

Towards User-centric Concrete Model Transformation

By

Iman Avazpour

A thesis presented to
the Faculty of Information and Communication Technologies,
Swinburne University of Technology, Melbourne,
in fulfilment of the thesis requirement for the degree of
Doctor of Philosophy

2013

Abstract

Model transformations are an important part of Model Driven Engineering (MDE). To generate a transformation with most current MDE approaches, users are required to specify (or provide) complex abstractions and meta-models and engage in quite low-level coding in usually textual transformation scripting languages. These abstractions are very different from concrete visual representation of source and target models and low level coding is hard to specify and maintain, especially for novice users. This specification technique provides pragmatic barriers for users of model transformations and prevents them from adapting MDE technologies.

This thesis introduces an approach for performing model transformation on concrete visualisations of example model elements. It allows end users to interactively specify rich, human-centric visualisations of complex data using a visual, drag-and-drop, by-example approach. End users can generate reusable visualisation implementations from these high-level specifications. Using these visualisations, users specify complex model element mappings between concrete visual notational elements using interactive drag-and-drop and reusable, spread sheet-like mapping formulae. Complex, scalable, efficient, accurate and reusable model transformation implementations are then generated from these by-example visual source-to-target mappings while high-level abstractions for transformation generation are automatically reverse engineered from visualisation examples. As a result, this approach helps to better incorporate a user's domain knowledge by providing familiar example concrete visualisations of models for transformation generation.

In addition, to better aid users to find correspondences in large model visualisations, an automatic recommender system is introduced. This provides suggestions for possible correspondences between source and target model elements using model characteristics and visual representations to generate guidance for large model mapping problems. These recommendations allow users to cut corners in specification of transformation correspondences by choosing among suggestions.

A proof of concept implementation of this approach, CONcrete Visual assistEd Transformation framework (CONVErT) is introduced which allows generation, design and use of varieties of notations including text, boxes and lines, shapes, etc. It integrates the use and definition of mapping functions and conditions and enables reverse engineering of metamodels.

The approach presented by this thesis was evaluated using set of visualisation and transformation case studies, a comparative analysis, a quantitative study and a user study. Case studies are chosen from a variety of domains to show the generality of the approach. The comparison study compares the approach and its tool support against a set of current research and industry approaches and toolsets. The quantitative study is devised to assess quality of the automatically generated transformation code against a

human expert's code and code automatically generated by an industry standard toolset. Finally, a user study complements the other evaluations and provides a report on typical end users experience with the approach and its prototype tool support.

Acknowledgement

I would like to thank:

- My supervisors (John Grundy and Lars Grunske) for their encouragement and guidance.
- My wonderful wife (Farnoosh Sadeghian) and my parents (Razie Mosaddegh and Alireza Avazpour) for their support, patience and motivations.
- My friends and colleagues for fruitful discussions.
- Participants of the user study.
- And anyone having a role during this past four years that I may not remember now.

Declaration

This is to certify that,

- This thesis contains no material which has been accepted for the award of any other degree or diploma, except where due reference is made in the text of the examinable outcome; and
- To the best of my knowledge contains no material previously published or written by another person except where due reference is made in the text of the thesis; and
- Where the work is based on joint research or publications, discloses the relative contributions of the respective workers or authors.

Iman Avazpour

November 2013

Melbourne, Australia.

Table of contents

1. Introduction	1
1.1. Background	1
1.2. Model visualisation	2
1.3. Model transformation	3
1.4. Thesis objective	6
1.5. Contributions	10
1.6. Thesis organisation	12
1.7. Summary	13
2. Literature review	14
2.1. Introduction	14
2.2. Modelling and MDE	14
2.2.1. Concrete versus abstract models	16
2.2.2. Model transformation and MDE	19
2.3. Model transformation	20
2.4. Transformation using concrete models	26
2.4.1. Model Transformation By-Example (MTBE)	26
2.4.2. Model Transformation By-Demonstration (MTBD)	29
2.5. Visualisations and transformation	30
2.5.1. Visual intractable schemas	31
2.5.2. Using concrete visualisations	32
2.5.3. Using concrete syntax in conjunction with abstract	34
2.6. Model transformation tools	34
2.7. Information and software visualisation	38
2.7.1. Model visualisation	38
2.7.2. Information Visualisation	40
2.8. User guidance	42
2.8.1. Metamodel matching	43
2.8.1.1. Instance-based matching approaches	43
2.8.1.2. Schema-based matching approaches	44
2.8.1.3. Hybrid approaches	46
2.8.2. Recommender systems	46
2.8.3. User guidance in transformation	48
2.9. Summary	49
3. Approach	51
3.1. Introduction	51
3.2. Approach	53
3.2.1. Visualisation	54
3.2.1.1. Visual notations	56
3.2.1.2. Mapping input data to visual notations	57
3.2.1.3. Visualisation composition	58
3.2.1.4. Visual aid for notation composition	59
3.2.2. Correspondence specification	60
3.2.2.1. Transformation rule representation	63
3.2.2.3. Correspondence Recommender	64
3.3. Scope	66

3.4. Evaluation	67
3.5. Summary	68
4. Visualisation	69
4.1. Introduction	69
4.2. Visualisation procedure	70
4.2.1. Visual notations	71
4.2.1.1. Interaction logic	74
4.2.2. Mapping input data to visual notations	76
4.2.2.1. Transformation functions	77
4.2.2.2. Transformation conditions	84
4.2.3. Notation composition	87
4.2.3.1. Visual aid for debugging transformation composition	89
4.3. Case studies	90
4.3.1. Bar chart visualisation	90
4.3.2. Minard’s map visualisation	99
4.3.3. UML class diagram visualisation	106
4.3.4. Java code visualisation	116
4.3.5. Computer Aided Design (CAD) visualisation	124
4.4. Summary	130
5. Transformation using concrete visualisations	131
5.1. Introduction	131
5.2. Transformation approach	132
5.2.1. Transformation rule specification	132
5.2.2. Transformation rule representation	138
5.2.3. Generating transformation scripts	139
5.3. Case studies	144
5.3.1. Mapping bar chart to pie chart	144
5.3.2. Mapping Minard’s map to pie chart	149
5.3.3. Mapping UML class diagram to Java	153
5.3.4. Mapping CAD designs to alternative tree visualisation	159
5.4. Summary	164
6. Correspondence recommender	165
6.1. Introduction	165
6.2. Correspondence recommender (“Suggerer”)	166
6.2.1. Calculating recommendations	168
6.2.1.1. Static similarity recommenders	170
6.2.1.2. Structural similarity recommenders	174
6.2.1.3. Propagated similarity recommender	176
6.2.2. Suggerer system ensemble	179
6.2.3. Recommendation representation	184
6.3. Summary	187
7. Tool support: <u>Concrete visual assisted transformation (CONVErT)</u>	189
7.1. Introduction	189
7.2. Overview of CONVErT	189
7.3. CONVErT’s User Interface	192
7.3.1. Visualiser	193

7.3.2. Mapper	197
7.3.3. Notation designer (Skin++)	201
7.4. Implementation	204
7.4.1. User interface design	205
7.4.2. Visual rendering	206
7.4.3. Abstraction	209
7.4.4. Suggester system	210
7.4.5. Transformation	212
7.5. Usage scenarios	215
7.6. Summary	220
8. Evaluation	221
8.1. Introduction	221
8.2. Comparative study	222
8.3. Quantitative evaluation	232
8.3.1. Suggester evaluation	233
8.3.2. Transformation code quality	239
8.4. User evaluation	246
8.4.1. Experimental setup	246
8.4.2. Experiment results	248
8.4.2.1. Visualisation	249
8.4.2.2. Transformation	254
8.4.2.3. Suggester	259
8.4.3. Threats validity	263
8.4.3.1. Threats to internal validity	263
8.4.3.2. Threats to external validity	263
8.4.3.3. Threats to construct validity	263
8.4.3.4. Threats to statistical validity	264
8.5. Summary	264
9. Conclusion and future work	266
9.1. Conclusion	266
9.2. Future work	268
9.2.1. Transformation generation and tool support	268
9.2.2. Transformation recommender system	269
9.2.3. Dynamic and enhanced visualisation	269
9.2.4. Other domains	270
References / Bibliography	272
Appendix 1 ATL transformation script example	287
Appendix 2 Ethics approval clearance	291
Appendix 3 User study tasks for first group	293
Appendix 4 User study tasks for second group	295
Appendix 5 Survey questionnaire	297
Appendix 6 Samples of citation formats used for evaluating Suggester	307
List of publications	310

List of figures

Number	Description	Page
1.1	Examples of (A) UML class diagram meta-model, and (B) Class diagram using concrete notation.	4
1.2	Example of (A) Java meta-model, and (B) Java source code.	7
1.3	Examples of correspondences between UML class diagram and Java code. Dashed arrows demonstrate more fine grained correspondences.	7
1.4	Sample correspondence relations between UML class diagram XML example and Java code XML example.	9
2.1	Example C++ code and its corresponding AST.	16
2.2	UML class diagram example.	17
2.3	Simplified UML metamodel (from [1]).	18
2.4	Transforming PIM to PSM and revers using refinement and abstraction transformations.	19
2.5	Sample of ATL transformation rule for transforming UML operations to Java methods (from [1]).	21
2.6	Example XSLT script and its source and target XML.	24
2.7	Sample mapping generation using ALTOVA MapForce.	31
2.8	Sample mapping generation using Form-based mapper.	33
3.1	High level description of our approach and transformation flow.	54
3.2	Using notation repository for generating visualisations.	55
3.3	High level structure of visual notations (left) and a concrete example describing a chart notation (right).	56
3.4	Mapping input data to visual notation's model data.	57
3.5	Composition of a bar chart visualisation.	59
3.6	Visual aid for notation composition.	60
3.7	Correspondence specification examples. Arrows depict drag and drop directions.	61
3.8	Architecture of a visual notation.	62
3.9	Examples of transformation rule representations. A) UML class diagram to Java class notation transformation rule. B) Bar to pie piece transformation rule.	64
3.10	A sample of recommendation list recommending correspondences between a bar chart and a pie chart.	66
4.1	Visualisation procedure.	70

Number	Description	Page
4.2	Model View Controller (MVC) set up from [2].	71
4.3	Adaptation of Model View Controller for visual notation design.	72
4.4	Sample bar chart visualisation.	72
4.5	a) Model, b) View, c) Annotated View, d) Controller of a bar chart visual notation and e) Final notation.	74
4.6	a) View, b) Model, c) Controller, and d) final bar notation.	74
4.7	Notation's MVC wrapped in interaction logic.	75
4.8	Brief architecture of system implementation. Notations will be saved in the notation repository.	75
4.9	Mapping sales records input to notations: a) Mapping a sales record to a bar, b) Mapping spread sheet to chart. Arrows depict drag and drop directions	77
4.10	Transformation function's structure.	78
4.11	A) A summation function, B) Its template. Arrow marks reverse operation. Information of input argument is lost during forward summation operation.	79
4.12	Mapping sales records to bar using summation function. Arrows depict drag and drop directions.	80
4.13	The generated transformation script resulted from the function of Figure 4.12.	81
4.14	Reverse transformation script resulted from the function of Figure 4.12.	81
4.15	Example of defining a new function using function template.	83
4.16	Transformation condition's structure.	84
4.17	A) A transformation condition, B) Its code template.	85
4.18	Using a condition for specification of bar's colour.	85
4.19	Transformation code script resulted from condition of figure 4.18.	86
4.20	Composition of visual notations to create bar chart visualisation.	87
4.21	Visualisation of a bar chart. User has right clicked on the bar chart and the internal elements of the bar chart notation are represented in a pop-up window	88
4.22	Visualisation composition debugging aid. The product of composing a bar in the bar chart is shown in a pop-up.	89
4.23	Visualisation composition debugging aid for function of a class diagram visualisation.	90

Number	Description	Page
4.24	Example of sales records XML input.	91
4.25	Bar chart notation, a) View code, b) Model XML, c) Annotated View, d) Controller transformation and e) Final notation.	92
4.26	Bar notation, a) View code, b) Model XML, c) Annotated View, d) Controller transformation and e) Final notation.	93
4.27	a) mapping sales records to bar using summation function and b) mapping spreadsheet to bar chart notation. Arrows depict drag and drop directions.	94
4.28	Transformation code resulting from mapping correspondences and summation function of Figure 4.27.	94
4.29	Alternative model for bar's notation.	95
4.30	New bar's view annotation.	95
4.31	Mapping sales records to new bar.	96
4.32	Specifying arguments of condition.	96
4.33	Mapping colour values to condition, and mapping condition result to Color element of bar notation.	97
4.34	Composition of notations using the new bar's notation.	98
4.35	Resulted coloured bar chart.	98
4.36	Minard's map (from [3]).	99
4.37	Minard's map recreated by Humphrey [4]	100
4.38	Troops movement notation's (a) View and (b) Model.	101
4.39	Annotated View of Troops movement notation with Model elements.	101
4.40	Map notation's (a) View, (b) Model.	102
4.41	Map notation's annotated View XAML.	102
4.42	Specifying correspondences between troop movement records and provided troop movement notation. Arrows indicate drag and drop directions.	103
4.43	Specifying correspondences between troop movement records and provided troop movement notation using functions (a) correspondences with campaign data input file and (b) using separate input file to specify colours. Arrows indicate drag and drop directions.	104
4.44	Specifying correspondences between input data and map notation.	105
4.45	Composing troop movement and map notations to generate complete visualisation. Arrows are provided by framework.	105
4.46	Minard's map resulting from our approach.	106

Number	Description	Page
4.47	An example of UML class diagram inputs.	107
4.48	Desired visualisation for example of figure 4.47.	108
4.49	UML attribute notation's (a) View and (b) Model.	108
4.50	Annotated view of UML attribute notation.	109
4.51	UML function parameter (a) View and (b) Model.	109
4.52	Annotated view of UML function parameter.	109
4.53	UML function notation's (a) View and (b) Model.	110
4.54	Annotated view of UML function notation.	110
4.55	UML association notation's (a) View and (b) Model.	110
4.56	Annotated view of UML association notation.	111
4.57	UML class notation's (a) View and (b) Model.	111
4.58	Annotated view of UML class notation.	112
4.59	UML class diagram notation's (a) View and (b) Model.	112
4.60	Annotated view of UML class diagram.	112
4.61	Specifying correspondences between class element and class notation. Arrows depict drag and drop direction.	113
4.62	Specifying correspondences between function parameters and parameter notation. Arrows depict drag and drop direction.	114
4.63	Composition of notations to generate Class diagram visualisation.	115
4.64	Example of class diagram visualisation of XYZ airline.	115
4.65	An example of Java code input represented by XML.	117
4.66	Java property's (a) View and (b) Model.	117
4.67	Java property View's annotations.	118
4.68	Java class's (a) View and (b) Model.	118
4.69	Java class view annotations.	119
4.70	Mapping Java class input elements to Java notation.	120
4.71	Mapping parameter element to Java parameter notation. Arrows depict drag and drop.	120
4.72	Using string merge function to alter Java class's name. Arrows depict drag and drop.	121
4.73	Composing notations to generate Java code visualisation. Arrows are provided by framework.	121
4.74	Resulted visualisation of the example in Figure 4.65.	122
4.75	Example Java code visualisation of airline application.	123

Number	Description	Page
4.76	Example input model of a CAD XML.	124
4.77	CAD room's (a) View and (b) Model.	125
4.78	Annotated view of CAD room notation.	125
4.79	CAD floor plan's (a) View and (b) Model.	126
4.80	Annotated view of CAD floor plan notation.	126
4.81	CAD design notation's (a) View and (b) Model.	127
4.82	Annotated view of CAD design notation.	127
4.83	Correspondence specification between shape element and room notation.	128
4.84	Correspondence specification between plan data and floor notation.	128
4.85	Composition of notations for CAD visualisation.	129
4.86	Example of the generated CAD visualisation.	130
5.1	Transformation generation procedure.	132
5.2	One-to-One correspondences between elements of a bar in bar char and elements of pie pieces in a pie chart. Arrows depict correspondences.	133
5.3	Correspondences between elements of a troop movement notation in Minard's map and elements of a pie piece in pie chart. Solid arrows depict indirect correspondences while dashed arrow depicts direct correspondence.	134
5.4	Correspondences between a bar in bar char and a pie piece in a pie chart. Solid arrow depicts parent correspondence while dashed arrows depict child correspondences.	135
5.5	Correspondences between a UML attribute and a Java property. Solid arrow depicts parent correspondence while dashed arrows depict child correspondences.	137
5.6	Using conditions to specify correspondences. A) Before values are specified to the condition arguments, B) after values are provided. Arrows show drag and drop directions.	138
5.7	Examples of transformation rule representation. Transformation rules are: A) UML class to Java class, B) A Room in 2D visualisation to a room notation in another 2D visualisation, C) A pie piece to bar and D) UML attributes to Java property.	139
5.8	Steps for generating transformation rule between UML class attribute and Java property.	140
5.9	Pseudo code representing the transformation template of step one in Figure 5.8.	140

Number	Description	Page
5.10	Pseudo code representing the transformation template after step two in Figure 5.8.	141
5.11	Transformation rule script for transforming UML attribute to Java property in XSLT.	141
5.12	Transformation rule specification between bar chart and pie chart. Arrows depict drag and drop.	143
5.13	Example bar chart and pie chart visualisations.	145
5.14	Mapping chart area notations.	146
5.15	Mapping a bar to a pie piece.	147
5.16	Transformation rules for transforming bar chart to pie chart.	147
5.17	Generated transformation script for transforming bar chart to pie chart.	148
5.18	End result of the bar chart to pie chart transformation.	149
5.19	Minard's map visualisation.	149
5.20	Specifying Minard's map to chart area transformation rule. Arrows depict drag and drop directions.	150
5.21	Specifying Troops movement notation to pie piece notation transformation rule. Arrows depict drag and drop directions.	151
5.22	Specifying Troops movement notation to pie piece notation transformation rule using subtraction and merge functions. Arrows depict drag and drop directions.	152
5.23	Transformation script generated as a result of rule specification of Figures 5.21 and 5.22 in XSLT.	152
5.24	Resulting pie chart visualisation.	153
5.25	Sample Visualisations of a UML class diagram (source) and Java visualisation (target).	154
5.26	Specifying transformation rule between UML class attribute and Java field property. Arrows depict drag and drop directions.	155
5.27	Specifying transformation rule between UML diagram and Java package.	156
5.28	Specifying transformation rule between UML class and Java class.	157
5.29	Specifying transformation rule between UML association and Java property.	158
5.30	Transformation rule script for transforming UML associations to Java property.	158
5.31	Resulting Java code visualisation.	159

Number	Description	Page
5.32	Defining a transformation rule for transforming a room in 2D CAD building to a room node in tree-based layout.	160
5.33	Using conditions to map 2D room notation to room node notation of a structure chart. Arrows depict drag and drop direction.	161
5.34	Defining a transformation rule for transforming a floor plan in 2D CAD building to a floor node in tree-based layout.	162
5.35	Defining a transformation rule for transforming a 2D CAD building to a tree-based layout.	163
5.36	Concrete representation of three rules required to transform a 2D CAD building to a tree-based layout.	163
5.37	Resulting tree structure chart.	164
6.1	Architecture of "Suggester" system.	167
6.2	Sample correspondences between UML class diagram example XML and Java code XML.	168
6.3	Sample correspondences between UML class diagram example visualisation and visualisation of Java code.	169
6.4	Example correspondences between UML class diagram example XML and Java code XML and their calculated score using name tag recommender.	171
6.5	Example UML class graph.	172
6.6	Abstraction graph of the UML class example in Figure 6.5.	172
6.7	Abstraction graph of a Java class.	173
6.8	Correspondences returned by value similarity suggester and their similarity scores.	173
6.9	Correspondences returned by type similarity suggester.	174
6.10	Sample graph of UML class diagram (A) and Java code (B).	176
6.11	Calculating IsoRank similarity for sample graphs [5].	178
6.12	Abstractions examples of two citation formats.	180
6.13	Sample recommendation list.	184
6.14	Result of accepting and rejecting recommendations.	184
6.15	Guide and Filter system for representing correspondence recommendations.	185
6.16	Example of recommendations that result in transformation rules for UML class diagram to Java code mapping.	186
6.17	Updated list of recommendations after selecting a parent correspondence.	187

Number	Description	Page
7.1	Components of CONVErT.	190
7.2	Using CONVErT's visualiser UI for mapping input model elements to visual notations. 1) Input model, 2) Predefined notation, 3) Designer canvas, 4) Recommendations, 5) Status panel.	194
7.3	Using CONVErT's visualiser UI for composing visual notations. 6) Customised notations' panel, 7) Notation composition Canvas, 8) Usage logs.	196
7.4	CONVErT's visual functions and conditions panel (9) and visualiser renderer (10).	197
7.5	CONVErT's mapper UI. 1) Source visualisation, 2) Target visualisation, 3) Highlighting elements, 4) Functions and conditions, 5) Recommendations and 6) Status panel.	198
7.6	CONVErT's rule designer. 4) Functions and conditions, 7) Rule designer canvas, 8) Panel for adding values, 9) A merge function, 10) Updated recommendations, 11) Rule designer status.	200
7.7	CONVErT's UI for visual representation of transformation rules (12) and 13) User logs.	201
7.8	Notation designer UI, 1) Name of the new notation, 2) Input XAML view, 3) Rendering of the input XAML, 4) Model data, 5) Status panel.	203
7.9	Using notation designer UI to annotate input view, 6) Annotated view 7) Controller transformation, and 8) Generated notation.	204
7.10	Components of renderer subsystem.	207
7.11	Sample visualisation file to be rendered by Renderer.	207
7.12	Example of notations retrieved from notation repository. a) Horizontal bar, b) Horizontal bar chart, c) Horizontal bar's model, and d) Horizontal bar chart's model.	208
7.13	Final rendered visualisation.	208
7.14	Components of CONVErT's abstraction subsystem.	209
7.15	Example input model representing Sales elements.	210
7.16	Abstraction graph of input file in Figure 7.15.	210
7.17	Components of Suggester system.	211
7.18	Components of CONVErT's transformation subsystem.	212
7.19	Transformation rule template for transforming a sales record to bar notation.	213

Number	Description	Page
7.20	Defining the transformation between sales element and bar's notation. Arrows depict drag and drop.	214
7.21	Resulting transformation code.	215
7.22	Transformation generation between a portion of input model and a visual notation.	216
7.23	Usage scenario for creating a transformation rule between portion of input model and a visual notation.	217
7.24	Transformation generation between notations scenario.	218
7.25	Usage scenario for creating a transformation rule between source and target notations.	219
8.1	Screen capture of ALTOVA MapForce. 1) Default tree-based representations, 2) Using a NAND function.	227
8.2	Mapping class diagram example to Java code visualisation example in CONVErT.	228
8.3	Representation of class diagram example and Java code example in MapForce.	229
8.4	Generated transformation rules for transforming UML class diagram to Java visualisation in CONVErT. Transformation rules are marked by 1.	230
8.5	Correspondence specification between UML class diagram example and Java code example in MapForce. Mapping correspondences are marked by 1.	230
8.6	Example correspondences for bar chart in ALTOVA MapForce. Arrows mark correspondences.	243
8.7	Example correspondences for bar chart in CONVErT. Arrows demonstrate drag and drop directions to specify correspondences.	243
8.8	Rendering the generated target model as a result of running transformation scripts of (A) CONVErT's and (B) ALTOVA MapForce.	245
8.9	Group based responses to question Q.13.	254

List of tables

Number	Description	Page
2.1	Comparison of most used transformation languages. + indicates support, (+) shows partial support and – shows no support.	23
2.2	Comparison of model transformation tools. + indicates support, (+) shows partial support and – shows no support.	36
4.1	List of default functions provided in proof of concept framework.	82
4.2	List of default conditions provided in proof of concept framework.	86
6.1	Correspondence recommenders used in Suggester system	170
8.1	Comparison of most used transformation languages and CONVErT's language. ✓ indicates support, (+) shows partial support and – shows no support.	223
8.2	Comparison of model transformation tools and CONVErT. ✓ indicates support, (+) shows partial support and – shows no support.	225
8.3	Summary comparison of ALTOVA MapForce and CONVErT.	233
8.4	Categories of all possible recommendations.	234
8.5	Resulted values calculated from the evaluation metrics.	236
8.6	Metrics and quality attributes to evaluate model transformations adopted from [6]. + indicates direct affect while – indicates adverse effects.	241
8.7	Comparison of transformation codes generated by CONVErT, ALTOVA MapForce and XSLT expert.	242
8.8	Demographic questions of the user study questionnaire and participant's responses.	249
8.9	User study questions for visualisation evaluation.	250
8.10	User study questions for transformation evaluation.	255
8.11	User study questions of Suggester system and user responses.	259

Chapter 1

Introduction

1.1 Background

Models have been around for many years and their application in science and engineering is evident. They describe some aspect of a System Under Study (SUS) [7]. This description varies depending on what aspects of the system are to be studied. For example, in aviation, engineers test models of planes in wind tunnel to assess and improve their aerodynamic efficiency. Planes here are the SUS and their aerodynamic characteristic is the aspect of the system being studied.

Many models for software engineering have been developed. These include flow charts and data flow models, entity-relationship models, object-oriented models, and behavioural models. However, often these models have been used in limited ways during software development lifecycle, e.g. for documentation.

Depending on application of software models and their modelling language, varieties of visualisations have been used to represent software models. The level of detail in these visualisations varies according to their targeted audience and may range from very high level to low level technical information.

Though a number of such software models have been proposed and used, the heavy use of models in software engineering has recently been motivated by the introduction of Model Driven Engineering (MDE) [8]. MDE promotes models as first class software

artefacts and aims to develop, maintain, and evolve software by performing transformations on these models; therefore, easing the development and maintenance of complex and large software systems [9]. It is fair to say then that, transformations are the “heart and soul” of MDE and Model Driven Development (MDD) [10].

A transformation, in software engineering terms, is automatic generation of a target model from a source model according to a transformation definition [11]. In the context of MDE, software development tasks are to be performed on models by means of transformations. For example, transforming higher level models to lower level models (e.g. UML models to program code) can be performed using refinement transformations [12]. Or in a maintenance example, refactoring transformations can be used to restructure models to improve quality of finished software product [13].

Following sections describe model visualisations and how model transformations are specified, and problems associated with specification approaches.

1.2 Model Visualisation

Due to complexity of software systems, there is a growing demand for approaches that incorporate visualisations in the software industry [14]. Software visualisation approaches concentrate on variety of applications including software landscape visualisation (e.g. Rigi [15] and software map [16]), hierarchical dataset visualisations (e.g SHriMP [17]), visual languages (e.g. Rimu and VML [18]). These visualisation approaches tend to reduce complexity of software development lifecycle.

With emergence of Meta-tools, generating visual languages and diagram based editors became easier and more feasible. Meta-tools allow generation of visualisation environments that provide facilities for users to interact with those visualisations. Example of such meta-tools are Marama tool-suit [19] and DiaGen [20].

Although many approaches for visualising data had been previously introduced (e. g. Skin [21] and relational visualisation notation [4]), none of these approaches have

successfully found their way in modelling environments and for MDE. In the context of MDE the most famous approach is Eclipse Modelling Framework (EMF) [22]. EMF models can be visually represented using the diagrammatic syntax of Graphical Modelling Framework (GMF) [23]. However, EMF and GMF are mostly targeted to technical users familiar with programming environments and IDEs and therefore have limited support for novice and none-programmer users.

This thesis seeks to provide a user friendly visualisation approach to visualise models and input data. It does so using by-example approach to transforming input data to visual notations. As a result, concrete visualisations are generated for each of the provided examples which may be very different (e.g. Charts, UML diagram and Java code).

1.3 Model Transformation

Transformations in software engineering are mostly performed on higher level abstraction of models, called meta-models. The primary use of these abstractions is to allow better generalisation and reduce the amount of coding required to implement transformation rules. Also, they provide a mechanism for evaluation, where it is possible to check if a resulting model conforms to the intended target.

To specify transformation on model abstractions, specialised model transformation languages such as Atlas Transformation Language (ATL) or Triple Graph Grammars (TGG) are used [24]–[27]. These transformation languages are designed to transform model instances that conform to source meta-models, to models conforming to target meta-model. For example, to transform a class diagram to Java code, a transformation script needs to be written in ATL (or other transformation languages). This transformation script includes correspondences between source abstraction (class diagram meta-model) and target abstraction (Java meta-model). These correspondences define relations between elements of both sides. Having this script ready, it can be applied on instances of source models (class diagrams), using a transformation engine, to transform them to target instances (Java code).

Transformation scripts usually include multiple transformation rules that are generally specified using textual representation of the transformation language of choice. This approach to transformation rule specification provides pragmatic barriers for non-software engineering users as support for development and debugging of scripts, or visual representation of rules and correspondences are very limited [28]–[32]. Also, correspondence relations of a transformation rule need to be specified on model abstractions and meta-models. Often, the representation of these abstractions are far removed from the representation of model instances they represent [33]–[35]. For example, Figure 1.1 shows an example of UML class diagram meta-model and a class diagram instance.

Apart from different representation, the way meta-models are represented results in hiding certain modelling concepts. In Figure 1.1 for example, it can be easily spotted that “orderDate” (a UML attribute) belongs to “Order” class by looking at class shape in the class diagram. To find the same relation in the meta-model, one has to follow “owner” association of “Attribute” element to “Classifier” and to “Class”. This phenomena is called *Concept Hiding* and has been regarded as a factor in complexity of meta-model comprehension [36]–[38]. For large meta-model and applications, spotting these relations would become fairly hard and complex.

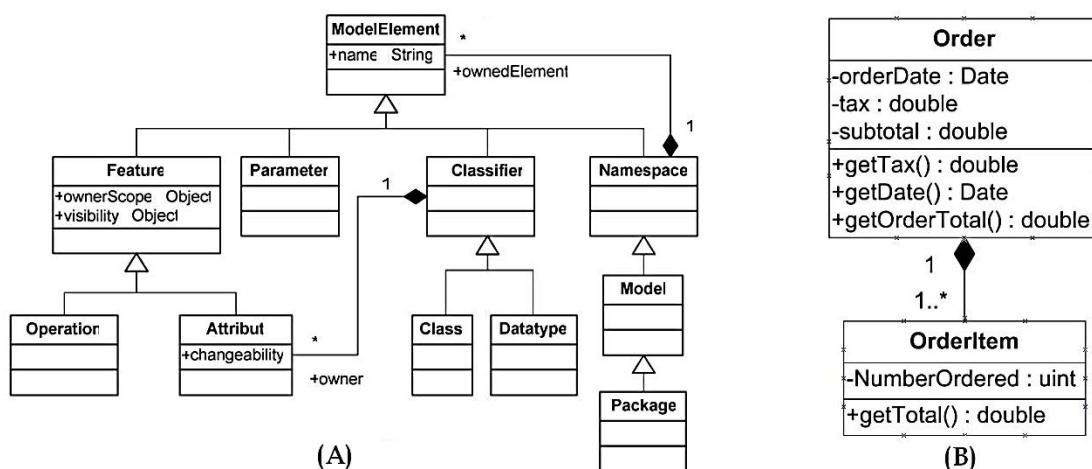


Figure 1.1 Example of (A) UML class diagram meta-model, and (B) Class diagram using concrete notation.

For computer scientists and software developers, using complex models, meta-models and textual coding is a routine practice; therefore, they will probably not experience great technical difficulty adapting current transformation approaches. However, for modellers that did not have usual education in type theory, programming, and transformation development, the complexity of meta-models and textual transformation representation may well be overwhelming and results in pragmatic barriers to defining and using model transformations [18].

These issues have motivated the research community to look for alternative approaches, namely, Model Transformation By-Example (MTBE) [30], [39]–[42] and By-Demonstration (MTBD) [38], [43], [44] approaches. These have their roots in programming techniques and date back to the research on innovative approaches to develop program source code, e.g. Programming By-Example [45].

MTBE and MTBD try to express a model transformation declaratively in the domain language of the source and target models. MTBE’s principle is to derive high level model transformation rules from initial prototypical set of interrelated source and target models. User has to provide multiple source and target instance pairs and specify their correspondences. The system then uses the defined correspondences to try to derive a generic transformation. To use these approaches, users must familiarise themselves with the syntax of the correspondence specification language. This is problematic as no visual approach for specifying correspondences on actual familiar notations (like UML Class Diagrams or source code) exists [29]. Also, since the rule derivation is semi-automatic, current approaches require multiple example pairs to exist and the user needs to refine derived rules, which makes adaptation of these approaches even harder.

In MTBD, changes that a programmer does to the model are recorded and then generalised to derive transformation rules. MTBD approaches incorporate a recorder component that monitors user interaction with the models and uses that recorded interaction as the basis of correspondence generalisation. Therefore, changes should be mostly applied in a predefined environment so that the recording component is able to monitor the interaction. These techniques are mostly limited to endogenous transformations where source and target model conform to the same meta-model [29].

The following section describes the objective of this thesis toward more user-centric model transformation, using a motivating example.

1.4 Thesis Objective

This thesis research addresses by example transformation approaches and tries to improve them. It does this by providing a visual correspondence specification approach where users use their desired graphical notations and model visualisations to specify source and target model correspondences. Also, since source and target models can become fairly complex even using visualisations, this thesis investigates ways to incorporate information retrieval techniques to provide guidance to users in form of “which elements of source and target models correspond”. To illustrate the intention of this thesis research, this section provides an example of MDE domain, transforming UML Class Diagrams to Java source code.

Assume John, a software developer, is working in an MDE-using team and has received a system analysis report for an application. Being an expert in UML diagram interpretation and a Java coder, he is familiar with concrete syntax of the diagrams and Java code. He is interested in transforming specific parts of UML diagrams provided by the analysis directly to his programming code, to increase team productivity, code quality and to ease software evolution. For example, he wants to create a model to code translator to transform specific features and parts in the analysis UML diagrams to specific Java code templates.

As mentioned previously, with current transformation approaches, transformation designers typically have to specify or provide meta-models for both class diagram and the Java source code. A schematic view of UML class diagram meta-model was previously provided in Figure 1.1. Figure 1.2 demonstrates simplified Java meta-model and a Java code example.

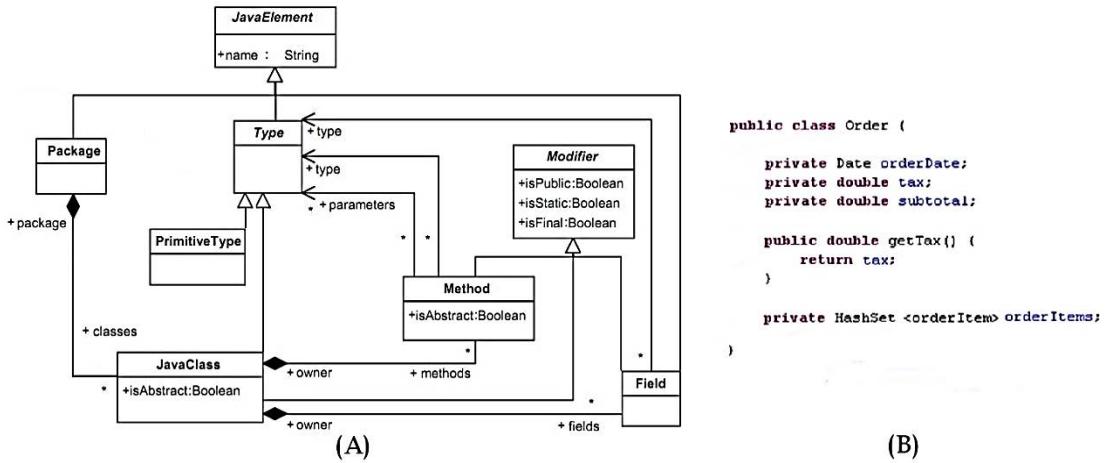


Figure 1.2 Example of (A) Java meta-model, and (B) Java source code.

For John, as an expert in the domain, corresponding elements in the UML diagram and in his Java code are obvious. He can clearly spot and relate classes, methods, and even statement snippets in both program code and class diagram. Some of such correspondences are depicted by Figure 1.3. For example, John can easily relate an attribute in a class diagram (e.g. `tailID`) to a property in Java code (e.g. `owner`) and their fine-grained elements e.g. name, type and identifier of an attribute (e.g. `+`, “`tailID`” and `String`) to identifier, type and name of a java property (e.g. `public`, `string` and “`owner`”). Figure 1.3 shows examples of these mappings with solid arrows and their internal fine grained mappings by dashed arrows.

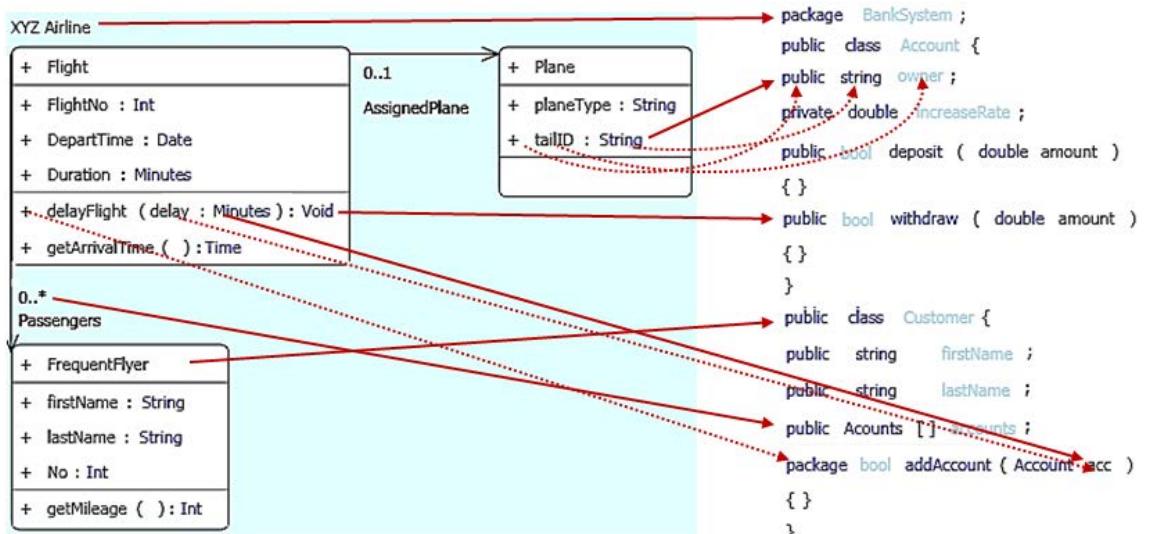


Figure 1.3 Examples of correspondences between UML class diagram and Java code. Dashed arrows demonstrate more fine grained correspondences.

Comparing meta-models of Figures 1.1 and 1.2 and the concrete representations of Figure 1.3 reveals that the concrete notation is much more visually appealing and familiar than the complex notations used in meta-model specification. Given that John may not have experience or knowledge of transformation languages and meta-modelling, specifying correspondences is better understood by him using one or more class diagrams and corresponding code examples, as shown in Figure 1.3. However, using current approaches to create such a model to code transformation, he has to work with the complex syntax of abstract UML and Java meta-models, and the low-level textual syntax and semantics of transformation languages, such as XSLT, ATL and QVT.

Appendix 1 provides an example of transformation script written in ATL to transform UML class diagram to Java code. This ATL example is adopted from ATL transformation Zoo [1]. Even though the examples have been significantly simplified, the transformation script includes more than 150 lines of code and demonstrates many meta-model constructs. In larger examples, this transformation can expand rapidly resulting in more complexity for transformation designers.

Provided that a By-example approach is being used for transformation, correspondences will be specified on examples of source and target models. Figure 1.4 shows examples of simplified UML class diagram and Java code in XML. Sample correspondences between these two examples are marked by red lines in the figure. This specification of correspondences using current techniques requires the user to use a correspondence language specific to the by-example approach being used. For instance, John would need to specify that an operation in class diagram directly corresponds to a method in Java code, and so does its return type, name, and its access privilege. He then has to repeat this for other available example pairs. For John as a user familiar with concrete visualisations of these models, it is more convenient to specify these correspondences on concrete visualisations similar to Figure 1.3. Therefore, this thesis seeks to provide such capable concrete visualisations so that he can simply drag and drop elements of his generated source and target visualisations to specify a correspondence.

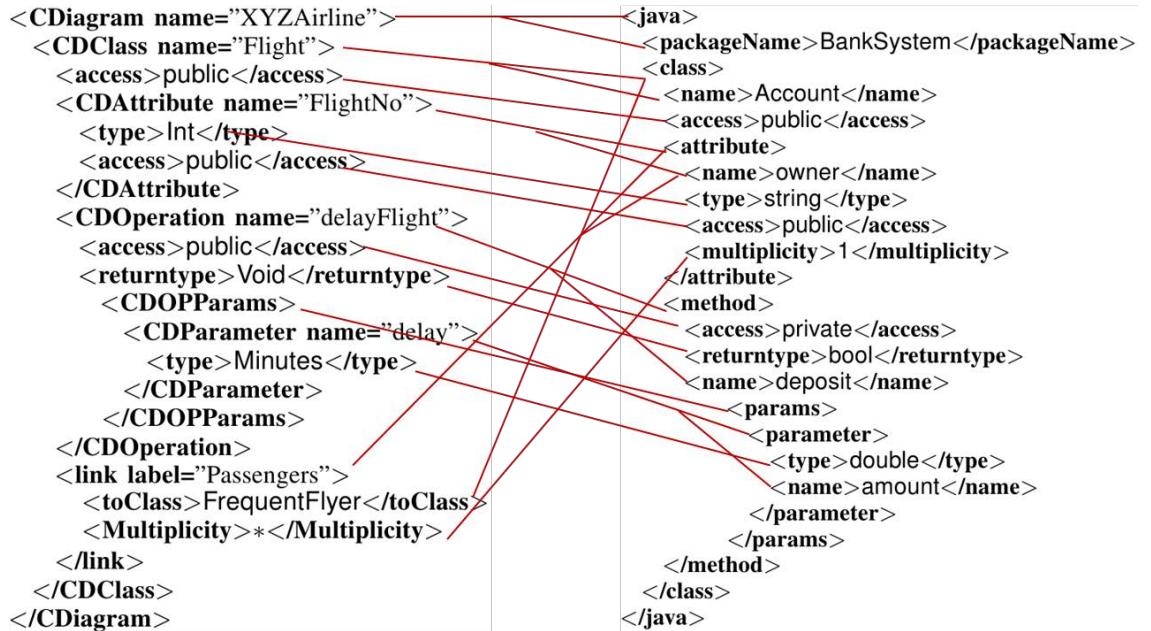


Figure 1.4 Sample correspondence relations between UML class diagram XML example and Java code XML example.

Although using visualisations improves the cognitive descriptive power and understandability of correspondence specifications, some software models get fairly complex due to the number of features they are describing. Computers deal with this complexity quite differently than humans. Therefore, in an ideal situation, an automatic mechanism can analyse input models and provide guidance to the user to specify correspondences [46]. For example, in the aforementioned example, an automatic routine might analyse the UML class diagram and java code and suggest that parameters in a function declaration of the class diagram could be matches for parameters in a java method. John can use these suggestions as guidance or select them as his source to target correspondences. Providing such semi-automated user support features for transformation specification is another key goal of this thesis.

1.5 Contributions

We address the user friendliness of model transformation generation in this thesis. Key contributions of this thesis are the following:

- Producing reusable model visualisation specifications using an interactive, by-example approach.
- Using a concrete, by-example model transformation metaphor.
- Model mapping and transformation specification by drag and drop between concrete visualisations.
- Utilising a set of recommenders using various recommender system techniques and generating mappings from recommendations.
- Supporting fully automated model transformation script generation from specified mappings.
- Providing scalable, easy-to-use, robust and extensive tool support for each of these facilities.
- Carrying out a comprehensive end user evaluation of our prototype toolset and overall approach.

In the following, these contributions are described in more details.

Target users of this thesis approach are the users that are not trained in complex transformation languages and meta-modelling, but are familiar with specific modelling languages and their visual notations. For these users, the correspondences between participating source and target models are relatively clear. However, to define these correspondences as transformation rules should not involve learning new modelling concepts and acquisition of significant new knowledge.

To achieve this objective, we have decided to provide a mechanism to use arbitrary, end-user-oriented visual notations in the model transformation process. This approach allows users to define (or reuse) model visualisations for provided source and target examples and use these specified by-example visualisations for correspondence specification between source and target examples. We also incorporate a guidance system that helps novice users and experts alike in the visualisation and transformation specification process. Each of these decisions contributes to making the complex model

transformation specification and generation process more user-centric than it currently is.

The adaptation of visual notations for correspondence specification requires a representation model for visual notations that allows semantic links between elements of a notation and the actual model data (context). These models should allow possible extensions and flexibility for users to create them in order not to be limited to the fixed number or type of notations. These visual notations are then mapped to input examples and composed to generate interaction capable visualisations. As a result, users will have the freedom to choose (or design their own) notations that are familiar to them and define transformations using visualisations generated from the notations.

A distinctive part of this thesis concerns the way support can be provided to the users for visualisation generation and complex model transformation specification. Part of this support comes with providing hints on visual notation composition of the visualisation process. To provide such support, the visualisation mechanism allows the user to review samples of visualisation to be created as a result of composing visual notations, before actually generating it. Therefore, if the resulting visualisation sample is not what the user expects, it can be altered to achieve desired results.

As large model visualisations and transformations can become very complex, the approach provides guidance on possible and likely correspondences that eventually create transformation rules. To achieve this, an automated recommender system analyses the interaction and input examples for recommending possible correspondences between model sub-structures. The user can see these recommendations as guidelines to develop transformation rules or simply use them to create final transformation artefact.

These contributions are realised in our proof of concept tool “CONcrete Visual assistEd Transformation” framework, or CONVErT for short. CONVErT provides a proof of concept implementation of each of the above research contributions and plays an important role in the validation of our design and approach. Its most important aspects include: (i) a concrete model visualisation specification framework; (ii) a set of proactive model correspondence suggesters; (iii) a model for model correspondence

rules; (iv) an XSLT code generator to implement specified model transformations; and (v) a reverse engineering mechanism for automatic extraction of a meta-model from the set of input models.

