central characteristics of QSTR applications: reasoning across a broad range of abstraction levels (which has been reinforced by a fragment analysis), the continuity assumption and the ubiquity of neighbourhoods for reasoning about changing scenarios, modelling infinite domains, and reasoning about objects in multi-dimensional models of space and time. These characteristics directly motivated the requirements, design, and validation methodologies in this thesis.

**Higher-level organisation of QSTR applications**

In this thesis a number of methodologies have been established that directly support developers in organising and designing QSTR application relations and the associated relation theories. Primarily, the QSTR application analogue of high-level object-oriented classes has been identified as a QSTR fragment. Fragments have been defined as cohesive, modular collections of relations and the corresponding relation theories (or constraints) that specify relation conversion, composition, and conceptual neighbours. QSTR calculi such as Allen’s Interval Algebra or Guesgen’s orientation calculus are fragments.

It has been shown that structuring QSTR applications into fragments greatly facilitates application analysis, thus supporting the high level organisation and development of QSTR applications. Simple fragment metrics such as the number of fragments and the number of relations per fragment have been shown to be effective in characterising the QSTR case study applications, by providing a heuristic measure of overall complexity and by highlighting salient operational and organisational properties. For example, it has been observed that in the TreeSap case study application the more abstract, higher level relations are dynamically defined by users during querying, whereas in QtvLight the more abstract relations of subjective impression are built into the design of the application.

Fragment analysis of the case study applications highlights the very hierarchical organisational structure of QSTR applications; QSTR applications consist of low level fragments that represent and reason about relatively concrete spatial and temporal scenario concepts such as QSTR calculi, and high level fragments that encode more abstract and typically highly
application-specific concepts such as subjective impressions or image categories. This insight directly motivated the development methodologies for supporting the design of low level and high level fragments in this thesis.

Establishing methodologies for specifying formal requirements of QSTR applications

Categories for formal QSTR application requirements have been established in the areas of external interface requirements, functional requirements, and performance requirements. External interface requirements have been derived from an analysis of the case study QSTR applications and a review of the relevant QSTR literature. The external interface requirements specify the ways in which a QSTR application can model a problem domain, such as the types of spatial and temporal entities being modelled, dimensionality, and the relevant properties of space.

Three key methodologies have been established to facilitate the specification of QSTR application functional requirements. Firstly, a novel idealised application methodology was presented, in which a developer specifies how their QSTR application would be required to behave if it were to receive perfect information about the scenario. An idealised application is defined using a sufficiently expressive interpretation language selected by the developer; the QSTR community has identified a number of useful domains of interpretation. In defining an idealised application the developer ignores both the limitations on the availability of scenario information during runtime and the abstractive nature of qualitative systems. The idealised application methodology is employed to assist in the evaluation of alternative QSTR calculi according to application-specific task requirements.

Secondly, in order to support the specification of functional requirements a complete enumeration of basic, purely qualitative tasks has been derived based on the underlying theoretical foundation of QSTR applications established in this thesis. Specifically, the set of tasks is derived from possible sequences of the three fundamental operations: object selection, scenario ambiguity refinement, scenario universe modification. General tasks that apply to any problem domain are query, modify, check consistency and infer. Problem domains that are modelled using multiple scenarios include additional tasks that focus on the relationship between scenarios. Unordered and partially ordered scenario tasks are merge and split. Ordered scenario tasks are check forward and backward consistency, envision, diagnose, complete sequence, ordered merge and ordered split.

Thirdly a template has been established that characterises the general behaviour of QSTR applications based on the previously derived enumeration of purely qualitative tasks. The template can be employed by developers to explicitly incorporating behaviour into the specification of functional requirements.
Finally important concepts have been defined that enable a developer to specify the performance requirements of QSTR applications. Performance requirements are characterised by the complexity of querying and the size and complexity of scenarios.

Establishing methodologies for designing QSTR applications

In this thesis a number of QSTR application design methodologies have been established. Design methodologies have been presented that enhance the accessibility of QSTR for software engineers by establishing a mapping between concepts in the QSTR domain and the object-oriented software domain. Further methodologies support the design of low level fragments by facilitating the evaluation of QSTR calculi with respect to the functional requirements of a QSTR application. High level fragment design is supported through methodologies that enable the development of custom, high level, application-specific relations and constraints.