1.6 Thesis organisation

This thesis is organised in the following nine chapters. Chapter 1 (this chapter) introduces the background and a motivating example for this work. It gives an overview on the objectives and summarises our key research contributions.

Chapter 2 reviews the state of the art in model transformation specification and generation. It provides discussions on modelling and MDE, model transformation, use of visualisations in transformation, model transformation tools, approaches to visualising information and models, and finally techniques to providing user support and guidance.

Chapter 3 provides an overview of our approach and methodology. It lists research questions being addressed in this thesis and our approach in solving them. It also depicts the scope and boundaries of this thesis research.

Chapter 4 describes our approach for model and information visualisation. It provides details of how intractable visual notations are defined, mapped to input data, and composed to generate complex visualisations. These visualisations allow user interactions in form of drag and drop and can be used for model transformation specification. Multiple case studies are provided in the chapter to demonstrate application of our visualisation approach for different domains.

Chapter 5 describes how visualisations can be used in transformation specification. Transformation rules are defined by drag and dropping visual notations of source and target model visualisations. From these visual by-example transformation rules, full transformation scripts are generated and used to transform data provided by source visualisation to the visualisation of target. Chapter 5 also provides multiple case studies using visualisations defined in chapter 4 to generate transformations.

Chapter 6 describes our approach in providing user guidance and support for transformation specification. A recommender system is designed to provide recommendations based on possible and likely correspondences between source and target models. Users can choose among recommended correspondences to generate transformation rules.

Chapter 7 describes our tool support and proof of concept framework CONcrete Visual assistEd Transformation (CONVErT). It provides details of CONVErT's implementation and user interface and expresses the technical and engineering decisions made on the framework.

Chapter 8 contains evaluation of the CONVErT approach. This evaluation has been divided into a comparison study, a quantitative evaluation, and user evaluation. A detailed comparison study is provided to check the approach provided in this thesis against a state of the art transformation and mapping tool (ALTOVA MapForce [47]). The quantitative evaluation is targeted to the evaluation of the recommender system being used for the user support mechanism. It also includes a quantitative evaluation of the automatically generated transformation script using a selection of transformation code quality attributes and metrics. Chapter 8 also includes details of our user evaluation and its experimental setup including user tasks and questionnaires. Analysis of the questionnaire responses by the users and a discussion are also provided.

Finally, key conclusions from this research are summarised in chapter 9 along with some key directions for future research.

1.7 Summary

This chapter provided an introduction and background of this thesis. It described thesis objectives using a motivating example and demonstrated its contributions on the basis of the motivating example. This chapter also briefly described the organisation of this thesis and provided an overview on the remaining chapters.

Chapter 2

Literature Review

2.1 Introduction

This chapter describes state of the art in model transformation. It describes modelling and the importance of transformations to realisation of Model Driven Engineering (MDE). It then investigates current transformation approaches with a focus on by-example transformations and visual mapping techniques. It provides a comparison of current approaches and where the approach presented in this thesis fits in that context. Since the approach presented in this thesis includes visualisation and user guidance, this chapter also provides a brief review of model and information visualisations approaches and user guidance mechanisms for interactive techniques.

2.2 Modelling and MDE

We start this section by definition of a Model:

“A model is a set of statements about some system under study (SUS) ... statement means some expression about the SUS that can be considered true or false”[7].

Based on this description, a model of, for example, a bridge describes a system under study (SUS) - in this case is a bridge. It makes statements on the structural capability of

the bridge which could be true or false. With regards to software engineering domain, for example, a UML class diagram is a model that describes an object oriented software system. The software system here is the SUS and the classes and relations between them can be considered its statements [7].

Models in MDE are defined in Technical Spaces (TS). Kurtev et al. define TS as:

“... a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. It is often associated to a given user community with shared know-how, educational support, common literature and even workshop and conference meetings” [48].

Examples of technical spaces conforming to this definition are XML-based languages, Ontology Engineering and Database Management Systems (DBMS) [27], [48]. MDE can also be seen as a technical space itself [49].

Models in general are diverse and may represent software processes, designs, code, configurations, performance or other data [50]. Unlike traditional software documentation models, models in MDE are not considered as design sketches. Instead they are the primary artefacts that drive the development process where development, maintenance and evolution of software is performed by transformations on models [8], [51]. In this configuration, a model transformation is performed during each design phase to complete micro-processes or to assist the designer complete the phase [52]. Transformations are therefore an integral part in realisation of MDE [9].

Model-driven transformations are also applied in other domains, for example in data processing to transform complex data from one form to another [53]. The most well-known proposal for MDE is the Model Driven Architecture (MDA) by Object Management Group (OMG) [54]. It plays a significant role in promoting MDE using selection of technologies readily available and previously adopted by OMG. MDA envisions higher order models, called Platform Independent Model (PIM), to be transformed to richer and more refined models, called Platform Specific Models (PSM), by means of refinement transformations. PIMs describe a system at a level of abstraction that is sufficient to allow use of their entire contents for implementing the system on different platforms [55]. A PSM in this configuration can conform to

multiple PIMs each focusing on different aspect of the modelled entity. Similarly, different PSMs can be generated from a PIM using different refinement transformations.

Depending on application domain, models can exist at different abstraction levels. For example, a class diagram as a model is in higher abstraction level than the Java code derived from it. Similarly, UML metamodel is in higher abstraction level than class diagram. Following subsection describes concrete and abstract models with regards to the level of abstraction each model is represented in and the amount of information it presents.

2.2.1 Concrete versus abstract models

The modelling domain and programming languages domain are very similar with regards to classification of artefacts as concrete and abstract. In the programming language domain, abstract syntax describes the concepts of a given language independently of the source representation of that language. These abstract syntaxes are primarily used by tools such as compilers for internal representation [56]. Concrete syntax, on the other hand, provides a user friendly way of writing programs and is more familiar for programmers. For example, an Abstract Syntax Tree (AST) can be considered a generic abstraction for arbitrary programming languages. A C++ program code generated from that AST is a specific concrete syntax which conforms to that abstraction. Figure 2.1 shows an example C++ method code and its AST. The same AST may also be used to generate Java code, or other programming languages.

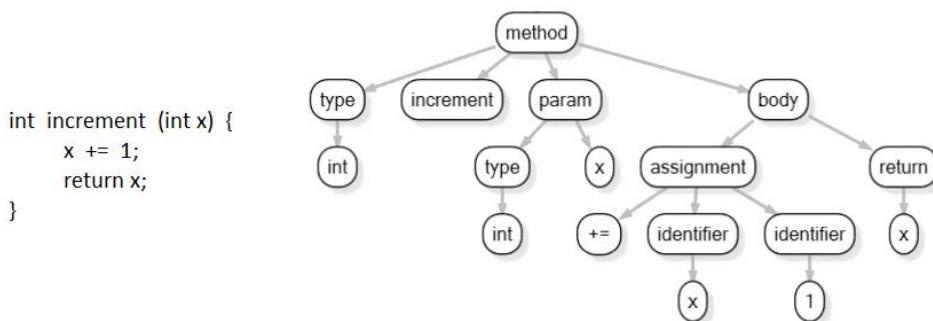


Figure 2.1 Example C++ code and its corresponding AST.

Similarly in the modelling domain, a concrete model is a model represented in the concrete representation of the modelling language. Depending on modelling language domain, concrete syntax can be specified using text, shapes or other means. For example, a graph is a concrete model represented using vertexes and links, or entity diagram is a concrete model specified using boxes and lines, UML class diagram is also concrete model using boxes, lines and labels as shown by Figure 2.2.

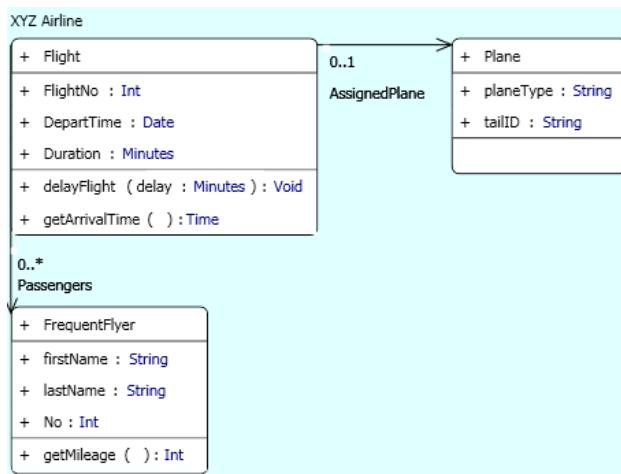


Figure 2.2 UML class diagram example.

Abstract syntax of models is mostly described by metamodels. A metamodel can essentially be considered as a model of models which includes set of concepts to create models [50]. The abstract syntax of a modelling language can be then identified with metamodels, while its representation can be defined in the concrete syntax [50]. For example, the abstract syntax in UML metamodel constrains the allowable structure and relationships between model elements represented as instances of meta classes [7]. An example of a simplified UML metamodel is provided in Figure 2.3.

A metamodel that captures abstract syntax of a model, describes the structure of all possible models derived from it. Metamodels are not however a complete definition of modelling languages since a language definition needs to describe also how a model is rendered by textual and/or graphical elements (concrete syntax) and what the intended meaning of each modelling concept is (semantics) [57].

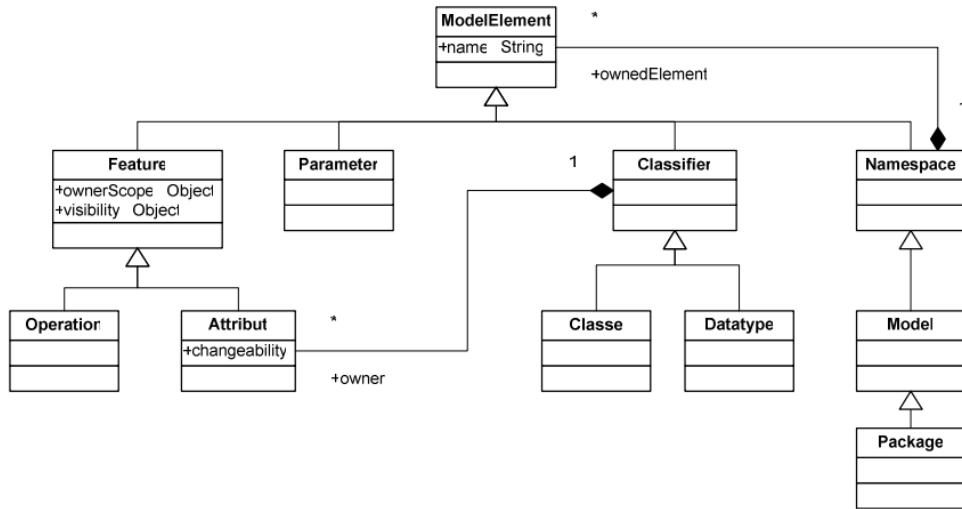


Figure 2.3 Simplified UML metamodel (from [1]).

A concrete model is usually more understandable for domain experts as it describes domain specific concepts which are familiar for the experts [35]. For example, it is much simpler for users (such as programmers) to write programs in Java, rather than directly using instances of Java meta-modelling concepts [56]. Or it is easier for business analysts to use a form-based metaphor for generating data mappings rather than using data schemas and mapping languages [32].

Abstract syntax can be used as an intermediate language such that multiple languages can be expressed in it [58]. It also provides a good generalisation of models which can reduce the amount of coding required for transformation generation between models. On the contrary, the abstract syntax may contain elements that cannot be expressed in the concrete syntax of the language [59]. Abstractions also hide certain concepts that are easily understandable using concrete syntax, and therefore make it harder for users to understand them [37]. Using abstract syntax, although helping in the reduction of efforts for designing model transformations, it also becomes harder and less user friendly for non-expert user modellers.

2.2.2 Model transformation and MDE

MDE is defined around the idea that “everything is a model” [60]. With this assumption, software development tasks will be performed by transformations on these models. Given the task to be performed, different model transformation approaches can be used. For example, transforming a PIM to PSM can be performed using refinement transformations. These transformations provide domain specific information required to be inserted in the PIM [12], [61]. The reverse direction can be performed using abstraction transformations. With abstraction transformations, some domain specific data will be abstracted away from the PSM that results in a more generic target model (PIM). Figure 2.4 demonstrates these transformations.

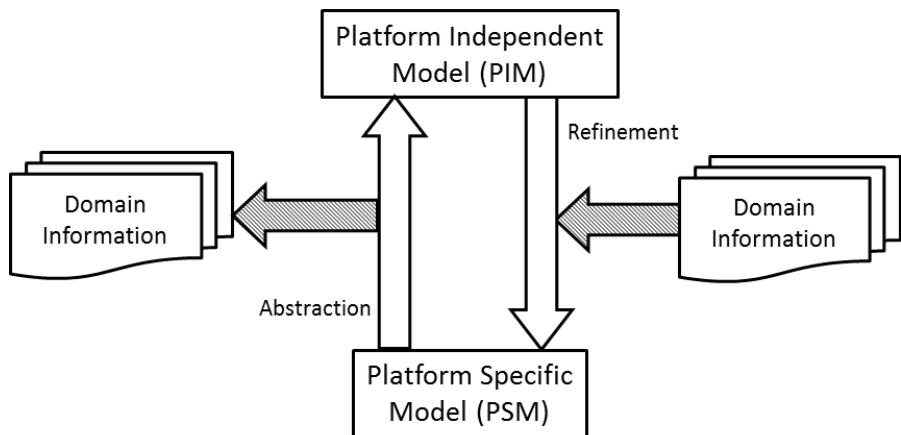


Figure 2.4 Transforming PIM to PSM and vice versa using refinement and abstraction transformations.

Similar to refinement and abstraction, migration and refactoring transformations can help alter models to perform specific software development task. Migration transformations for example, help transform models of one modelling language to another modelling language while maintaining the same level of abstraction. An example is transforming class diagrams to relational database diagram [25]. Refactoring transformations help restructuring models while keeping them at the same abstraction level and conforming to the same metamodel. Refactoring transformation can help improve quality of software models [13].

Similar to transformation task mentioned above, multiple other software development tasks exist that can be performed by model transformations in the context of MDE. A collection of these tasks can be found in model transformation literature [62]–[66]. The following sections review a selection of most widely used approaches and languages for implementation of model transformations.

2.3 Model transformation

In-line with importance of model transformation for realisation of MDE, many approaches, tools and transformation languages have been proposed. To compare these approaches, a list of characteristics of model transformations is provided in this section. These characteristics are selected from previous literature and classification surveys of model transformations [62], [63], [65].

Technical Space Transformations can be classified according to the application domain or technical space of source and target models they are applied to. Some transformation approaches are hard coded for specific tasks. These transformations are usually transformations written with general programming languages (e.g. C, Java) and are built for specific source and target models. There are a number of transformation languages available that are built purposefully for certain domains. For example TXL transformation language is designed for transformation between programming languages [67]. Other transformation languages may consider alternative domains and technical spaces with larger variety of source and target types.

Syntax (*Concrete versus Abstract*) Transformations can be specified on concrete or abstract syntax of source and target. We refer to this as Input Artefact Syntax, i.e. syntax of input source and target models and their abstraction level. Most general purpose transformations are specified and applied on abstraction (metamodel) of source and target. For example to write transformations with ATL, users have to specify correspondences between elements of source and target metamodel. As a result, syntax of the input artefacts when transformation is specified by ATL is abstract. For example, Figure 2.5 demonstrates an ATL transformation rule for transforming UML operations to Java methods. **Transformations on concrete syntax are generally easier to understand**

for non-expert users due to use of familiar concrete syntax e.g. the box and line shapes in a class diagram. Transformations on abstract syntax, on the other hand, are more scalable since they allow application of transformation rules on more high level syntax of models and therefore there is no need to specify multiple transformation rules for similar lower level elements. For example, user does not need to provide a transformation for each UML attribute separately in a class diagram.

For specification of transformations and writing transformation scripts, current transformation approaches generally use textual syntax of transformation languages or a fixed graph-based visualisation of source and target models. We refer to this as ***Specification Syntax***. Textual and graph-based specifications, although hard for non-experts, are generally referred to as concrete syntax [57]. For example ATL provides a textual syntax similar to programming languages to specify transformations. Graph grammars have a graphical view of the metamodels being used as source and target. Although they help a lot in understandability of metamodel artefacts, the transformations still need to be specified in textual syntax.

```

rule O2M {
    from e : UML!Operation
    to out : JAVA!Method (
        name <- e.name,
        isStatic <- e.isStatic(),
        isPublic <- e.isPublic(),
        owner <- e.owner,
        type <- e.parameter->select(x|x.kind=#pdk_return)->asSequence()->first().type,
        parameters <- e.parameter->select(x|x.kind<>#pdk_return)->asSequence()
    )
}

```

Figure 2.5 Sample of ATL transformation rule for transforming UML operations to Java methods (From [1]).

Rule application control Depending on how transformation rules are executed, their transformation rule application control can be imperative or declarative. Imperative rule applications require specific control flow to be provided to describe how (and in what order) the transformation rules are supposed to be executed. Declarative rules focus on what should be transformed to what instead of how or in what order.

Transformation Engineering (Exogenous versus Endogenous) Exogenous transformations transform models conforming to different metamodels. For example, transforming UML class diagram to Java code is an exogenous transformation. Endogenous transformations on the other hand transform models that are specified in the same metamodel or modelling language. A good example of endogenous transformations is refactoring transformations.

Transformation Scenario (Vertical versus Horizontal) Transformations can be vertical, i.e. source and target are at different levels of abstractions, or horizontal, i.e. source and target are at the same level of abstraction. Considering this characteristic, transformations of PIM to PSM (and its reverse), as defined by MDA, or program synthesis that transforms a high level programming language to a low level or machine language, are examples of vertical transformations. On the other hand, transforming for example UML class diagram to Relational Database Management System (RDBMS) models is a horizontal transformation since source and target have different metamodels. UML class diagram metamodel is generally specified by MOF and RDBMS metamodel is specified by database schemas.

Exogenous and endogenous transformations can be either horizontal or vertical. For example refactoring transformations can be considered as horizontal and endogenous transformations, refinement transformations are endogenous and vertical. While languages migration transformations are horizontal and exogenous, and code generation transformations are vertical and exogenous [65].

Directionality (Unidirectional versus Multidirectional) Unidirectional transformations only specify source to target transformation, i.e. only one dimension. Bidirectional transformations specify source to target (forward direction) and its reverse, that is, target to source transformation. If multiple sources or targets are involved, multidirectional transformations can specify transformation in multiple directions. Bidirectional transformations are a special type of multidirectional transformations where a single source and a single target are being used in transformation. Bidirectional transformations can be achieved by providing separate transformations for forward and reverse directions.

Table 2.1 lists a group commonly used model transformation languages. It provides a comparison according to the technical spaces that the languages are designed for, Specification syntax of the languages, supported syntax of the artefacts being used as source and target, whether rule application is imperative or declarative, whether the language supports vertical, horizontal, exogenous or endogenous transformations and whether it supports transformation rules for single or multiple directions. Directionality here refers to implicit support for writing for example bidirectional rules rather than having a separate unidirectional transformation for each direction. Following paragraphs briefly describe the transformation languages and their characteristics.

Table 2.1 Comparison of most used transformation languages. + indicates support, (+) shows partial support and – shows no support.

Transformation language comparison	TGG [24]	ATL [68]	XSLT [69]	QVT [70]	MOLA [71]	ETL [72]
Technical Space						
Model	+	+	-	+	+	+
Text	-	+	-	+	-	-
XML	+	+	+	+	+	-
Specification Syntax						
Text	+	+	+	+	-	+
Graph	+	-	-	+	+	-
Visualisation	-	-	-	-	-	-
Input Artefact Syntax						
Abstract	+	+	+	+	+	+
Concrete	-	-	-	-	-	-
Rule application control						
Imperative	-	(+)	+	+	+	+
Declarative	+	+	+	+	+	+
Transformation Scenario						
Vertical transformation	+	+	+	+	+	+
Horizontal transformation	+	+	+	+	+	+
Transformation Engineering						
Exogenous	+	+	+	+	+	+
Endogenous	+	(+)	+	+	+	+
Support for directionality						
Unidirectional	+	+	+	+	+	+
Bidirectional	(+)	-	-	(+)	-	-
Multidirectional	-	-	-	-	-	-

Triple Graph Grammar (TGG) [73] is among the most used model transformation languages. TGG provides a means for declarative specification of transformation between pairs of source and target graphs which are connected using a correspondence graph. The correspondence graph records information and constraints on the matches between source and target graphs. TGG allows specification of bidirectional transformation rules. TGG has been extensively utilized in FUJABA and GReAT tool suites [74]–[76] for model transformation. Graph based approaches like TGG can benefit the use of visualisations for source and target in form of graphs. For example, a graphical representation of model transformations for TGG was provided by Grunske et al. [25].

Atlas Transformation Language (ATL) is the transformation languages of ATLAS Model Management Architecture (AMMA) [26], [27], [68]. It allows transformations to be specified for metamodels and uses a concrete textual syntax. ATL is a declarative language; however, there are helper functions provided which can enable limited imperative rule applications.

XSLT is a transformation language for manipulating XML data [69]. It is a functional language that allows both imperative and declarative programming. Imperative rules can be generated using powerful XPath matching functionality [77]. XSLT accepts source and target in XML or similar formats like XHTML. An example XSLT script is provided in Figure 2.6.

Source	Transformation Script	Target
<pre><?xml version="1.0" ?> <data> <user> <username>Foo</username> <name>Bar</name> </user> <user> <username>Foo2</username> <name>Bar2</name> </user> </data></pre>	<pre><xsl:stylesheet xmlns:xsl="http://www.w3.org/2005/xpath-functions"> <xsl:template match="/data"> <root> <xsl:apply-templates select="user"/> </root> </xsl:template> <xsl:template match="user"> <user> <xsl:attribute name="userID"> <xsl:value-of select="username" /> </xsl:attribute> <xsl:value-of select="name" /> </user> </xsl:template> </xsl:stylesheet></pre>	<pre><?xml version="1.0" ?> <root> <user userID="Foo">Bar</user> <user userID="Foo2">Bar2</user> </root></pre>

Figure 2.6 Example XSLT script and its source and target XML.

Although models can be specified in XML, and string values can be used in XML, we do not categorise XSLT as a transformation language that can be used in Model or text technical spaces. Neither can it be classified as working on concrete syntax as XML needs to be parsed to represent concrete syntax in form of text, graphics, etc.

Query Views Transformation (QVT) is the transformation language of OMG's MDA [70]. Like ATL it provides model-to-model transformation defined on both XMI-based metamodels and textual concrete syntax. It includes three languages, QVT operational, QVT Relations and QVT Core. QVT operational is an imperative language for model-to-model transformation based on EMF models. QVT Relations, on the other hand, is a declarative language defined as part of QVT which provides relationships between MOF models. It has a textual and graphical representation for specification of relations between MOF models. Operational language defines forward and backward transformations separately and requires consistency to be preserved manually, while Relations language describes how the source and target relate to each other in a bidirectional manner. QVT core is a lower level language which both Relations and Operational languages can be translated into and is inspired by the Triple Graph Grammars approach [78].

MOdel transformation Language (MOLA) is a graphical language based on graph transformations that provides graphical constructs for specification of transformations [71]. It provides graphical structures, which are quite similar to UML activity diagrams, to define transformation rules [79].

Epsilon Transformation Language (ETL) is the model transformation language of the Epsilon model management infrastructure [72]. It offers rule scheduling by specifying lazy rules that are only executed when they are explicitly called, and by guarded rules that are only executed if their guard evaluates to true. ETL provides both declarative and imperative style of programming.

As can be seen in Table 2.1, the languages for specifying transformations mostly use the abstraction of source and target models as input artefact syntax. Use of these abstractions introduces high barriers for non-expert users, since they require users to learn and use complex meta-models which are far removed from the modelling

language of source and target models [29], [37], [80]. Also the need for specification of such transformation using textual syntax of transformation languages adds to the problems of model transformation specification [18]. As an alternative, approaches have tried to address this problem by automation of transformation rule derivation using concrete model examples[29], [40], [81], [82], transformation generation by demonstration [38], [45], or automatic alignment of models and metamodels to eliminate the need for learning transformation languages [83]–[85]. The following subsections provide an overview on these approaches.

2.4 Transformation using concrete models

Approaches to perform transformation on concrete models are grouped in two categories: Model Transformation By Example (MTBE) and Model Transformation By Demonstration (MTBD). They have their roots in programming techniques and date back to the works on alternative approaches to develop program source code, Programming By Example [45]. These techniques leverage user interaction and well-formed rules to replace the text centric source code design. In line with these techniques, MTBE/MTBD tries to express the transformation declaratively in the domain language of source and target models without having to specify metamodels or transformation languages. Following sections describe these approaches in detail.

2.4.1 Model Transformation By-Example (MTBE)

MTBE’s principle is to derive high level model transformation rules from initial prototypical set of interrelated source and target models. This concept was first used by Varro et al. incorporating graph transformations [40]. The idea is to provide multiple source and target model pairs, and ask a user (domain expert) to specify source and target model element correspondences. The system then uses these correspondences to derive transformation rules. This approach was later improved by replacing user centric heuristics with predefined knowledge in form of Inductive logic [41]. Inductive logic

programming is intersection of inductive learning and logic programming and aims at construction of first order clausal theories from examples and background knowledge. There, the designer assembles prototype mapping models by showing how source and target elements should be interrelated and based on this input the system should synthesize the set of model transformation rules. Although this approach eliminated the use of metamodels, the modeller must now focus on generating the inductive logic as meta information for the transformation engine in a way that the system can derive transformation rules.

These approaches can handle only one-to-one relationships between target and source elements and while defining the inductive logic is a separated task than model transformation itself, the transformation designer should have knowledge of logic programming to define it. Also, these approaches are not adaptable to situations where only a few examples are available [39].

Wimmer et al. introduced a conceptual framework for mappings generated from models using the syntax of the modelling language [36], [86]. Their conceptual framework allowed users to define both models inside the framework and specify mappings and corresponding elements. Then a model transformation generator would generate transformation code. Later, they introduced special mapping operators to give the user more expressivity for defining model mappings [86]. It allowed the definition of semantic correspondences on concrete syntax, from which ATL rules could be derived.

Using the examples that are provided to the system, Kessentini et al. proposed Model transformation as optimisation by example (MOTOE). This approach views model transformation as a problem to be solved using fragmentary knowledge [82]. They proposed to view model transformation as an optimisation problem. Two strategies were chosen: first, parallel exploration of different transformation possibilities on example source and target models by means of a global search heuristic. And second, use a hybrid global/local heuristic to improve initial transformations [39]. Since the number of solutions becomes huge, the problem becomes optimisation of transformation generation on the solution space of possible transformation rules. Block constructs were introduced as a previously performed transformation trace between a set of constructs in the source model and a set of constructs in the target model. Then,

finding a good transformation is based on finding the combination of constructs that maximises the inside block coherence and between block coherence. This approach tries to propose a transformation even when rule induction is impossible since it chooses the closest possible rule. However, since all possibilities are considered, it will become very complex. Moreover, rule generation is not deterministic since multiple runs might result in different rule generation.

Kessentini et al.'s work differs to MTBE, since MTBE focuses on automatic transformation between models containing same information in different forms. This work allows interactive and incomplete transformation. Examples in MTBE are complete models and therefore, generated transformation can be tested on example models. In this approach however, since models may be partially complete they are not immediately suited for testing.

Recently Faunes et al. used an evolutionary computation algorithm to derive model transformation rules [81]. Their approach did not require detailed mappings between models and could not produce executable rules. Derivation of transformation rules in their approach is guided by the ability of generated rules to successfully transform the provided examples, which guarantees that they are executable with the right control sequence. Their selection process favours the rules with the highest fitness value, i.e. the resulted target matches the intended example best [81].

Although many-to-many transformations are possible with some general model transformation approaches, MTBE approaches are all applicable on one-to-one concrete mappings and cannot consider many-to-many cases. García-Magariño et al. [30], proposed an improvement to MTBE by generating M:N rules based on constraints defined for simulating input patterns of several elements and preserving them in a dictionary. They embed mappings in models by indicating how information should be transformed from input to output and used metamodels to declare how a model is to be constructed [30]. By defining generic transformation grammar in EBNF, the generated transformation could possibly be implemented by transformation languages like QVT, ATL, etc. Although their method provided a possibility of defining many-to-many transformations, which is a key limitation of many transformation mechanisms, due to

extensive use of metamodels and metadata in the dictionary, we cannot classify it as concrete by-example approach to model transformation.

Semi-automatic generation of transformation rules in MTBE approaches often leads to an iterative manual refinement of generated rules. Therefore the model transformation designers may not be isolated completely from knowing the transformation languages and metamodel definitions [38]. The inference of transformation rules depends on given sets of mapping examples, so one or more mapping examples must be available to set up a precise prototype mapping. Users are required to learn and use the syntax of a correspondence specification language to specify seeding correspondences. This is often problematic as no visual approach for specifying correspondences on actual familiar notations exists [87]. Seeding the process with such examples is not always an easy task in practice [39]. Also, current MTBE approaches focus on mapping the corresponding concepts between two different models and providing complex mappings like arithmetic or string operations is not possible.

2.4.2 Model Transformation By-Demonstration (MTBD)

Model transformation by demonstration (MTBD) approaches are based on an expert performing transformation tasks and recording of the process steps by a recorder [38], [43], [44]. Robbes et al. take this approach and design a system that is capable of capturing changes a programmer would perform to the program code [44]. This system then generalises the changes to form abstract changes for reuse. They model their system as an evolving Abstract Syntax Tree (AST) with nodes representing class, package, methods, variables and statements. Each AST is accompanied by a history to record changes. Those changes would be then generalised to form abstract changes, allowing them to be reused. Their approach is most suited for refactoring applications of program code.

Sun et al. proposed a similar approach for model transformation where the user does the transformation on one instance of the model and the system records the user's interaction. The system then generalises the recordings and imitates the procedure on selected portions of source model that satisfy the pre-conditions [38]. Users demonstrate

how model transformation should be done by editing (e.g. add, delete, comment, update) the model instance to simulate model transformation process step by step. Then, a recording and inference engine (MT-Scribe) captures user operations and generates a transformation pattern using inference. This pattern specifies the precondition of transformation and sequence of operations.

Brosch et al. introduced “Operation recorder” that is capable of recording atomic operations on models and creating composite operations based on recordings [43]. Operation recorder is capable of creating composite operations based on recordings and can accept input models defined in ECore. It then generates composite user defined refactoring by subsuming multiple set of atomic changes. Operation recorder is independent of any particular modelling language as long as it is based on ECore. Changes to the models are not recorded on runtime, instead, they are recorded when modifications are complete, i.e. initial model is checked against modified model.

MTBD generates transformation of models within the same metamodel [88]. Therefore, it cannot be used for transformation applications which require exogenous model transformation. The biggest issue with demonstration-based approaches is their high reliance on the recording system (e.g. MT-Scribe [38]). These systems are generally tool specific and integrating them with other transformation approaches (than the one they were designed for) might not be always possible. Moreover, using MTBD in exogenous model transformation is challenging since finding both source and target models that can be monitored by recording agents does not seem feasible. Therefore, most of these approaches address endogenous transformation tasks.

2.5 Visualisations and transformation

Textual specification of model transformation scripts, although very efficient for software engineers, introduces pragmatic barriers for general users [80], [87], [89]. As a result some approaches have tried using visualisations. These approaches can be grouped into three categories. First, approaches that visualise abstractions and schemas and leverage user interactions with the elements of these visualisations for generating

transformation scripts. Second, approaches that provided concrete visualisations of source and target models. And finally approaches that tried to address transformation using concrete syntax in conjunction with abstract syntax of the models. In the following subsection, we briefly review these approaches.

2.5.1 Visual intractable schemas

This category of transformation generation techniques uses predefined visualisation of schemas and user interaction with the visualisations to make transformation more user-friendly. A good example of these approaches is ALTOVA MapForce [47]. MapForce provides default tree-like visualisations for schemas of both source and target as shown by Figure 2.7. To generate mappings users drag and drop schema elements and specify mapping correspondences. From these correspondences, the transformation script is generated. In another approach, visual representations of EDI message meta-models were used as source and target. Similarly, users would generate transformation specifications by drag and dropping specification of model correspondences between these visual elements [31].

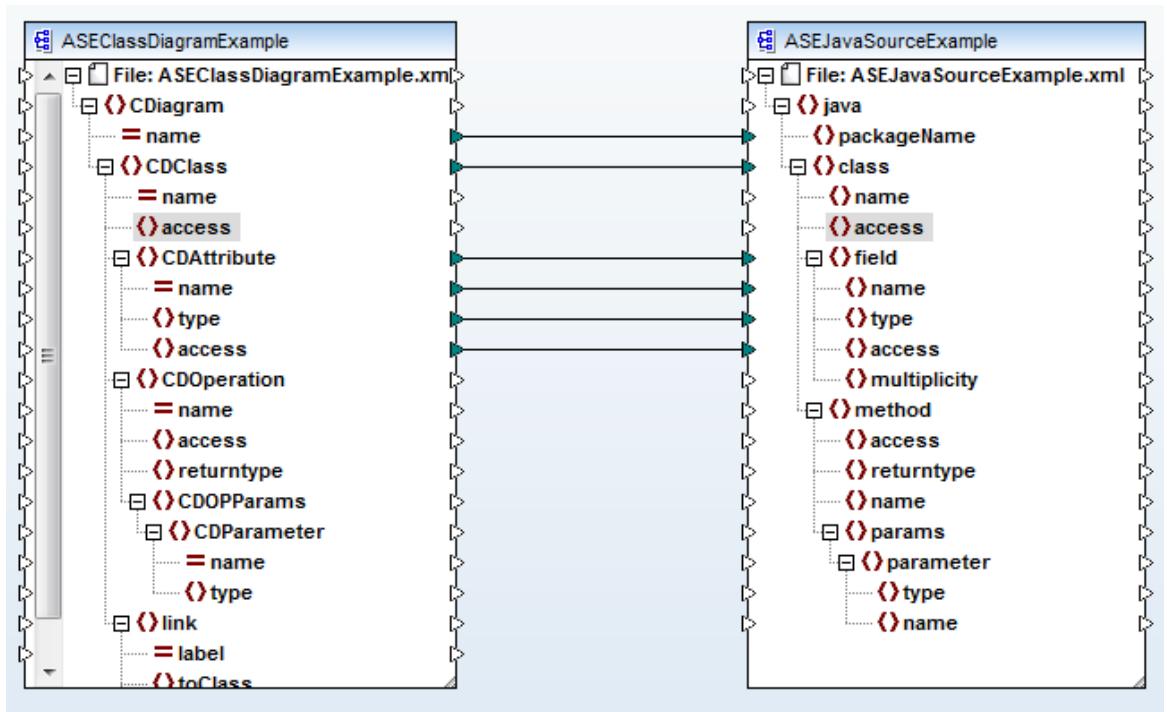


Figure 2.7 Sample mapping generation using ALTOVA MapForce.

Although user interaction and visualisation helps the transformation generation process, the fact that the artefacts being used in these approaches are still abstractions makes them targeted to more advanced users [90]. As a result, providing improved mapping specification environments require users to think in terms of meta-models and abstract meta-model correspondences, not leveraging their model domain knowledge in a human-centric way. Next section describes techniques that use concrete visualisations of models for transformation generation.

2.5.2 Using concrete visualisations

Concrete visualisations can improve understandability of mapping and transformation generation [29], [32]. A recent study on comparison of three transformation languages namely Concrete syntax-based Graph Transformation (CGT), ATL and Attributed Graph Grammar (AGG) suggested that due to use of graphical concrete syntax, CGT is more concise and requires considerably less effort from the modeller than the other two which use textual abstract syntax [91]. CGT uses a default concrete syntax similar to Business Process Modelling (BPM) and therefore the syntax is familiar for the modeller's domain knowledge.

A concrete-like representation to make models familiar to the transformation designers was taken in form-based mapper by Li et al. [32]. It uses a concrete visual metaphor based on the concept of business forms, to visualise complex business data and provided an understandable data transformation mechanism for system users (in their case, business analysts). A screen shot of this form-based mapper is shown in Figure 2.8. A closely related approach is presented by Stoeckle et al. [92], [93]. In their approach however, they try to provide multiple views for source and target models. A notation converter generator generates required transformations or transforming between different views of the provided models.

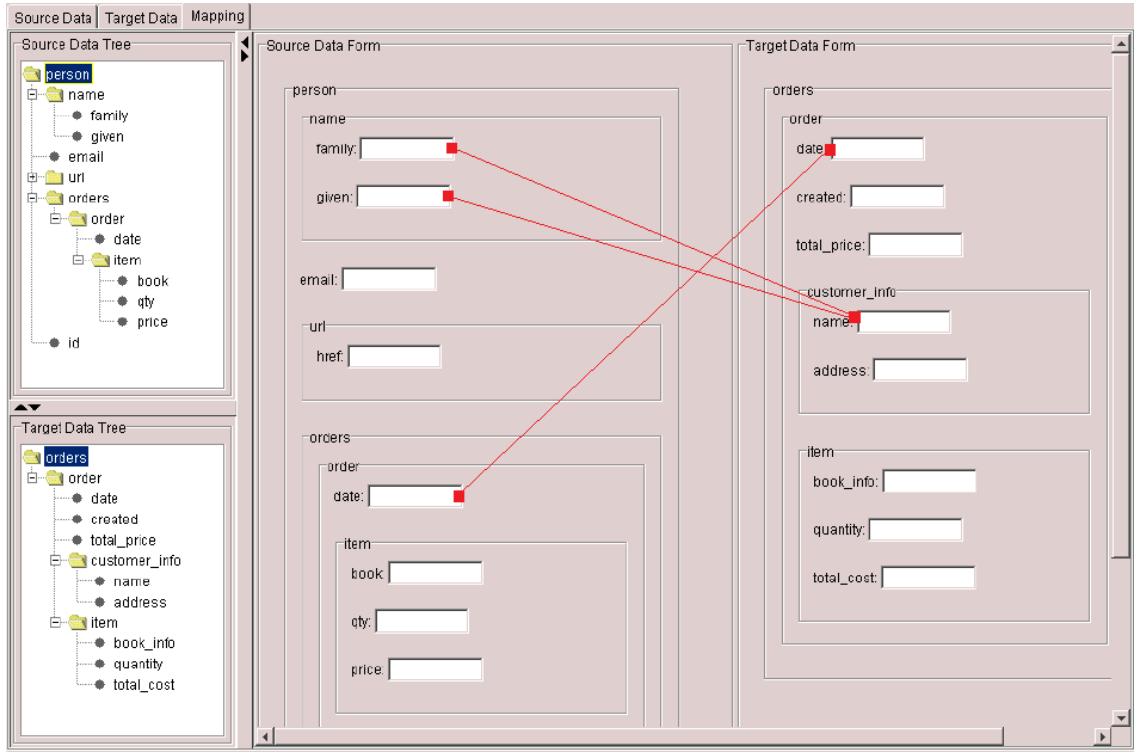


Figure 2.8 Sample mapping generation using Form-based mapper.

Schmidt proposed using a UML2-profile to define transformations as pattern in concrete syntax of UML2 [94]. This way, users have the ability to define model transformation in the same visual language as models. A UML profile is a package and contains some restrictions on the possible extensions of a reference metamodel (e.g. UML and CWM). Stereotypes with parameters can be used to define syntax and model instances of metamodels. The modeller defines UML2 model and patterns. A generator uses this pattern to create a profile for the application of the pattern and some data containing transformation and constraints for expansion of patterns. A modifier then takes transformations and constraints to generate an expanded UML2 model. Their approach, however, was only capable of handling in-place transformations of UML2 to UML2 models.

2.5.3 Using concrete syntax in conjunction with abstract

Approaches in this category involve using concrete syntax in conjunction with abstract. For instance, since in graph transformation, patterns of the left hand side (LHS) and right hand side (RHS) graphs are defined in abstract syntax and more readable concrete syntax is not used in the transformation rules, Baar and Whittle proposed separation of modelling language from pattern language [33]. Their approach was to write the graph transformation rules directly in the concrete syntax of the modelling language and extract the metamodel of the pattern language from that of the modelling language [33]. However, their method required alteration of concrete syntax to include labelling of objects and optional occurrence of attributes and links.

Visser used syntax definition formalism (SDF) to incorporate existing (meta-) programming languages with concrete syntax notation [58]. This combination was used in transformation generation using Stratego [95]. Using concrete notations for generating object programs helps to better understand the abstract syntax of the meta-language elements to be mapped. Although the use of concrete syntax makes meta-programs more readable than abstract syntax specifications, the approach requires both meta-language and object language syntaxes to be provided as an input to the platform.

2.6 Model transformation tools

This section provides a comparison of the mostly used and available transformation tools. This comparison is provided in Table 2.2. It compares several transformation tools based the application domain supported by tools, specification syntax of transformation, transformation cardinality, input artefact syntax, the way users interact with the tool, supported directionality and user support mechanism. Some of these categories have been described in this chapter before.

Among the categories of Table 2.2, *transformation cardinality* checks the type of transformation correspondences that can be specified. The options could be one to one, one to many or many to many correspondences.

User interaction category tries to capture the level of simplicity of using tools for novice transformation developers. This interaction can be performed using textual specification of correspondences which is hard for novice users. It could also be interactive specification where users click on elements, or ask tool to perform certain tasks. Some tools also allow drag and drop of visual elements to specify correspondence which is much easier for non-expert users.

User support mechanism concerns possible support mechanism to make transformation more user-friendly. This support could be in form of providing concrete visualisations of input models. For concrete visualisations, the fixed tree-based visualisation of input models is not considered as a support mechanism, as is used in most schema mapping approaches. Other type of guidance mechanisms could include providing interactive guidelines to users to finish the transformation task. This interactive guidance may be provided in form of recommendations, or providing a review of the resulting mapping target.

Majority of the transformation frameworks being discussed here are based on graph transformations. These include ATOM3 [96], VIATRA2 [97], GReAT [76], UMLx [98], and BOTL [99], [100]. These approaches are mostly integrated in Eclipse framework.

VIATRA2 integrates graph transformation and abstract state machines (ASM) to manipulate graph based models [97]. It uses a dedicated transformation language (VIATRA2 Transformation Language or VTL) which is composed of three sublanguages that provide support for multilevel meta-modelling, pattern and rule-based model transformations, and template-based code generation. It uses Visual and Precise Modelling (VPM) metamodels which provide a visual concrete syntax to represent metamodels but the framework does not allow arbitrary concrete syntaxes.

Table 2.2 Comparison of model transformation tools. + indicates support, (+) shows partial support and – shows no support.

Tool comparison	ATOM3 [96]	VIATRA2 [97]	ALTOVA [47]	CLIO [101]	GReAT [76]	ATL [68]	UMLx [98]	BOTL [100]	Form-based [32]
Application domain									
Model to model	+	+	+	+	+	+	+	+	(+)
Model to text	-	+	+	-	-	+	-	+	-
Text to text	-	-	-	-	-	+	-	-	-
Specification Syntax									
Text	+	+	-	-	+	+	-	-	-
Graph	(+)	-	-	-	+	-	+	+	-
Visualisation	-	-	(+)	(+)	-	-	-	-	(+)
Transformation Cardinality									
1-to-1	+	+	+	+	+	+	+	+	+
1-to-M	+	+	+	+	+	+	+	(+)	+
N-to-M	+	+	+	+	+	+	-	(+)	+
Input Artefact Syntax									
Abstract	+	+	+	+	+	+	+	+	-
Concrete	-	-	-	-	-	-	-	-	(+)
User interaction									
Textual	+	+	-	+	+	+	+	+	-
Interactive	-	-	+	+	-	-	-	-	+
Drag-and-drop	-	-	+	-	-	-	-	-	+
Support for directionality									
Unidirectional	+	+	+	+	+	+	+	+	+
Bidirectional	+	-	-	-	-	-	(+)	+	-
Multidirectional	-	-	-	-	-	-	-	-	-
User support mechanism									
Interactive guidance	-	-	-	+	-	-	-	-	-
Visualisation	-	-	-	-	-	-	-	-	(+)

ATOM3 provides concrete visualisation for source and target graphs and allows users to define rules by using concrete graph-based rules [96]. It uses TGG as the transformation language and is therefore capable of generating many-to-many rules. Similar to other graph based approaches, if the source and target models are not graphs, concrete syntax cannot be provided for them.

Graph Rewriting and Transformation language (GReAT) uses a combination of pattern specification language, a graph transformation language, and a control flow language

[76]. Transformation rules in GReAT can be defined using Generic Modelling Environment (GME) based graph visualisations[76].

UMLx provides a visual language for model transformation that uses standard UML class and object diagrams to define meta models, and uses object diagram to define inter-schema transformations [98]. As a result, it provides a graphical visualisation of rules that is similar to UML standards. Visual transformation rules in UMLx are transformed to XSLT and then applied on meta models.

ALTOVA MapForce is a schema mapping generation tool that provides users with tree-based visualisation of source and target schemas [47]. It allows definition of correspondences using drag and drop of elements from tree visualisations. Functions can be used to generate more complex mappings and unidirectional mapping is generated in XSLT, XQuery, Java, C# and C++.

Form-based Mapper proposed by Li et al. is a data mapping targeted for specific users, i.e. business analysts [32]. Therefore, it is limited to a certain application domain of transforming business data in terms of forms. Its visualisation mechanism also is designed to consider business forms as source and target. Although form-based mapper allows using interactive drag and drop metaphor on concrete visualisations, since the visualisation is targeted to specific application domain and users, it has been shown to partially support concrete input artefact syntax in Table 2.2.

Basic Object-oriented Transformation Language (BOTL) is a relational transformation language and system that provides the ability to use graphical description techniques and integrated algorithmic descriptions to graphically define mapping rules [99], [100]. It provides a UML-like notation for graph rewriting rules working on pairs of models or graphs. BOTL provides bidirectional transformation for bijective relations, for non-bijective relations however, the consistency preservation is not clearly defined [102].

Clio is the data transformation and mapping generator that was developed for information integration applications [101], [103]. Clio provides declarative mappings to be specified between source and target schemas. It supports XML schema and relational schema and can generate mappings in XQuery, XSLT, SQL, and SQL/XML queries [101]. Clio provides a fixed tree-based visualisation of source and target schemas and

allows users to drag and drop elements of these schemas to specify schema mapping correspondences. Clio also provides a set of correspondence matchers to create initial correspondences. These matches can then be incrementally updated [103]. Clio generates mappings in form of queries and therefore N-to-M mapping can be specified.

Although ATL is not a transformation tool and comes integrated in Atlas Model Management Architecture (AMMA), a dedicated plug-in in Eclipse framework can be used to develop transformations in ATL. Using Eclipse IDE users can specify transformations by writing ATL scripts.

There are a number of transformation tools and approaches that are used for program code transformations. Examples are Tree transformation languages (TXL) [67], Alchemist [104], ASF+SDF[105] , JTS/Jak [106], Stratego [95] and Gra2Mol [107]. Since application of these approaches is for specific domains (e.g. program transformation, textual syntax) they are not included in this comparison.

2.7 Information and Software Visualisation

There is a growing demand for approaches that incorporate visualisations in the software industry and beyond [14]. Accordingly, apart from using visualisations for model transformation, a significant contribution of this thesis is on generating concrete visualisations. This section provides a brief review on approaches for visualisations. The review is based on visualisation techniques for models and information visualisations.

2.7.1 Model visualisation

Among the early approaches of software visualisations are SHriMP and Rigi [15], [17]. Rigi was designed to provide a structural view of large software systems and effectively present the information accumulated during the development process [15]. SHriMP is a software visualisation environment specific to hierarchical datasets with non-

hierarchical relationships between the nodes and was integrated in Rigi framework [17]. However, these tools suffered from adaptability and flexibility in industrial settings [14].

More recently, a framework for Model Driven Visualization using Eclipse Modelling Framework (EMF) was introduced by Bull et al. [14]. This framework is capable of generating flexible visualisations using meta-modelling and transformations. It has been implemented within Eclipse [108]. The EMF models used in this approach allow generation of code from a model.

Cerno-II is a visualisation system capable of constructing graphical views of the execution state of object-oriented programs [109]. It has three layers: 1. Display layer which works by Display Specification Language. 2. Abstraction that consists of abstractors responsible for extracting running program data and map it to display layer and 3. Program layer that consists of objects and data structures of the program being visualised. Cerno-II uses display specification language to design new representations for displays. Each descriptor in this language is a functional expression specifying the general format of a type of display (boxes, lines, etc.). Alignment of descriptors is based on horizontal and vertical lists. Skin is a visual functional language for flexible user interface component construction using icon and connector model [21]. It can be considered as a specialisation of Cerno-II. Skin provides icon-like graphics that once connected using special connectors can form visualisations.

With emergence of Meta-tools, generating visual languages and diagram based editors became easier and more feasible. Meta-tools allow generation of visualisation environments that provide facilities for users to interact with those visualisations. An example of such meta-tools is Marama tool-suit [19]. Marama is a set of Eclipse plug-ins that support rapid specification and implementation of other software tools such as Domain Specific Visual Languages (DSVL) and modelling tools. It was used to develop other modelling environments like MaramaEML as a multi-view business process modelling environment [110], MaramaAI for multi-lingual requirements engineering [111], and MaramaMTE for performance engineering [19].

DiaGen is another example of a meta-tool for generating diagram editors [20]. DiaGen uses hyper-graphs for language definition and describes a diagram as a set of diagram components and the relationships between attachment areas of connected components of the model. Like Marama, DiaGen has been used in creating diagram-based applications, for example in an editor for graph-based languages [112].

CIDER is a tool developed for building Smart Diagramming Environments (SDE) [113]. It uses Constraint Multi-test Grammars (CMG) for specification of diagrammatic syntax. The Constraint Multi-set Grammar (CMG) formalism is a kind of attributed multi-set grammar. A CMG specification has two parts: symbol definitions and production rules. Symbols have geometric and semantic attributes. Each type is either terminal or non-terminal. Terminal symbols correspond to the primitive graphic objects in the diagram while non-terminal symbols are more complex objects built-up from these.

The approach presented by Ernst et al. provides visualisations for software application landscapes (software map) [16]. For each cluster map of the system, they have identified a semantic model and a symbol model. The semantic model contains actual values for the objects of the model and their attributes. They propose to use transformations to link the gap between semantic model (the data to be visualised) and symbolic model (visualisation). Then transformation rules like “every business application is transformed to a rectangle with text as the name using white background colour” are used [16].

2.7.2 Information Visualisation

The context of this thesis is on model visualisations. However, since information can be provided in form of a model, model visualisation approaches could also be applied for visualising information. Similarly, some approaches on information visualisation can also be used to provide alternative visualisations for models.

A survey of information visualisation tools has been provided by Pantazos et al. [114]. They have evaluated a group of industry and academic information visualisation tools for three types of users (Novice, Savvy, and Expert). Novice users have no

programming skills but have domain knowledge and basically interact with predefined visualisations. Savvy users, have some basic skills and an understanding of the domain. While expert users are users with very good programming skills who construct advanced visualisations and have no domain knowledge. Five dimensions were used to compare visualisation approaches: Who constructs the visualisation, What types of visualisations user can develop, Does it support construction of advanced visualisations, How are visualisations specified and created, and finally, Does it have a development environment. The study concluded that the current information visualisation tools do not support Savvy users in construction of advanced visualisations [114].

Relational Visualisation Notation (RVN) for generating multi-dimensional visualisations was introduced by Humphrey [4]. RVN is a graphical notation that allows users to specify visual designs without the use of programming. RVN is composed of three parts: Semantic data models, which provide facilities for describing, storing and retrieving information according to the relational model. Graphics relations that visually represent information relations, define the information and graphical models of the visualisation as well as the transformation between them. And finally design diagrams, which combine multiple information and graphic relation into a visualisation design specification. The graphics relation has a schema for graphics and a schema for information and set of bindings between them. The graphics schema is templates made of boxes, lines, graphics iteration and graphic selection. It defines the visualisation's visual structure like parameterised icons. The binding between graphics schemas and the information schema is algebraic expressions i.e. formulae which make it easier for users without programming skills to make visualisations. Design diagrams are directed, acyclic graphs that combine source relations to produce output graph relations.

UVis is a tool suit that addresses visualisation creation for non-programmer end users with advanced spread sheet knowledge and basic relational database understanding [115]. Users can compose customisable visualisations using formulas and building blocks such as box, line, and labels. uVis controls have three kinds of properties: control properties (the pre-defined properties a control supports), uVis properties (additional properties for visualizations), and user's properties (properties created by the user) [115]. Each formula represents an SQL statement and can refer to data in the database

or any of the properties. UVis studio tool is visual studio based tool which uses drag and drop approach similar to visual studio.

Where information can be structured as graphs, graph visualisation approaches can be used. Herman et al. provide a survey on approaches to create graph visualisations [116]. Their survey is specially focused on how to navigate large graphs reducing visual complexity through reorganisation of the data.

The Visual Wiki is an approach for visualising information using a combination of textual and visual representation of same body of knowledge [117]. A Visual Wiki has four components: concept, text, visualisation, and the mapping between them. Concept describes the purpose and content of visual components. Text and visualisation components use a language to represent the content of the underlying knowledge base: a visual and a natural language. Finally the mapping determines how the two representations (visual and textual) are linked together and how they influence each other [117]. Its application was shown in generation of ThinkFree, an IT knowledge management system for tertiary institutions [118]. Visual wikis can be generated using VikiBuilder which is a visual wiki meta-tool [119].

2.8 User guidance

With the scale of today's software engineering applications, users are presented with an ever-increasing load of information. One response from research and industry to the problem of information overload is Recommender Systems [120]. Recommender systems help users find information and make decisions where they lack experience or can't consider all the available data [120]. They have been previously used and tested in many e-commerce applications (examples are [121]–[123]).

This thesis approach introduces a recommender system that specifically focuses on correspondences between elements of concrete visual model representations to guide users in specifying their transformation rules. This approach builds on model matching techniques and recommender systems. The following sections provide a brief review of

approaches to model matching and recommender systems in software engineering. It is followed by a subsection on previous approaches to provide guidance to users of model transformations.

2.8.1 Model matching

Model (or metamodel) matching techniques try to find an alignment for relating two or more models. This alignment can then be used to semiautomatically generate transformations between two models. These generated transformations can then be adopted and validated by an expert as a set of transformation rules [85]. Model matching is therefore very similar to MTBE in terms of finding possible correspondences between source and target models.

Matching approaches can be categorised into three categories based on the artefacts being used as source and target to be matched and their abstraction level. These categories include schema based, instance based and hybrid approaches. Following subsections provide details on current approaches in each category.

2.8.1.1 Instance-based matching approaches

Instance based model matching approaches are the closest to MTBE. They use instances of source and target models to find and explore possible alignments. QuickMig is such an approach that uses instance-based matchers on manually created examples to generate alignments [124]. Similar to QuickMig, SmartMatcher also uses manually created instances [125]. However, the objective of both approaches is to create a matching alignment between source and target schemas, not the instances. The actual and targeted outputs of the matching algorithm in SmartMatcher are compared and the differences are propagated back to adopt the functional relationship model in form of the mapping between LHS and RHS models [125].

Kache et al. introduced an approach for reverse engineering of transformation rules in data intensive systems using data mining approaches [126]. They have classified all transformation rules into three groups, value-based correlation, aggregation, and

arithmetic transformations. Their approach requires source and target to be relational datasets with logical definition for both schemas. They also consider a primary foreign key to be present for joining source and target and to keep record of transformation rule data. The discovery is done in three phases: 1. Pre-processing: they execute a set of tests depending on the type of transformation rule group to be discovered. 2. Data mining: use mining techniques on the results of phase one. 3. Post-processing: derive transformation rule from data mining output. These steps are repeated for each of the three groups of transformation rules.

Yeh et al. used a semantic matcher to find a mapping between two knowledge representations encoded using same ontology [127]. When mapping two representations, there are mismatches that occur between elements of representations. This work tries to find instances of mismatches that encode sufficient similar content. Then these instances are generalised into transformation rules for use in semantic matching. Structural representation is mostly encoded as graph. Therefore semantic matching can be considered a graph matching problem.

2.8.1.2 Schema-based matching approaches

Schema matching (or metamodel matching) approaches are generally similar to instance based matching in terms of methods of finding correspondences. However, they intend to search metamodels and abstractions of source and target models. As a result, the found correspondences are already generalised and are in abstract level.

Some approaches to metamodel matching proposed using similarity heuristics on schema labels and types, and similarity propagation. Bossung et al. used mapping agents that look for label similarities in source and target schemas [46]. They focus on automated mapping generation of XML schema by using analysis agents which traverse both schemas and apply a set of heuristics to find correspondence between elements. Heuristics such as same name, same type were used to automate data mapping. Clio by IBM also used a similar approach [101]. Clio uses value correspondences for mapping the schemas and interprets sets of value correspondences to compute mappings for the most common schema heterogeneities known from the database field [101].

Other approaches consider structural similarities as well as label similarities. For example SAMT4MDE+ finds structural similarity between metamodel elements using weighted score of structural similarity to determine mapping specifications between metamodels [128]. Voigt and Heinze proposed structural comparison focusing on common sub-graph [84]. The approach has three stages: a planarity check, a planarization, and graph edit distance calculation. To obtain better results, set of correct mappings are provided by user as seeds which serve as starting points for similarity calculation. Sequential matching systems like Similarity Flooding propagate similarities in a graph between nodes using fixed point computation [84]. In their internal graph representation, classes, packages and attributes are each mapped onto a vertex and references are mapped to edges.

Similarity flooding is a graph matching algorithm that uses labelled directed graphs as input [129]. It first converts input models to directed labelled graphs and uses iterative fixed-point computation to find similar nodes in both graphs using string matching. It then calculates propagation of similarity of two nodes to their neighbours. By limiting suggestions (e.g. thresholding similarity scores) a set of matching results are prepared. Falleri et al. used this approach to automatically find mappings between two metamodels [83].

Dolques et al. proposed combining string similarity and schema matching to automatically retrieve the links and corresponding elements of source and target instances [42]. Their proposed architecture was to first generate the correspondence links of source and target by a matching engine, and ask an expert to check and validate the links. The matching engine would then check source and target and provide a candidate matching model. The expert would check the model for validation. Using this model they used anchor prompt approach in a two-step process designed for ontology matching to find matches [42]. Their approach works on EMF therefore all models should conform to Ecore metamodel.

A number of similar approaches are shared among ontology, schema, and metamodel matching domains [130]. For example, ModelCVS transits ECore based metamodels of ModelWare to OWL-based ontologies of OntoWare [37]. Using OntoWare and ontology matching, matches are found. Then the reverse transition will result in

metamodel matches of the input metamodels. If metamodels have common terminology, for example when matching UML to UML2, ModelCVS results in good matches due to use of ontologies.

A model management operation (ModelGen) that automatically translates a source schema expressed in one metamodel into an equivalent target schema expressed in a different metamodel, along with mapping constraints between the two schemas is introduced by Bernstein et al. [131]. Given a source model and schema, it is possible to generate the given target model's schema using special ModelGen operator [132]. It is defined in a meta-metamodel level where every metamodel conforms to. As a result, model management tools using ModelGen are claimed to be generic.

2.8.1.3 Hybrid approaches

Schema-based approaches can be improved by using model instances. For example, in Bossung et al. using example instances helps mapping agents relate schema elements more accurately [46]. SmartMatcher, uses a collection of mapping operators (predefined in the system) and tries them on schemas to find appropriate mappings [125]. A set of initial mappings should be provided to the system to narrow the search space, otherwise the operators of the mapping language have to be applied randomly which take a lot of time. The mapping model is created from alignments in INRIA alignment format [133]. It has the capability of being used to derive transformations in multiple languages based on source and target schemas [133].

2.8.2 Recommender systems

In general, there are three types of recommender systems: content-based, collaborative filtering and the hybrid recommender systems. Content-based recommenders learn the preferences of their users based on historical usage data, or available information of the items [134]. Collaborative filtering recommendations recommend items based on its similarity to items used by other users with similar profiles to the current user [135]. These systems recommend an item to a user if users with similar interests have used that

item previously. Hybrid approaches tend to combine collaborative and content-based methods [136], [137]. They leverage advantages of each approach and attempt to mitigate the limitations of each approach as well.

Software artefacts have become very large and may include varieties of source code, models, code, APIs and other artefacts. This provides a significant pressure on developers and maintainers of software to carry on dedicated tasks. Accordingly, recommender systems in software engineering have been focused on increasing productivity of developers by providing task specific recommendations. An example of these tasks is code reuse to reduce implementation efforts. CodeBroker is a development environment that promotes reuse by enabling software developers to reuse available components [138]. It analyses comments in the code and uses a combination of text similarity and signature similarity to find suitable methods among available library contexts.

In large software projects, locating specific portions of the project or code is a challenging task. Robillard introduced a recommender system that helps developers find items of interest [139]. It analyses the topology of a graph of structural dependencies of a software system and recommend set of items that might be of interest to the developer. Similarly, Hipikat helps new developers joining a development team in finding source code, email discussions or bug reports related to a specific query [140]. It provides a development environment using Eclipse IDE and records all of the artefacts produced during the development. Rascal uses a recommender agent to track usage histories of a group of developers and recommends components that are expected to be needed by individual developers [141]. The components that are believed to be most useful to current developers will appear first in the recommendation list.

Dhruv is a recommender system that provides debuggers with a list of recommended artefacts relevant to a bug report [142]. It uses a three-layer community model based on developers. The first layer considers users, and contributors. The second layer considers content e.g. code, bug reports, and forum messages, and final layer includes interactions between these. Dhruv uses a web-based environment and recommends objects according to the similarity between a bug report and other bug reports, code, and mailing lists on the web [142]. Similarly, DebugAdvisor helps debuggers search

through diverse data repositories associated with large projects to find solutions to fixing a specific bug [143].

For some applications, developers might need to find related expertise to perform a software engineering task. In such situations, Expertise Browser can be used to recommend relevant expertise [144]. It recommends experts by detecting past changes to a given code location or document and assumes previous developers that altered the document have expertise in it.

Not all recommender provide recommendations by providing data. Mylyn is a recommender system that helps users of an IDE by hiding irrelevant information provided by the IDE and hence improve programmer productivity [145]. It identifies and blurs classes in a large software project that are less relevant to the task.

2.8.3 User guidance in transformation

User guidance mechanisms have been integrated in tools for many application domains. Examples are code completion [146]–[148], diagram completion domains [149]–[151], Model completion [152]–[154], and Domain Specific Visual Languages (DSVL) [155]. Despite increasing attention to supporting users in labour intensive tasks of software engineering, we are not aware of any research, techniques or approach that is specifically generated to support users in model transformation specification. Previous research has been mostly focused on how transformations should be generated and the technologies to enabling it [29], [62], [63], [66], [156].

Siikarla et al. investigated how model transformations should be developed and what are the roles involved in the design phases [157]. They claimed that different modelling formalisms need different expertise and stakeholders need to use different notations at different levels of abstraction. As a result and due to the fact that construction of model transformation needs constant feedback, they proposed an iterative and incremental application to developing model transformations which consisted of three roles: transformation design phase expert, transformation architect and transformation programmer. The transformation design phase expert has knowledge of specific design phase and provides examples of correspondences in the source and target models. A

correspondence example captures expert's intuitive knowledge by describing structures in target models that should be resulted from the given source model. Transformation definition, which defines high level structural behaviour of transformation code, will be then generated from patterns by transformation architect and implemented by transformation programmer.

The intelligent agents approach proposed by Bossung et al. finds possible correspondences between elements of source and target model schemas [46]. Although the user can accept or reject correspondences, it was mostly targeted at automatic generation of model transformation scripts rather than guidelines for model transformation designer and was designed for schema mapping applications. ALTOVA MapForce also takes a much limited approach by providing automatic correspondence mapping of exactly similar schema labels [47]. User can ask MapForce to map exactly similar name labels of the schemas automatically to save time. Our approach to providing user guidance is built on Bossung's approach.

Among data mapping applications, Alexe et al. developed Muse as a mapping design wizard [158]. Muse uses two components, Muse-D and Muse-G, to guide data mapping designer in generating the final mapping specification for relational and nested schemas. Muse-D provides set of unambiguous mappings that can provide sample example outputs. Muse-G on the other hand, is used to guide the designer to find required mapping groupings semantics that can lead to designed output [158].

2.9 Summary

This chapter has provided an overview on some state of the art in modelling and model transformations. It briefly reviewed Model Driven Engineering (MDE) and different transformation languages and tools. In reviewing model transformation techniques, special attention was made to Model Transformation By-Example (MTBE) and Model Transformation By-Demonstration (MTBD) and transformation approaches that use some type of visualisation to allow user interaction for development of model

transformations. Approaches for providing visualisations in information and model visualisation domains were also discussed.

In-line with contributions of this thesis in providing user guidance and support for model transformation specification, state of the art in design, use and application of recommender systems has also been investigate. Due to the similarity of model matching and metamodel matching approaches to model transformation by example, an overview on automatic matching techniques was also provided and their application in providing user support was also discussed.

Chapter 3

Approach

3.1 Introduction

This research investigates the use of concrete representations of complex models to make model transformation specification and generation process more user-centric. The notion of “concrete” here refers to the notations that are generally used for defining models. These notations may include textual (e.g. source code or documents) or graphical notations (e.g. boxes and lines used in diagrams, graphics used in charts, etc.).

The term “users” refers to users that are not trained in complex transformation languages and meta-modelling or type theory, but are familiar with specific modelling languages and their concrete visual notations. For these users, the correspondences between participating source and target models and their semantics are relatively clear.

Also, by graphical notations, we are not referring to actual graphical notation of input models. Instead this approach seeks to providing users with a user friendly method of generating concrete visualisations. These visualisations can be similar or different to actual model representations and more toward visualisations that users are familiar with. Therefore, the first main research question is defined as:

RQ1: *Can concrete model visualisations be effectively generated in a visual and interactive by-example approach?*

Our approach addresses this question by letting users define or choose notations provided by our framework (defined by other users or themselves previously), and map them to input model examples. From this interaction, a model-to-visual notation mapping will be generated. These specified by example model-to-visualisation notation mappings will then be composed to generate complete and complex visualisations.

Using generated concrete representations rather than abstract, our hypothesis is that users will find it more understandable to define correspondences between source and target model elements using their concrete notation, rather than by using meta-model notations as with most current approaches. These correspondences will eventually lead to transformation rules between the underlying models. Therefore our second main research question is derived as:

RQ2: Can a model transformation be effectively generated using concrete by-example visualisations?

Nevertheless, the models being used in software engineering today may get large and this has direct effect on the complexity and scale of their visual representation. Therefore, concrete visualisation alone may not contribute enough to better comprehension of large models. This factor affects expert and novice users alike and will frame our third main research question:

RQ3: How can interactive guidance be provided to users of model transformation systems?

We incorporate a guidance system that helps users in using the transformation process. This system helps by providing recommendations on source and target representations. These recommendations indicate which elements of source and target model are likely to match. Users can then view them as guidance or choose among them for correspondence specification.

The approach presented in this thesis will not be acceptable, unless an appropriate tool support is provided and users can use the tool to evaluate the approach. Such tool support should also integrate different aspect of the approach in a useable and scalable manner. As a result, research question four is defined to address this:

RQ4: *Can our approach be implemented in a usable, scalable and user friendly tool?*

The contributions of this thesis are implemented and validated in a proof of concept prototype CONcrete Visual assistEd Transformation framework, or CONVErT for short [80], [89]. CONVErT provides a proof of concept implementation of each of the research contributions, and plays an important role in validation of our approach.

To address these separate and yet complementary questions, we have devised a collection of research questions and motivating scenarios. Following sections are dedicated to description of these in more details.

3.2 Approach

The high level scenario of our approach is depicted in figure 3.1. It describes the artefacts that take part in transformation specification as source and target interchangeably. The arrows show transformation direction. Transformations from examples to visualisation have been depicted by arrows of the same colour and shading to indicate that the process of transforming example data to visualisation for both source and target is similar. As indicated, all transformations here are bidirectional, i.e. once the forward transformation is generated, its reverse is generated automatically (where possible, or an alternative is specified by the user if not).

The key idea is that source and target examples are first transformed to specialized visual representations¹. Visual notations in the visualisation are capable of being used directly for correspondence specification, i.e. they can be dragged and dropped on each other to specify correspondence links. To generate transformation between visualisations, many such correspondences are required to be specified.

¹ Here after we may use the terms “visualisation process” and “transforming to visual notations” interchangeably as they refer to the same procedures in our approach.

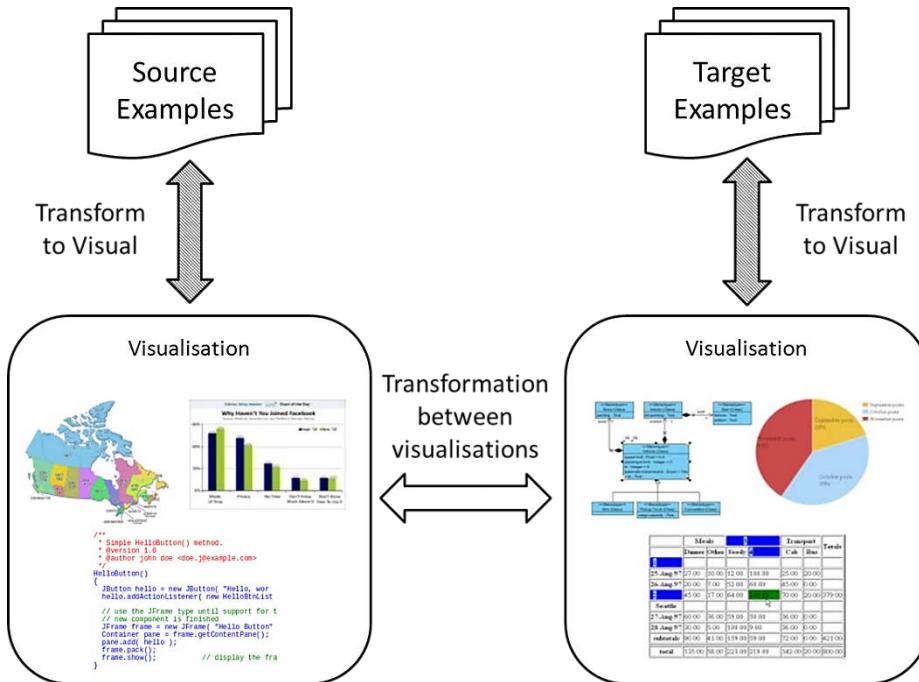


Figure 3.1 High level description of our approach and transformation flow.

The visualisation process and transformation between visual notations are separate tasks and can be distributed among users. For example, a user may be responsible for specifying visualisations for provided model data in form of charts and tables. Other users use these charts and tables to generate transformation specifications instead of using the underlying provided data. Since the system is responsible for the round tripping between these two processes, they can always integrate and reuse their transformation results to produce a round trip transformation. The following sections review our visualisation and transformation specification approach in more detail.

3.2.1 Visualisation

Using complex information in a visual format is more acceptable and effective for human beings because visual representations use the capabilities of our powerful human visual system [159]. Visualisations and diagrams can reduce the amount of search needed for finding elements and information [160]. Therefore, Our first research question and focus point is on how we can employ this capability to our advantage to improve user-friendliness of model transformation specification.

To use visual representations for model transformation (or any other field that requires visual representations), a mechanism for generating and rendering visualisations needs to be available. This mechanism should define how the values in data are mapped to corresponding visual representations, and how the visual notation is to be represented to the user. Therefore, objective of the visualisation will be to provide this mechanism, and further for our approach, to use the mechanism for model transformation correspondence specification. By developing the visualisation approach presented in this thesis research, we will also contribute to “*Creating complex visualisations from arbitrary data (here input models).*”

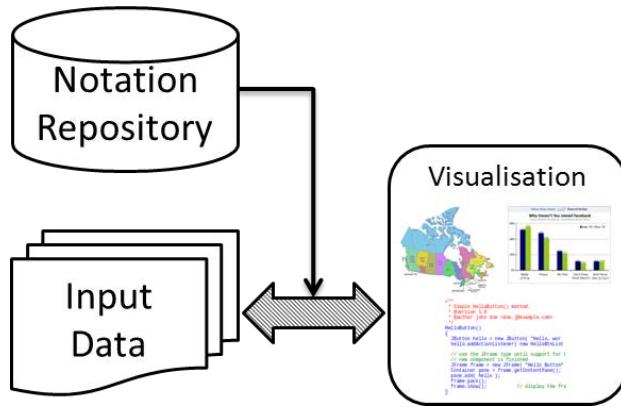


Figure 3.2 Using notation repository for generating visualisations.

The models and hence their visual notations come from variety of domains and different data. Therefore, as is depicted by figure 3.2, a first impression is to provide a notation repository so that the users can compose complex visualisations from existing basic notational elements. A problem with such architecture is that to add a new visual notation and visualisation, this approach would require low level programming for defining and rendering visual notational shapes. This is not a desirable procedure. A more effective procedure would allow arbitrary notations to be defined and used as visualisation. Therefore, following questions are devised:

RQ1.1: How can a variety of visual notations be defined and integrated to the system?

RQ1.2: How can we define correspondence links between data and visual representation?

Following section is dedicated to how we approach these questions.

3.2.1.1 Visual Notations

A useful visualisation mechanism should allow users to define a variety of shapes, colours, textures and graphics as visual elements. Being inspired by the Model View Controller (MVC) approach [2], our decision was to separate the notational visual representations (View), from representative data (Visual Model) and provide a Controller for updating model values in the view. However, unlike traditional MVC [2], the Controller here is not a collection of interfaces between Models and Views. Instead, the controller is a transformation that transforms the model data to the view and is generated using provided semantic links. This way, each visual notation would have a data portion and a rendered visualisation (see figure 3.3). The visualisation is depiction of the model data to visual elements; therefore, it only provides one direction (forward transformation).



Figure 3.3 High level structure of visual notations (left) and a concrete example describing a chart notation (right).

The modified MVC approach allows users to create or provide model visualisations by using already available view examples and specifying links between the data and those visualisations. As a result the visualisation mechanism is flexible and variety of visual

notations can be generated in the system and users will have the freedom to choose (or design) desired notations.

3.2.1.2 Mapping input data to visual notations

Given that visual notations are generated and available, to visualise input data, the provided data should be mapped to the notation data. Once the data is transferred to the notation's model data, it can be represented by the notation's view using the controller transformation. As a result, next research question would be defined as:

RQ1.3: How can the mapping between model elements and visual notations be specified?

To answer this question, our approach follows a drag and drop procedure for specifying mappings between input data and notations. Elements of input data are provided to users with a default representation, and users drag and drop elements on elements of the visual notation's model. To perform this mapping, our approach automatically generates transformation rules for transforming (portions of) input data to the model of the desired notations. Figure 3.4 shows an example of such interaction where the data is being mapped to visual notation.



Figure 3.4 Mapping input data to visual notation's model data.

In Figure 3.4 the input data is being mapped to a visual notation's model data. This will result in a model-to-visual notation transformation rule. This transformation rule is embedded in the notation, making it a customised notation for that portion of input data. Unlimited number of customised notations can be generated this way.

3.2.1.3 Visualisation composition

Next step to have a complete visualisation is to be able to generate visualisations using defined customised notations. These notations need to be composed to create a complete visualisation. This defines our next research question:

RQ1.4: Can the defined customised visual notations be composed and linked together to generate more complex and complete visualisations?

To answer this question, our approach uses a dedicated notation composition procedure. Each of these customised notations represents a model-to-visualisation transformation rule. Composition of these transformation rules will result in a transformation script that can transform a bigger portion of input data to a more complete visualisation, hence a model-to-visualisation transformation script.

To perform this composition, the design of our notations allows specifying place holders in the notation. These place holders specify where other notations might be added. Figure 3.5 shows a composition of a bar chart using the chart area and a bar notation. The bar notation in this figure has been linked to bars element placeholder of the chart notation. This composition results in generation of a transformation script that transforms the input model data to bar chart visualisation composed of chart area and bars.

Multiple customised notations can be linked to a placeholder. This is to allow alternative visual notations to be embedded inside a notation. This feature allows specification of multiple alternative visualisations based on certain conditions. For example in a bar chart, the bars could be specified by rectangle shapes or cylinders depending on a shape element in host bar chart's model. If no such condition is

provided and duplicate notations are linked to a placeholder, the system automatically picks the first notation.

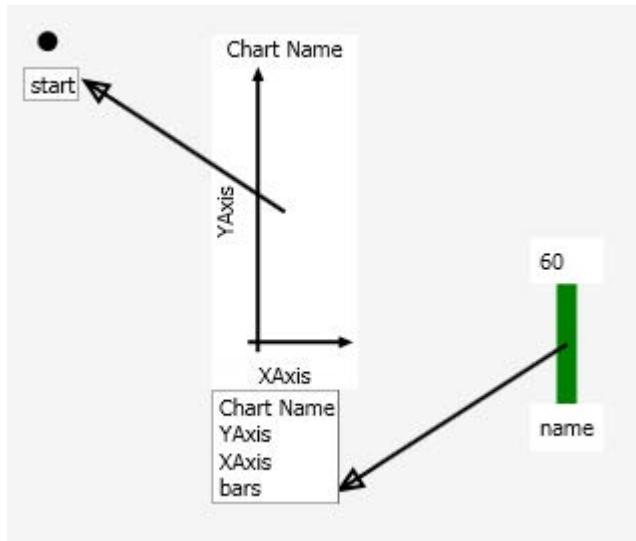


Figure 3.5 Composition of a bar chart visualisation.

3.2.1.4 Visual aid for notation composition

Composition of customised notations can be a complex task. When the number of customised notations increases and visualisation becomes more complex, it might become hard to follow the composition procedure. Given that the composition process will provide the scheduling of model-to-visual transformation rule inside notations, it is important that the notation composition is performed correctly. As a result following question is raised:

RQ1.5: *In what form should guidance be provided to users on composition of notations?*

To provide support for composition and hence being able to schedule transformation rule sequencing, the rendering mechanism is designed in a way that it is capable of rendering partial visualisations. As a result, when visual notations are being composed, even though the resulting composition (and hence transformation) may not be complete, the system is capable of rendering the partial result. Therefore, users can review the

result of so far composed notations on the spot and perform corrections as they see fit. Figure 3.6 shows an example of this visual aid.

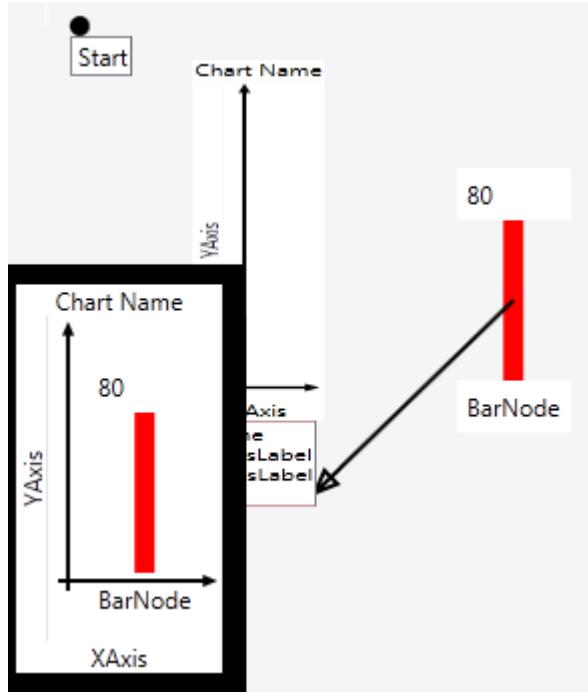


Figure 3.6 Visual aid for notation composition.

3.2.2 Correspondence specification

The intention of defining desired visualisations was to improve the comprehension of input models, and hence defining correspondences between source and target models, in a model transformation specification. With visualisations available, next step would be to specify correspondences between source and target visualisations as the starting point of transformation generation. Consequently we asked:

RQ2.1: *Can we perform correspondence specification (and hence transformation specification) on actual visual notation of input models?*

A correspondence is a link between an element in the source model and an element in the target. Such a link may simply imply that the value of the source element should be copied to the target element (and vice-versa in a bidirectional case). Once visualisations of source and target are available, correspondence links between two elements (notations) in a visualisation are specified by drag and dropping one element onto the

other. This way a link can be specified between dragging element (source notation) and the element it is being dropped on (target notation). Examples of such an interaction can be found in Figure 3.7.

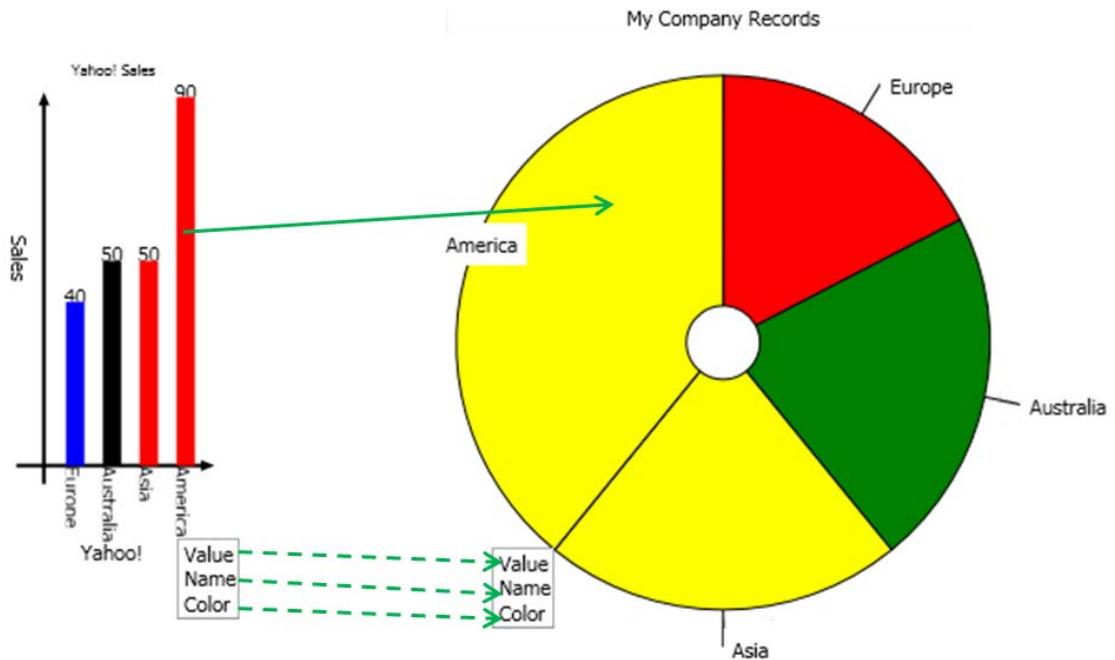


Figure 3.7 Correspondence specification examples. Arrows depict drag and drop directions.

To make arbitrary notations capable of being used in correspondence specifications, each notation needs to be aware of the interaction logic (drag and drop events) and transformation logic (bits and pieces required for generation of transformation code like forward and reverse transformation templates and abstraction). These elements have been included into our visual notation architecture along with rendering logic. Each notation carries the elements required in our drag and drop approach as well as rendering mechanism and transformation templates. Figure 3.8 illustrates the architecture of each visual element. This architecture allows visual elements to be capable of being dragged and dropped onto other visual elements. Once a visual element is dropped onto another element, the forward and reverse transformation templates of each will be defined according to the interaction direction.

Example 3.1 Dragging element E1 on element E2 will result in the abstraction of E1 to be used as initial Reverse transformation template of E2 and abstraction of the E2 as Forward transformation template of E1. These abstractions include the structure and type of the input data that each element carries and is automatically reverse engineered. The other two templates (Reverse template of E1 and Forward template of E2) are assigned accordingly by assuming the interaction was performed in the other direction. This way both forward and reverse transformations follow the same routine.

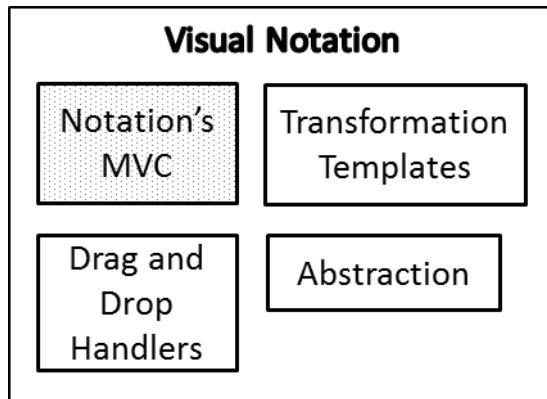


Figure 3.8 Architecture of a visual notation.

A complete transformation rule will be generated using visual elements once all their attributes are assigned to their corresponding elements on the target side (and hence completing the forward and reverse templates). If required, functions and conditions can be used to generate more complex rules.

Example 3.2 To transform a bar in a bar chart to a pie piece in a pie chart, its value, colour and label could be dragged and dropped on the corresponding attributes of the pie piece similar to example of Figure 3.7. Once correspondences are defined, a transformation rule is formed to transform a bar to the pie piece (and reverse when possible).

3.2.2.1 Transformation rule representation

Transformation rules are an integral part of any transformation system. A complete transformation specification usually consists of a combination of multiple transformation rules. The division of transformation specification in rules improves reusability and helps better debugging of the specification. On the other hand, once dealing with large models, many such rules will be defined which can affect understandability. Given that first class artefacts of our approach are visual notations, a textual representation of a transformation rule would be out of place and not suitable. Therefore, transformation rules should be represented by visual notations too. Thus a representation mechanism should be available for transformation rules as well. Thus we define research question 2.2 and 2.3:

RQ2.2: Can a transformation rule be represented visually?

RQ2.3: How to create a visualisation for transformation rules?

Our approach for answering these research questions uses the visual notations of source and target visualisation to present transformation rules. Each visual element embeds the forward transformation template that creates another visual element (the target). By applying this template on the embedded data, a visual element of the target model with its notation is created. As a result it will be possible to access target notation from each element. Putting each visual element and its target notation together will provide a schematic view of the transformation rule, i.e. users can see each transformation rule by the source and the target notation that can be generated as a result of applying the rule and hence, a representation for transformation rules. Examples of such rule representations are provided in figure 3.9.

Although visual notations increase the cognitive and comprehension of models compared to textual representations, when dealing with large scale models and specially for novice users, it is still hard to perform complex tasks like transformation specification and finding correspondences. Next section describes our approach to providing support to users in form of recommendations, so that they can better identify likely correspondences.



Figure 3.9 Examples of transformation rule representations. A) UML class diagram to Java class notation transformation rule. B) Bar to pie piece transformation rule.

3.2.3 Correspondence Recommender

Correspondence specification between source and target for large models can easily get complex and time-consuming, adversely affecting transformation specification procedure. This affects both novice and expert users alike. Therefore, users should be supported by guidance mechanisms to help them define model transformation between visual notations and hence our third main and subsequent research questions.

RQ3: *How can interactive guidance be provided to users of model transformation systems?*

RQ3.1: *In what form should guidance be provided to users of model transformation?*

In this context, our approach delivers this support by providing hints on possible and likely correspondences between source and target that can eventually create transformation rules. To achieve this, an automated recommender system (“Suggester”) is designed which analyses user interaction and input examples for recommending possible correspondences between models and their sub-structures. The recommendations provided by the Suggester mechanism can be used directly to develop transformation rules or used as guidelines to create final transformation artefact. Note that depending on where the Suggester is being used (for visualisation or transformation between visual notations) source and target examples could be input data or visualisation data.

The main task of a recommender system in general is to provide guidance to users for choosing among multiple options. However, the accuracy of this guidance depends on type of user (e.g. expert or novice), type of application, and their intended purpose

among others. Although this guidance hints do not have to be accurate all the time, higher accuracies will result in better trust in the recommender system by users. Never the less, producing recommendations that are already known to users will gradually cause users to ignore it over time [161]. The trade-off between different dimensions of the recommender system should be considered [162]. As a result research question 3.2 is designed to target design of the recommender system.

RQ3.2: What is the best technique to generate acceptable recommendations?

To design a recommender system for our correspondence recommendations that produces accurate recommendations, and yet satisfies other dimensions to some extent, our approach follows ensemble learning techniques [163]. It combines similarity scores provided by a collection of recommenders to produce final list of recommendations. Each of these recommenders uses a predefined similarity heuristic and analyses source and target model examples and ranks element pairs by similarity scores. The combination of these scores creates the final list of recommendation.

If recommendations are provided to users as a fixed list with no interaction, they will not provide a helpful guidance when dealing with large models. As a result it should be possible for users to interact with these recommendations. Considering this, research questions 3.3 and 3.4 are raised:

RQ3.3: How can users best interact with recommendations?

RQ3.4: How can user response be used and integrated into the guidance mechanism?

The Suggester mechanism provides users with initial and seeding recommendations so that they can start transformation generation with higher confidence. Our approach provides recommendation in interactable list where users can accept or reject recommendations (see Figure 3.10). By selecting a recommendation, the underlying recommended correspondence is applied. Also, selecting a recommendation indicates user's interest in the recommendation. A feedback mechanism analyses user interaction and promotes the recommender system. If user rejects a recommendation, the system penalises the recommender system and therefore this interaction helps the learning mechanism inside Suggester system to improve its recommendation capability.

- ✓ ✗ Map `ChartData` To `NPCData`
- ✓ ✗ Map `ChartData/Name` To `NPCData/ChartName`
- ✓ ✗ Map `ChartData/Bars` To `NPCData/Pieces`
- ✓ ✗ Map `ChartData/Bars/BarNode` To `NPCData/Pieces/NPPData`
- ✓ ✗ Map `ChartData/Bars/BarNode/Value` To `NPCData/Pieces/NPPData/Value`
- ✓ ✗ Map `ChartData/Bars/BarNode/Name` To `NPCData/Pieces/NPPData/Name`

Figure 3.10 A sample of recommendation list recommending correspondences between a bar chart and a pie chart.

3.3 Scope

This thesis research tries to address concrete based model visualisation and transformation in a generic way. It is based on the assumption that specifying model visualisations and model transformations based on concrete instead of abstract representations is easier for end-users, in particular those without a corresponding education. This hypothesis has been verified by previous literature (e.g. [29], [32], [91]) and therefore is not part of this thesis. As a result, the focus of this thesis is more on providing an approach to realise concrete and example based visualisation and transformation.

Due to time constrains implementation of the approach has been focused on model examples in XML and CSV. We believe these categories cover a broad spectrum of model examples and are good samples to prove applicability of this approach. Also, for transformations, XSLT has been chosen as transformation language of choice. However, the templates and the transformation rule structures are generic and adaptable to other transformation languages. Therefore, the concepts and methods can be used by other transformation languages as well.

The transformation specification in this approach can generate bidirectional transformation (both forward and reverse) for bijective correspondences. However, there are situations where generation of reverse direction is not straight forward. For example, when adding two values of the source model to produce a value in target model, the reverse direction cannot be automatically defined since information of the original values is lost during forward transformation. We call these transformation

Lossy transformations and although we have designed approaches to perform such transformations, they are not considered in this thesis.

Similarly, when applying transformation rules using pre conditions, the reverse transformation should consider the forward transformation condition. For example if a colour is to be chosen based on a value in the source model, the reverse transformation should check for applicability of the condition. This is usually done by model checking resulted reversed source model against the condition. This model checking is not considered in this thesis. Nevertheless, some consideration has been made and extension points are available to provide for possible future research inclusion.

3.4 Evaluation

The evaluation strategy of this thesis is based presentation of case study examples, comparison study, a quantitative study and a user experiment. For each contribution of our approach, multiple case study examples are provided. These examples are provided at the end of the chapters that describe contributions.

Our comparison study (provided in Chapter 8) compares our approach against a state of the art transformation tool and approach. It provides examples of how users interact with both tools and what are the procedures involved for performing transformation tasks.

The quantitative analysis part of our evaluation is provided in Chapter 8 and examines the correctness of the recommendations produced by the proposed recommender system of the Suggester mechanism using Precision, Recall and F-Measure metrics. It also provides a study of quality of the automatically generated transformation code by our approach against transformation codes produced by a human expert and that automatically generated by a state of the art mapping tool. It uses quality attributes and metrics from model transformation literature and introduced by Van Amstel et al. [6].