Mapping between the concepts in object oriented (OO) software and the QSTR application domain

A primary contribution of this thesis has been to establish a mapping between object oriented (OO) domain concepts (such as classes, object attributes, object operations and class relationships) and the central concepts in QSTR applications. This greatly enhances the accessibility of QSTR applications, as it allows the adoption of prominent software development tools from the Unified Modelling Language (UML) for designing, organising, and visualising QSTR applications. The basis of the mapping is the concept of QSTR fragments. As discussed previously, an analysis of the case study QSTR applications has emphasised the hierarchical nature of fragment organisation. Significantly, two important types of constraints that specify the interaction between lower and higher level fragments are mapped directly to two types of OO relationships, namely fragment definitions and fragment generalisations.

Applying certain UML diagrams directly to QSTR applications is not practical, particularly UML sequence diagrams which visualise the flow of information between software components. This is due to the imperative nature of OO software as opposed to the more declarative, rule-based nature of QSTR applications. UML sequence diagrams have been adapted by introducing a simple notational device that allows a developer to specify easily modifiable groups of fragments. The adapted sequence diagrams emphasise the flow of information between groups of fragments and abstract from the ordering of information flow within a group of fragments.

Designing low level fragments: selecting low level QSTR calculi

A low level fragment design methodology has been established that builds on the functional requirements methodology of specifying idealised applications. A key concept called functional
consistency has been defined to evaluate the performance of a QSTR application with respect to the functional requirements specified in the idealised application. The primary contribution in low level fragment design has been the derivation of simple equations that identify the classes of scenarios (as expressions in the interpretation language) for which a QSTR application is functionally inconsistent; a QSTR application that is functionally inconsistent for a given scenario will produce output that differs from the idealised application. It has been proved that functional consistency can be determined by analysing readily available properties of QSTR calculi such as weak composition and the effectiveness of algebraic closure. The developer can use these mathematical tools to analyse the efficacy of myriad QSTR calculi according to their specific task requirements. Moreover, this establishes a basic foundation for further research into methodologies that facilitate the selection of QSTR calculi. An adapted path-consistency algorithm has also been presented for generating a QSTR calculus-based application up to 3-consistency from a fine-grained idealised application.

**Designing high level fragments: data-based metrics and decision tree learning for analysing fragment definitions**

Methodologies for the design of high level fragments have been established that build on the formalisation of fragment definitions. Specifically, two methodologies have been presented that enable the developer to assess the efficacy of fragment definitions with respect to field data and domain-specific contextual information, and to utilise the hierarchical interactions between fragments for deriving high level conceptual neighbourhoods.

Firstly, the software engineering concepts of object-oriented cohesion and coupling have been adapted for QSTR applications. In QSTR applications, coupling is a measure of how effectively a set of low level relations distinguishes between different high level relations; cohesion is a measure of how effectively a set of low level relations defines a high level relation. Coupling and cohesion are calculated by using a dataset of exemplary objects (either in one large scenario or spread across a set of scenarios) that reflect the correct interactions between fragments, such as the detailed results of field studies. A clustering approach is presented based on the information obtained from cohesion and coupling. Clusters can be used to derive high level neighbourhood graphs; this has been demonstrated by deriving a neighbourhood graph for image categories in the Image Retrieval QSTR application. A variation of cohesion and coupling has been presented that measures the accuracy and precision of the fragment definitions as the average discrepancy and variance between inferred relations and expected relations.

Secondly, decision tree learning has been employed for generating fragment definitions from datasets and deriving conceptual neighbourhoods based on the generated definitions. Decision trees succinctly express fragment definitions; each vertex in the tree is annotated with a reference domain relation and a state (i.e. holds, not holds, and not applicable) and each leaf is annotated with an abstraction domain relation and a state. A path from the root to a leaf
represents a fragment definition, where tuples in the relation state at each vertex in the path are conjoined to be an improper subset of the tuples in the relation state at the leaf. It has been demonstrated using the Image Retrieval application that a developer can employ decision tree learning to generate fragment definitions from field data, and then compare the resulting fragment definitions with their manually formalised fragment definitions.

**Designing high level fragments: high level neighbourhoods**

A high level fragment design methodology has been presented that enables developers to define and utilise high level neighbourhoods that are consistent with the interactions between fragments. The standard definition of conceptual neighbours was shown to be ineffectively weak when employed in the wider context of QSTR applications; specifically, when using the standard conceptual neighbour definition two relations possibly being neighbours results in relations almost always being neighbours. The established methodology extensively generalises the definition of conceptual neighbourhoods to enable the design of high level neighbourhoods that are consistent with the structure of fragments in a QSTR application. In order to develop more accurate neighbour definitions two conditions are imposed. The first condition restricts the set of paths considered for determining neighbour status, and the second condition groups paths together into equivalence classes.