Finally, our evaluation is concluded with a user study of our approach and toolset. This user study is designed to capture user experiences with the toolset for generating visualisations and transformations. This user study is also provided in Chapter 8.

3.5 Summary

This chapter described research questions being addressed in this thesis and brief description of the approach taken by this thesis to address them. We seek to address two main research questions which are 1. Visualisation of arbitrary model data into more understandable concrete representations and use them for model transformation specification, and 2. Provide guidance mechanism for defining correspondences in a model transformation specification task. The subsequent questions arising from these main research contributions were introduced in this chapter and a brief description of how we approach them was presented. The list of research questions addressed by this thesis is provided as follows:

Research Questions:

1. Can concrete model visualisations be effectively generated in a visual and interactive by-example approach?
 - 1.1. How can a variety of visual notations be defined and integrated to the system?
 - 1.2. How can we define correspondence links between data and visual representation?
 - 1.3. How can the mapping between model elements and visual notations be specified?
 - 1.4. Can the defined customised visual notations be composed and linked together to generate more complex and complete visualisations?
 - 1.5. In what form should guidance be provided to users on composition of notations?
2. Can a model transformation be effectively generated using concrete by-example visualisations?
 - 2.1. Can we perform correspondence specification (and hence transformation specification) on actual visual notation of input models?
 - 2.2. Can a transformation rule be represented visually?
 - 2.3. How to create a visualisation for transformation rules?
3. How can interactive guidance be provided to users of model transformation systems?
 - 3.1. In what form should guidance be provided to users of model transformation?
 - 3.2. What is the best technique to generate acceptable recommendations?
 - 3.3. How can users best interact with recommendations?
 - 3.4. How can user response be used and integrated into the guidance mechanism?
4. Can our approach be implemented in a usable, scalable and user friendly tool?

Chapter 4

Visualisation

4.1 Introduction

The approach presented in this thesis for model transformation uses concrete, example model visualisations for specification of complex model transformations. Source and target examples are first transformed to visual concrete notations. Then the defined notations in these visualisations are used directly for interactive correspondence specification and then for model transformation script generation. To realise this approach, a mechanism and procedure is required to generate these model visualisations. The notations used in the model visualisations need to enable user interaction in the form of drag and drop to specify notation correspondences. These correspondences are then used to generate underlying model transformation rules.

This chapter describes how interaction-capable notations and visualisations are created using this approach. It provides examples from a variety of fields to demonstrate the applicability of this visualisation approach. The following sections describe the steps required to create visualisations and how elements of each step are produced. In summary, this chapter describes our approach for addressing following research questions:

1. *Can concrete model visualisations be effectively generated in a visual and interactive by-example approach?*
 - 1.1. How can a variety of visual notations be defined and integrated to the system?

- 1.2. How can we define correspondence links between data and visual representation?
- 1.3. How can the mapping between model elements and visual notations be specified?
- 1.4. Can the defined customised visual notations be composed and linked together to generate more complex and complete visualisations?
- 1.5. In what form should guidance be provided to users on composition of notations?

4.2 Visualisation Procedure

The brief procedure and steps to create a visualisation are shown in Figure 4.1. Notation generation (step 1) involves creating a notation from provided visual contents (View) and mapping it to a defined Model. The combination of the two results in a notation which will be saved in a repository for reuse. The notations provided in the repository are then mapped to elements of the input models (to be visualised) to create customised notations for that model (step 2). Once all required customised notations are generated, they can be composed to create complete visualisations (step 3).

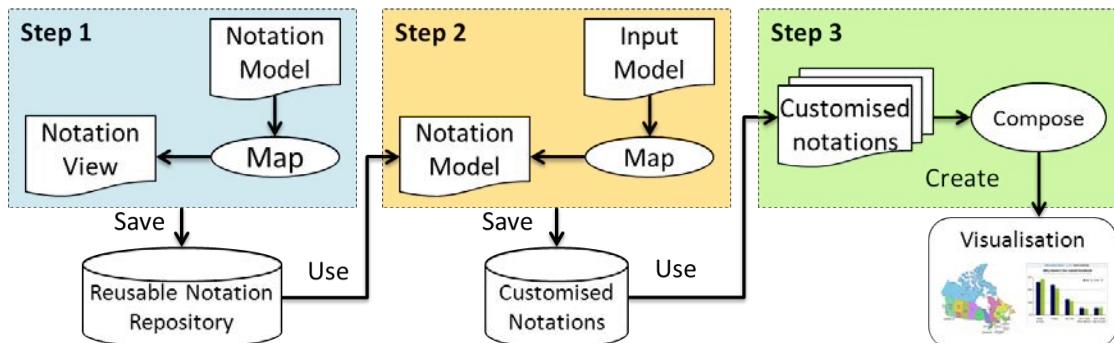


Figure 4.1 Visualisation procedure.

As Figure 4.1 suggests, visual notations are the centre-part of our visualisation approach. They represent the front line of our approach in using concrete visualisation and capable user interaction. The next section describes notation structure and provides examples of how they can be created and used for model visualisation. It specially describes our approach in answering research questions 1.1 and 1.2.

4.2.1 Visual Notations

Let's start this section by definition of visual notation.

Definition 4.1 A *visual notation* in this approach provides a visual illustration (of a portion of whole) of model data and a means for user interaction.

Since notations are the primary components for creating transformation rules, they need to include certain transformation related artefacts. These artefacts help realisation of correspondences and transformation rule templates.

The architecture of notations is inspired by the Model View Controller (MVC) architecture [2]. Three-way division of an application in MVC entails separating (1) the parts that represent the model of the underlying application domain, (2) the way the model is presented to the user, and (3) the way the user interacts with it [2], as can be seen in Figure 4.2. Therefore, in MVC programming, objects of different classes take over the operations related to the application domain (the Model), the display of the application's state (the view), and the user interaction with the model and the view (the controller).

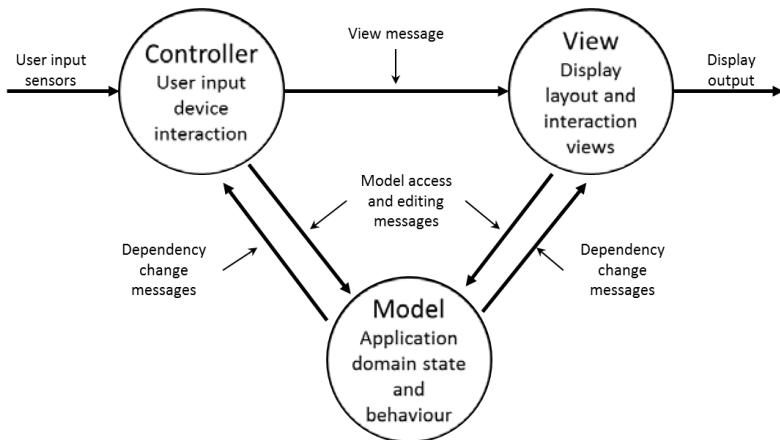


Figure 4.2 Model View Controller (MVC) set up from [2].

In our adaptation of MVC for visualisations however, a visual notation is described as combination of a View (the visual representation), a Model (domain data represented by the View) and a Controller which controls the links between the Model and the View (as can be seen in Figure 4.3). In such a configuration, any updates to the Model will be

applied to the View by the controller, which itself is created by user-provided annotations in the View. These annotations define correspondence relations between Model and View.

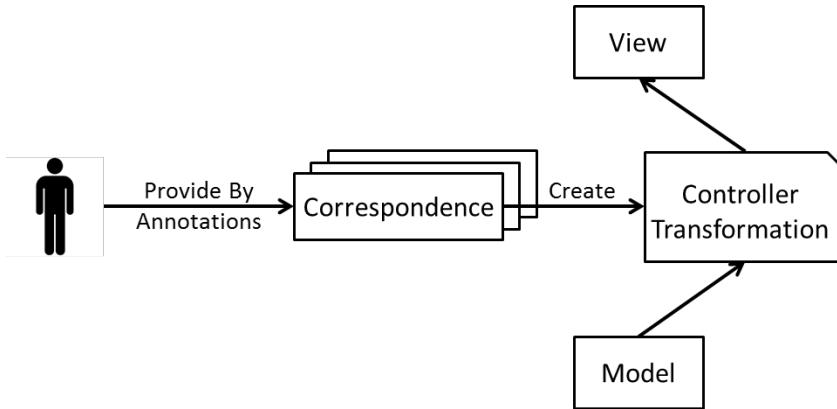


Figure 4.3 Adaptation of Model View Controller for visual notation design.

Example 4.1 In a bar chart visualisation of Figure 4.4, bars represent values of a certain category by visually depicting that view using their height. Since multiple bars may exist in a bar chart for a category, each bar is also accompanied by a name for the value it represents. Therefore a bar's model should specify the value and the name of the bar.



Figure 4.4 Sample bar chart visualisation.

The Controller of this MVC configuration is a transformation which inserts the domain values defined in the Model to the View and as a result updates the View with new

values. To generate this Controller transformation, annotations should be provided in the View representing correspondences between the View and the Model. These annotations include one-to-one correspondence relationships, and one-to-many and iterative correspondences between model and view. To specify these correspondences, a simple annotation scripting is used in our approach consisting of `linkto="<element>"` for specifying one-to-one correspondences, and `callfor="<element>"` for specifying one-to-many correspondences.

Notations can host other notations (e.g. a bar chart will host multiple bars). To clearly define the position which notations are to be placed in a host notation, a placeholder should be provided in the host notation's Model. These place holders are specified by iterative correspondences. To define the Controller for these two notations, provided View's code should be annotated. Annotated Views are read by a transformation code generator and a Controller transformation script is generated for each view. In this transformation script `linkto` annotations are translated to value fetch scripts and `callfor` annotations are translated to call for templates. As a result, when the Controller transformation script is executed, it will fetch and copy the values provided to its Model to their corresponding visual elements in the View. It will also register a declarative call for templates to be applied on the data provided to the placeholder elements of notation's Model.

Example 4.2 Model, View and Controller of a bar chart and a bar are shown in Figures 4.5 and 4.6. Each bar chart represented here has three Labels describing the chart name, Y axis and X axis which should be defined by its Model. It also creates two axis arrows representing the chart area. On the other hand, each bar has a Name and a Value. A representative of these two notation Models are marked by "b" in both Figures. A placeholder should be provided in bar chart's Model to specify where bar notations being inserted should copy their Model; therefore, a "bars" element is provided in bar chart's Model (see Figure 4.5b). In Figure 4.6 the values provided by a bar's Model ("Name" and "Value") represent 1-to-1 mapping correspondence with elements on the View therefore they will be provided by one-to-one annotations (e.g. `linkto="Name"`). Same is true for the bar chart, the labels representing axis and chart names represent 1-

to-1 relations. The place holder in the Model of the bar chart is in one-to-many relationship as multiple bars may be present in a bar chart as a result, it will be annotated by a one-to-many correspondence annotation i.e. `callfor=“Bars”`.

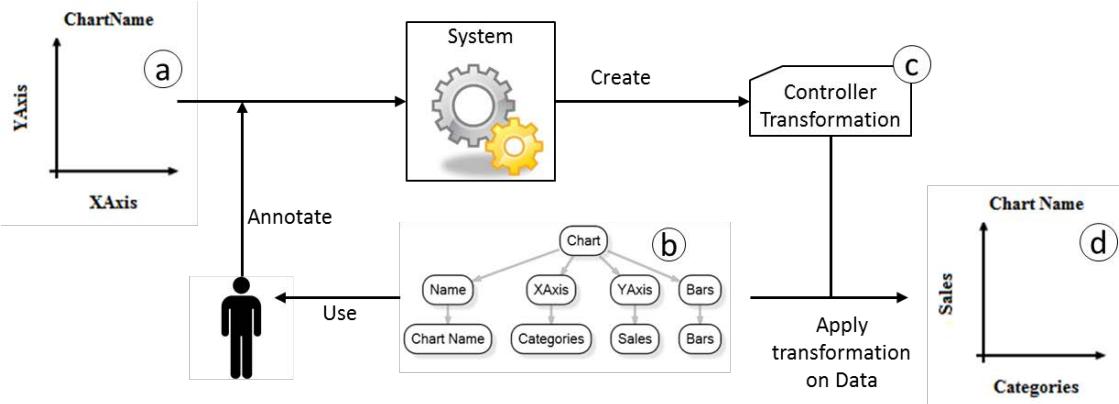


Figure 4.5 a) View, b) Model, c) Controller and d) Final bar chart visual notation.

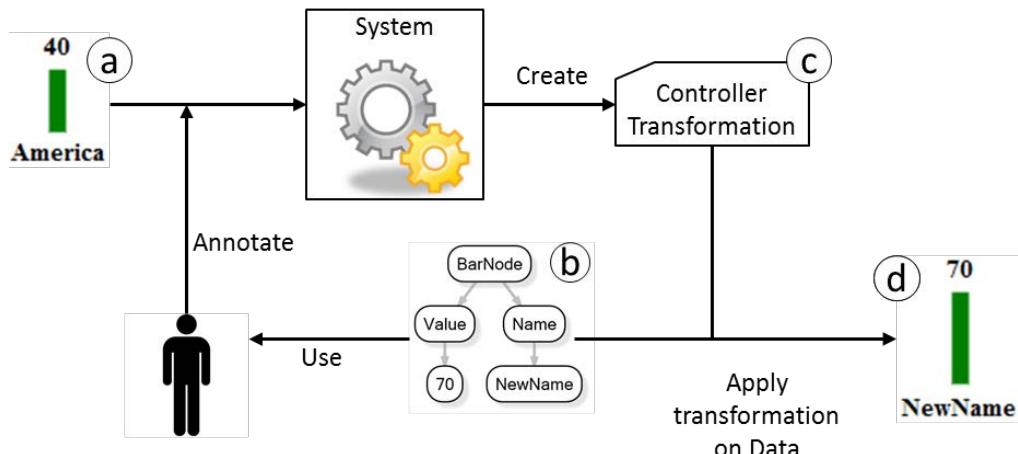


Figure 4.6 a) View, b) Model, c) Controller, and d) final bar notation.

4.2.1.1 Interaction Logic

To generate interaction (drag and drop) for notations, a standard interaction mechanism is provided for all notations as in Figure 4.7. It includes handlers for drag and drop, data structures for abstractions and transformation templates. Once notation's MVC is created, the system automatically wraps each notation's MVC in this structure. As a result, every notation in this visualisation is interaction capable. This wrapping also provides capabilities to see internal elements of notation's Model by right clicking on

the notation. If user right clicks on a notation, these elements are represented in a pop-up window.

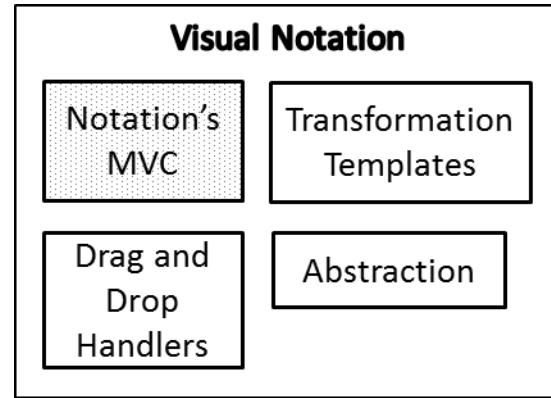


Figure 4.7 Notation's MVC wrapped in interaction logic.

Once a visual notation is created, it will be saved in notation repository for reuse. Figure 4.8 depicts a brief architecture of system implementation including the notation repository. The renderer mechanism of the system uses controller transformation in each notation to create visualisation rendering. It uses a visitor pattern to check the parts to be visualised and find matching controller transformations. It then creates a complete transformation script to transform the input to be visualised to renderable visualisation.

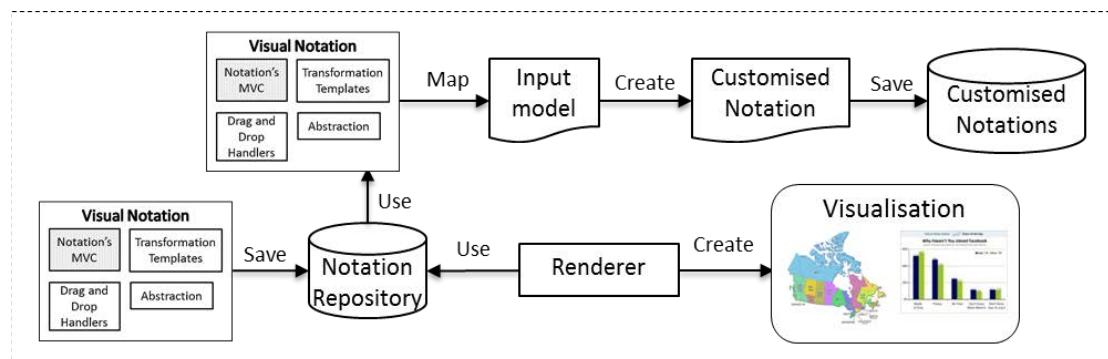


Figure 4.8 Brief architecture of system implementation. Notations will be saved in the notation repository.

Each notation in the repository can be used for mapping variety of input models to the visualisation represented by the notation. Next section elaborates more on this mapping from input model examples to visual notations and accordingly provides our response to research question 1.3.

4.2.2 Mapping input data to visual notations

Step two of our model visualisation approach involves mapping example model data to visual notations. This step creates transformation rules for transforming specific parts of input models to the notation's Model data. Users are required to map elements of their input model to elements of visual notations Model by defining mapping correspondences using drag and drop. These drag and drop interaction triggers a transformation rule template to be created from to transform the elements being dragged to host notation's data. These templates are generated initially from the Models embedded in dragging notation and the host notation. To enable drag and drop of input elements a default tree-like representation of the input model data is provided for users.

Example 4.3 Assume we have an XML representation of a company's sales records and would like to visualise it using bar chart visualisation. Each bar in this bar chart will be representative of a sales record. To create these bars, first step is to drag and drop a sales element from example sales model data onto a bar notation as shown by Figure 4.9a. This interaction triggers a transformation rule template to be created from a sales record element to the bar notation's data. Next, corresponding internal elements of sales record and the notation's Model should be linked, i.e. sales record's Region attribute should be dragged and dropped on bar's Name and Amount should be dragged on bar's Value, as shown by Figure 4.9a. These internal correspondences fill the transformation rule template. Note that this tasks needs to be performed only once for all sales records. Same procedure should be performed for chart notation and the spread sheet element as can be seen in Figure 4.9b.

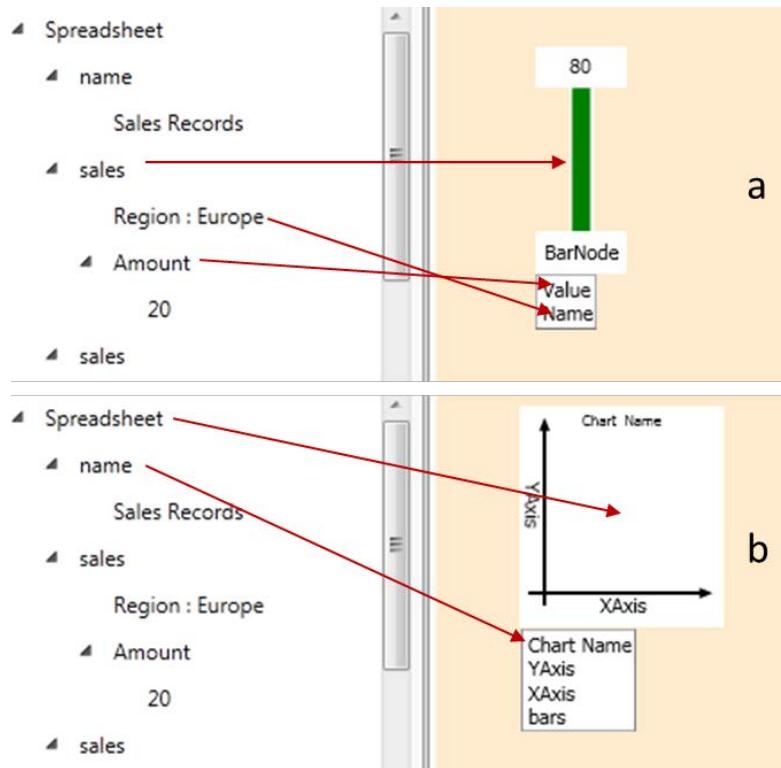


Figure 4.9 Mapping sales records input to notations: a) Mapping a sales record to a bar, b) Mapping spread sheet to chart. Arrows depict drag and drop directions.

Once data mapping is complete, the new customised notations are saved. This results in creation of a customised notation and a transformation rule that transforms a portion of input model to the notation's Model (and its reverse where possible). In Example 4.3 for instance, a transformation rule will be generated to transform each sales record to the Model part of bar's notation. Note that at this point the notation can generate its View according to the Model using the Controller defined in step one.

4.2.2.1 Transformation functions

Not all correspondences between visual notation elements are simple 1-to-1 relations. Therefore, a variety of model transformation functions, such as summation, merging, subtraction, and textual parsing need to be used, among others. These functions enable the specification and generation of more complex correspondences and hence generation of more complex model element-to-visual notation and visual notation-to-visual notation mappings.

The structure of a transformation function is depicted by Figure 4.10. The model and usage of each function follows visual notations with a difference that function's view is provided by an image rather than a visualisation. Similar to visual notations, a function can be dragged and dropped and right clicking on them reveals their input and output arguments.

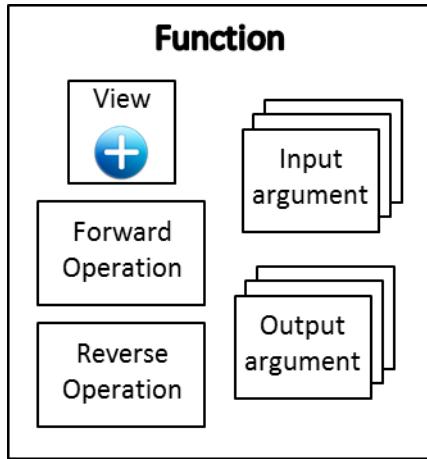


Figure 4.10 Transformation function's structure.

Functions are defined by templates that specify input and output arguments, forward and reverse operations to be performed by the function, and a representative image that can be provided by users. One-to-one correspondences in our approach result in reverse transformations being generated automatically. To generate more complex one-to-many, many-to-one, or many-to-many correspondences using functions, the reverse operation (if possible) should be provided by the function designer. If reverse operation is not possible, a default operation can be provided instead.

In the case of arithmetic operations, the reverse direction is not possible when a group of values are used to calculate a final value. It is due to the fact that the information provided by original values is lost during forward operation. These transformations are called *Lossy* and while we have worked on support for them, addressing them is not detailed in this thesis and is left for future work.

Figure 4.11A shows a summation function with its two inputs and an output. The internal code template of the function is depicted by part B. As can be seen, since the reverse operation of the two added values is not possible unless at least one of the original values is stored, the reverse operation (marked by arrow) divides the output to calculate two input values in reverse. An example of using functions in mapping correspondence specification is provided by Example 4.4.

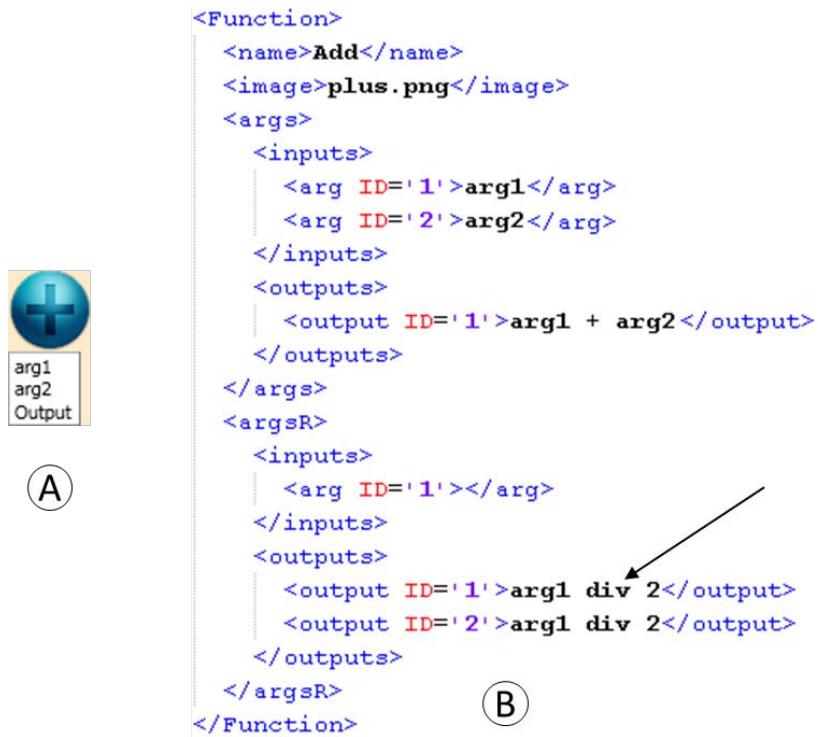


Figure 4.11 A) A summation function, B) Its template. Arrow marks reverse operation. Information of input arguments is lost during forward summation operation.

Example 4.4 Assume sales records of example 4.3 consisted of two amounts for representing each sales element. Since bar notation takes only one amount for its Value, these two amounts should be added before mapping to the bar's value. Figure 4.12 demonstrates how the summation function can be used to calculate the bar's Value according to the sum of two input values. The arrows demonstrate how dragging and dropping will be performed by the user in this case.

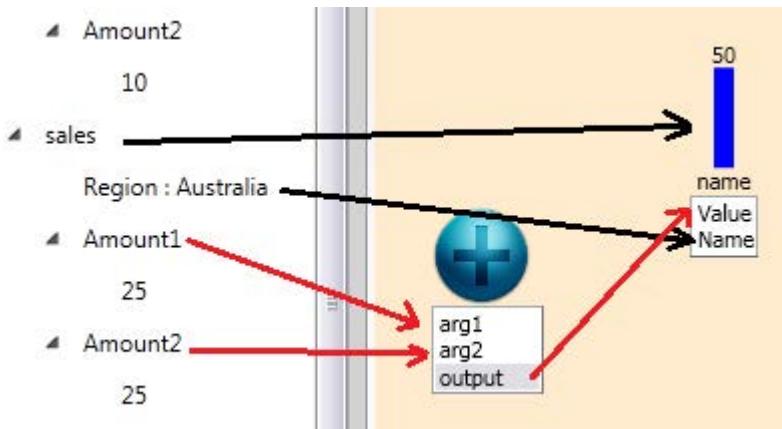


Figure 4.12 Mapping sales records to bar using summation function. Arrows depict drag and drop directions.

Once data mapping is complete, the new customised notations are saved. This results in creation of a customised notation and a transformation rule that transforms a portion of input model to the notation's Model (and its reverse where possible). In Figure 4.12 for instance, a transformation rule will be generated to transform each sales record to the Model part of bar's notation. A transformation code generator reads function templates and generates transformation code for the specified transformation language. For example, the resulted transformation script using the summation function of Figure 4.12 in XSLT is provided by Figure 4.13. This code transforms the sales element and its internal elements to a bar node's data model. Note that argument numbers are automatically updated by transformation code generator to prevent similar argument names in the full transformation script. The reverse transformation script is also shown by Figure 4.14.

```

<BarNode xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:variable name="arg225" >
    <xsl:value-of select="Amount1" />
  </xsl:variable>
  <xsl:variable name="arg226">
    <xsl:value-of select="Amount2" />
  </xsl:variable>
  <xsl:variable name="output227" >
    <xsl:value-of select="$arg225 + $arg226 " />
  </xsl:variable>
  <Value>
    <xsl:copy-of select="$output227" />
  </Value>
  <Name>
    <xsl:value-of select="@Region" />
  </Name>
</BarNode>

```

Figure 4.13 The generated transformation script resulted from use of the function of Figure 4.12.

```

<sales xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:variable name="arg227" >
    <xsl:value-of select="Value" />
  </xsl:variable>
  <xsl:variable name="output225">
    <xsl:value-of select="$arg227 div 2" />
  </xsl:variable>
  <xsl:variable name="output226" >
    <xsl:value-of select="$arg227 div 2" />
  </xsl:variable>
  <xsl:attribute name="Region" >
    <xsl:value-of select="Name" />
  </xsl:attribute>
  <Amount1>
    <xsl:copy-of select="$output225" />
  </Amount1>
  <Amount2>
    <xsl:copy-of select="$output226" />
  </Amount2>
</sales>

```

Figure 4.14 Reverse transformation script resulted from use of the function of Figure 4.12.

By default a set of simple functions are provided in our framework implementation. Table 4.1 provides a list of these functions and the operation they perform. There has

been no rational behind choosing the representative images for function notations. Users can alter these notational images according to their preference.

Table 4.1 List of default functions provided in proof of concept framework.

Function	Notation	Inputs	Forward Operation	Outputs	Reverse Operation
Summation Adds two arguments		Arg 1 Arg 2	Arg1 + Arg2	Out	Out div 2
Subtraction Subtracts two arguments		Arg 1 Arg 2	Arg1 – Arg2	Out	Return Out
String Merge Merges two strings		Arg 1 Arg 2	Merge (Arg1, " ", Arg2)	Out	Split (output, " ")
String split Splits two strings from first occurrence of Space character		Arg 1	Split (Arg1, " ")	Out 1 Out 2	Merge (Arg1, " ", Arg2)

Additional functions can be defined using function's template by providing their input and output arguments and the required operations. These functions can be saved in a function repository for reuse. Function templates provide dedicated spaces for specifying the task that the function performs. This task should be provided by function designer depending on the arguments and according to the transformation language code that is to be generated from this function. Example 4.5 shows how an additional functionality can be provided using function templates.

Example 4.5 This example shows a function that takes two strings as input arguments and returns three string outputs using combination of input strings and constant values as specified in Figure 4.15. For example, if “Sales” and “Europe” are provided as inputs to this function, it will return “Sales Amount”, “Europe” and “Sales of Europe” as outputs. Reverse operation calculates original input values based on the outputs.

```

<Function>
  <name>Mix</name>
  <image>mix.jpg</image>
  <args>
    <inputs>
      <arg ID='1'>Arg 1</arg>
      <arg ID='2'>Arg 2</arg>
    </inputs>
    <outputs>
      <output ID='1'>concat(concat(arg1 , ' '), 'Amount')</output>
      <output ID='2'>arg2 </output>
      <output ID='3'>concat(concat(arg1 , ' of '), arg2 )</output>
    </outputs>
  </args>
  <argsR>
    <inputs>
      <arg ID='1'></arg>
      <arg ID='2'></arg>
      <arg ID='3'></arg>
    </inputs>
    <outputs>
      <output ID='1'>substring-before(arg1 , ' Amount')</output>
      <output ID='2'>arg2 </output>
    </outputs>
  </argsR>
</Function>

```

Figure 4.15 Example of defining a new function using function template.

Once used, functions are read by transformation code generator. The operation of each function is translated to the transformation language of choice and the resulted outputs are saved in variables inside transformation rule scripts. As a result, when defining functions, function designer needs to have previous understanding of the transformation language of choice. For instance in Example 4.5, *concat* and *substring-before* functions are functions provided by XSLT language. For other transformation languages, their dedicated functionality should be stated in the function templates.

Pointers to variables of each function will be placed inside transformation rules in places where function outputs are to be used. For example see the resulted function script of Figure 4.13.

4.2.2.2 Transformation conditions

Transformation conditions are used in a similar way to transformation functions and control when and how correspondences are applied. Figure 4.16 shows a transformation condition's structure. As can be seen in the figure, a difference of conditions and functions is the missing output arguments. This is due to the fact that the output of a condition is not known beforehand. It depends on the values being sent to the condition. As a result, unlike functions, conditions do not have an explicit output.

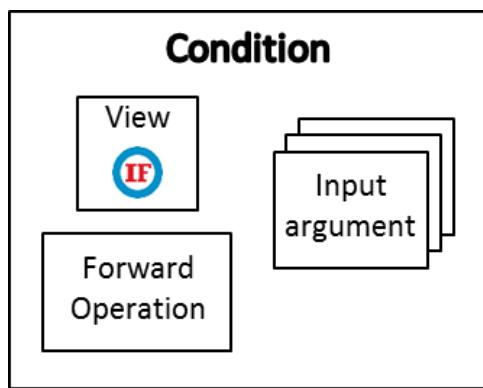


Figure 4.16 Transformation condition's structure.

Similar to transformation functions, conditions are also defined using a template. This template defines the arguments to base the conditions on, the condition expression(s) and the values to be transferred as output if the condition is met. In case none of the conditions are met, “else” statements can be provided as well.

Example 4.6 Figure 4.17a shows a condition that checks two arguments (*arg1* and *arg2*) and if the values provided by these arguments are equal, passes the value that has been dragged to the condition expression. If not, the value dragged to “Otherwise” will be used as output. The internal template of this condition is provided by Figure 4.17b.

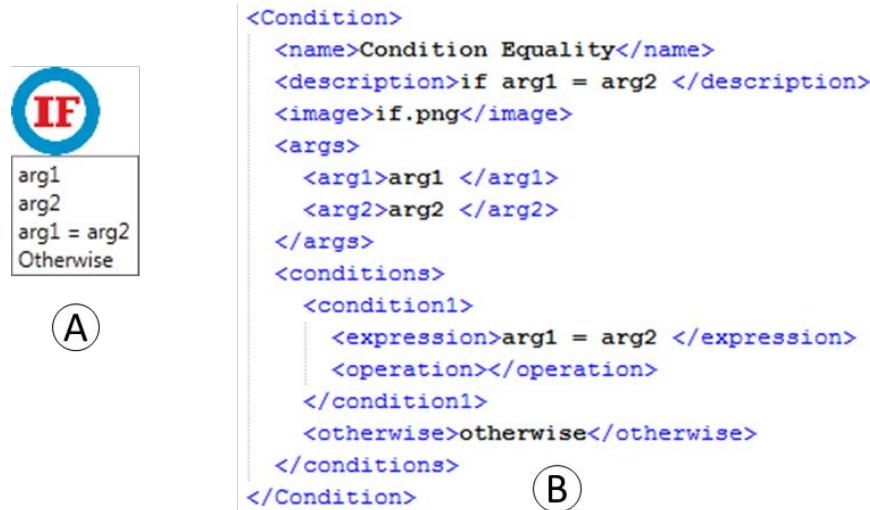


Figure 4.17 A) A transformation condition, B) Its code template.

To specify a conditional correspondence, after drag and drop of required source elements on condition's arguments, the condition notation itself is dragged and dropped on target element.

Example 4.7 Figure 4.18 shows using a condition for specification of a model to visual mapping. The conditions check if the value provided with Amount element is more than or equal to 40. If so the colour to be returned by the condition is Blue, otherwise Red will be returned. The condition itself is then dragged and dropped on colour element of the bar. This condition specification will result in the code script of Figure 4.19. This script will be included in the transformation rule script.

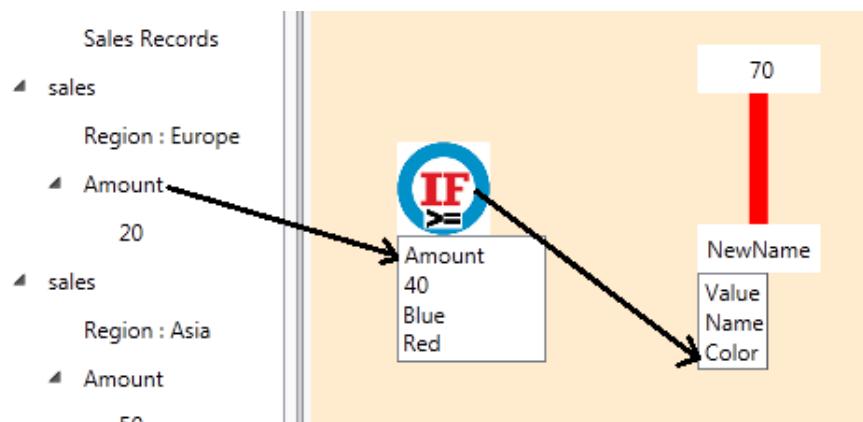


Figure 4.18 Using a condition for specification of bar's colour.

```
<Color xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:choose>
    <xsl:when test="Amount &gt;= 40 ">Blue</xsl:when>
    <xsl:otherwise>Red</xsl:otherwise>
  </xsl:choose>
</Color>
```

Figure 4.19 Transformation code script resulted from condition of figure 4.18.

In case there is a requirement to have reverse operation for conditions, a possible solution is to check whether the condition is valid for source through model checking.

For instance in Example 4.7 the reverse operation could check if the colour of a bar is blue, the value to be copied to sales Amount is indeed more than or equal to 40. This model checking, however, is outside the scope of this thesis. The assumption here is that if the values generated by forward transformation rules that use conditions exist in the target, their respective source portion of the transformation rule should be generated in reverse. Therefore, the reverse operation creates a non-conditional transformation.

Table 4.2 lists the default conditions provided in our framework implementation. Similar to functions, additional conditions can be added to the framework using provided template.

Table 4.2 List of default conditions provided in proof of concept framework.

Condition pseudo code	Notation	Number of Arguments
if Arg1 = Arg2 then pass Arg3 else pass Arg4		4
if Arg1 >= Arg2 then pass Arg3 else pass Arg4		4
if Arg1 = Arg2 then pass Arg3 else if Arg1 > Arg2 then pass Arg4 else pass Arg5		5

4.2.3 Notation composition

Is response to research question 1.4 on composition of notations, step three allows users to link, combine and embed customised notations. A customised notation represents a model element-to-visual notation transformation rule. To have a complete transformation script, the prepared collection of transformation rules should be scheduled according to their call sequence. With traditional transformation scripting languages this is achieved by asking users to write codes for this script, similar to procedural programming, and by providing metamodels. In our approach however, our assumption is that there is no metamodel available and user is not willing to code. Therefore, by using composition of notations we infer the target visualisation's metamodel and call sequencing of the transformation script.

By linking a notation to a placeholder element of another, the host notation knows the transformation rule embedded in the notation being dragged should be called in place of the element. This is in order to affect the embedded model element-to-visual notation mapping. This linking results in scheduling of model element-to-visual notation transformation rules.

***Example 4.8** In composing notations of Figure 4.20, by linking the bar model element-to-visual notation visualisation component defined earlier (see Figure 4.9) to a bars element of a bar chart model element-to-visual notation visualisation, it is specified that the bar chart contains set of bars.*

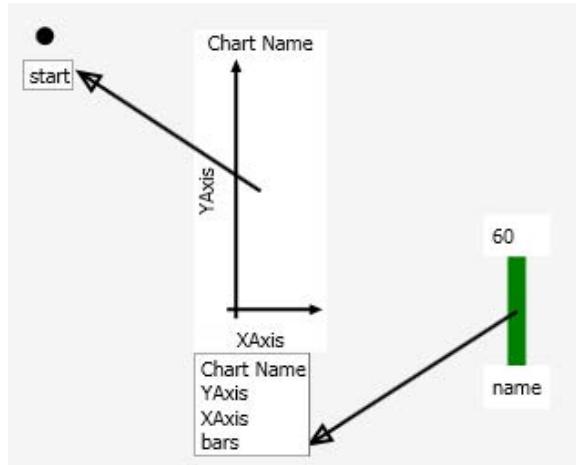


Figure 4.20 Composition of visual notations to create bar chart visualisation.

Linking a notation to a *start* element will define the top-most (first to be run) transformation rule for the completed model transformation specification. This tells the transformation scheduler to start generating transformation code from the rule linked to *start* element. For example, in Figure 4.20, the bar chart notation's transformation rule is the first rule to be called to transform a company records model element to a bar chart notation representation. It then calls bar's transformation rule to generate a bar representation for each sales record.

The composition process results in a complete transformation script that transforms the input model to the visualisation of the composed notations. As stated previously, since all visual notations resulting from this transformation are wrapped by interaction logic of visual notation, the whole visualisation and its composing notations can be interacted with in form of being dragged and also other notations can be dropped on them.

Example 4.9 Figure 4.21 shows the resulting bar chart of the transformation generated from the composition of Figure 4.20. A user has right clicked on the bar chart and the internal model elements are being represented as a result in the pop-up window. Note that since no value has been provided to XAxis and YAxis labels, default values provided by bar chart notation's model have been used.

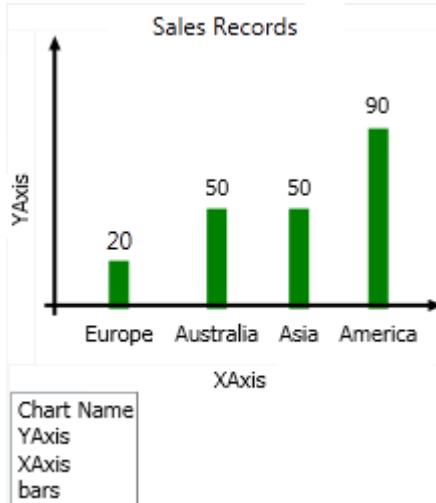


Figure 4.21 Visualisation of a bar chart. User has right clicked on the bar chart and the internal elements of the bar chart notation are represented in a pop-up window.

Composition of multiple model element-to-notation mappings into a complete visualisation specification also allows the system to build a meta-model from the underlying Model of each notation element. Our approach uses this meta-model for model validation purposes.

4.2.3.1 Visual aid for debugging transformation composition

To provide support for scheduling and being able to debug rule sequencing (and hence answering research question 1.5), the rendering mechanism is designed in a way that it is capable of rendering partial visualisations. As a result, when dragging a transformation rule on a notation placeholder element, even though the resulting rule may not be complete, the system is capable of rendering the partial result. Therefore, users can review the result of so far completed rule on the spot and perform corrections as they see fit.

Example 4.10 In composing a bar chart visualisation, when a bar is linked to bar chart's "Bars" element, the provided debugging aid depicts the result of that bar being inserted in the bar chart. Figure 4.22 depict the provided debugging aid. Also as another example, Figure 4.23 shows debugging aid during generation of a

class diagram. User has dropped parameter notation on “Parameters” element of a function notation.

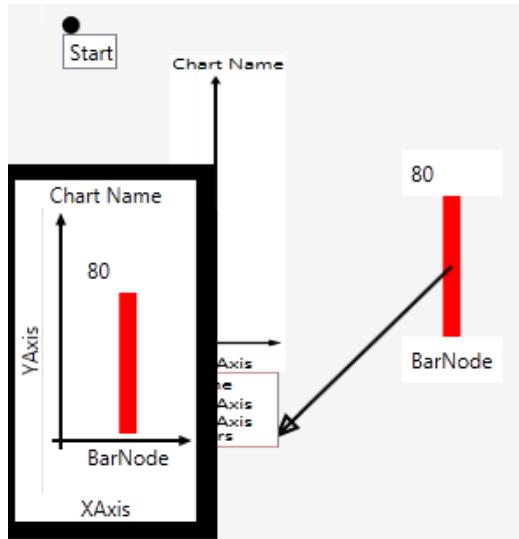


Figure 4.22 Visualisation composition debugging aid. The product of composing a bar in the bar chart is shown in a pop-up.

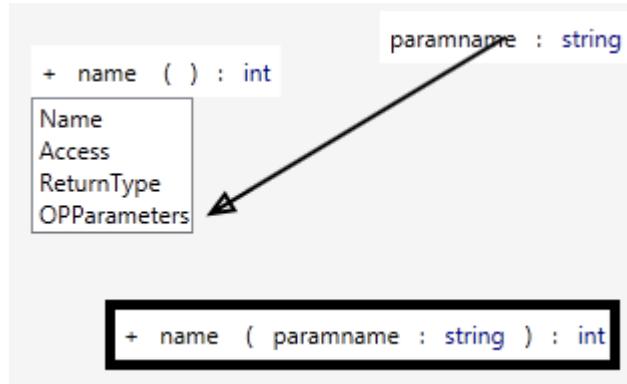


Figure 4.23 Visualisation composition debugging aid for function of a class diagram visualisation.

4.3 Case studies

This section provides number of case studies to show applicability of the approach for different visualisations and input models. It will focus also on the implementation specific decisions taken for the proof of concept prototyping of the approach.

4.3.1 Bar chart visualisation

This section provides detailed visualisation of the bar chart used throughout the chapter. The input file to be visualised is provided as an XML and contains sales records of a company as seen in Figure 4.24.

```
<Spreadsheet>
    <name>Sales Records</name>
    <sales Region="Europe">
        <Amount1>30</Amount1>
        <Amount2>10</Amount2>
    </sales>
    <sales Region="Australia">
        <Amount1>25</Amount1>
        <Amount2>25</Amount2>
    </sales>
    <sales Region="Asia">
        <Amount1>20</Amount1>
        <Amount2>30</Amount2>
    </sales>
    <sales Region="America">
        <Amount1>40</Amount1>
        <Amount2>50</Amount2>
    </sales>
</Spreadsheet>
```

Figure 4.24 Example of sales records XML input.

Our implementation uses Windows Presentation Foundation (WPF) and XML Application Markup Language (XAML) for views. Models to be linked to these Views are provided in XML. For example, for the bar chart visualisation the View, Model and controller transformations of bar chart and the bars are provided in Figures 4.25 and 4.26.

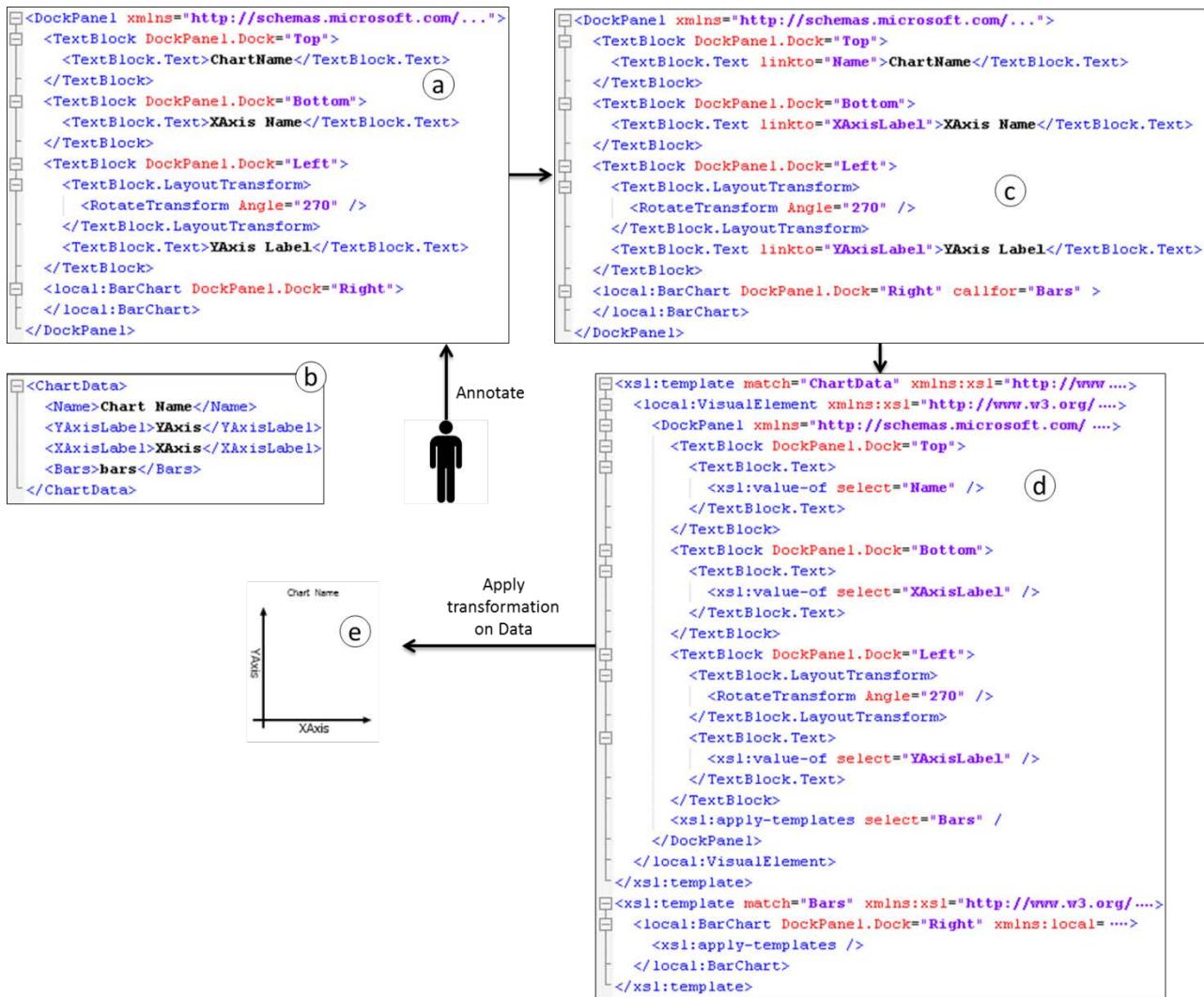


Figure 4.25 Bar chart notation, a) View code, b) Model XML, c) Annotated View, d) Controller transformation and e) Final notation.

WPF allows visualisation logic (if required) to be separately implemented in C# or Visual Basic as accompanying classes. For instance in bar chart visualisation of Figure 4.25a, a BarChart class (derived from Canvas class) has been implemented in local namespace which normalises the height of bars and positions them according to bar chart's height and width. Since possible bars are to be included inside this bar chart class, the 1-to-many mapping correspondence is annotated in this element by callfor="Bars" annotation (see Figure 4.25c). The resulting Controller transformation of bar chart calls for other transformations to transform the data being inserted in "Bars" element to visual bars and include them in bar chart for height normalisation and placement.

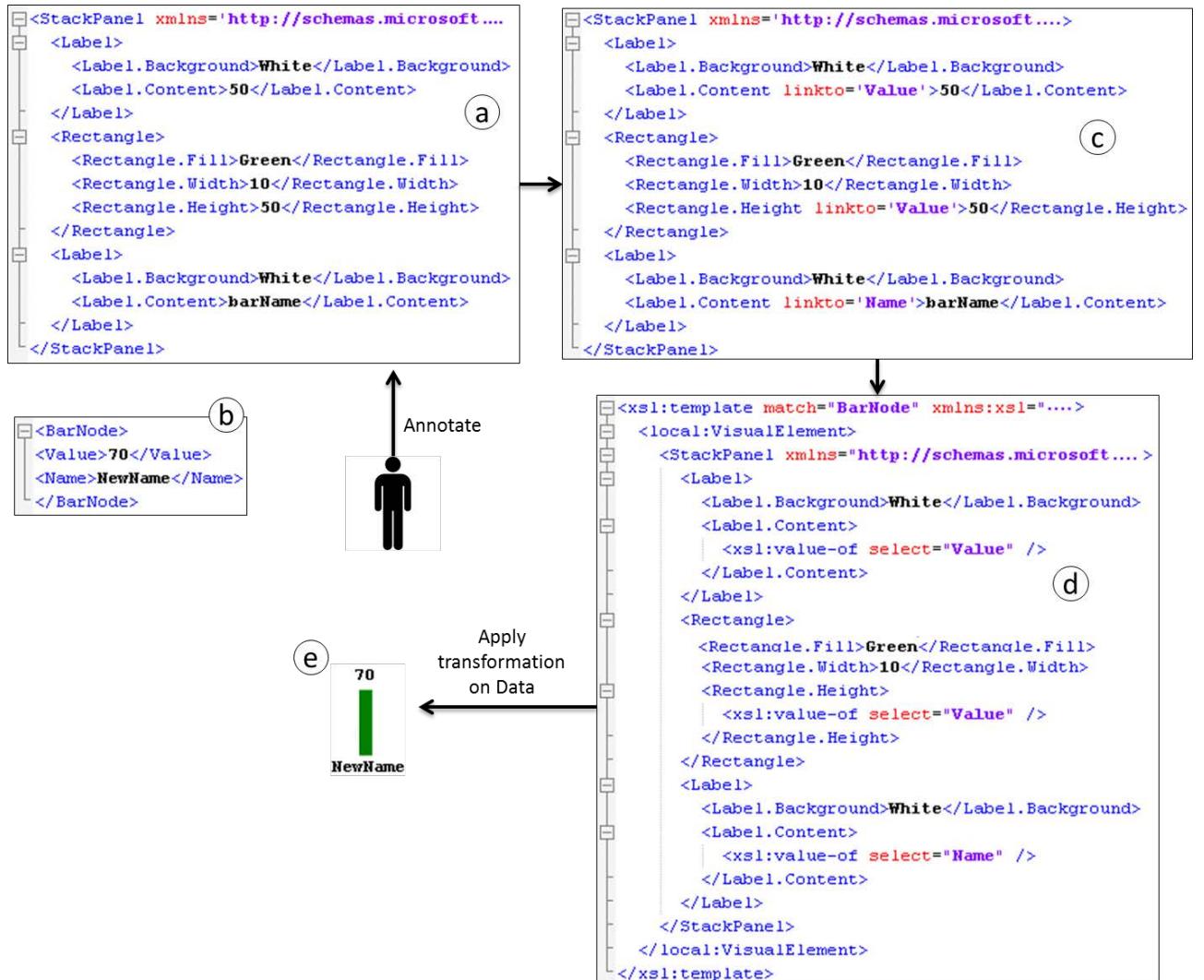


Figure 4.26 Bar notation's a) View code, b) Model XML, c) Annotated View, d) Controller transformation and e) Final notation.

That bar notation of Figure 4.26 has a value and a name as its model elements. These elements are in one-to-one relationship with elements of the view i.e. height of the bar, the label on top of the bar, and the label below each bar. These correspondences are provided by linkto annotations in Figure 4.26c.

Once these notational elements are defined, they need to be mapped to input model elements to specify model-to-visualisation example. Considering that the file to be transformed to bar chart visualisation has two values to be added to represent the bar's value, Figure 4.27a shows how the values can be added and linked using a summation function. The transformation rule resulting from the mapping and correspondences in Figure 4.27a is provided in Figure 4.28.

It is worth mentioning here that specification of correspondences between notation Views and Model is somewhat a complex task and requires basic understanding on XAML graphics. However, we believe with utility of visualisations expanding over time and as new additional visualisations are needed, this becomes less of an issue. Our assumption here is that generating notations is not a task to be performed by end user and end users will be provided with the prepared notations.

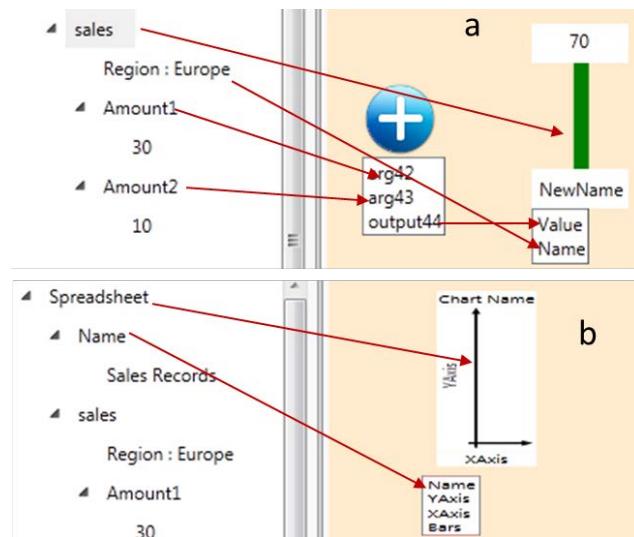


Figure 4.27 a) mapping sales records to bar using summation function and b) mapping Spreadsheet to bar chart notation. Arrows depict drag and drop directions.

```

<BarNode>
  <xsl:variable name="arg1">
    <xsl:value-of select="Amount1" />
  </xsl:variable>
  <xsl:variable name="arg2" >
    <xsl:value-of select="Amount2"/>
  </xsl:variable>
  <xsl:variable name="output1">
    <xsl:value-of select="$arg1 + $arg2" />
  </xsl:variable>
  <Value>
    <xsl:copy-of select="$output1"/>
  </Value>
  <Name>
    <xsl:value-of select="@Region" />
  </Name>
</BarNode>

```

Figure 4.28 Transformation code resulting from mapping correspondences and summation function of Figure 4.27.

Same mapping operation should be performed for bar chart notation (see Figure 4.27b). Once model to visual notation mapping are defined their notations should be composed to generate a full visualisation and model to visualisation transformation script. Figure 4.20 earlier showed an example of this composition with the resulting visualisation depicted by Figure 4.21. Now let's assume there is a need to update the visualisation and include colour in the bar notation's data. To perform this alteration, a new bar notation needs to be defined. Figure 4.29 shows the model data required for the new bar.

```

<BarNode>
  <Value>70</Value>
  <Name>NewName</Name>
  <Color>Red</Color>
</BarNode>
```

Figure 4.29 Alternative model for bar's notation.

The bar's view already accounts for bar's colour (see Figure 4.26a). However, it was set to Green by default. Hence, the annotation for generating the controller transformation of the new bar should consider the one to one relationship of Color element in the data and the colour of bar in the view. Figure 4.30 represents the new view annotation with the added annotation for the bar colour.

```

<StackPanel>
  <Label>
    <Label.Background>White</Label.Background>
    <Label.Content linkto='Value'>50</Label.Content>
  </Label>
  <Rectangle>
    <Rectangle.Fill linkto='Color'>Green</Rectangle.Fill>
    <Rectangle.Width>10</Rectangle.Width>
    <Rectangle.Height linkto='Value'>50</Rectangle.Height>
  </Rectangle>
  <Label>
    <Label.Background>White</Label.Background>
    <Label.Content linkto='Name'>barName</Label.Content>
  </Label>
</StackPanel>
```

Figure 4.30 New bar's view annotation.

With the new bar's notation generated, input model element to be represented with this new bar should be mapped to its data. These model to visualisation mappings are provided in Figure 4.31. Apart from the newly added Color element in notation's model, the rest of the mapping are similar to previous bar's mapping.

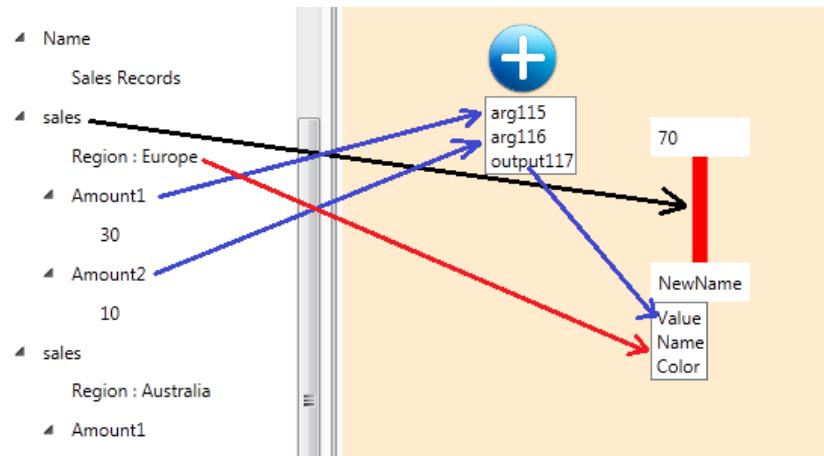


Figure 4.31 Mapping sales records to new bar.

Let's assume the colour to be specified for the new bar is to be specified according to the values in the Amount1 and Amount2 of the sales element, in a way that if Amount1 is more than Amount2 the colour of bar would be blue, if they are equal bar should be black and red otherwise. Figure 4.32 shows how a condition can be used in this case. The condition is dropped on designer canvas and Amount1 and Amount2 are drag and dropped on its arguments.

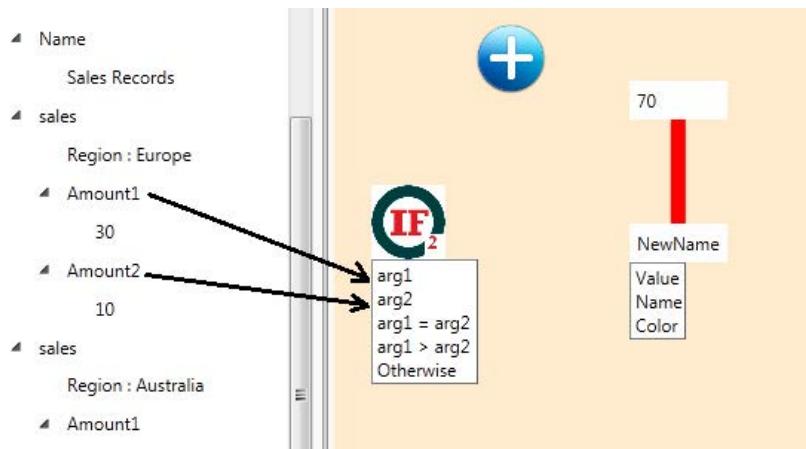


Figure 4.32 Specifying arguments of condition.

Note that it is possible to provide consistency checks in elements for example to check if the element that is dragged to “arg1” actually represents a numerical value. These checks should be provided and implemented in the visual elements and are outside the scope of our work here. If wrong elements have been dragged and dropped they can be over written by drag and dropping the correct elements. Based on the amounts, a colour value is to be passed by the condition. These colour values can be provided using a separate input. Figure 4.33 shows this separate input and how colour values are linked to condition expressions by drag and drop. Since functions do not have specific output (the output is selected based on the condition expressions) user has to drag and drop the function expression on the Color element of the notation as shown by arrows in Figure 4.33.

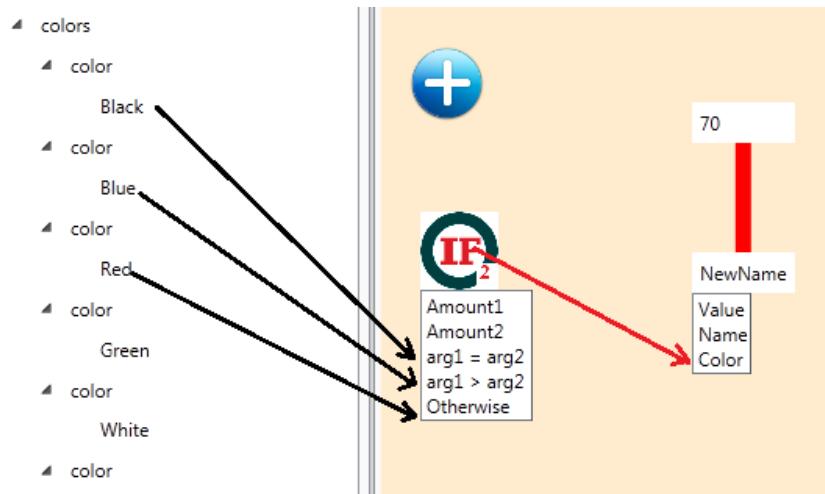


Figure 4.33 Mapping colour values to condition, and mapping condition result to Color element of bar notation.

Now that the new bar is defined, it should be included in the composition. Note that the new bar is the only notation that has been altered. Since other notations have not changed (bar chart notation in this case) they can be reused in the composition. Figure 4.34 shows the composition using the new bar’s notation. The arrows in Figure 4.34 are provided by framework for better tracking of notation composition. As mentioned before, the bar chart notation is the previous notation and has been reused.

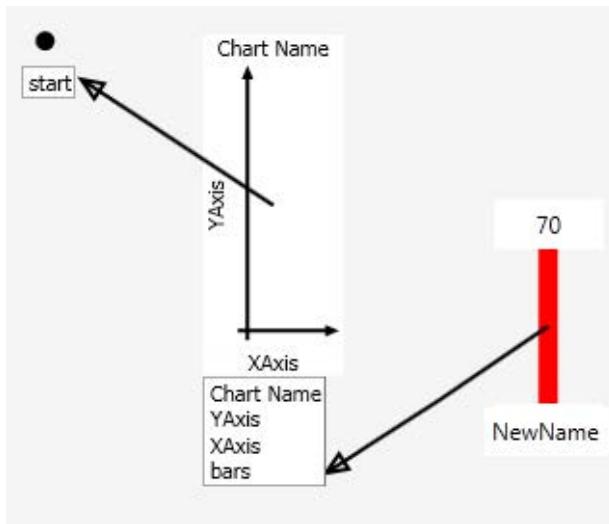


Figure 4.34 Composition of notations using the new bar's notation.

The result of composition of Figure 4.34 will be a transformation from sales records to bar chart visualisation with different colours for bars based on the provided values of the amounts similar to Figure 4.35.

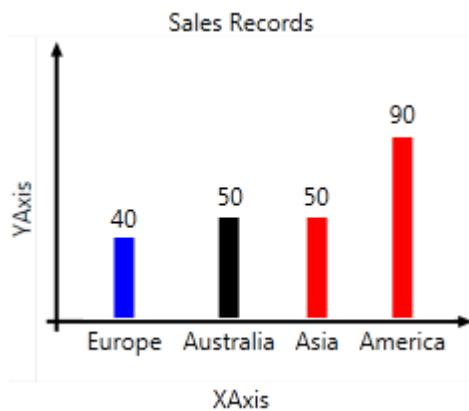


Figure 4.35 Resulted coloured bar chart.

4.3.2 Minard's Map visualisation

This case study section provides a worked example of creating a simplified version of Minard's map visualisation, as can be seen in Figure 4.36. Minard's map is a famous visual depiction of the French Grande Army's campaign for the invasion of Russia in 1812 by Charles Joseph Minard. It is widely considered as one of the best statistical graphs by the visualisation community [3], depicting number of troops, locations, campaign movements and status, and temperature information.

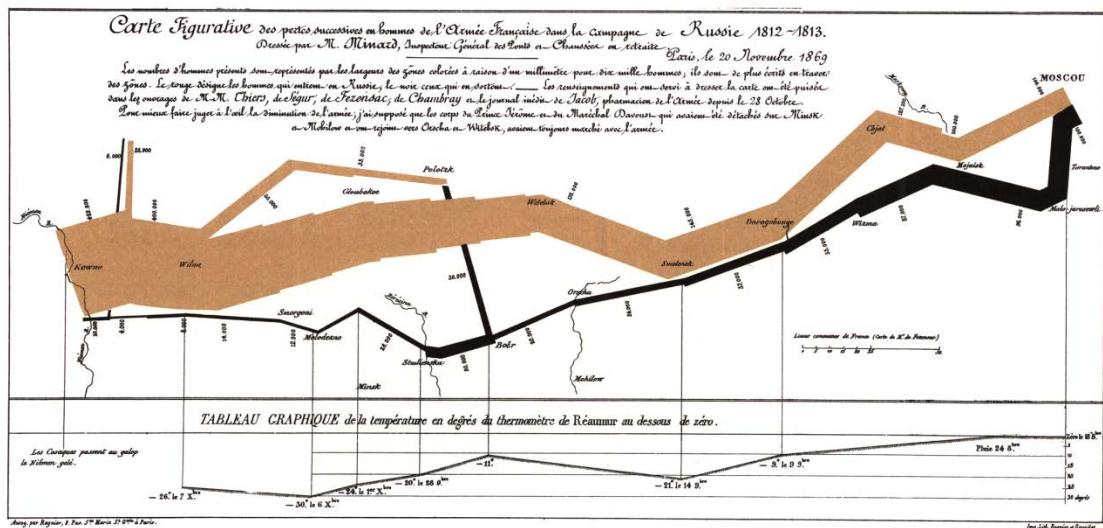


Figure 4.36 Minard's map (from [3]).

To reproduce this visualisation, we follow a similar approach to that used by Humphrey [4]; in that, at each location, a circle with its radius representing number of troops is drawn. Connecting these circles depicts troop movement. Therefore, each troop movement is represented by two circles depicting number of troops at starting point and destination and the lines connecting them. The colour of these movements defines whether troops were advancing (defined by Red) or retreating (defined by Black). Humphrey's recreation is depicted by Figure 4.37.

Our version of this visualisation uses two notations and we have asked a designer to craft required notation Views. These Views are provided using Windows Presentation

Foundation (WPF). For simplicity in the process, temperature information of the map is omitted and coordinate of the locations are considered relative to visualisation canvas.

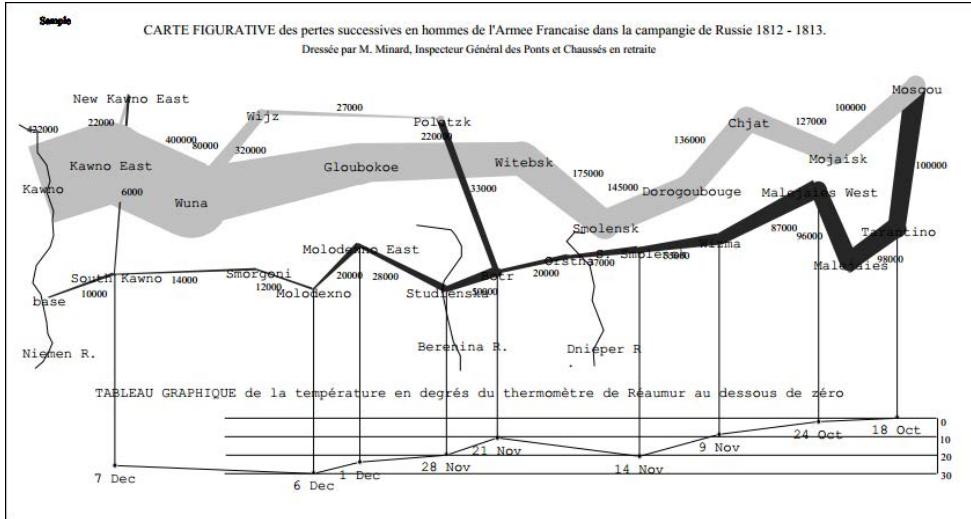


Figure 4.37 Minard's map recreated by Humphrey [4].

The first notation depicts troop movements from one position to another. Based on coordinates and the number of troops at the starting point and the number arriving at the destination, a shape will be drawn as specified in the notation View (Figure 4.38a). The algebras for creating this shape and the circles at the starting point and destination, for filling angular gaps, are provided in [4]. Our approach provides a way to specify and reuse this in C#. This notation requires a data part that represents start and destination location names and coordinates, number of troops starting the movement and arriving at destination, and the colour of the shape which represents whether troops were advancing or retreating. Therefore the data part for this notation will be similar to Figure 4.38b. Elements of this model are all in one-to-one relation with notation attributes. Therefore, they will all be specified for the controller transformation using “linkto” annotations as depicted in Figure 4.39.

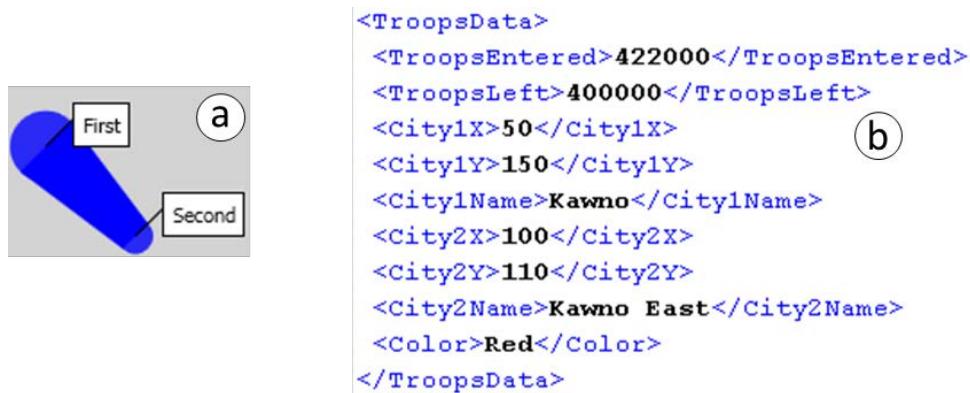


Figure 4.38 Troops movement notation's (a) View and (b) Model.

```

<local:TroopsMovement>
  <local:TroopsMovement.StartX linkto="City1X">5</local:TroopsMovement.StartX>
  <local:TroopsMovement.StartY linkto="City1Y">5</local:TroopsMovement.StartY>
  <local:TroopsMovement.EndX linkto="City2X">55</local:TroopsMovement.EndX>
  <local:TroopsMovement.EndY linkto="City2Y">55</local:TroopsMovement.EndY>
  <local:TroopsMovement.StartLocationName linkto="City1Name">First</local:TroopsMovement.StartLocationName>
  <local:TroopsMovement.EndLocationName linkto="City2Name">Second</local:TroopsMovement.EndLocationName>
  <local:TroopsMovement.ShapeColor linkto="Color">Blue</local:TroopsMovement.ShapeColor>
  <local:TroopsMovement.TroopsStarting linkto="TroopsEntered">400000</local:TroopsMovement.TroopsStarting>
  <local:TroopsMovement.TroopsEnding linkto="TroopsLeft">200000</local:TroopsMovement.TroopsEnding>
</local:TroopsMovement>

```

Figure 4.39 Annotated View of Troops movement notation with Model elements.

The second notation shows the map with its description on top (Figure 4.40a). As a result, its Data should provide the description information (Figure 4.40b). The map notation should host troop movement notations, therefore, a placeholder for troops notations should also be provided in map notations model. When the user is annotating the View, “callfor” annotation is provided in the View according to Figure 4.41. The Minard class declared in XAML in Figure 4.41 is derived from the Canvas and allows for hosting of other visual elements. Setting the position of troop movement notations relative to the Canvas is embedded within movement notations once the coordinates are defined. Therefore, when placed on the hosting map (Minard Canvas), those notations are already positioned according to their coordinates.

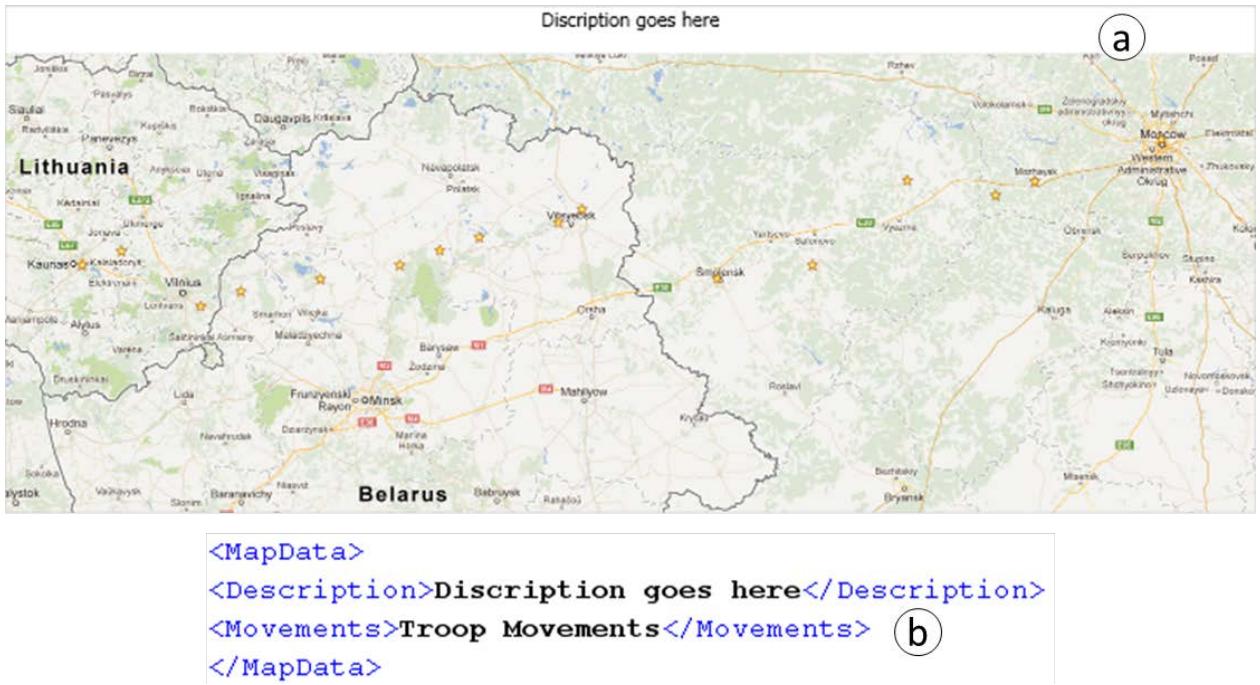


Figure 4.40 Map notation's (a) View, (b) Model.

```

<StackPanel Orientation="Vertical" Height="290" Width="716">
<TextBlock Height="27" TextAlignment="Center" Background="White">
<TextBlock.Text linkto="Description">Description of the map
</TextBlock.Text>
</TextBlock>
<Canvas Height="263" Width="716">
<Canvas.Background>
<ImageBrush ImageSource="NapoleonMap.bmp" />
</Canvas.Background>
<local:Minard Height="263" Width="716" callfor="Movements">
</local:Minard>
</Canvas>
</StackPanel>

```

Figure 4.41 Map notation's annotated View XAML.

The provided input data to be visualised is an XML data file (the input model) which includes a map description and a list of troop movement records. These records consist of start and destination location names and coordinates, number of troops starting and lost during the journey, and a status string which defines whether they were advancing or retreating.

To generate the transformation rule for visualising each movement record as a movement notation, users need to drag and drop a record element from input to a troop movement notation. Figure 4.42 shows mapping specification of records to troop movement notation.

Correspondence specification of troop movement notation starts by linking record element from input model to the troops notation marked by 1 in Figure 4.42. The bold solid black arrow depicts the drag and drop direction for specifying this correspondence. As a result, a troop movement notation will be generated for each Record element. Considering input data and the notation's Model data of Figure 4.42, we can conclude that coordinates, name of locations, and number of troops at the start are in a one-to-one relationship with their corresponding elements on the notation's Model data. Therefore their correspondences can be specified by direct drag and drop of input data elements on the notation's Model elements as shown by black dashed arrows in Figure 4.42. However, the notation requires the number of troops at the destination, whereas the input data record provides number of troops lost during the journey. Also the status of the movement is declared by Advancing or Retreating strings in the input while this has to be defined by colour in the notation. As a result, to specify these correspondences, user of our approach has to use provided functions and conditions.

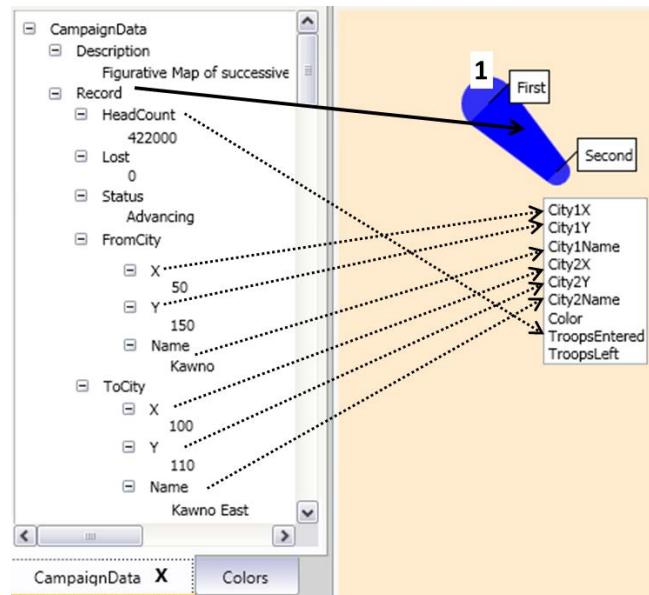


Figure 4.42 Specifying correspondences between troop movement records and provided troop movement notation. Arrows indicate drag and drop directions.

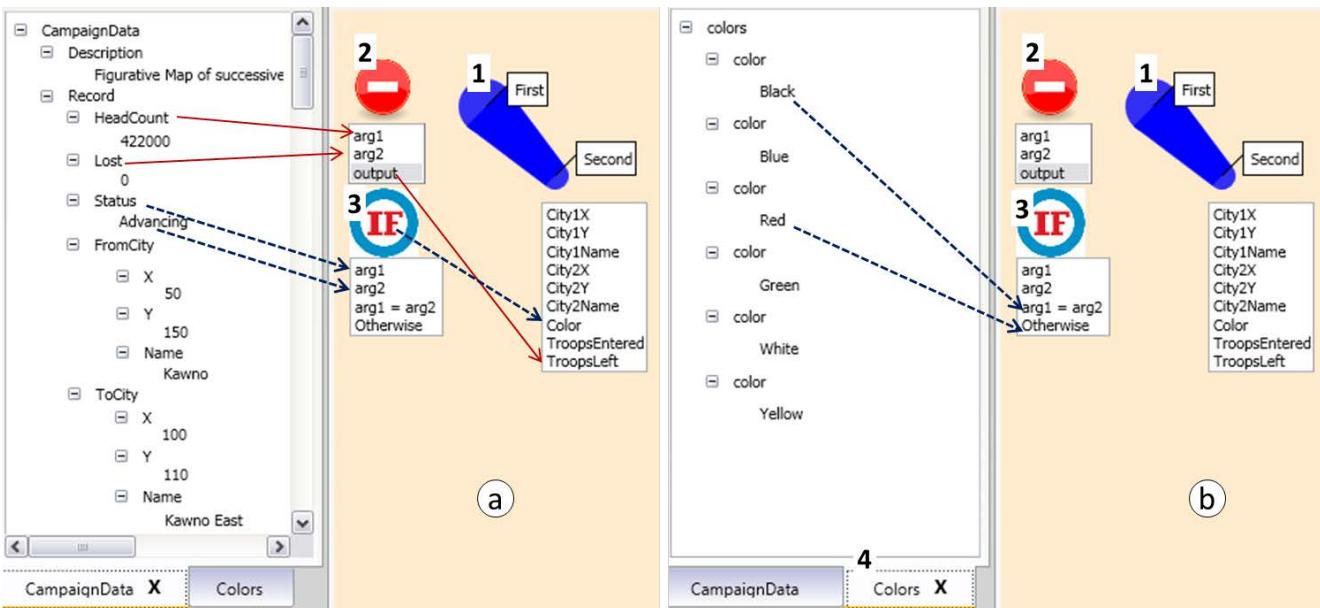


Figure 4.43 Specifying correspondences between troop movement records and provided troop movement notation using functions (a) correspondences with campaign data input file and (b) using separate input file to specify colours. Arrows indicate drag and drop directions.

Troop movement status can be defined by using a condition which provides a colour according to the status string. To generate this, user drops a condition on the Canvas (as marked by 3 in Figures 4.43a and 4.43b) and links corresponding elements. The (navy) bold dashed arrows depict drag and drop directions for correspondences of this condition. Colour names are provided as a separate input file (See 4 in Figure 4.43b). The required colours are then dragged and dropped on condition elements (See Figure 4.43b). Similarly, for specifying the number of troops at the destination, the user is provided with a subtraction function which subtracts troops lost from number of troops starting (Head Count) to calculate the required value (marked by 2 in Figure 4.43). The user then maps its result to troops arriving at the destination element “TroopsLeft” (red solid arrow in Figure 4.43a).

To specify correspondences for map notation, users need to drag a campaign data element from the input model on the map and link its description to the description element of the map notation, as shown by Figure 4.44. Note that by default space consuming notations are shrunk to save space on the designer Canvas. The “Movements” element is the place holder for troop movement notations which will be linked in the notation composition step.



Figure 4.44 Specifying correspondences between input data and map notation.

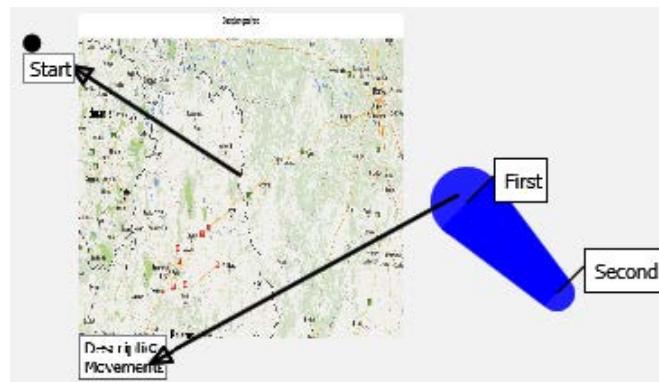


Figure 4.45 Composing troop movement and map notations to generate complete visualisation. Arrows are provided by the framework.

Now that both notations are designed and their correspondences to the input data are defined, the only step remaining to have a full visualisation is to compose it by linking the movement notation and the map notation, as shown by Figure 4.45. Once done, the generated transformation from this composition will be applied on the input data to produce a map visualisation and the resulting visualisation is shown by Figure 4.46.

Figurative Map of successive losses in men of the French army in Russian Campaign 1812 ~ 1813



Figure 4.46 Minard's map resulting from our approach.

4.3.3 UML class diagram visualisation

This case study demonstrates how a UML class diagram can be generated for example class diagram inputs. Figure 4.47 depicts a simplified sample of these examples provided in XML.

```

<CDiagram name="Organisation">
  <UMLClass name="Department">
    <access>public</access>
    <Attribute>
      <name>DepName</name>
      <type>String</type>
      <access>public</access>
    </Attribute>
    <Operation name="setName">
      <access>public</access>
      <returntype>void</returntype>
      <CDOPParams>
        <CDParameter name="dname">
          <type>String</type>
        </CDParameter>
      </CDOPParams>
    </Operation>
    <Operation name="getName">
      <access>public</access>
      <returntype>String</returntype>
    </Operation>
    <link label="Has">
      <toClass>Personnel</toClass>
      <Multiplicity>*</Multiplicity>
    </link>
  </UMLClass>
  <UMLClass name="Personnel">
    <access>public</access>
    <Attribute>
      <name>Name</name>
      <type>String</type>
      <access>public</access>
    </Attribute>
  </UMLClass>
</CDiagram>

```

Figure 4.47 An example of UML class diagram inputs.

As in previous example, a visual notation needs to be defined for each distinct part of input model. For this UML class diagram example, these parts include: UML attributes, operations and their parameters, classes, associations and the diagram itself. A desired visualisation for such an example would be similar to Figure 4.48.

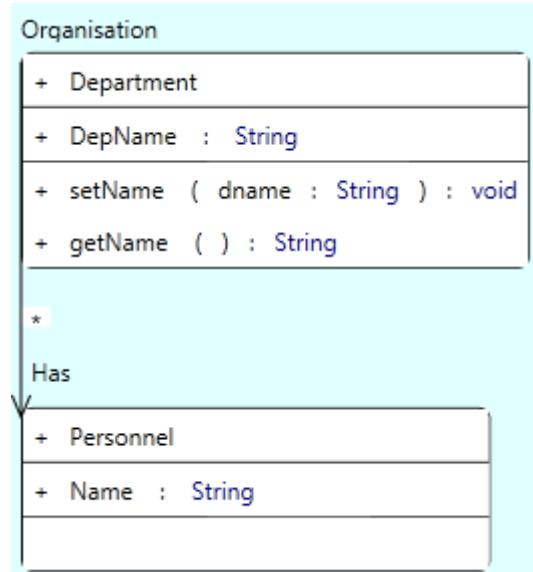


Figure 4.48 Desired visualisation for example of figure 4.47.

To generate notations required for this visualisation, a designer has crafted the notation views. These views are created using a combination of XAML shapes and C# logic. This C# logic controls how elements of these shapes are laid out on other notation views. For each of these views a model data has to be provided. Views have to be linked to model data to create notations. For example Figure 4.49 shows view and model of UML attribute notation.

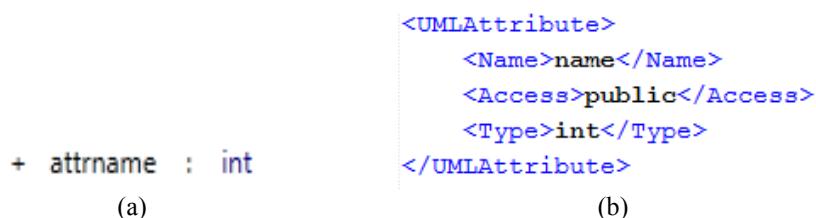


Figure 4.49 UML attribute notation's (a) View and (b) Model.

As shown by Figure 4.49b, the data includes attribute's name, type and access. These values are in one to one relation with elements of the view and should be annotated in

the provided view by linkto annotation so that the controller can be generated. Figure 4.50 shows the annotated view of attribute notation.

```
<local:UMLAttribute Background="White" >
  <local:UMLAttribute.AttributeName linkto="Name">attrname</local:UMLAttribute.AttributeName>
  <local:UMLAttribute.AttrAccess linkto="Access">public</local:UMLAttribute.AttrAccess>
  <local:UMLAttribute.AttributeType linkto="Type">int</local:UMLAttribute.AttributeType>
</local:UMLAttribute>
```

Figure 4.50 Annotated view of UML attribute notation.

The provided view for the attributes includes the required code for altering Access values. For example if the provided value for attributes access is Public it will generate a +, and similarly for private a – and so on. It will use blank if access value is not provided.

Other notations should be similarly generated using model data and the provided views. Figures 4.51 and 4.52 show view, model and annotated view of a UML function parameter.

<code>pName : int</code> (a)	<code><UMLParameter></code> <code> <Name>paramname</Name></code> <code> <Type>string</Type></code> <code></UMLParameter></code> (b)
---------------------------------	---

Figure 4.51 UML function parameter (a) View and (b) Model.

```
<local:UMLParam Background="White">
  <Label>
    <Label.Content linkto="Name">pName</Label.Content>
  </Label>
  <Label>:</Label>
  <Label Foreground="Navy">
    <Label.Content linkto="Type">int</Label.Content>
  </Label>
</local:UMLParam>
```

Figure 4.52 Annotated view of UML function parameter.

Figure 4.53 shows model and view of a UML function. The parameters are to be included inside this notation. Therefore, a place holder is provided in function model (“OPParameters”). This place holder is accordingly annotated by “callfor” annotation in Figure 4.54.

```

<UMLOperation>
  <Name>name</Name>
  <Access>public</Access>
  <ReturnType>int</ReturnType>
  <OPParameters>params</OPParameters>
+ OperationName ( ) : int    </UMLOperation>
(a)                                (b)

```

Figure 4.53 UML function notation’s (a) View and (b) Model.

```

<local:UMLOperation Background="White" >
  <local:UMLOperation.OperationName linkto="Name">OperationName</local:UMLOperation.OperationName>
  <local:UMLOperation.Access linkto="Access">public</local:UMLOperation.Access>
  <local:UMLOperation.ReturnType linkto="ReturnType">int</local:UMLOperation.ReturnType>
  <StackPanel Orientation="Horizontal" callfor="OPParameters" />
</local:UMLOperation>

```

Figure 4.54 Annotated view of UML function notation.

UML associations are composed of an arrow, cardinality, and a label for association name. Figure 4.55 shows the view and model of UML associations. The provided elements of association model are in one to one relationship with elements of the view. As a result they have been annotated using “linkto” annotation in Figure 4.56.

```

<UMLAbsociation>
  <Name>assocname</Name>
  <EndClass>class2</EndClass>
  <Multiplicity>*</Multiplicity>
</UMLAbsociation>
(a)                                (b)

```

Figure 4.55 UML association notation’s (a) View and (b) Model.

```

<local:UMLAssociation>
  <local:UMLAssociation.AssociationName linkto="Name">has</local:UMLAssociation.AssociationName>
  <local:UMLAssociation.EndClass linkto="EndClass">Class2</local:UMLAssociation.EndClass>
  <local:UMLAssociation.EndCardinality linkto="Multiplicity" >*</local:UMLAssociation.EndCardinality>
</local:UMLAssociation>

```

Figure 4.56 Annotated view of UML association notation.

The shape for a UML class diagram designed in our example is composed of a box that includes class's access and name in the first compartment, class's attributes in the second compartment and operations in the third compartment as is shown in Figure 4.57. In our configuration of classes, associations are also included in the class they start from.

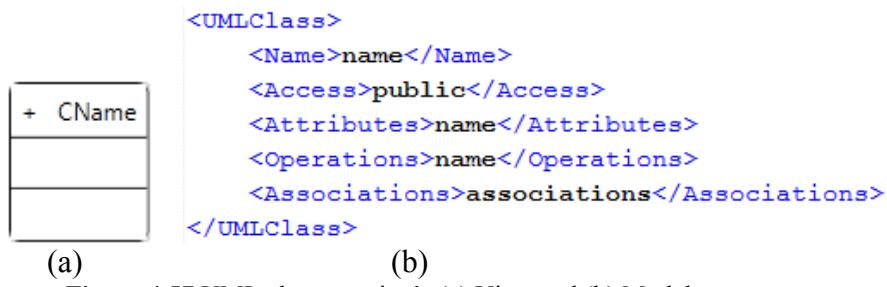


Figure 4.57 UML class notation's (a) View and (b) Model.