**Implementation**

Strategies have been presented for implementing QSTR application designs in software. The approaches to implementation focus on the use of declarative languages and relational database systems for encoding relations and constraints, and implementing general reasoning procedures. Basic software architectures have been presented that integrate the implementation of high level custom fragments with specialised QSTR reasoners that provide implementations of standard QSTR calculi. Specifically, QSTR calculi are combined with custom high level fragments using loose integration. A loose integration of QSTR calculi and custom fragments enables modular software development, and thus provides the most efficient approach with respect to the development resources required for application design and validation.

A strategy for implementing constraints has been presented based on a general set theoretic constraint form. The general form of constraints divides conditions into three categories based on whether they admit or reject tuples from a set; the conditions are referred to as triggers, overrides and independent. Although any logical form is satisfactory for implementing QSTR applications, this construct has a number of desirable design properties; the construct is modular, orthogonal, and syntactically similar to the portion of Structured Query Language (SQL) that is equivalent to first-order logic. Basic principles for implementing deductive closure and
scenario consistency checking algorithms have been presented, based on the observation that all QSTR tasks can be implemented as a sequence of queries and updates.

Finally, principles for lowering the resources required to represent a given qualitative scenario have been presented. A central principle is that the developer can minimise the amount of redundant information represented in the model by employing intermediate fragments. A novel complexity metric called H-complexity is used to inform a developer about the influence that design decisions have on the complexity and performance requirements of QSTR application components.

**Establishing methodologies for validating QSTR applications**

In this thesis a QSTR validation process methodology has been established. The validation process consists of creating a test suite containing black-box and white-box tests, using the test suite to refine the application design, and then quantifying the efficacy of the test suite to determine when to terminate the testing process. An evaluation methodology has also been established to assess the fault-detecting performance of various QSTR application validation approaches.

**Adapting software engineering validation methodologies**

A number of validation methodologies have been adapted from well known techniques in software engineering. In particular, two well known white-box testing methodologies have been adapted from software engineering, namely unit testing and integration testing, that support a developer in creating an appropriate test suite. The most crucial components of the QSTR unit testing methodology are two scenario equivalence classes defined based on a decision table representation of scenarios. QSTR integration tests exercise the interaction between pairs of expressions.

Furthermore, two methodologies for analysing the efficacy of a test suite at detecting faults have been adapted from software engineering. Six test coverage metrics have been defined using an analogue to the control flow graph called the sentence interaction graph (SIG). Analogous to statement and branch execution in imperative software, the key concepts of expression execution (vertex) and sequences of expression executions (edge and k-path) have been defined as a basis for the coverage metrics. Mutation testing has also been adapted for QSTR applications. The key component of QSTR mutation testing is a set of six mutation operators based on analogous mutation operators applied to imperative and OO software. Importantly, the distinction between neighbouring relations is exercised by replacing relations with their neighbours in expressions, and by manipulating the edges in neighbourhood graphs; hierarchical distinctions are excised by replacing relations with their specialisations and generalisations.
**H-complexity test coverage metrics**

In contrast to the validation techniques adapted from software engineering a novel metric called H-complexity has been presented which is derived specifically from the relational theoretic model that underlies QSTR applications. H-complexity is derived from the novel concept of homogeneous (H) sets, defined as atomic definable sets; no query exists that can separate two objects within the same H set, thus objects within an H set are equivalent and indistinguishable.

H-complexity is employed to develop two main QSTR application validation tools. Firstly, the formulation of H-complexity provides the developer with key insights about the interaction between salient expression properties and the complexity of the language required to encode the expression. A developer can use these basic properties of complexity to manage development resources and complexity when modifying their design.

Secondly, H-complexity has been used to define four test coverage classes that a developer can employ to assess the efficacy of a test suite. The H-complexity test coverage classes are: tuple state coverage, basic query coverage, H set coverage, and scenario coverage. A crucial component of the H-complexity coverage methodology is a set of guidelines that allow a developer to define a focused, refined test space. A comparative analysis of H-complexity test coverage classes and the test coverage classes adapted from software engineering has shown that the two coverage approaches complement each other; the adapted coverage metrics provide a focused test set derived from the content of the design (specifically, the conditions required to execute constraints) and the H-complexity metrics provide a broader test set derived from the language.

**Meta-validation methodology**

A QSTR evaluation validation methodology has been established that enables a developer to conduct experiments that empirically investigate whether a given set of QSTR validation techniques are effective at detecting faults in QSTR application designs. The evaluation methodology is formalised as a sampling process that selects from a repository of QSTR applications. A sample of QSTR applications are seeded with faults using the mutation testing methodology. Test approach efficacy was evaluated in two ways. Firstly test methods were assessed according to the proportion of faults detected per test on average, referred to as functional efficacy. Secondly test methods were assessed according to the predicted cognitive burden required by a human oracle to determine the expected result of a given test.

To ensure that the developer can access a statistically significant sample size for the experiments, an approach for randomly generating QSTR applications has been presented. Random variables have been defined that represent the model of QSTR applications developed in this thesis.
Experiments were conducted to evaluate the efficacy of the validation methodologies. The results of the experiments have been used to identify the most effective utilisation of the respective validation methodologies according to the software development process being employed. Random testing is suggested as an ideal testing approach for agile software development processes or during the early, prototyping stage of QSTR application development. Vertex testing is an ideal general validation approach as it achieves high mutation scores with relatively small testsets; in particular, vertex condition testing achieves only a marginally higher mutation score on average while requiring over twice the number of tests. Unit testing can be employed to validate small, focused aspects of safety critical components.

12.1 Future Research

The most promising area of future research is in establishing an integrated QSTR application development environment analogous to the Garp3 workbench, as presented in Section 1.3. Research is currently being conducted by Bhatt, Dylla, and Hois on developing integrated frameworks for software tools that assist in the design of ambient environments. This is an extremely significant contribution in facilitating the development of QSTR applications. In conjunction to this, the development methodologies presented in this thesis establish a foundation for future research towards a more general framework for facilitating the development of QSTR applications that address a wide range of disciplines.

12.2 QSTR Application Development using Functional Consistency Analysis and Future Research

This section presents the areas of the functional consistency methodology that can benefit from further research. One important area of future research is enabling developers to specify idealised applications in a natural way (e.g., using graphical languages). Human-computer interaction research is also required to determine intuitively meaningful ways of reporting the functional consistency results to developers.

Further research is required to determine whether the conjectured intractability of generating the strongest QSTR-based application from an idealised application is of practical concern to developers. The analysis of QSTR applications in the case studies (refer to Section 2.8) suggests that this will not be an issue; in all QSTR applications that have been studied the theories that are used to express patterns require very few variables, i.e., in the order of (at most) tens of variables as opposed to hundreds or thousands.

Further research is also required for analysing applications that deliberately respond to inconsistent scenarios, for example, to execute a sensor calibration routine. The problem is that
if applications must respond to inconsistent scenarios then in general the application loses completeness. QSTR applications are complete in the sense that if an application does not produce output $\alpha$ for a scenario then the user can be confident that the application is correct in not producing $\alpha$.\footnote{In contrast, QSTR applications are not sound; they may erroneously produce $\alpha$ in scenarios for which the idealised application does not produce $\alpha$. It is interesting to note the duality between QSTR applications being complete but not sound, and QSTR calculi being sound but not complete (refer to Section 2.2).} If the application is supposed to produce $\alpha$ according to $\Theta^A_\alpha$ for inconsistent scenarios, then the user cannot guarantee that the application is correct for not producing $\alpha$ in some scenario, because in general it cannot be guaranteed that the scenario is actually consistent, as presented in Section 2.6.

12.3 Discussion of Limitations and Future QSTR Meta-Validation Research

This section highlights important directions for further evaluation methodology research that could enhance the ability of QSTR application developers to analyse QSTR validation methodologies. The key areas of further development are the QSTR application model, the fault seeding methodology, measuring cognitive efficacy, managing the resources required to conduct mutation testing, selecting the appropriate sampling strategy, and compiling a gold standard QSTR application repository.

The first issue is the validity of the QSTR application model. The effect of any QSTR application model is to distinguish between different types of constraints, thus allowing the developer to specify distributions over particular classes of constraints within the constraint space. The problem is determining whether the model reflects realistic QSTR applications, which requires real QSTR applications for comparison. A broad collection of real QSTR applications is required to identify the concepts that need to be modelled (for example, to determine whether defining a probability distribution over fragment definitions is useful) and to develop accurate probability distributions for the parameters in the model. Another modelling issue is determining conditional probability distributions. For example, one important pattern identified from the case studies (refer to Section 5.2, Table 5.3) is that fragments that are high up in the hierarchy of definitions have fewer relations on average than fragments lower down in the definition hierarchy.