Since each class notation includes attributes, operations, and possible associations, the place holders of these elements are provided in class model (Figure 4.57b). These placeholders are then accordingly annotated by “callfor” annotations in the class view’s code shown in Figure 4.58. Class’s name and access are in one to one relationship with their corresponding elements of the view and therefore are marked by “linkto” annotations.

```

<local:UMLClass MinHeight="75" MinWidth="20">
    <local:UMLClass.ClassName linkto="Name" >CName</local:UMLClass.ClassName>
    <local:UMLClass.ClassAccess linkto="Access">public</local:UMLClass.ClassAccess>
    <StackPanel Orientation="Vertical" Background="White">
        <Canvas Background="White" callfor="Associations" />
        <Border BorderBrush="Black" BorderThickness="0,0,0,1" MinHeight="25" >
            <StackPanel Orientation="Vertical" Background="White" callfor="Attributes" />
        </Border>
        <Border MinHeight="25" >
            <StackPanel Orientation="Vertical" Background="White" callfor="Operations" />
        </Border>
    </StackPanel>
</local:UMLClass>

```

Figure 4.58 Annotated view of UML class notation.

Figure 4.59 shows View and Model for diagram notation. This notation is essentially a canvas which allows depiction of diagram name on top and organises class notations using grid layout formation. Classes are place on canvas and ordered according to their appearance order in the input file. Once all classes are placed, diagram canvas triggers class's association rearrangement so that they can link their associations correctly according to position of the association's *To* class. This rearrangement functionality is provided in the class views logic code. Annotated view of this notation is shown in Figure 4.60.

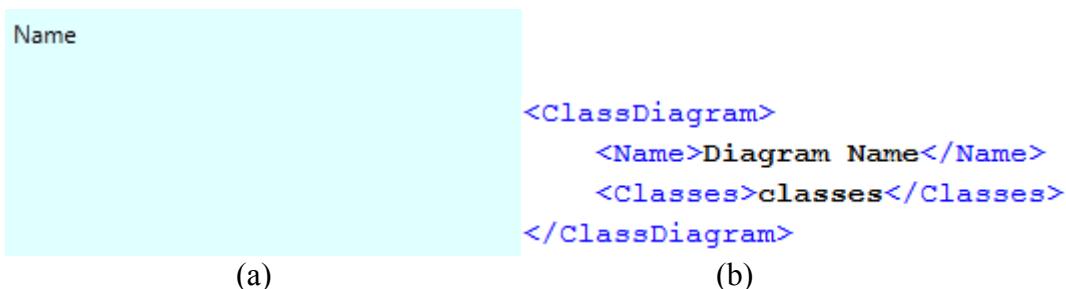


Figure 4.59 UML class diagram notation's (a) View and (b) Model.

```

<local:ClassDiagram>
    <local:ClassDiagram.DiagramName linkto="Name" >Name</local:ClassDiagram.DiagramName>
    <Canvas callfor="Classes"/>
</local:ClassDiagram>

```

Figure 4.60 Annotated view of UML class diagram.

Once required notations are generated, correspondence links between elements of the notation and input elements should be specified. For example for defining notations for UMLClass elements, user drops UML class notation to the designer canvas and drag and drops class element of the input model on the notation as shown by solid arrow in Figure 4.61. This interaction will trigger the creation of a transformation rule for transforming that portion of the input model (UMLClass element in XML input) to the host notation's model.

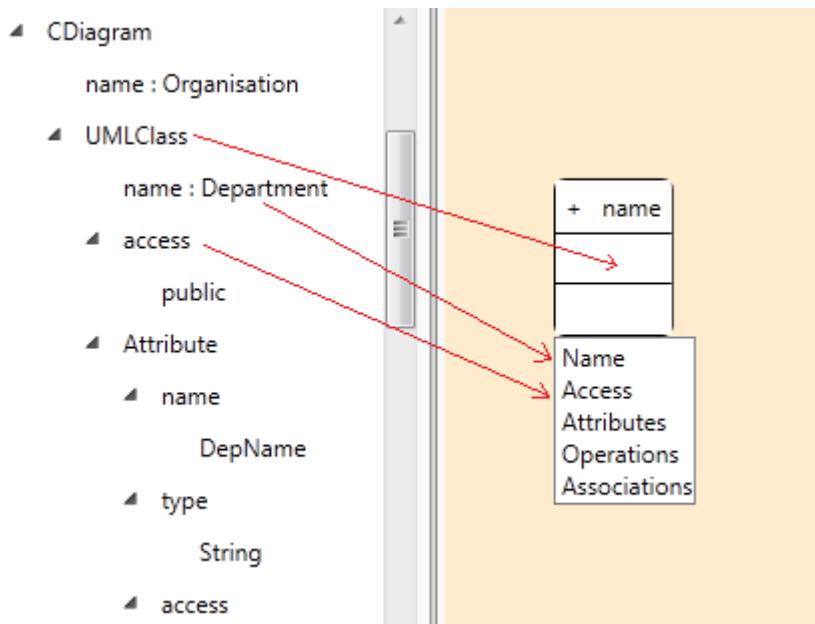


Figure 4.61 Specifying correspondences between class element and class notation. Arrows depict drag and drop direction.

Each notation may have internal elements that correspond to elements of their model data. They can be viewed in a popup window by right clicking on the notation. For example, our UML class notation here has an access identifier, a name, a placeholder for attributes, a place holder for associations and a place holder for operations as its model. These placeholders specify where other notations are going to be included.

Figure 4.61 shows three correspondences: Correspondence between UMLClass element and the notation, correspondence between UMLClass name and class notation's Name, and correspondence between UMLClass's access and class notation's Access. Please

note that since UMLClass's name is a XML attribute and not an element, framework has shown it differently in the default tree view. The place holders are not required to be linked to at this stage. When user is composing notations, other notations will be linked to these place holders. Same procedure will be repeated for other notations. For example Figure 4.62 shows how elements of a function parameter can be linked to their corresponding notation.

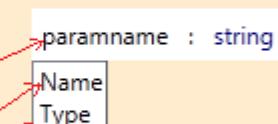
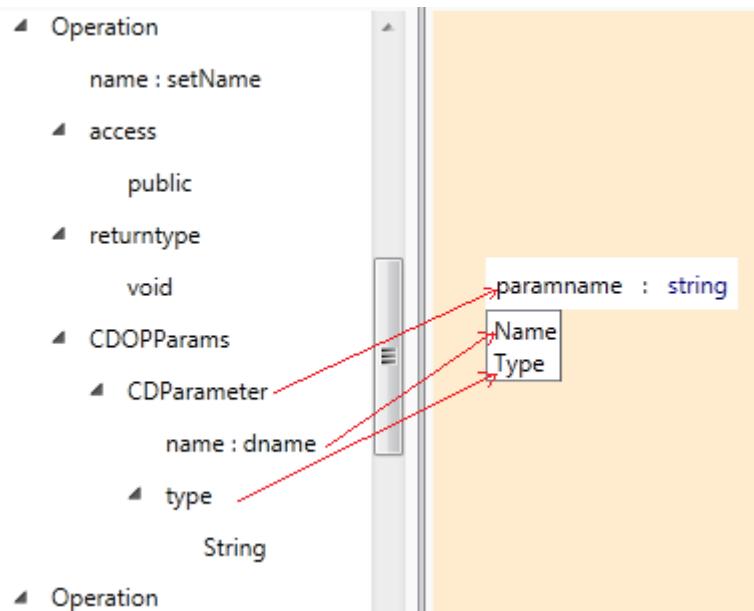


Figure 4.62 Specifying correspondences between function parameters and parameter notation. Arrows depict drag and drop direction.

To have a complete transformation script, the prepared collection of transformation rules in notations should be scheduled according to their call sequence. This is achieved by using notation composition. Once all notations for a visualisation are defined, they should be composed to create a complete visualisation. To do so, user drops all generated customised notations on the scheduling canvas and links them according to their specific place holders as depicted by Figure 4.63.

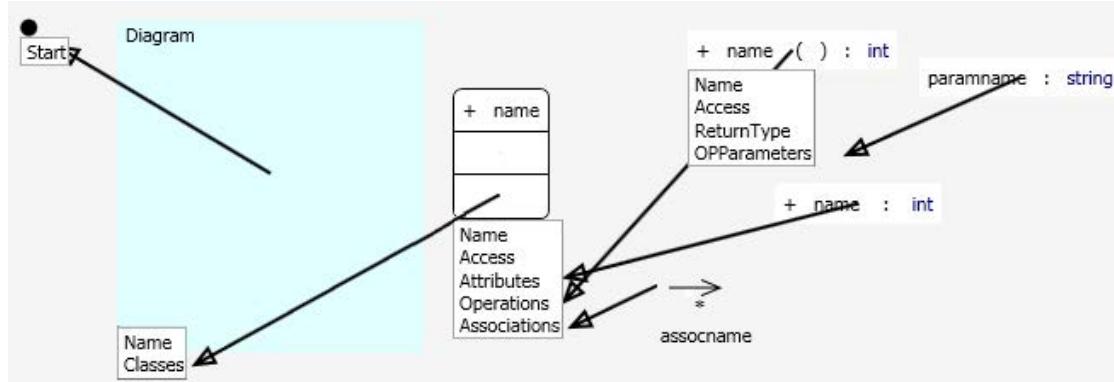


Figure 4.63 Composition of notations to generate Class diagram visualisation.

As previously seen on other case studies, notation composition will result in scheduling of embedded transformation rules in each notation. In composition of Figure 4.63 the transformation rule in diagram notation is the first rule to be called to transform a UMLDiagram model element to a UML diagram notation. It then calls the UML class transformation rule, and the scheduling continues accordingly for other linked notations.

By using the compositions specified in Figure 4.63, a complete XSLT script to generate concrete visualisations of class models will be generated for rendering class model examples similar to Figure 4.48. This transformation script can be reused for other examples. For example Figure 4.64 shows the result of applying same transformation script to an XML input representing class diagram of XYZ airline application.

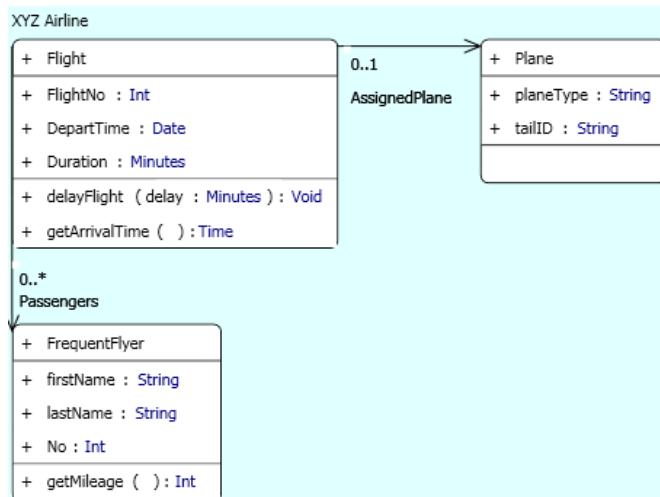


Figure 4.64 Example of class diagram visualisation of XYZ airline.

4.3.4 Java code visualisation

Source code is a type of concrete syntax. However, to be able to benefit from drag and drop and interaction of our approach and framework, it should be visualised using interaction capable notations. This section provides an example of visualising Java source code. Assume XML representations of Java code is available and is similar to example shown in Figure 4.65.

To generate visualisation for this example, a visual notation has to be created for each distinct part of this input model once. For example, to visualise an example Java code XML similar to Figure 4.65, a visual notation for Java package, Classes, attributes, methods and method parameters has to be created first. Similar to previous case studies, we assume a designer designs the views for the required notations. The task for generating the notation then would be to specify model data elements required for these notations and annotate views accordingly to generate the controller transformation between the model and the view.

```

<java>
    <packageName>BankSystem</packageName>
    <Class>
        <name>Account</name>
        <access>public</access>
        <Attribute>
            <name>owner</name>
            <type>string</type>
            <access>public</access>
            <multiplicity>1</multiplicity>
        </Attribute>
        <method>
            <access>private</access>
            <returntype>bool</returntype>
            <name>deposit</name>
            <params>
                <parameter>
                    <type>double</type>
                    <name>amount</name>
                </parameter>
            </params>
        </method>
    </Class>
</java>

```

Figure 4.65 An example of Java code input represented by XML.

For example, to generate a notation for Java properties, the provided view and the required model data are depicted by Figure 4.66. This notation by default includes the array indicators (“[]”). The logic behind notation controls the value to be put inside the brackets. If no value is provided, or the multiplicity of the property is one, it will omit the brackets.

```

<Field>
    <Access>public</Access>
    <Name>field</Name>
    <Type>int</Type>
    <Multiplicity>1</Multiplicity>
private string [] myfield ;      </Field>
(a)                                (b)

```

Figure 4.66 Java property's (a) View and (b) Model.

Elements of the View should be linked to elements of the model. It can be done by annotating the View according to elements provided by Java property's model. This annotation is depicted in Figure 4.67.

```
<local:JavaField Background="White">
  <local:JavaField.FieldName linkto="Name" >myfield</local:JavaField.FieldName>
  <local:JavaField.FieldAccess linkto="Access" >private</local:JavaField.FieldAccess>
  <local:JavaField.FieldType linkto="Type" >string</local:JavaField.FieldType>
  <local:JavaField.Multiplicity linkto="Multiplicity" >1</local:JavaField.Multiplicity>
</local:JavaField>
```

Figure 4.67 Java property View's annotations.

Other notations can also be generated similar to Java property. For example Java class notation's view and model are provided in Figure 4.68 and its annotated View is depicted in figure 4.69. Since Java class notation includes other notations like attributes and functions, their place holders are provided in the model data and are accordingly annotated in the View (see Figures 4.68 and 4.69).

```
public class MyClass
{
  <JavaClass>
    <Access>public</Access>
    <name>class</name>
    <properties>props</properties>
    <methods>methods</methods>
  </JavaClass>
  (a) (b)
  <Figure 4.68 Java class's (a) View and (b) Model.>
```

```

<StackPanel>
    <StackPanel Orientation="Horizontal">
        <Label Foreground="Navy">
            <!--> <Label.Content linkto="Access">public</Label.Content>
        </Label>
        <Label Content=" class " Foreground="Navy" />
        <Label Foreground="LightBlue">
            <!--> <Label.Content linkto="name">MyClass</Label.Content>
        </Label>
    </StackPanel>
    <Label>{</Label>
    <StackPanel Orientation="Vertical" callfor="properties" />
    <Label />
    <StackPanel Orientation="Vertical" callfor="methods" />
    <Label>}</Label>
</StackPanel>

```

Figure 4.69 Java class view annotations.

The generated notations should be linked to example data by specifying correspondence links between elements of the input examples and elements of notation data. For example, to generate a visualisation for a Java class, a Java class notation has to be placed (dropped) on the designer canvas and class element of the input model should also be dropped on the notation as shown by arrows in Figure 4.70. This interaction will trigger the creation of a transformation rule for transforming that portion of the input model (class element in XML input) to the host notation's model. The internal elements should also be mapped accordingly. For example, class notation has access, name that should be specified. Placeholders for properties and methods will be used in notation composition step.

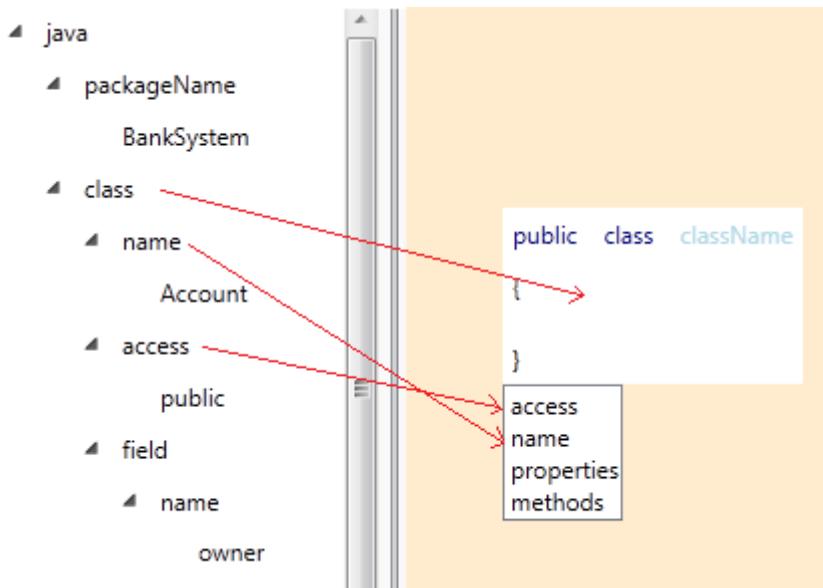


Figure 4.70 Mapping Java class input elements to Java notation.

As another example, consider mapping elements of the input XML representing method parameters to parameter notation. Figure 4.71 shows the interactions and correspondences required for this mapping.

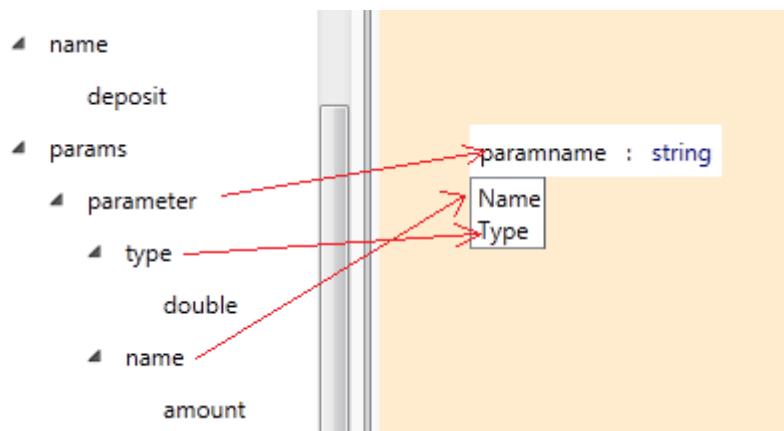


Figure 4.71 Java class View's annotations.

A range of “mapping functions” are available to be used to manipulate content from source to target visualisation. For example, if it is required to alter the name of Java class by appending a “_Class” to its name, a string merging function (marked by a in

Figure 4.72) can be used. These mapping functions are used similar to the notations, i.e. they can be dropped on the designer canvas, and desired input elements can be linked to their internal elements (i.e. function's input arguments) by drag and drop. If a function has outputs, they can be dragged to other desired elements of notation.

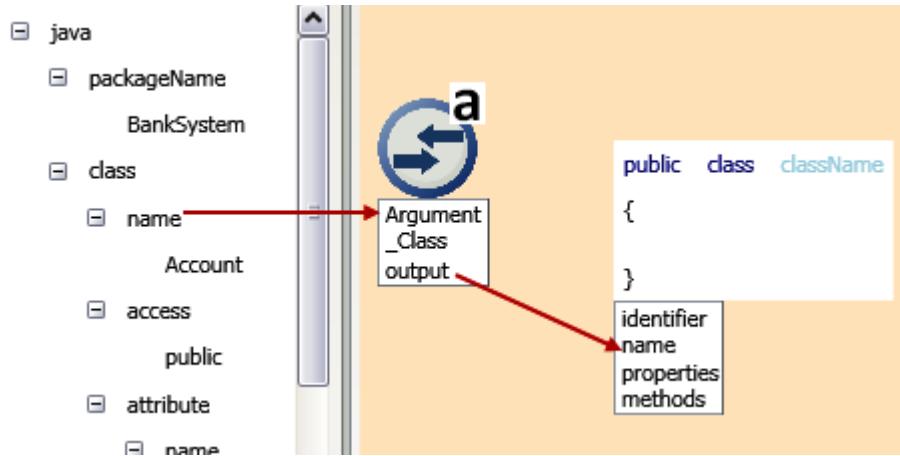


Figure 4.72 Using string merge function to alter Java class's name. Arrows depicts drag and drop.

To have a complete Java visualisation (and hence a transformation script from Java XML to visualisation), the prepared collection of notations should be composed. Once all notations for Java visualisation are defined, they should be dropped on scheduling canvas (provided in the framework) and linked according to their specific place holders as depicted by Figure 4.73.

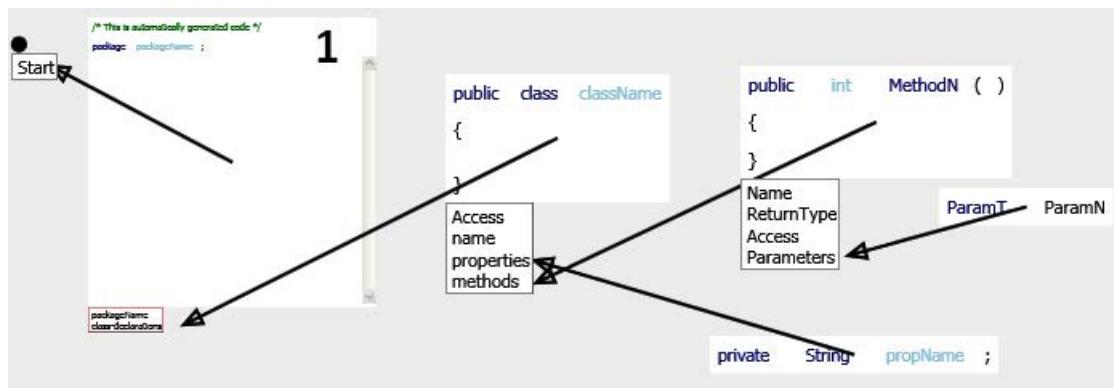


Figure 4.73 Composing notations to generate Java code visualisation. Arrows are provided by framework.

According to this composition, Java package notation's (marked by 1 in Figure 4.73) internal transformation is the first to be called to transform a Java package model element to a package notation. It then calls the class transformation rule, and the scheduling continues accordingly for other linked notations. Applying the resulted transformation on the example of Figure 4.65 will result in visualisation of Figure 4.74.

```
/* This is automatically generated code */

package BankSystem ;
public class Account
{
    public string owner ;

    private bool deposit ( double amount )
    {
    }
}
```

Figure 4.74 Resulted visualisation of the example in Figure 4.65.

Figure 4.75 shows an example of applying this transformation on another example representing Java source XML of an airline application.

```

/* This is automatically generated code */

package XYZ_Airline ;
public class Flight
{
    public int FlightNo ;
    public date DepartTime ;
    public Minutes Duration ;
    public FrequentFlyer [] Passengers ;
    public Plane AssignedPlane ;

    public void delayFlight ( minutes delay )
    {
    }

    public time getArrivalTime ( )
    {
    }
}

public class Plane
{
    public string planeType ;
    public string tailID ;
}

public class FrequentFlyer
{
    public string firstName ;
    public string lastName ;
    public int No ;

    public int getMileage ( )
    {
    }
}

```

Figure 4.75 Example Java code visualisation of airline application.

4.3.5 Computer Aided Design (CAD) visualisation

Visualisations have been used in variety of application domains. One domain that has benefited a lot from visualisation is Computer Aided Design (CAD). CAD visualisations allow designers to see various parts of their designs and correct imperfections before building them. This case study shows how CAD examples can be provided to our framework and visualised. Here we assume these examples are provided in XML files similar to example of Figure 4.76. It shows an example of CAD design data of a building. It is composed of plan levels that may include rooms of different types. For simplicity, the geometry data has been merged into one element. However, it could have been provided in different elements or even input files.

```
<CADData>
  <Name>New Green Building</Name>
  <Plan>
    <Name>Living Area</Name>
    <Shape>
      <Geometry>M 0,0 L 100,0 L 100,80 L 85,100 M 100,105 L 0,105 L 0,0</Geometry>
      <Name>Open Kitchen</Name>
      <Type>Kitchen</Type>
    </Shape>
    <Shape>
      <Geometry>M 0,0 L 100,0 L 100,80 L 85,100 M 100,105 L 0,105 L 0,0</Geometry>
      <Name>Washing Area</Name>
      <Type>Toilet</Type>
    </Shape>
    <Shape>
      <Geometry>M 0,0 L 100,0 L 100,80 L 85,100 M 100,105 L 0,105 L 0,0</Geometry>
      <Name>Room 1</Name>
      <Type>BedRoom</Type>
    </Shape>
  </Plan>
...
</CADData>
```

Figure 4.76 Example input model of a CAD XML.

Three notations are required for this case study, a notation for room shapes, a notation for floor plans and a notation for whole building. Similar to previous case studies, a designer has provided the views for these notations. These notation views are generated using XAML graphics that unlike other case studies do not include logic code. The layout of the graphics used in these views is controlled by XAML controls. Figure 4.77

shows the view and model for room notation. Each room has geometry, name, and a type. Therefore, the provided model should include these elements.

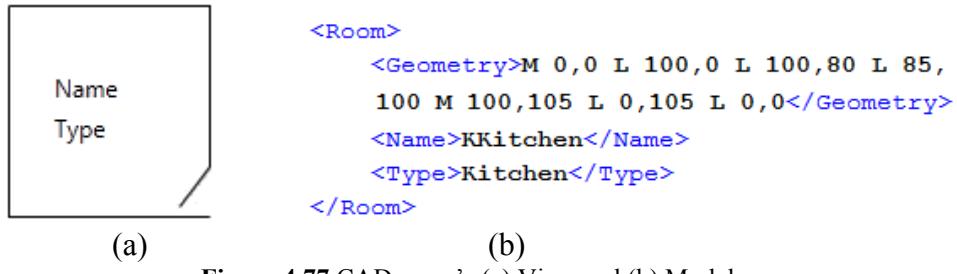


Figure 4.77 CAD room's (a) View and (b) Model.

Since all the elements of this room model are in one to one relationship with the elements of the view, they should be annotated in the view code using “linkto” annotations. Figure 4.78 shows the annotated view.

```

<Canvas Width="20" Height="20" Background="White" >
    <Path Stretch="Fill" Fill="White" Stroke="Black" >
        <Path.Data linkto="Geometry">M 0,0 L 100,0
        L 100,80 L 85,100 M 100,105 L 0,105 L 0,0</Path.Data>
    </Path>
    <Label Canvas.Top="30" Canvas.Left="20">
        <Label.Content linkto="Name">Name</Label.Content>
    </Label>
    <Label Canvas.Top="50" Canvas.Left="20">
        <Label.Content linkto="Type">Type</Label.Content>
    </Label>
</Canvas>

```

Figure 4.78 Annotated view of CAD room notation.

Floor plans include the rooms and show how they are arranged. They also have a label to identify each floor. The view and model of a floor plan is depicted by Figure 4.79.

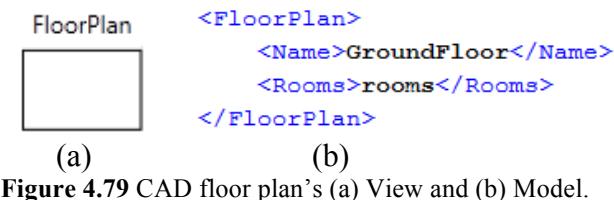


Figure 4.79 CAD floor plan's (a) View and (b) Model.

Since each floor plan houses multiple rooms, the annotation for rooms should reflect the one-to-many relationship. Figure 4.80 shows the annotated view of Figure 4.79. By default, each floor plan lists included rooms from top.

```

<StackPanel Orientation="Vertical" >
  <Label>
    |   <Label.Content linkto="Name" >FloorPlan</Label.Content>
  </Label>
  <Border BorderBrush="Black" BorderThickness="1,1,1,1">
    |   <StackPanel Orientation="Vertical" callfor="Rooms">
    |   </StackPanel>
  </Border>
</StackPanel>

```

Figure 4.80 Annotated view of CAD floor plan notation.

The final notation for our CAD visualisation is the CAD design notation which embeds other notations. Its view includes a stack panel embedded in a scroll viewer for housing multiple floor plans. So in case the design become large, users can scroll to see the whole design and it will not interfere with other elements of UI. The view and model of this notation and the annotated view are depicted by Figure 4.81 and Figure 4.82.

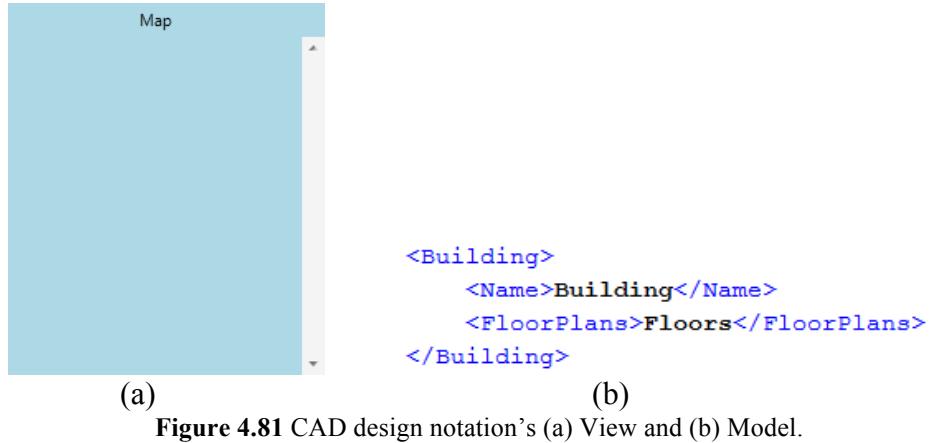


Figure 4.81 CAD design notation's (a) View and (b) Model.

```

<StackPanel Background="LightBlue" Orientation="Vertical" >
  <Label HorizontalAlignment="Center">
    <Label.Content linkto="Name" >Map</Label.Content>
  </Label>
  <ScrollViewer MinHeight="250" MinWidth="250" >
    <StackPanel Orientation="Vertical" callfor="FloorPlans">
      </StackPanel>
    </ScrollViewer>
  </StackPanel>

```

Figure 4.82 Annotated view of CAD design notation.

Next step is to map input elements to generated notations. This can be done by drag and dropping input elements on notational elements. For example, Figure 4.83 shows how elements of a shape from input model can be linked to a room notation. When room notation is drop on designer canvas, user drags the shape element on room notation. This will trigger a transformation rule for transforming that portion of the input model (Shape element in XML input) to the room notation's model. The internal elements of the shape and room notation's model should be linked by drag and drop too. For example, the room notation in Figure 4.83 has a Geometry, Name and Type. These internal elements can also be linked by drag and dropping elements as shown by arrows in the figure. These correspondences will be included in the transformation rule template that has been triggered.

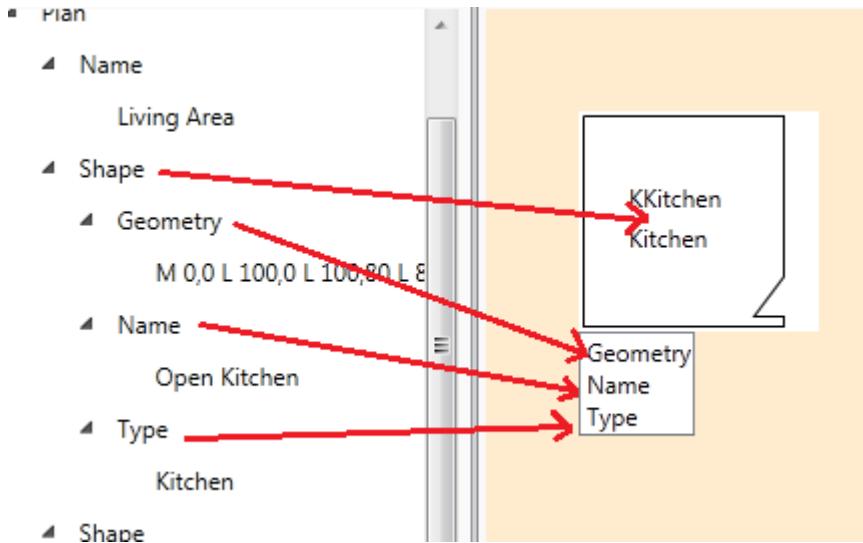


Figure 4.83 Correspondence specification between shape element and room notation.

Figure 4.84 shows another example of linking input model elements to a floor plan notation to transform each plan in input model to a floor plan notation. The **Rooms** element of floor plan notation will be used in notation composition as it is a placeholder for room's notation.

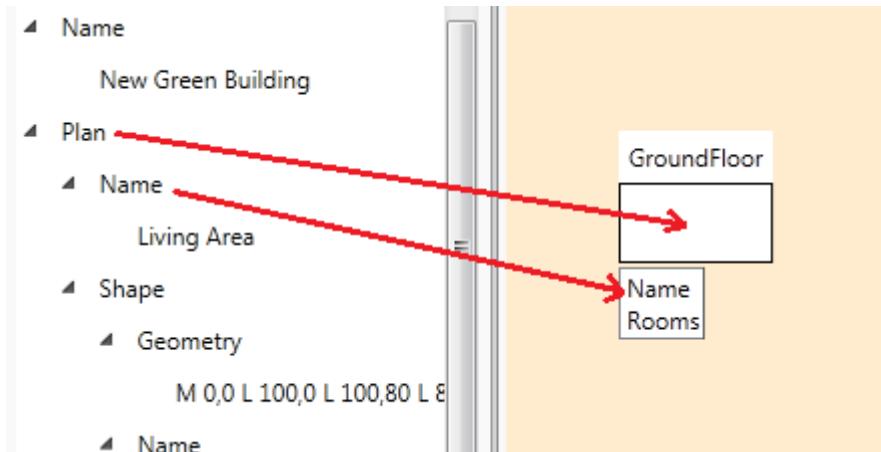


Figure 4.84 Correspondence specification between plan data and floor notation.

To have a complete visualisation and hence a model to visualisation transformation script, the prepared collection of defined notation rules should be scheduled according

to their call sequence. Composition of notations specifies how a full model visualisation is formed from a set of sub-element visualisations and as a result, in this example, notations are composed as depicted by Figure 4.85. This figure also shows how place holders help identification of which element should be included in which notation. These place holders will be replaced by calls to transformation rules of linking notations. For example in composition of Figure 4.85 since the room notation is linked to “Rooms” element of floor notation, a call to the transformation rule embedded in the room notation will be placed in floor notations “Rooms” element.

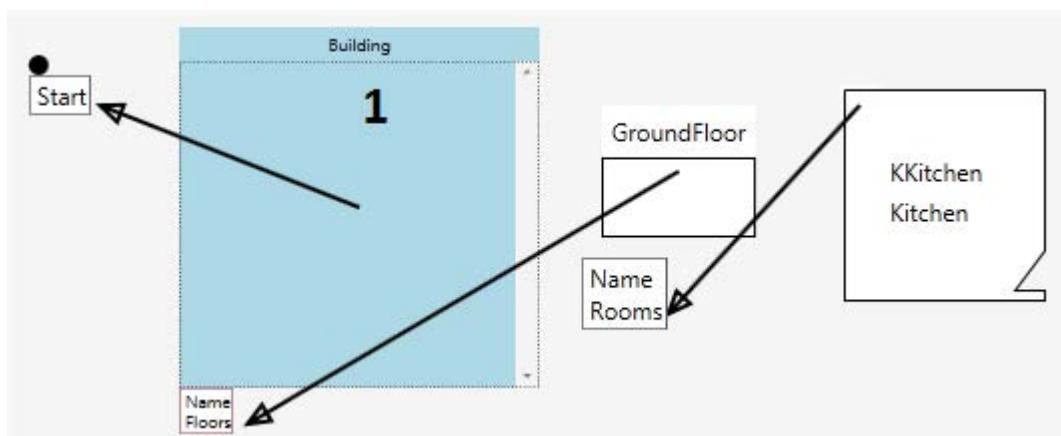


Figure 4.85 Composition of notations for CAD visualisation.

This composition results in the scheduling of model element-to-visual notation transformation rules and thereby a model to visualisation transformation script will be generated. For example, by using the compositions specified in Figure 4.85 a complete XSLT script to generate concrete visualisations of CAD models will be generated for rendering those model examples to visualisations similar to the visualisation of Figure 4.86. Note that the generated XSLT transformation script can be reused and applied to all examples of the CAD input to provide an automatic concrete visual notation renderer. These generated concrete 2D visualisations are implemented as WPF elements and allow interaction with their composing notations. The individual elements of a concrete visualisation can be dragged, dropped on other elements, and right clicking them reveals their internal elements.

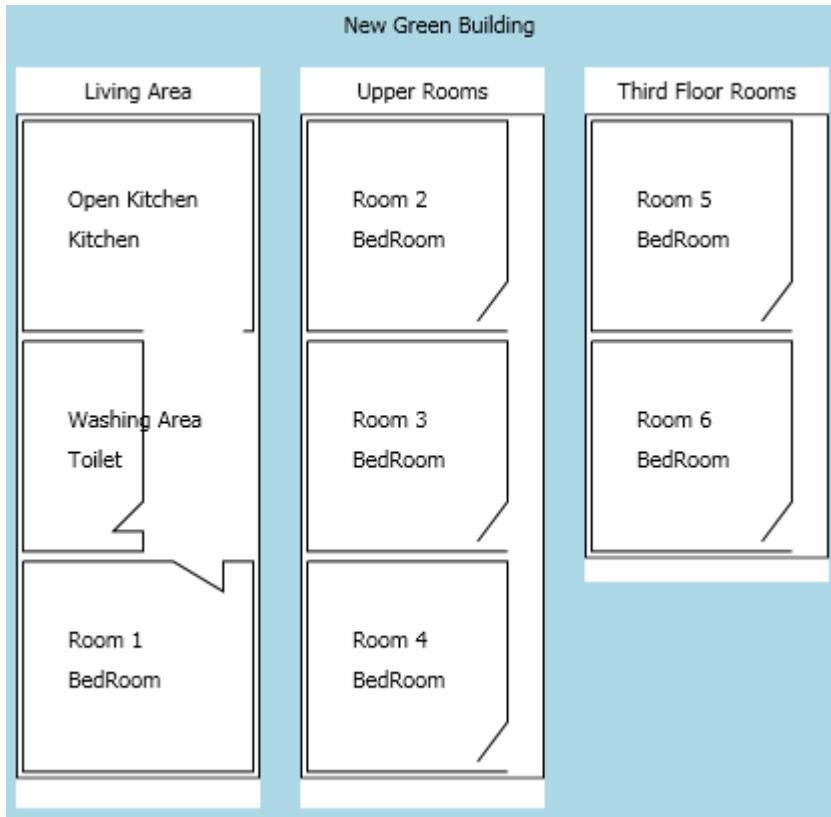


Figure 4.86 Example of the generated CAD visualisation.

4.4 Summary

This chapter described the visualisation approach provided in by this thesis. It described how visual notations are created using designer provided views and linked to model data using a controller transformation. The created notations are capable of being drag and dropped and elements of input models can be linked to them and their internal model elements using drag and drop approach. Once notations are generated and linked to input model examples, they should be composed to generate full visualisations. Composition of notations results in generation of model to visualisation transformation script and can be reused and applied on similar examples to generate visualisations.

Five case studies were provided in this chapter to indicate applicability of the presented approach for varieties of application domains. These case studies included visualisation generation for bar charts, a recreation of Minard's map, UML class diagrams, Java code and CAD designs.

Chapter 5

Transformation using concrete visualisations

5.1 Introduction

A major motivation for this thesis research was to address complexity of model transformation generation by providing familiar and concrete visual notations as first class artefacts in transformation generation process. These concrete visualisations help to better incorporate users' domain knowledge into specifications of source and target model correspondences and thus into generated model transformation rules.

Chapter 4 described the creation of such concrete visualisations in detail. This chapter introduces ways in which these visualisations can be used in transformation rule specification using drag and drop between visualisations of example models. Low level model transformation scripts are automatically generated using these drag and drop interactions. This chapter addresses research question 2 and its following sub-questions:

2. Can a model transformation be effectively generated using concrete by-example visualisations?
 - 2.1. Can we perform correspondence specification (and hence transformation specification) on actual visual notation of input models?
 - 2.2. Can a transformation rule be represented visually?
 - 2.3. How to create a visualisation for transformation rules?

5.2 Transformation approach

Given that source and target visualisations are available, the transformation procedure between the two visualisations, as illustrated in Figure 5.1, involves: 1) Mapping notations of source visualisation to target visualisation to create transformation rules, 2) automatic reverse engineering of a meta-model (abstraction) from source and target visualisations, and 3) automatic generation of transformation script using defined rules and the reverse engineered abstraction. The following subsections describe each of these steps in detail.

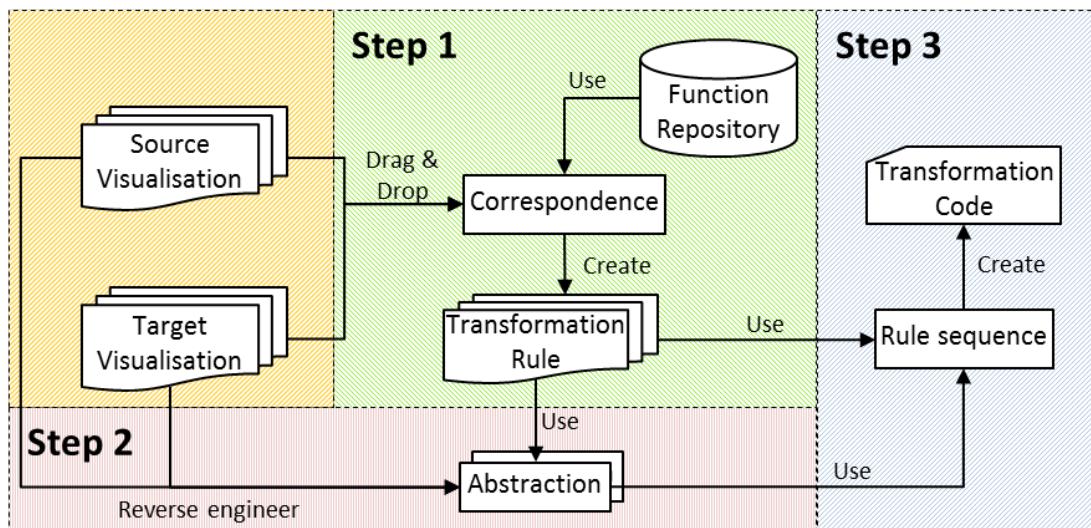


Figure 5.1 Transformation generation procedure.

5.2.1 Transformation rule specification

Concrete and familiar visualisations provide better facilities for spotting correspondences between source and target model visualisations. First, let us revisit definition of correspondence in our approach:

Definition 5.1 A *Correspondence* is a relation between elements on both sides of the transformation. It specifies whether element(s) of the Left Hand Side (LHS) model play a role in deciding element(s) of the Right Hand Side (RHS) model.

A correspondence can be 1-to-1, 1-to-Many or Many-to-Many depending on number of elements participating in the relationship from both sides. These correspondences can specify a direct relationship between LHS and RHS models or an indirect relationship. Therefore we have:

Definition 5.2 A *Direct correspondence* defines a direct relationship between LHS² element and the RHS element. It usually results in the value of the LHS element being copied to the RHS element.

Definition 5.3 An *Indirect correspondence* is the relationship between LHS and RHS model elements that is specified through functions and conditions or other correspondences.

Example 5.1 Consider transformation example of bar chart to pie chart visualisation. Assume each bar has a name, a value and a colour, and each pie has a name, value and colour accordingly. Three 1-to-1 direct correspondences between elements of each bar and elements of each pie piece can be specified, as shown by Figure 5.2.

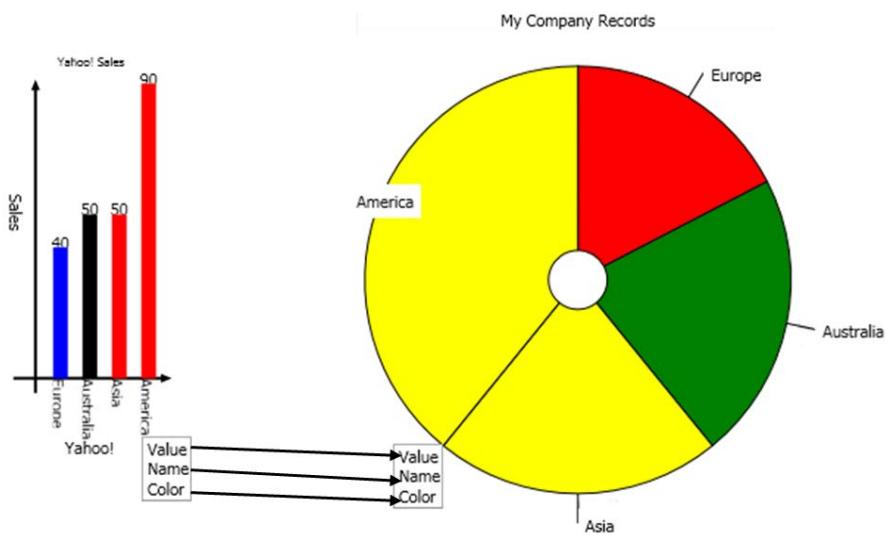


Figure 5.2 One-to-One correspondences between elements of a bar in bar char and elements of pie pieces in a pie chart. Arrows depict correspondences.

² From this point forward, we might use LHS model and source model interchangeably. Same is true for RHS model and target model.

Example 5.2 Consider transformation of a troop movement notation in Minard's map to a pie piece in a pie chart. Assume each troop movement notation has number of troops when the movement started in the first city, number of troops when it ended and a colour representing the status (advancing/retreating). The pie chart on the RHS is representative of the number of troops lost during the movement and the status of the campaign and the name of the two cities involved in the movement. As a result, to specify the transformation between the two notations one can specify following three correspondences (as shown by figure 5.3): 1) Direct correspondence between troop movement colour and pie piece colour. 2) Indirect correspondence between the number of troops in first and second cities, and the value of the pie piece. A subtraction function has to be used to calculate the number of troops lost during the movement. 3) Indirect correspondence between name of the two cities and name of the pie piece. A merging function is used to merge the two names.