The second issue is the applicability of the fault seeding methodology to QSTR application. Relatively few studies have been conducted to determine the most effective methodology for fault seeding imperative software [25].\footnote{Andrews et al. [25] have discussed the difficulties of evaluating fault seeding approaches.} Even fewer studies have been conducted for fault seeding knowledge-based systems [131, 132], which are closely aligned with QSTR applications. Experiments must be conducted that investigate whether mutation testing produces realistic

\[\text{output } \alpha \text{ for a scenario then the user can be confident that the application is correct in not producing } \alpha.\]
faulty QSTR applications. However, as previously discussed in Section 9.6.2 there is no data on the types of real faults that occur in QSTR applications, and no information to determine whether the coupling effect hypothesis between complex faults and simple faults holds or does not hold for QSTR applications.

Thirdly, further research is required to determine whether the factors of cognitive efficacy that have been presented in this chapter are realistic. Studies of domain experts must be conducted to determine the most crucial indicators of cognitive burden in QSTR testing approaches, similar to the framework established in [44].

An additional issue is that mutation testing can produce a large number of variant programs, and processing the complete set of mutants may require more testing resources than the developer has available. Based on the results of the experiments, this does not appear to be a significant issue for QSTR applications that have up to approximately 100 relations and 40 constraints, as the number of mutants generated was practical for conducting the experiments. Despite this, managing the number of generated mutants is likely to become a problem for larger applications that contain hundreds of constraints. One technique for making mutation testing more practical is to take a sample of all possible mutants, referred to as weak mutation testing [90, 130]. Research is required to investigate the effect on the average mutation scores of testing approaches when using varying degrees of weak mutation testing.

Further research is required to explore alternative sampling strategies. The sampling approach adopted for these experiments was simple random sampling with replacement. It is feasible that more relevant information about the efficacy of different testing approaches could be obtained by identifying and targeting particular classes of QSTR applications. For instance, stratified random sampling is a sampling approach that partitions the population into distinct groups. One example is to stratify QSTR applications according to the application domain, such as robotics, GIS, construction IT, and so on. Experiments could then address more specific research questions about whether a particular testing approach is more effective in particular application domains. The external interface requirements defined in 4.2 provide criteria for potentially important strata, such as the spatial granularity (e.g. desktop, indoor, or outdoor), the interaction between relations (e.g. simple relations with superficial interactions compared to complex relations that are deeply interdependent), and so on. Another alternative sampling approach is cluster sampling, where the sampled units are grouped into larger aggregate units. This approach is required when it is not feasible to directly sample QSTR applications. For example, in the interest of obtaining more uniformly implemented applications, it may be more practical to select applications based on the research groups responsible for their development; in this case the applications are clustered based on research groups, and the research groups are the sampled units.

Finally, a uniform repository of real QSTR applications would greatly advance the potential for research in QSTR validation methodologies. A central objective of QSTR validation
research is to identify test coverage classes that have a high density of fault detecting tests. The issue is that test coverage subsumption does not imply superior classes of fault detecting tests, and thus comparative test class analysis in isolation cannot be used to evaluate the efficacy of a given validation approach. A gold standard repository that reflected realistic distributions of the characteristics of QSTR applications, particularly the types and frequencies of design faults that occur in practice, would permit thorough empirical and analytical research into the efficacy of competing validation methodologies.

An integrated QSTR application development environment would facilitate the development of uniform QSTR applications by researchers in a range of disciplines, thus providing a basis for compiling a repository of real QSTR applications. This would be the QSTR application analogue of the recent initiative in the QSTR community of compiling a repository of benchmark problems for QSTR calculi [124].


REFERENCES


REFERENCES


Appendix A

This appendix contains Prolog scripts used to perform the evaluation methodology experiments presented in Chapter 11.