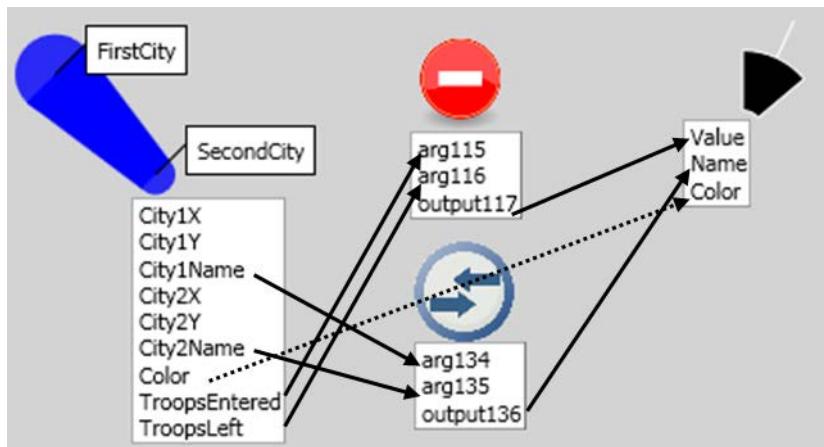


Figure 5.3 Correspondences between elements of a troop movement notation in Minard's map and elements of a pie piece in pie chart. Solid arrows depict indirect correspondences while dashed arrow depicts direct correspondence.

From these definitions, it can be concluded that a direct correspondence will result in a 1-to-1 mapping, but the opposite is not necessarily true. For example, an element on the LHS might have relation to another element on the RHS through a function.

Defining correspondences is very similar to structural programming paradigms, i.e. some correspondences may affect a larger portion of the source or target models or both and therefore, might include other correspondences. As a result, two categories of correspondences can exist. We call them *Child* and *Parent* correspondences.

Definition 5.4 A *Child correspondence* is a direct or indirect correspondence that specifies a relationship by its own.

Definition 5.5 A *Parent correspondence* is a correspondence that includes set of other correspondences. This set should have at least one child correspondence.

Example 5.3 In mapping a bar chart visualisation to a pie chart of Example 5.1, each bar on the source (bar chart) has parent correspondence relation with each pie piece in the target (pie chart). The value of the bar, its name and colour, have child correspondence relationships with corresponding value, name and colour of the pie pieces. Figure 5.4 depicts these correspondences.

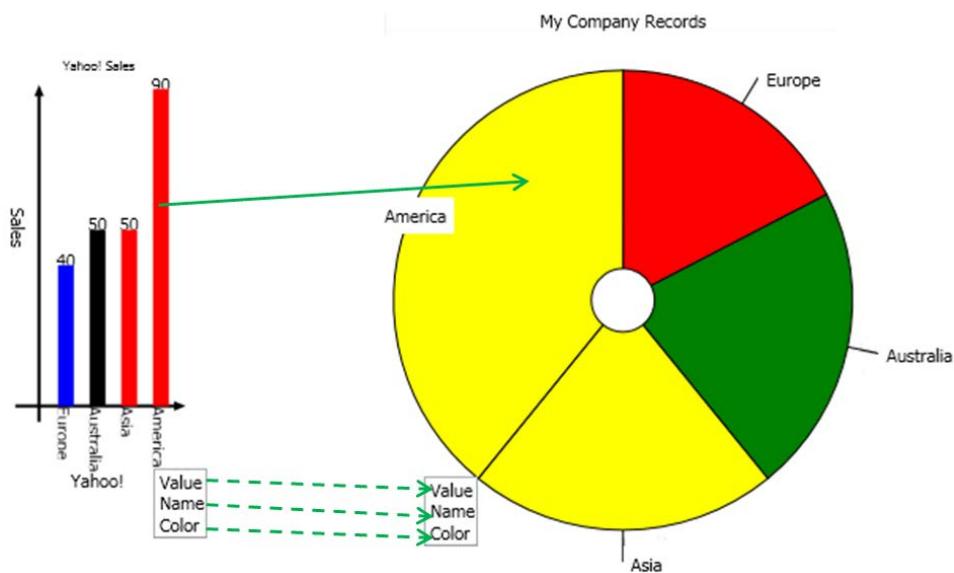


Figure 5.4 Correspondences between a bar in bar char and a pie piece in a pie chart. Solid arrow depicts parent correspondence while dashed arrows depict child correspondences.

With these two categories, it would also be easier to reuse correspondences. Considering these definitions, a transformation rule can be defined as follows.

Definition 5.6 A *transformation rule* in our approach is defined by a parent correspondence and may include a set of child correspondences, operations and calls to other transformation rules. It can include both direct and indirect correspondences and may include additional operations.

Applying transformation rules on (part of) source model will result in generation or modification of (part of) target model. A transformation rule might be called multiple times and might be applied on multiple sources to result target(s).

Example 5.3 The parent correspondence of Figure 5.4 represents a transformation rule that transforms a bar in bar chart to a pie piece in pie chart. It includes three child correspondences. Another transformation rule which transforms bar chart to pie chart will call this rule multiple times for each bar to create pie pieces in pie chart.

Figure 5.5 shows another example from software engineering domain. A transformation rule can be defined by the parent correspondence between UML class diagram's attribute and a Java class property. This parent correspondence includes three child correspondences to correspond Name, Access and Type.

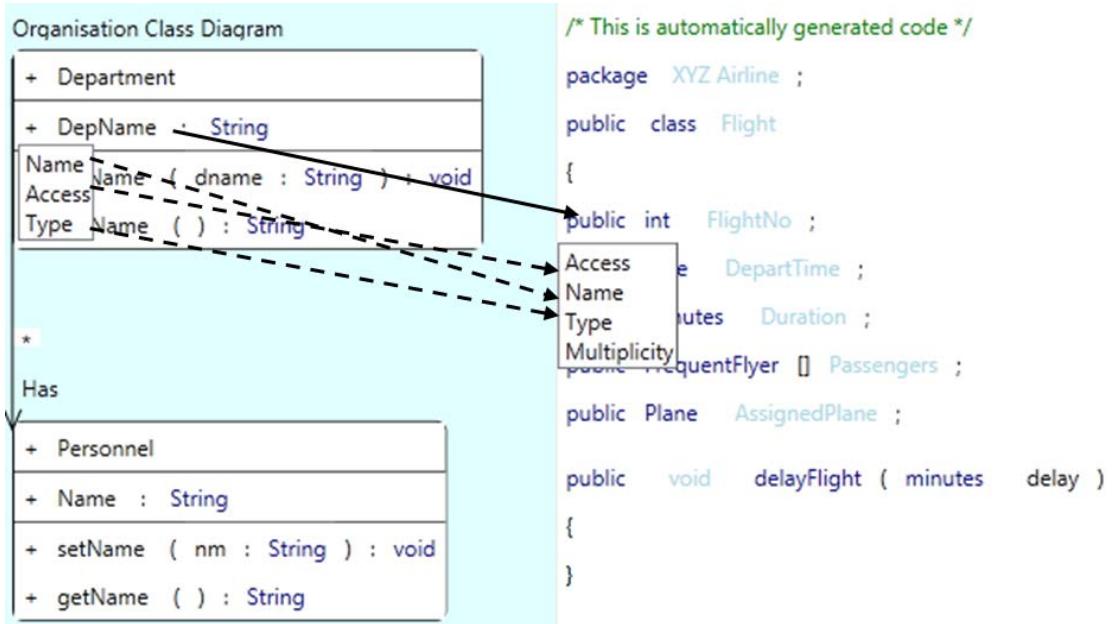


Figure 5.5 Correspondences between a UML attribute and a Java property. Solid arrow depicts parent correspondence while dashed arrows depict child correspondences.

Transformation rules range from simple direct correspondences to complex set of mixed correspondences and transformation rule calls. A variety of functions can be used to define these correspondences depending on the transformation task at hand. Similar to model-to-visualisation transformation generation step, varieties of functions and conditions can be specified to define more complex correspondences and hence transformation rules. These functions and conditions are used similar to visualisation step. Example 5.4 demonstrates use of a condition in mapping a bar notation to pie piece notation.

Example 5.4 Figure 5.6 demonstrates using a condition to specify colour of a pie piece according to the values provided by bar notations in a bar chart. Value of the bar is dragged to arg1 and the value to be checked against is provided to arg2. Then specific colours are provided in condition statements accordingly. The condition should then be dragged to the colour element of the pie piece.

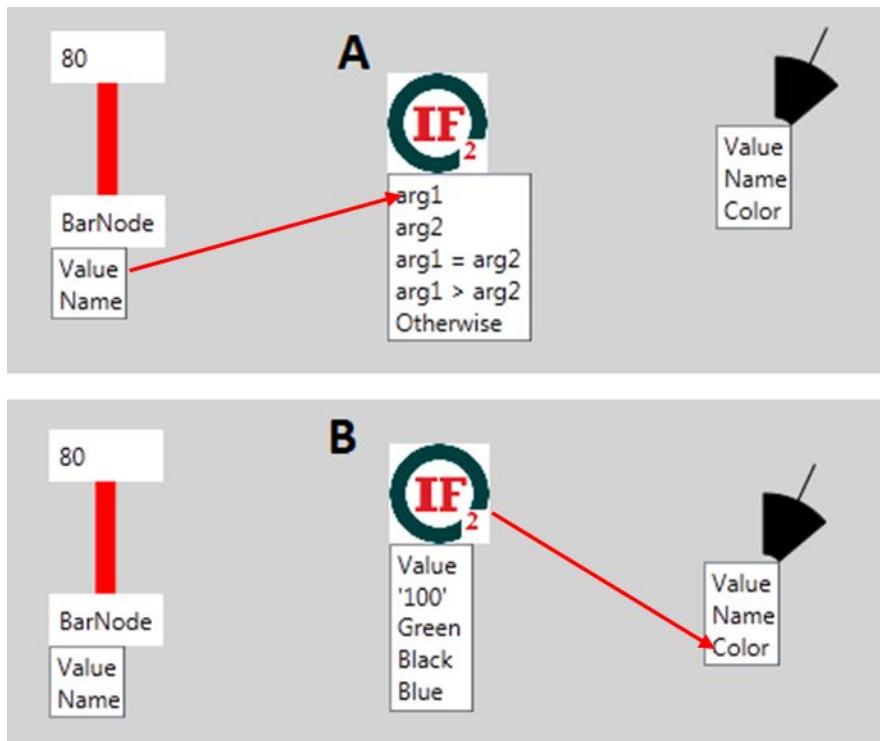


Figure 5.6 Using conditions to specify correspondences. A) Before values are specified to the condition arguments, B) after values are provided. Arrows show drag and drop directions.

5.2.2 Transformation rule representation

Transformation rules are inseparable part of any model transformation system. A complete transformation specification usually consists of combination of multiple transformation rules. If transformation specification involves large models, many such rules will be defined which can affect understandability of the process and debugging. Given that first class artefacts of our approach are visual notations, we argue that a textual representation of a transformation rule would be out of place and not suitable. Therefore, transformation rules are represented by visual notations too (see our research questions 2.2 and 2.3 on visual representation of transformation rules).

Each transformation rule is represented by the source and target notations it is representing. Putting each visual element and its target notation together will provide a schematic view of the transformation rule, i.e. users can see each transformation rule by the source, and the target notation that will be generated as a result of applying the rule on source notation. This provides a good mechanism for representing transformation rules visually. Figure 5.7 demonstrates examples of these rule representations.

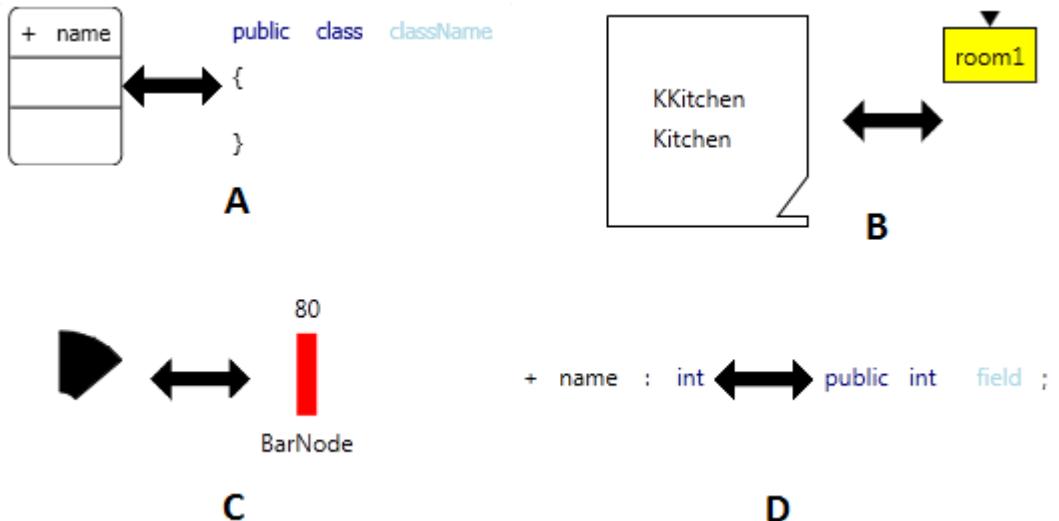


Figure 5.7 Examples of transformation rule representation. Transformation rules are: A) UML class to Java class, B) A Room in 2D visualisation to a room notation in another 2D visualisation, C) A pie piece to bar and D) UML attributes to Java property.

5.2.3 Generating transformation scripts

Transformation rules are defined using visual notations and their visual representations. Each transformation rule transforms the model underlying the source notation to the model of target notation. The MVC embedded in each notation is responsible for depicting elements of notation's model to the visual view. As a result, the transformation between notations only considers the model data.

By dragging and dropping a source notation to a target notation, their underlying models are used as templates for transformation rules. The child correspondences that represent the internal model elements of the two notations will be included in the transformation templates to form a complete template. Example 5.5 elaborates more on the procedure.

Example 5.5 Consider the transformation rule between UML attribute and Java property. Figure 5.8 demonstrates two steps of the required interaction. Step one: a UML attribute is dropped on a Java property. This will result in assignment of the underlying model of Java property as target and internal elements of forward transformation template. The pseudo code provided in Figure 5.9 shows the resulted template. The internal elements of these notations need to be mapped as

well. Therefore next step (marked by 2 in Figure 5.8) specifies the correspondence between *Name* of the UML attribute and *Name* in Java property. This correspondence will be reflected in the triggered transformation rule as shown by Figure 5.10. The remaining correspondences will be accordingly specified and reflected.

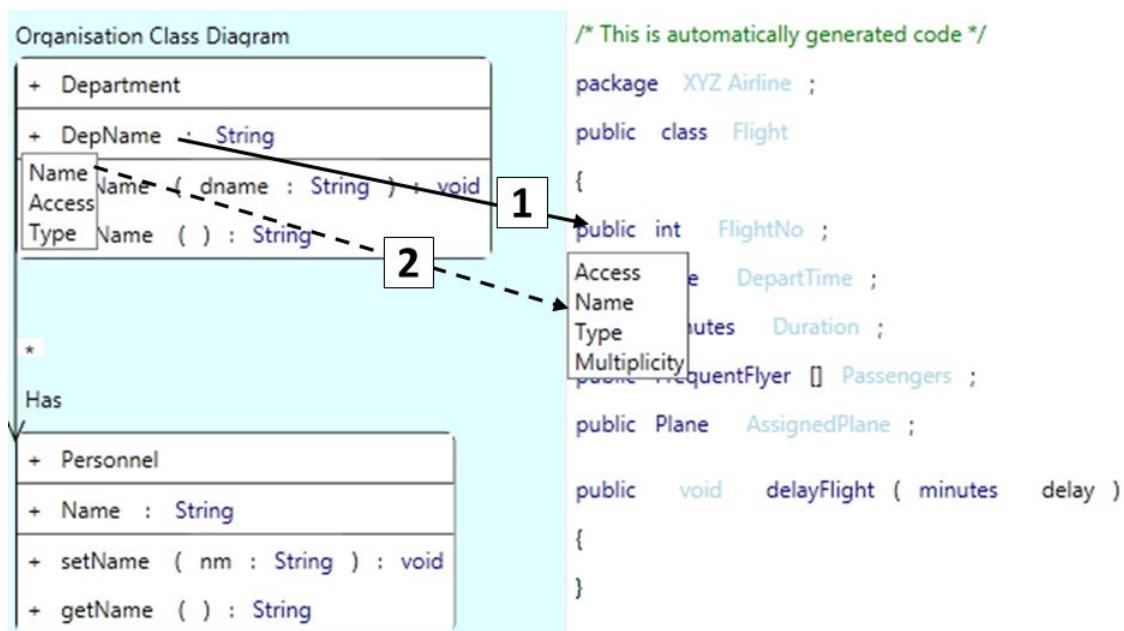


Figure 5.8 Steps for generating transformation rule between UML class attribute and Java property.

Transform UML attribute to {

Java Property

Internal model:

{

Access

Name

Type

Multiplicity

}

}

Figure 5.9 Pseudo code representing the transformation template of step one in Figure 5.8.

```

Transform UML attribute to {
  Java Property
  Internal model:
  {
    Access
    Map UML attribute's Name To Java Property's Name
    Type
    Multiplicity
  }
}

```

Figure 5.10 Pseudo code representing the transformation template after step two in Figure 5.8.

Once correspondence specification between two notations is complete, a transformation code generator reads these templates and generates the transformation rule scripts according to the transformation languages of choice. Example 5.6 demonstrates a sample of the generated transformation code.

Example 5.6 Consider the transformation rule defined in Example 5.5. Assuming that all elements of UML attribute (Name, Access, and Type) are mapped to their corresponding elements in Java property, the transformation rule generated by a code generator that generates XSLT would be similar to Figure 5.11. Note that since there exists no correspondence for Java property's multiplicity, its default value is chosen from Java property's notation data.

```

<xsl:template match="UMLAttribute" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <Field>
    <Access>
      <xsl:apply-templates select="Access" />
    </Access>
    <Name>
      <xsl:apply-templates select="Name" />
    </Name>
    <Type>
      <xsl:apply-templates select="Type" />
    </Type>
    <Multiplicity>1</Multiplicity>
  </Field>
</xsl:template>

```

Figure 5.11 Transformation rule script for transforming UML attribute to Java property in XSLT.

In case other transformation languages are of interest, code generators for those languages can be provided similar to the XSLT code generator used in Example 5.6. A complete visualisation to visualisation transformation may include multiple transformation rules. These transformation rules will be included in the transformation script to transform source visualisation to target visualisations.

Unlike the model to visualisation transformation which follows an imperative transformation rule control, the rule application control of visualisation to visualisation transformation follows a declarative approach. For model-to-visualisation transformation, the user would specify how model-to-visualisation rules are called and scheduled through composition of notations. Here, for visualisation-to-visualisation transformation rules are not required to be explicitly called. Instead a call to apply possible rules will be placed and the transformation engine will choose the possible transformation rules. This declarative approach allows transformation rules to be called for the composing visual notations inside each notation without user needing to explicitly specify them.

Example 5.7 In bar chart to pie chart transformation example, when mapping chart areas, users define three correspondences as shown by Figure 5.12. First correspondence defines the transformation rule templates for transforming bar chart notation to pie chart notation. Second correspondence defines the internal elements of the bar chart (its Name and Bars elements) to be transformed to internal elements of the pie chart. The Bars element of the bar chart includes other bars. As a result, if the bar to pie piece rule has been defined, the call for templates that has been put inside Bars element as a result of the correspondence, will result in calling bar-to-pie piece rule.

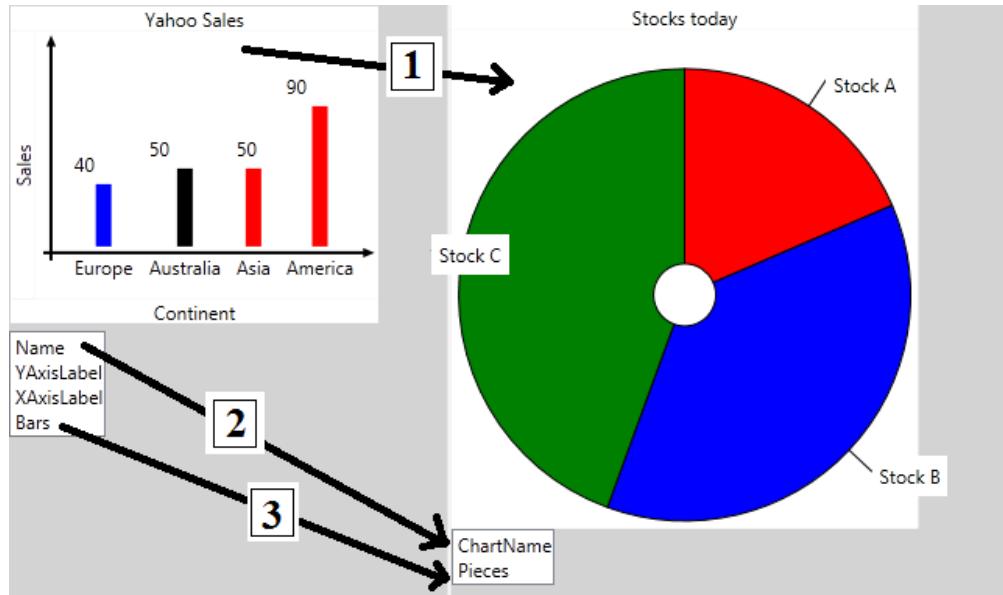


Figure 5.12 Transformation rule specification between bar chart and pie chart. Arrows depict drag and drop.

Note that the order of defining rules is not important. Users can define rules in order they wish. Once all rules are defined, the transformation code generator checks and uses all available rules in repository.

Although rule application control is declarative, before generating the transformation script, it must be clear that which transformation rule should be used first to start the declarative rule calling procedure. Our approach here uses the reverse engineered abstraction of both source and target visualisations to decide the starting rule.

The automatic reverse engineering mechanism uses a graph lattice as the basis for the abstraction. It incorporates a visitor pattern that traverses input examples and inserts new structures of the models it faces to the graph lattice. Therefore, it creates a complete abstract structure from provided visualisation example. The defined transformation rules are checked against source and target abstractions based on source structures they are to be applied to and the target structure they create, to find a rule applicable to the top most element of the abstraction. Once this rule is found, it automatically marks it as starting rule of the transformation script. The system then generates a full source model to target model transformation script based on the defined rules and hence our response to research questions 2 and 2.1.

Having taken both imperative and declarative approach for model-to-visualisation and visualisation-to-visualisation might imply that the transformation code generator should use transformation languages that support both declarative and imperative rule control mechanisms. Declarative transformations do not need explicit scheduling while imperative transformations allow better consistency checks. Since the visualisation approach is separated from the transformation, it is possible to use different transformation languages for each step, i.e. an imperative transformation language for model-to-visualisation and a declarative transformation language for the visualisation-to-visualisation transformation step.

The generated transformation code can be applied to any source model conforming to the example(s) used in the specification to produce a target model. Following section provides a group of case studies to further discuss this approach.

5.3 Case Studies

This section provides series of example case studies using our proof of concept tool implementation to specify and generate transformations using example visualisations. It describes transformation between bar chart and pie chart visualisation, Minard's map and pie chart, UML class diagram to Java, and CAD to alternative tree visualisations. The visualisation procedures of these examples are described in chapter 4.

5.3.1 Mapping bar chart to pie chart

Representing data using charts is common in many application domains. This case study demonstrates situations where an alternative visualisation is desired for same underlying data. It demonstrates how the data represented by a bar chart can be transformed into pie chart visualisation and hence a case of alternative visualisation for same underlying data. It uses the bar chart visualisations previously demonstrated in this thesis. Figure 5.13 shows examples of these visualisations.

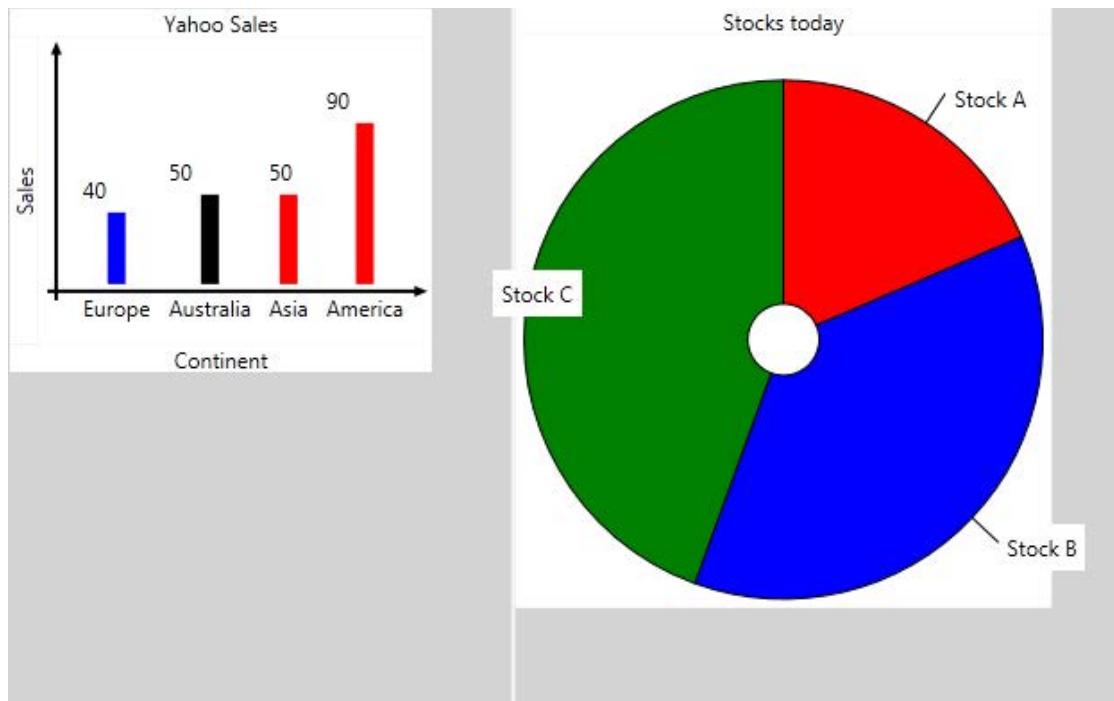


Figure 5.13 Example bar chart and pie chart visualisations.

As can be seen in Figure 5.13, bar chart visualisation is consisted of chart notation and set of bars. Pie chart is also consisted of pie area and the pie pieces. Therefore, for transforming bar chart to pie chart, two transformation rules will be required: a transformation rule to transform chart area in bar chart to chart area in the pie chart, and the transformation rule to transform each bar to a pie piece.

To generate the bar area to pie area transformation rule, it is required that the bar chart area notation be dragged and dropped on the pie chart area (as demonstrated by 1 in Figure 5.14). This interaction will trigger the transformation rule templates for both forward (bar chart area notation to pie chart area notation) and reverse (pie chart area to bar chart area) transformation.

Each of these notations has internal elements that should be mapped as well. The notations provide the internal elements in pop-ups that are displayed by right clicking on each notation. So to map internal elements (after popups are displayed) source elements need to be dragged and dropped on elements of target notation. This

interaction is marked by 2 and 3 in Figure 5.14. Here the bar chart area's name is to be mapped to pie chart's name and bars will be mapped to pie pieces. Once done, saving the rule will result in the default notations of both source and target chart areas to be saved as visual representative of the transformation rule.

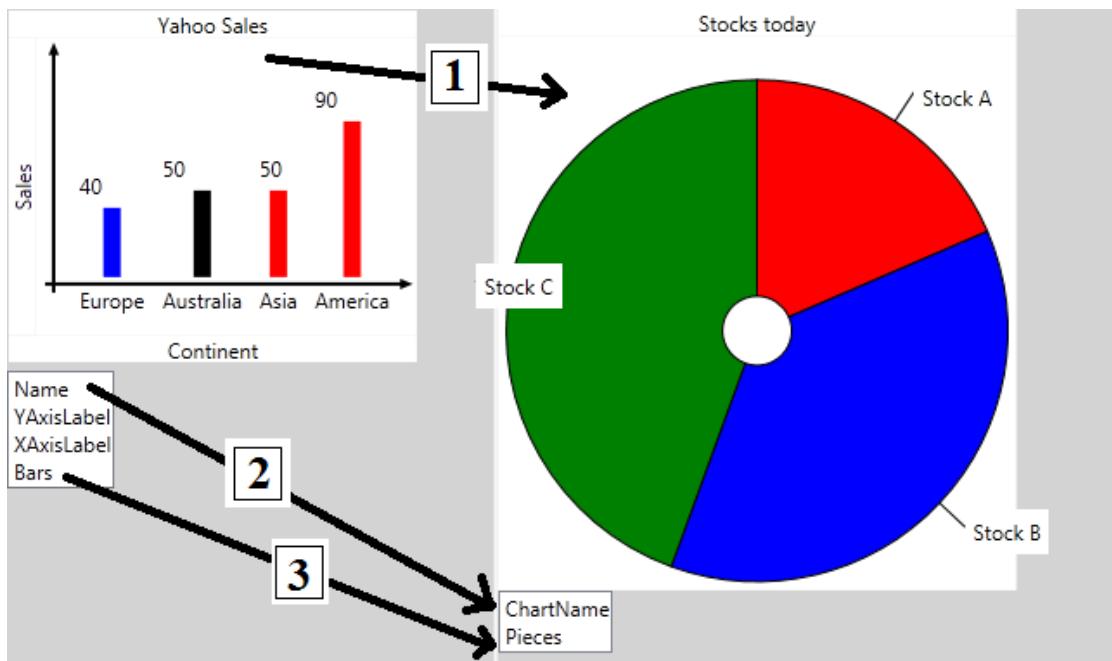


Figure 5.14 Mapping chart area notations.

To generate the second transformation rule, a bar notation needs to be dragged and dropped on a pie piece. Internal elements of the bar (name, value and colour) also need to be mapped to internal elements of the pie piece. Figure 5.15 demonstrates this interaction by arrows accordingly.

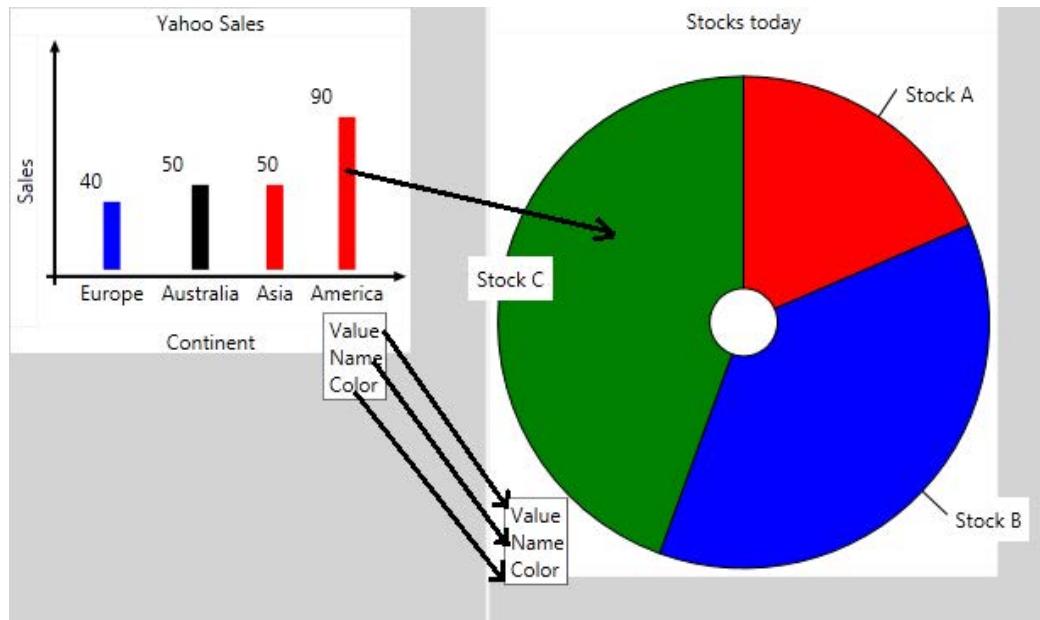


Figure 5.15 Mapping a bar to a pie piece.

Figure 5.16 shows the generated transformation rules. Note that since the chart areas are empty and there are no pie pieces or bars, the renderer does not show the pie boundary or bar chart axis.

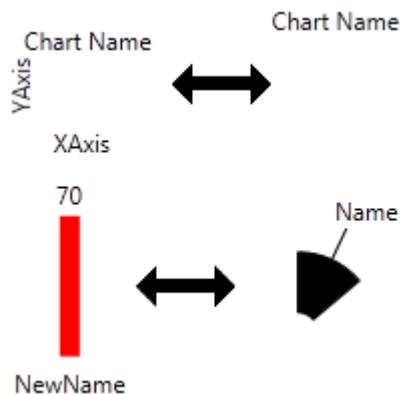


Figure 5.16 Transformation rules for transforming bar chart to pie chart.

To generate the transformation script, the reversed abstraction of bar chart is checked against the generated rules to find the starting rule (bar chart area notation to pie chart area notation's rule). The transformation script is then generated by calling the starting rule and including remaining transformation rules in the script. The remaining

transformation rules will be included in the script and called implicitly. Figure 5.17 shows the generated script in XSLT. Executing the generated transformation script will result in a new pie chart visualisation that represents the data of the bar chart visualisation as depicted by Figure 5.18.

```
<xsl:stylesheet xmlns:xsl="http:... " version="1.0">
  <xsl:template match="/">
    <xsl:apply-templates select="ChartData" />
  </xsl:template>
  <xsl:template match="ChartData">
    <NPCData>
      <ChartName>
        <xsl:apply-templates select="Name" />
      </ChartName>
      <Pieces>
        <xsl:apply-templates select="Bars" />
      </Pieces>
    </NPCData>
  </xsl:template>
  <xsl:template match="BarNode">
    <NPPData>
      <Value>
        <xsl:apply-templates select="Value" />
      </Value>
      <Name>
        <xsl:apply-templates select="Name" />
      </Name>
      <Color>
        <xsl:apply-templates select="Color" />
      </Color>
    </NPPData>
  </xsl:template>
</xsl:stylesheet>
```

Figure 5.17 Generated transformation script for transforming bar chart to pie chart.

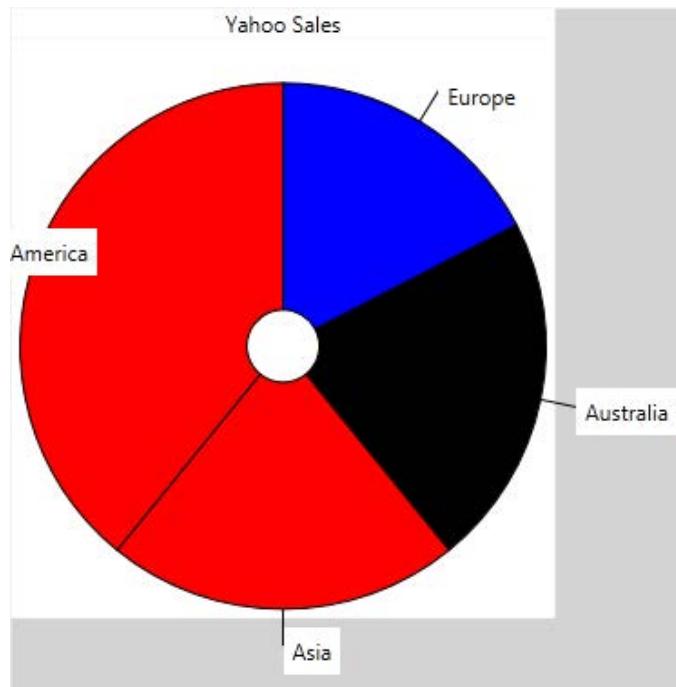


Figure 5.18 End result of the bar chart to pie chart transformation.

5.3.2 Mapping Minard's map to pie chart

This case study demonstrates how the data embedded in a complex visualisation (Minard's map) can be extracted and transformed to a model underlying a different visualisation (pie chart).

Given Minard's map visualisation in Figure 5.19, assume that there is a requirement for visualising the number of troops lost during the campaign at each key movement step as a pie chart, transformed by-example from this map visualisation.

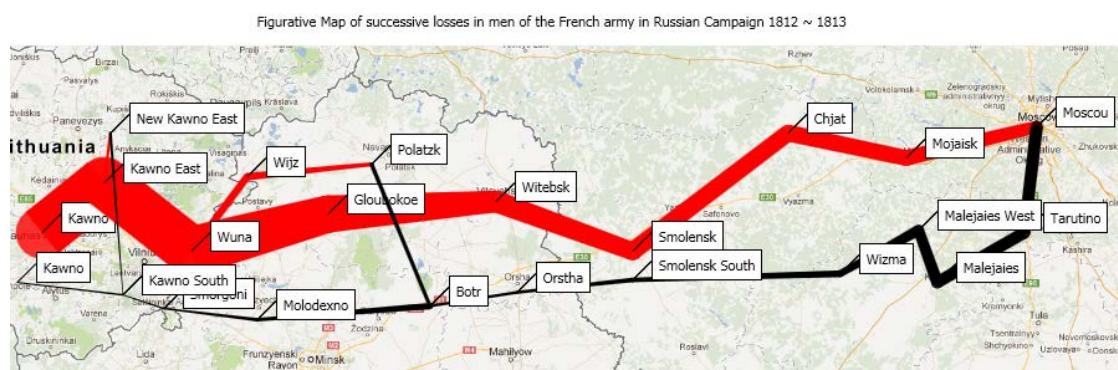


Figure 5.19 Minard's map visualisation.

To perform this transformation, we need to generate a rule for transforming the map to a chart and a rule for creating a pie piece from each troop movement notation. To generate the first rule, the user is required to drag and drop map notation on the chart area. The internal elements of the map will also correspond to elements of the chart; therefore, they should be linked as well. Figure 5.20 shows these correspondences by green arrows. Saving these correspondences will generate a map to chart transformation rule.

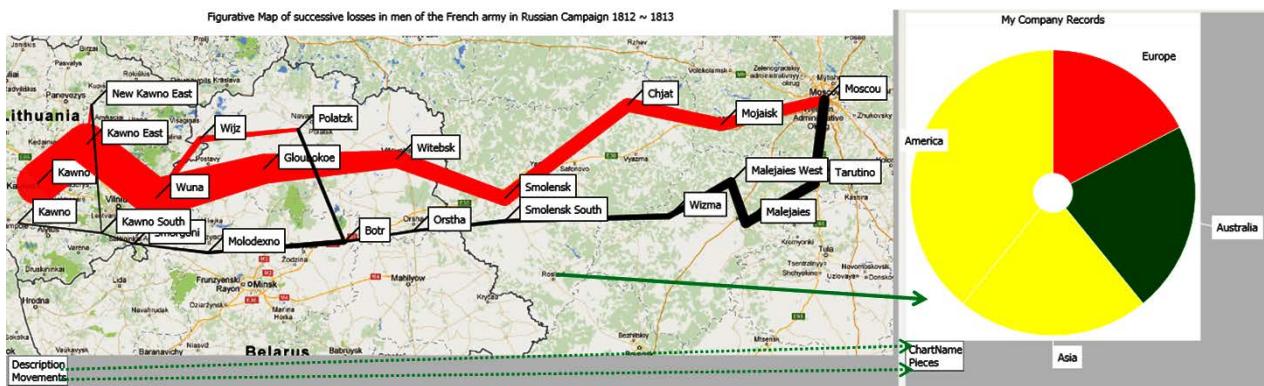


Figure 5.20 Specifying Minard's map to chart area transformation rule. Arrows depict drag and drop directions.

A second transformation rule is required for generating a pie piece notation from a troop movement notation. The user drags a movement notation onto a pie piece as shown by Figure 5.21. Each pie piece includes a value, a name and a colour. The colour element is in a one to one relationship with the troop notation's colour since we need to have the information regarding advancing or retreating status in that part of the journey. As a result, the colour element will be directly linked as shown by figure 5.21.



Figure 5.21 Specifying Troops movement notation to pie piece notation transformation rule. Arrows depict drag and drop directions.

Minard's map visualisation does not include a specific data element for number of troops lost at each movement. Therefore this value needs to be calculated from available data in the visualisation using provided mapping functions.

Once a notation is dragged on another in our framework, their default notations are provided in a separate window in case functions and conditions needed to be used. This is to prevent source and target visualisation windows from getting crowded. An example of this separate window and default notation visualisations is provided in Figure 5.22 for our example of troop movement notation to pie piece notation transformation rule.

To generate the value needed to be represented by each pie piece as the number of troops lost during the movement, a subtraction function needs to be used. Each pie piece needs to indicate that its data is representative of which troop movement notation. As a result each pie piece must include the name of the movement using the name at starting point of the movement and the name at the destination. A merging function is used here to merge two city names and include a “to” between them to generate the name for each pie piece. The required input elements to be used in these functions need to be dragged and dropped on function arguments and the function arguments will be dragged and dropped on the internal elements of the pie piece. These interactions are depicted by Figure 5.22. The result of this interaction will be the transformation rule script of Figure 5.23. Note the inclusion of function variables in the transformation script. Argument numbers have been updated by the transformation code generator.

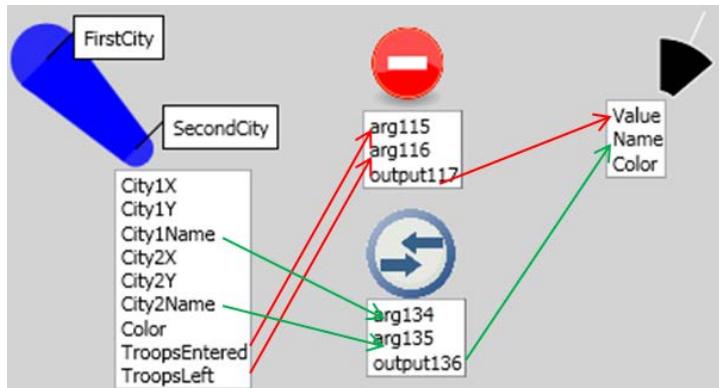


Figure 5.22 Specifying Troops movement notation to pie piece notation transformation rule using subtraction and merge functions. Arrows depict drag and drop directions.

```

<xsl:template match="TroopsData" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <NPPData>
    <xsl:variable name="arg99" >
      <xsl:value-of select="City1Name" />
    </xsl:variable>
    <xsl:variable name="arg100" >
      <xsl:value-of select="City2Name" />
    </xsl:variable>
    <xsl:variable name="output101" >
      <xsl:value-of select="concat(concat( $arg99 , &quot; to &quot;), $arg100 ) " />
    </xsl:variable>
    <xsl:variable name="arg96" >
      <xsl:value-of select="TroopsEntered" />
    </xsl:variable>
    <xsl:variable name="arg97" >
      <xsl:value-of select="TroopsLeft" />
    </xsl:variable>
    <xsl:variable name="output98" >
      <xsl:value-of select="$arg96 - $arg97 " />
    </xsl:variable>
    <Value>
      <xsl:copy-of select="$output98" />
    </Value>
    <Name>
      <xsl:copy-of select="$output101" />
    </Name>
    <Color>
      <xsl:apply-templates select="Color" />
    </Color>
  </NPPData>
</xsl:template>

```

Figure 5.23 Transformation script generated as a result of rule specification of Figures 5.21 and 5.22 in XSLT.

Once these two transformation rules are defined, transformation script for generating the pie chart can be generated. Using the abstraction of the source model (Minard's

map), system already knows that the starting transformation rule is the Minard's map to pie chart area transformation rule. The rest of the transformation rules (troop movement to pie piece in this case) will be implicitly called from starting transformation rule onwards. The resulting full visualisation of this example is depicted in Figure 5.24.

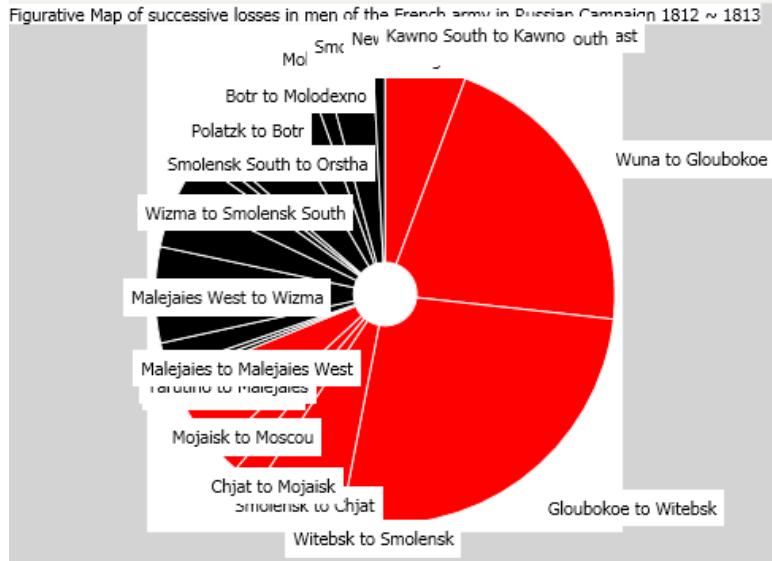


Figure 5.24 Resulting pie chart visualisation.

5.3.3 Mapping UML class diagram to Java

This case study demonstrates a transformation example from software engineering and MDE domain. It demonstrates how visualisation of a class diagram can be used to transform the underlying model to the visualisation of Java code. Given that visualisations of both UML class diagram and Java code are available, transformation between them can be generated by drag and dropping their notations. Figure 5.25 shows example of these source and target visualisations.

To perform this visualisation-to-visualisation transformation, transformation rules should be defined for UML diagram to Java code notation, UML Class to Java class, UML attributes to Java properties, UML operations to Java functions, and UML operation parameters to Java function parameters. In this visualisation configuration, each UML association is to be transformed into a Java property. The cardinality of this

association defines the multiplicity of that Java property. Therefore, a transformation rule is also required for UML association to Java properties.