A.1 Unit Tests

The following is a unit test as defined in the experiment procedure, Section 11.4.1.¹

\[
\begin{align*}
\text{testname}_1 & : - \text{write}('UT\_0\_disjoint\_1'). \\
test\_1(\text{Result}) & : - \text{setof}((S,A_1,B_1), (\text{rel}_1(S,A_1,B_1); \text{rel}_0(S,A_1,B_1)), \text{Result}); ~\text{Result}=[]. \\
\text{expected}\_1(\text{Result}) & : - \text{setof}((S,A_1,B_1), (\text{rel}_2(S,A_1,B_1); \text{rel}_0(S,A_1,B_1)), \text{Result}); ~\text{Result}=[]. \\
\text{scenario}_0 & : - \text{reset}\_\text{scenario}, \text{assert} (\text{rel}_1(s,a_1,b_1)), \text{assert} (\text{rel}_0(s,a_1,b_1)). \\
\text{scenario}_1 & : - \text{reset}\_\text{scenario}, \text{assert} (\text{rel}_1(s,a_1,b_1)), \text{assert} (\text{no}\_\text{rel}_0(s,a_1,b_1)). \\
\text{scenario}_2 & : - \text{reset}\_\text{scenario}, \text{assert} (\text{no}\_\text{rel}_1(s,a_1,b_1)), \text{assert} (\text{rel}_0(s,a_1,b_1)). \\
\text{scenario}_3 & : - \text{reset}\_\text{scenario}, \text{assert} (\text{no}\_\text{rel}_1(s,a_1,b_1)), \text{assert} (\text{no}\_\text{rel}_0(s,a_1,b_1)). \\
\text{passes}\_\text{test}_1 & : - \text{test}\_1(\text{R}), \text{expected}\_1(\text{E}), ~! , \text{R}=\text{E}. \\
\text{passes}\_\text{all}\_\text{tests}_1 & :- \text{scenario}_0, ~! , \text{passes}\_\text{test}_1, \text{scenario}_1, ~! , \text{passes}\_\text{test}_1, \text{scenario}_2, ~! , \text{passes}\_\text{test}_1, \text{scenario}_3, ~! , \text{passes}\_\text{test}_1.
\end{align*}
\]

A.2 Vertex Tests

The following is a vertex execution test as defined in the experiment procedure, Section 11.4.1.

\[
\begin{align*}
\text{testname}_1 & : - \text{write}('VT\_1'). \\
\text{passes}\_\text{all}\_\text{tests}_1 & :- \text{scenario}_0, ~!, \text{passes}\_\text{test}. \\
\text{testname}_2 & : - \text{write}('VT\_2\!'). \\
\text{passes}\_\text{all}\_\text{tests}_2 & :- \text{scenario}_1, ~!, \text{passes}\_\text{test}. \\
\% 'f' is the faulty application, 'c' is the clean application \\
\text{scenario}_0 & : - \text{reset}\_\text{scenario}, \text{assert} (f:rel_7(a_1,b_1)), \text{assert} (c:rel_7(a_1,b_1)). \\
\text{scenario}_1 & : - \text{reset}\_\text{scenario}, \text{assert} (f:rel_5(a_1,b_1)), \text{assert} (c:rel_5(a_1,b_1)). \\
\text{passes}\_\text{test} & : - f:\text{infer}, c:\text{infer},
\end{align*}
\]

¹In Prolog notation, capitalised words such as A1 are variables, the symbol % is used for comment lines, the symbol :- is implication, i.e. \( p_1 :- p_2 \) means \( p_1 \leftarrow p_2 \), and the connectives , and ; are conjunction and disjunction, respectively.
Chapter A

A.3 Test Script

The following is a Prolog testing script for unit testing as defined in the experiment procedure, Section 11.4.1.

```prolog
:- save_program('./initState.po').
:- open('C:/Experiments/Tests/UnitTesting/TestResults_App_0.txt',write, Stream),set_output(Stream).
:- restore('./initState.po'), consult('C:/Experiments/Tests/UnitTesting/App_0/Fault_0/UT.pl').
:- write('
Fault_0, '), testname_1, write(':'), ((passes_all_tests_1, write('MISSED'));write('DETECTED')).
:- restore('./initState.po'), consult('C:/Experiments/Tests/UnitTesting/App_0/Fault_1/UT.pl').
:- write('
Fault_1, '), testname_1, write(':'), ((passes_all_tests_1, write('MISSED'));write('DETECTED')).
:- current_output(S), flush_output(S),close(S).
:- halt.
```

Running the test script produces a log of test results as follows.

Fault_0, UT_0_disjoint_1:DETECTED
Fault_0, UT_1_disjoint_2:MISSED
Fault_0, UT_2_disjoint_3:MISSED
Fault_0, UT_3_disjoint_4:MISSED
Fault_1, UT_0_disjoint_1:MISSED
Fault_1, UT_1_disjoint_2:MISSED
Fault_1, UT_2_disjoint_3:MISSED
Fault_1, UT_3_disjoint_4:MISSED