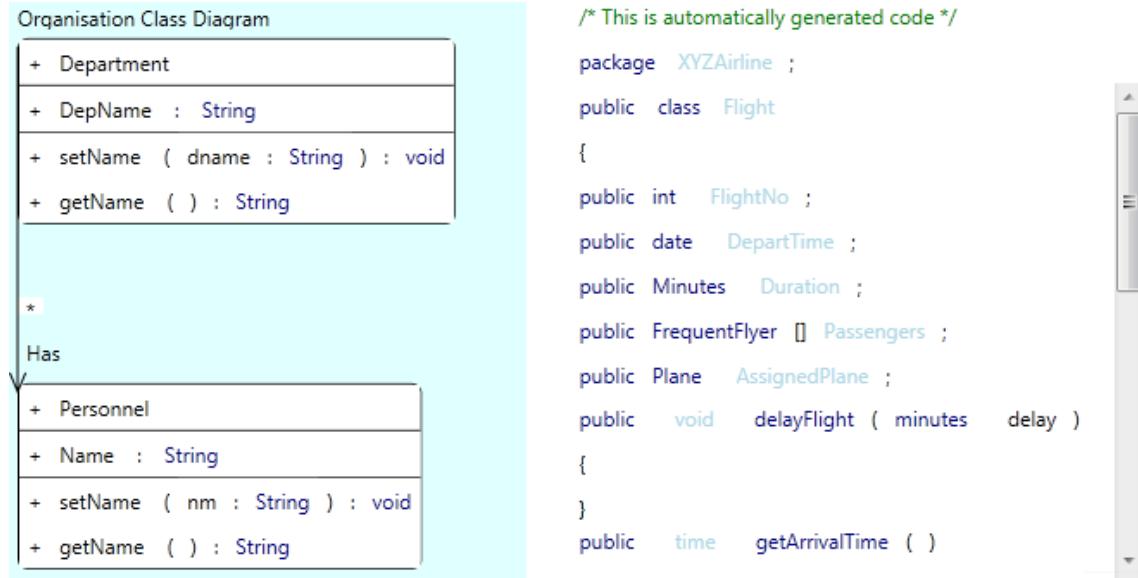


Figure 5.25 Sample Visualisations of a UML class diagram (source) and Java visualisation (target).

Figure 5.26 shows an example of creating a transformation rule for a UML attribute to a Java code property. To create this rule, user needs to drag a UML attribute to a Java field property, as depicted by solid black arrow, and match their internal elements, as shown by dashed black arrows.

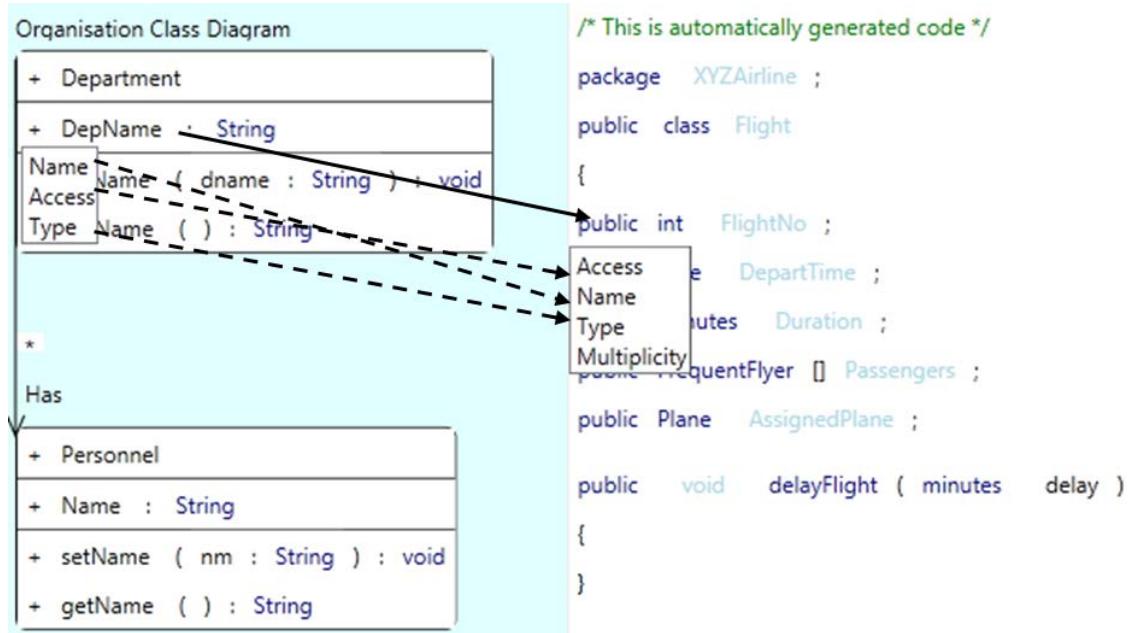


Figure 5.26 Specifying transformation rule between UML class attribute and Java field property. Arrows depict drag and drop directions.

Note that the two visualisations do not need to represent same data, as in Figure 5.26 where the class diagram represents an organisation but the Java code is representation of an airline package. Also, a level of consistency checking can be provided in each visualisation. For example, package name of the Java code knows that its name should not include white spaces, or Java attributes use default multiplicity of 1 when not specified and when blank is provided it is assumed to be N. These checks can be provided depending on the application during notation design. An alternative is to use functions and conditions when specifying transformations.

Figure 5.27 shows how UML diagram is mapped to Java package. As can be seen UML diagram's name contains spaces, these spaces will be deleted by controller of Java package when UML diagram's name is mapped to Java package's name.

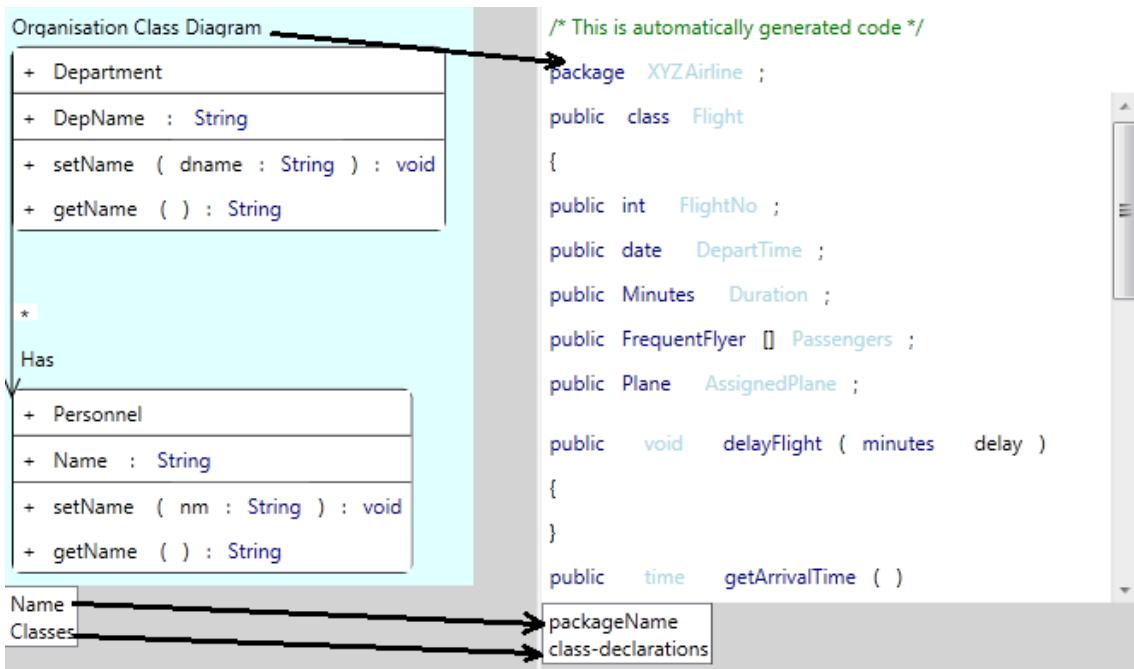


Figure 5.27 Specifying transformation rule between UML diagram and Java package.

Figure 5.28 shows how a class in class diagram is mapped to classes in Java code. As shown by the figure, since both attributes and associations are represented by properties in Java code, they have been mapped to properties in Java class element.

Similar to step two of specifying a concrete visualisation for a model, mapping functions are available to create more complex transformation rules between the concrete visual model mappings. For example, when mapping associations to a Java property, an association might have multiplicity defined by “*”, whereas a Java field property might have either a number or void as its multiplicity. A condition can be used to specify such a correspondence.

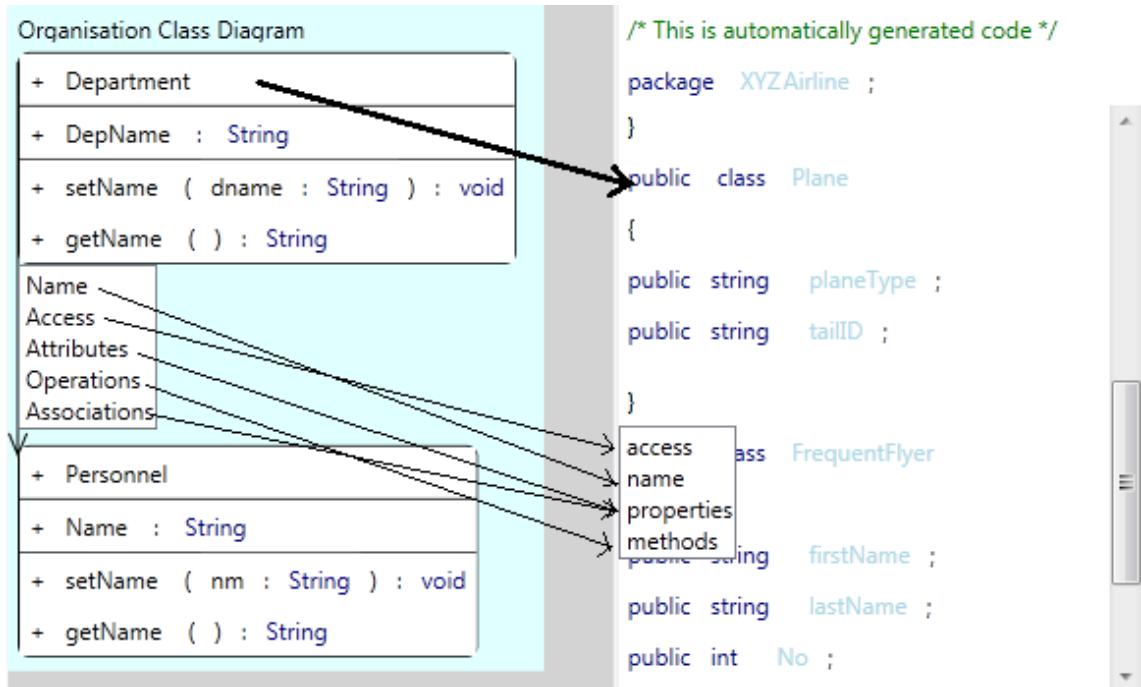


Figure 5.28 Specifying transformation rule between UML class and Java class.

By dragging a UML association to a Java field (or any notational element to another) their default notations will be shown on rule designer canvas to better provide space for using functions and conditions. Figure 5.29 shows the notations and the condition. The condition function in the figure tests whether “Multiplicity” of the association is equal to “*”. If so, it passes a blank character as output; otherwise it copies the value provided by “Multiplicity” to the output. Other correspondences between UML association and Java property can be defined either in this window or on original visual notations of the visualisation. If constant values need to be provided, (like spaces or “*”) they can be specified using provided facilities. The resulted transformation code script for this rule is shown by Figure 5.30. Note that since the association does not have access element, the code generator uses the default “public” value provided by the model of Java property.

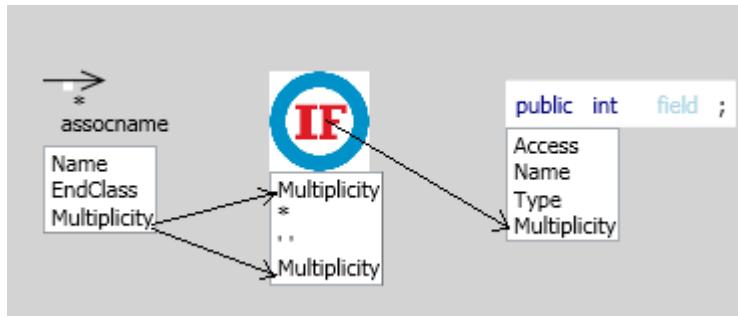


Figure 5.29 Specifying transformation rule between UML association and Java property.

```

<xsl:template match="UMLAssociation"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <Field>
    <Access>public</Access><!--default-->
    <Name>
      <xsl:apply-templates select="Name" />
    </Name>
    <Type>
      <xsl:apply-templates select="EndClass" />
    </Type>
    <Multiplicity>
      <xsl:choose>
        <xsl:when test="Multiplicity = '*'>*</xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="Multiplicity"/></td>
        </xsl:otherwise>
      </xsl:choose>
    </Multiplicity>
  </Field>
</xsl:template>

```

Figure 5.30 Transformation rule script for transforming UML associations to Java property.

Once all required rules are defined, transformation script for transforming UML diagrams to Java code can be generated. The code generator searches for starting rule which in this case is the UML diagram to Java package rule and generates the code script. The generated target as a result of applying this transformation script on the example class diagram of Figure 5.25 is shown in Figure 5.31. Note that the types are transferred to Java visualisation with the same capitalisation as UML diagram. In case it

is desired to have lower case type names, such functionality can be provided by using functions, or consistency checks inside visual notations.

```
/* This is automatically generated code */

package OrganisationClassDiagram ;
public class Department
{
    public String DepName ;

    public Personnel [] Has ;

    public void setName ( String dname )
    {
    }
    public String getName ( )
    {
    }
}
public class Personnel
{
    public String Name ;

    public void setName ( String nm )
    {
    }
    public String getName ( )
    {
    }
}
```

Figure 5.31 Resulting Java code visualisation.

5.3.4 Mapping CAD designs to alternative tree visualisation

This case study provides an example where Computer Aided Design (CAD) applications need to exchange complex models [164]. Consider the scenario where an architect might want to create an organisation's building structure chart based on an available CAD design. Assume that visualisation transformations for both models have

been provided beforehand, where the design model is visualised with a 2D building layout and the structure chart model via its diagrammatic representation. This case study shows how a transformation between elements of source design to elements of the target structure chart can be generated.

Figure 5.32 shows an example of mapping part of a detailed building design to a detailed structure chart. Elements of the chart structure should be created based on elements in the design. For example, drag and dropping a room on a corresponding room node in the tree and specifying their internal elements defines a transformation between their notations accordingly. Figure 5.32 shows this example for creating a transformation rule for mapping a 2D room shape (from source model visual notation) to a building structure node (in target model notation). To create this rule, user needs to drag a room notation element to a building node notation element, as depicted by solid black arrow, and match their internal elements, as shown by dashed black arrow.

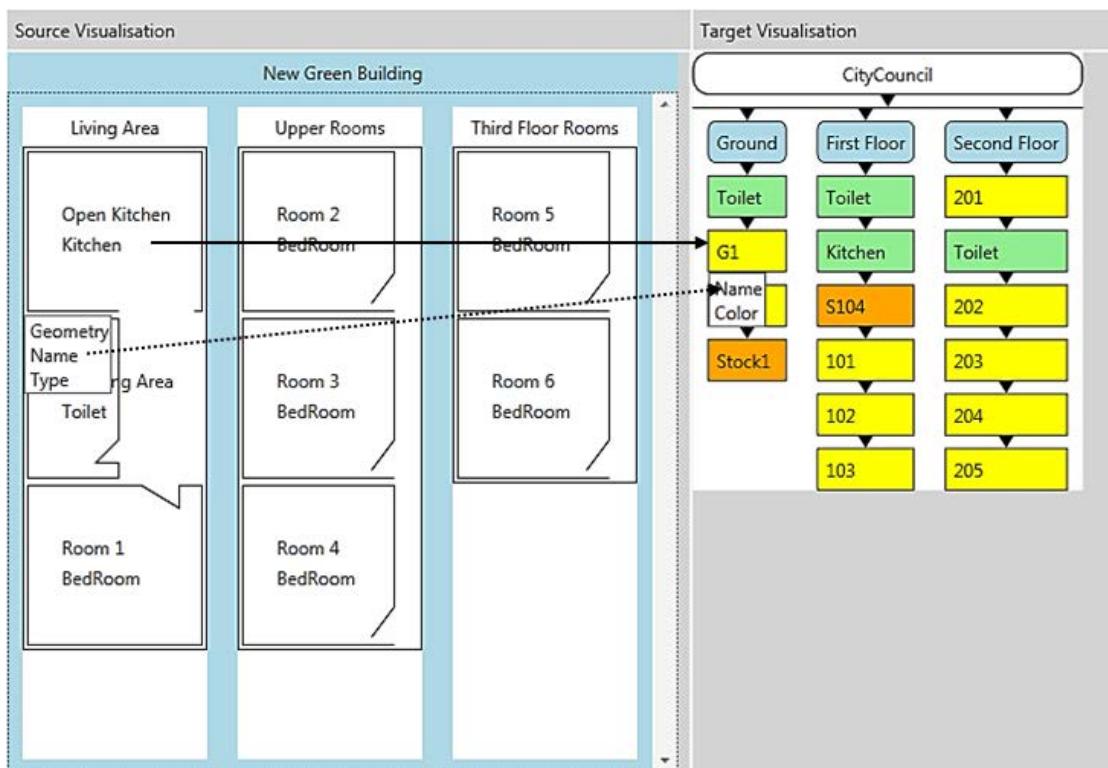


Figure 5.32 Defining a transformation rule for transforming a room in 2D CAD building to a room node in tree-based layout.

The nodes in the building tree structure are colour coded depending on room types. Room types in CAD visualisation however, are defined by their type string. To generate these colours, users can use conditions and specify colours based on room types. By dragging a room notation to a room node (or any notational element to another) their default notation representations will be shown in rule design view to better provide space for using functions. Users can navigate to that canvas and specify conditions as depicted by Figure 5.33.

The condition function in Figure 5.33 tests whether room's "Type" is equal to for example "Kitchen". Note than in general, arguments of functions and conditions are depicted by "arg" and a number (e.g. arg1, arg2). Once values are dragged and dropped on these arguments, they are replaced by the dragged value. Figure 5.33 captures the screen shot after 'Kitchen' element has been dragged and dropped on second argument of the function and hence the argument has been replaced by 'Kitchen'. Different colours (in this case Green) can be specified according to user's preference through the provided UI and dragged to the condition expression. Similarly, a colour can be defined if the condition expression was not satisfied by dragging the colour to "Otherwise" element. The value provided by the condition will be then assigned to the element of the target (in this case tree node's colour) as depicted by arrows in Figure 5.33.

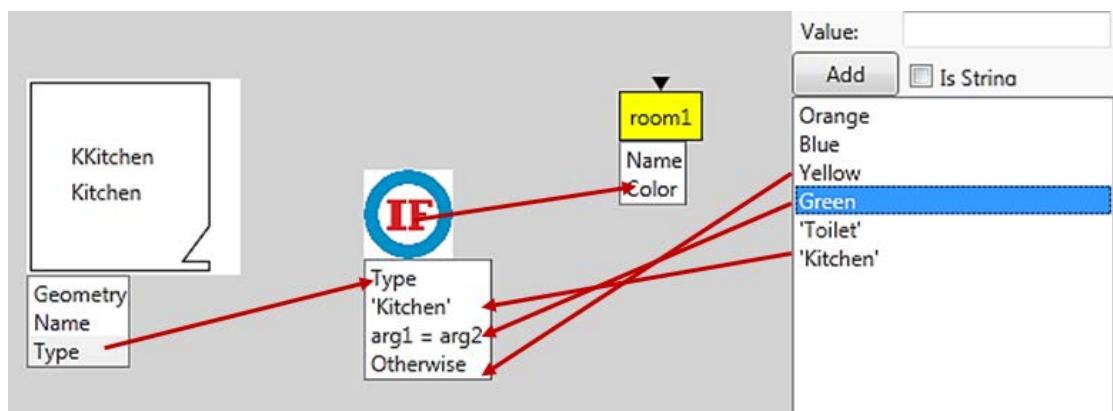


Figure 5.33 Using conditions to map 2D room notation to room node notation of a structure chart.
Arrows depict drag and drop direction.

To have complete transformation, two more rules need to be specified. First, the rule to transform floor plans in CAD design to floor nodes in the tree structure, and second, the rule to transform the CAD design to tree building. Elements inside notations for these rules are in one to one relation with each other. These rules can be specified by dragging and dropping their notations and their internal elements. Figures 5.34 and 5.35 show the creation of these rules accordingly.

The defined rules are depicted by the frame work using the concrete representation of the respective source and target notations that they transform. Figure 5.36 presents the concrete representation of these three rules. As stated before, since our approach uses the reverse engineered abstraction of the source and target visualisations, the two visualisations do not have to represent same data. For example in this case study, CAD design is representing a Green Building design while the tree structure is for a city council building.

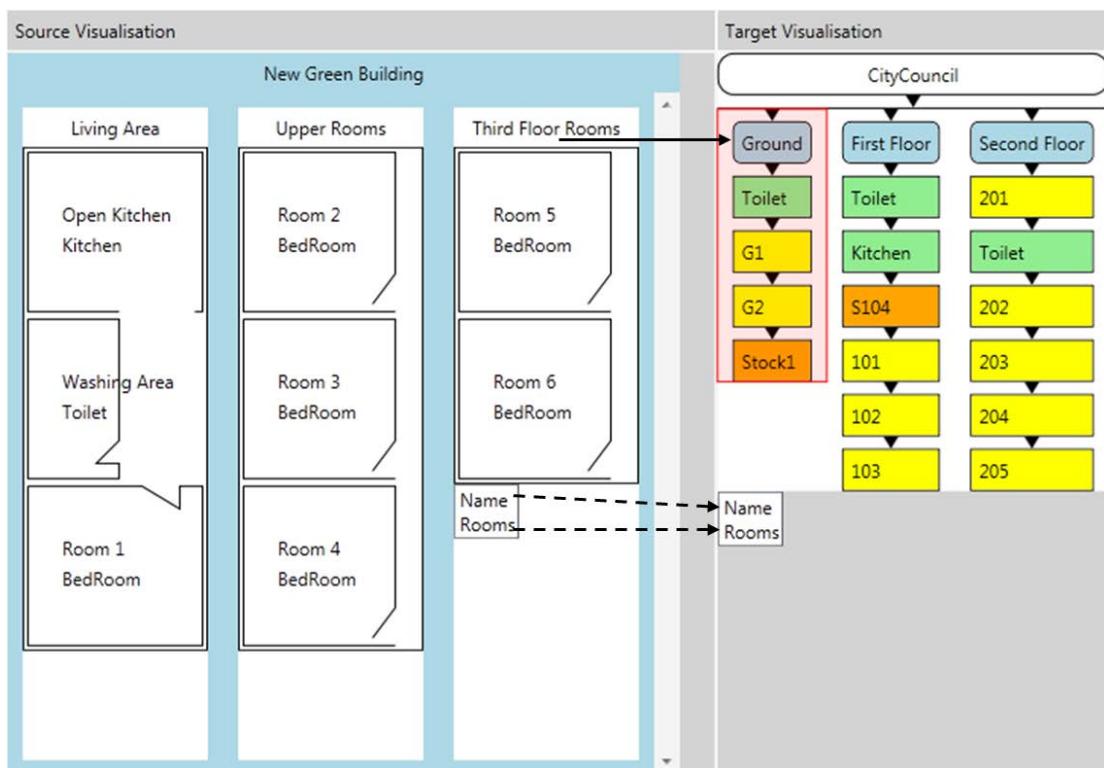


Figure 5.34 Defining a transformation rule for transforming a floor plan in 2D CAD building to a floor node in tree-based layout.

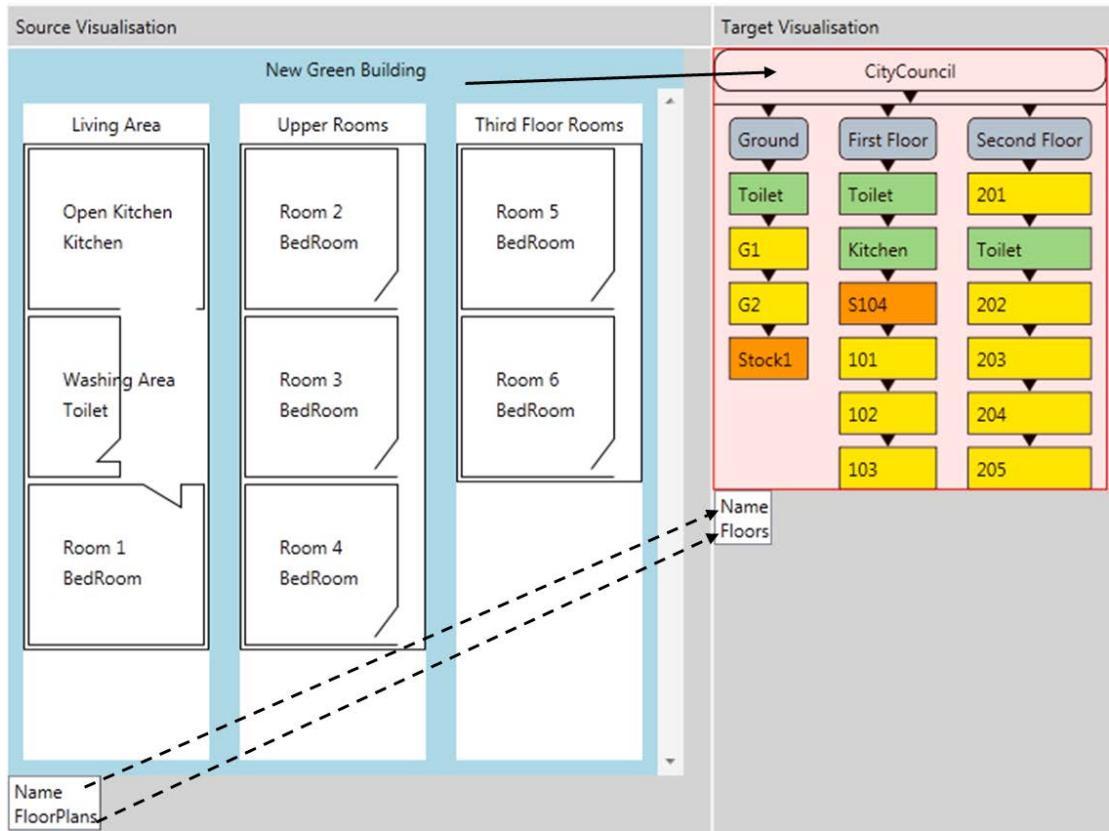


Figure 5.35 Defining a transformation rule for transforming a 2D CAD building to a tree-based layout.

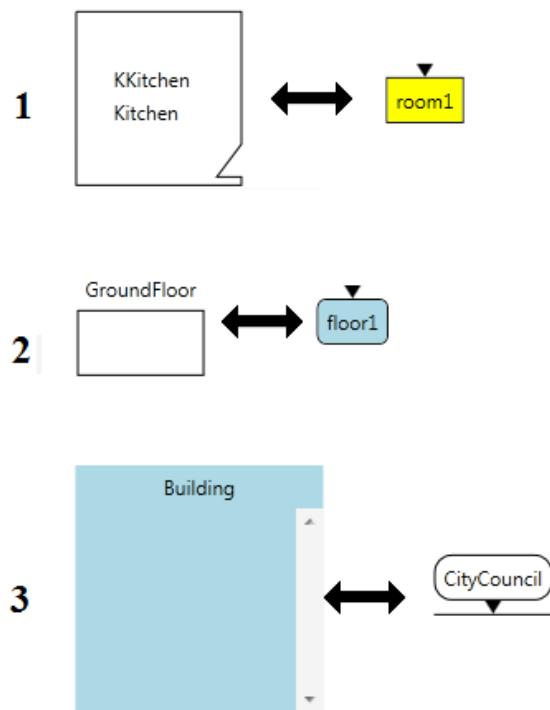


Figure 5.36 Concrete representation of three rules required to transform a 2D CAD building to a tree-based layout.

Once the required rules are defined, transformation script for this transformation can be generated. Applying this script to source CAD visualisations will transform the data represented by them to visualisation of the tree-based layout target. For example, applying the full transformation script on the source in this case study, will result in the visualisation of Figure 5.37.

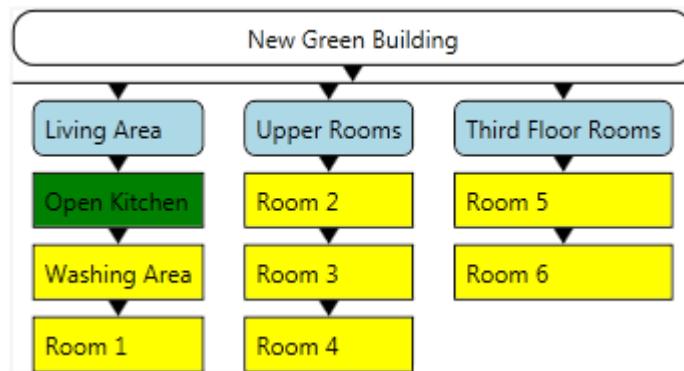


Figure 5.37 Resulting tree structure chart.

5.4 Summary

This chapter provided our approach in using concrete visualisations for model transformation specification. Concrete visual notations and their internal elements are dragged and dropped to generate transformation rules. From these visually specified transformation rules, transformation scripts are generated to transform source visualisation to target visualisations. Where possible the reverse direction is also generated automatically.

This chapter provided series of case studies to show applicability of our transformation approach for multiple domains. These case studies mostly used the visualisations generated in chapter 4 as source or target model visualisations.

Chapter 6

Correspondence Recommender

6.1 Introduction

Finding correspondences between source and target model elements for specification of transformation rules can be a challenging task. This can especially affect novice users more than experts. Even with incorporation of concrete visualisations, finding model element correspondences can get hard in large scale and more complex models.

This chapter describes our approach to providing guidance and support to transformation users in the form of correspondence recommendations. These recommendations are targeted to both novice and expert users. It helps novices explore possible correspondences and learn how source and target models can be linked. On the other hand, it supports experts in spotting correspondences for large models and visualisations, and helps them save time by selecting correspondences from suggested recommendations instead of drag and dropping visual notations. In summary, research question RQ3 and its following sub-questions are being addressed in this chapter:

3. How can interactive guidance be provided to users of model transformation systems?
 - 3.1. In what form should guidance be provided to users of model transformation?
 - 3.2. What is the best technique to generate acceptable recommendations?
 - 3.3. How can users best interact with recommendations?
 - 3.4. How can user response be used and integrated into the guidance mechanism?

6.2 Correspondence recommender (“Suggester”)

Our approach to support users provides recommendations on possible and likely correspondences between source and target that can help create transformation rules. To achieve automated support for correspondences, an automated recommender system (a “Suggester”) is designed that analyses input examples and user interaction in order to recommend possible correspondences between models and their sub-structures.

The main task of a recommender system in general is to provide guidance to users for choosing among multiple options [123]. Although this guidance hints do not have to be correct all the time, better correctness will result in more user trust in the recommender system [165]. Correctness is commonly measured with precision and recall metrics. These metrics consider the proportion of correct or incorrect metrics over a set of recommendations [161]. However, due to multi-dimension nature of recommender systems their accuracy should be measured according to application domain and their intended tasks [162].

For example, consider a recommender system that recommends set of commands to users of an Integrated Development Environment (IDE) to improve their efficiency [166]. If the recommended commands are the ones that the user is already aware of, although very accurate according to precision and recall, they will not improve user’s productivity. As a result, producing correct recommendations that are already known to users will gradually cause users to ignore it over time [167]. This factor (referred to as novelty) is usually measured by what proportion of the recommendations the user has already used or selected before, against the newly recommended items, and as a result directly affects accuracy. According to the application of recommender system, other dimensions to consider for their evaluation may include diversity, coverage, utility, serendipity, trustworthiness, learning rate, and robustness [162].

Among the three types of recommender systems (content-based, collaborative filtering and the hybrid) we have adopted the content-based approach for the design of our recommender system. Content-based recommenders prepare recommendations based on available data and information of items [134]. Content based recommenders would provide better applicability in off-line applications and would adapt to new models and contents faster than collaborative filtering approaches [168].

Previous research on design of recommender systems has shown how combination of recommendations resulting from different approaches can benefit the overall accuracy and acceptance of the recommendations [169]. Over thirteen thousand teams participated in Netflix competition to design a movie recommender that could improve an existing system [170]. The winner however, was the approach that combined the results returned by a group of recommender models that were not good-enough as stand-alone recommenders. It was shown that this combination allows recommenders to complement each other and produce better results [169].

The Suggester system introduced in this thesis, uses a mixture of content based recommenders and ensemble learning techniques [163]. The architecture of this system is outlined in Figure 6.1. The recommendations provided by the Suggester system include parent or child correspondences and can be used directly to develop transformation rules or used as guidelines to create final transformation artefacts.

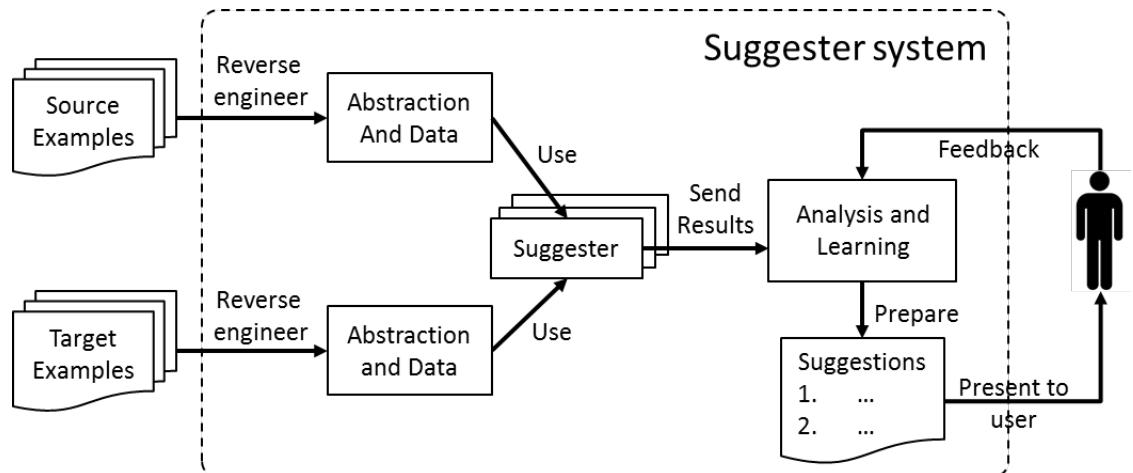


Figure 6.1 Architecture of "Suggester" system.

The content-based approach of this system allows a combination of information retrieval techniques to be used for analysing input model contents and identify similarities. Similarity scores provided by a collection of recommenders are used to produce final list of recommendations. Each of these recommenders uses a predefined similarity heuristic and analyses source and target model examples and ranks element

pairs by similarity scores. Next section describes how these scores are calculated individually and combined to achieve final recommendation list and hence our approach to answering research question 3.2 on generation of acceptable recommendations.

6.2.1 Calculating recommendations

To better demonstrate the application of correspondence recommender, Figures 6.2 and 6.3 show a simplified UML class diagram example and XML representation of Java code, and examples of their visualisations. Correspondences between these examples are shown by red lines in figures. As can be seen from figures, given that these example models were larger, finding correspondences would become a very hard task even for experts.

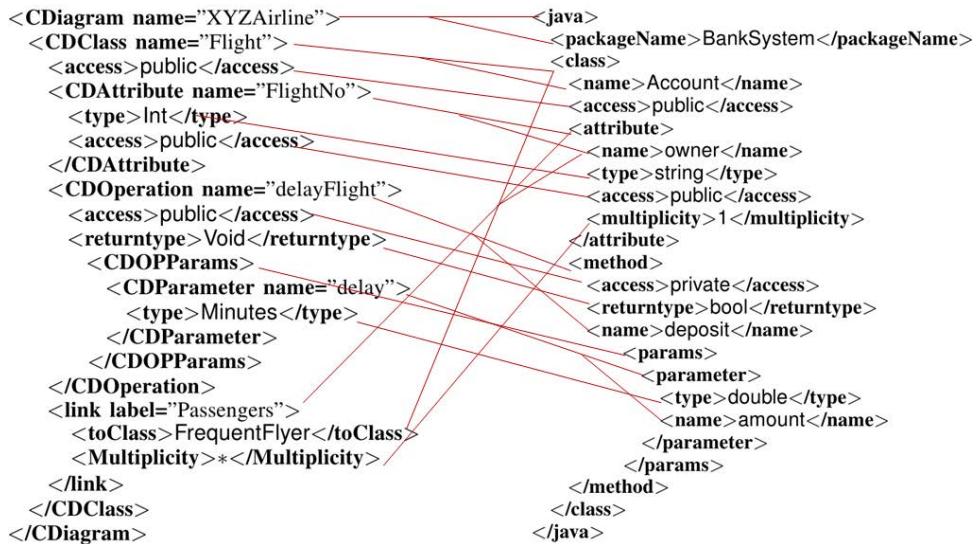


Figure 6.2 Sample correspondences between UML class diagram example XML and Java code XML.

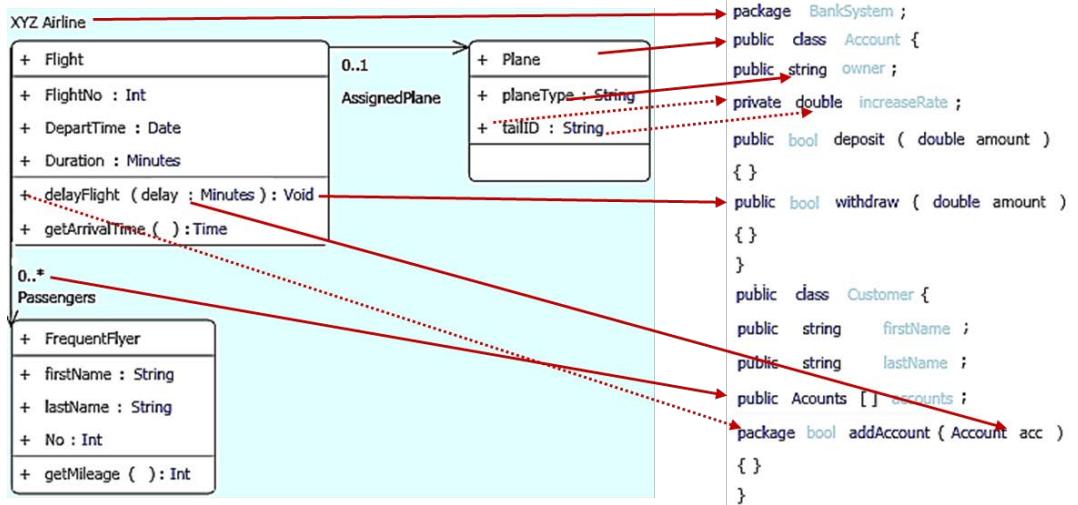


Figure 6.3 Sample correspondences between UML class diagram example visualisation and visualisation of Java code.

The ensemble learning adopted in this thesis research uses a set of similarity heuristics. These heuristics analyse source and target model examples according to a predefined similarity function. Table 6.1 provides a list of correspondence recommenders used in Suggerer system. These similarity functions range from static analysis of name tags and values to structural and propagated similarities. The result of these heuristics' analyses is returned to the system as a set of similarity matrices. These matrices are then used in an ensemble learning to finalise the similarity and calculate the recommendations.

Design of the Suggerer system allows adding more recommenders using provided component interfaces. Therefore, if need be to have other recommenders with different similarity function, they can be added to the system as extensions.

Table 6.1 Correspondence recommenders used in Suggester system

Correspondence recommender	Similarity heuristic
Static similarity	Value similarity: Similarity of the element values Name similarity: String matching similarity of element name tags Type similarity: Similarity of element types
Structural similarity	Neighbourhood similarity: Based on similarity of Neighbours of an element Graph similarity: Based on similarity of graph structure at element
Propagated similarity	IsoRank similarity: Similarity based on recursive analysis of neighbouring elements.

Early experiments with Suggester system revealed that due to the range of similarity heuristics and their similarity calculation overhead, the analysis of the actual source and target model examples would become costly and not efficient for large models. Therefore, instead of analysing actual source and target model examples, similarity functions are applied on the reverse engineered abstraction. This abstraction already preserves the structural constructs and name tags, and presents a good candidate for similarity calculation. With regards to smaller examples however, applying similarity recommenders on actual examples or their abstraction does not provide significant difference in terms of calculation time or recommender accuracy.

The similarity recommenders used in our Suggester system are described in following sections.

6.2.1.1 Static similarity recommenders

Static similarity recommenders judge similarity of source and target model constructs by comparing pairwise similarity of their elements. These recommenders are name tag similarity, value similarity and type similarity. The adaptation of these similarity functions is inspired by previous research [42], [46], [126].

Name tag similarity checks pairs of name tags of elements in source and target model examples. It incorporates a string matching technique that gives higher scores to “exactly similar” items than “somehow similar” items. It checks whether the name tags of both source and target match and assigns one score to each value being embedded in the other and one score if the values are exact matches.

Example 6.1 Consider UML class diagram and Java source code XML examples of Figure 6.4. Classes in the class diagram are defined by “CDClass” name tag and classes in the Java code are defined by “class” tag, since the tag of UML class includes Java class tag, name tag similarity suggester assigns score of one to the pair. UML class has an “access” element construct and Java class also has an “access”, the suggester gives this pair a score of three since the name tag of UML class’s “access” tag is included in the Java class’s “access” and vice versa (which accounts for two). Also since the values are the same, it adds another score. This will result in the cumulative score of three for “exactly similar” name tag pair “access”. Figure 6.4 shows a selection of these correspondences and their scores based on name tag similarity heuristic.

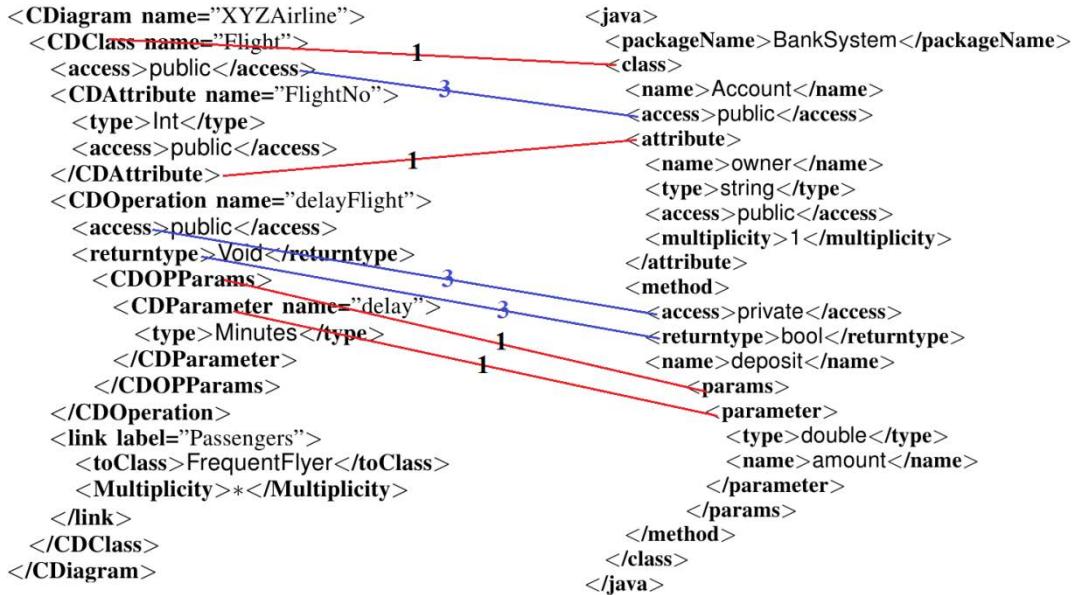


Figure 6.4 Example correspondences between UML class diagram example XML and Java code XML and their calculated score using name tag recommender.

Value similarity recommender checks values of model elements in both source and target model examples. To enable value similarity checks on reverse engineered abstraction, each construct in the abstraction was altered to accommodate the values seen in that construct throughout the model examples. Although this alteration had effects on size of the abstractions, it allowed more efficient analysis of model values for value similarity recommender and thus providing more usable recommendations. Value similarity recommender checks all the values that are represented in each construct and adds a score for each similar pair it finds. It then applies the total score to the construct pairs that possessed these values.

Example 6.2 Assume a UML class diagram's class example is given as the graph of Figure 6.5. This class has two attributes and two operations. One operation of this class also includes a parameter. Given this example XML the abstraction graph would look like the graph of Figure 6.6.

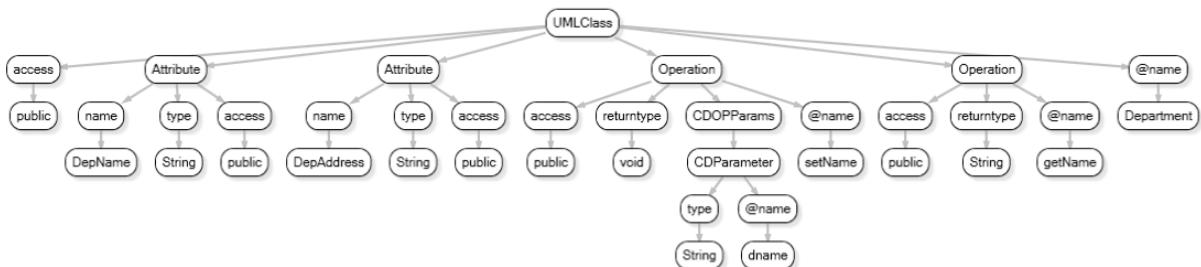


Figure 6.5 Example UML class graph.

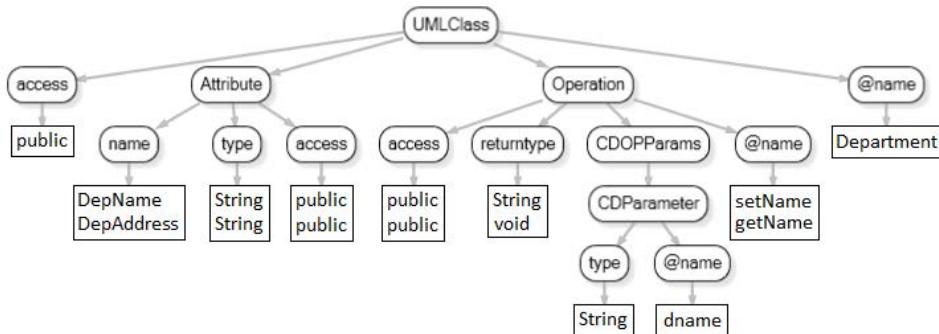


Figure 6.6 Abstraction graph of the UML class example in Figure 6.5.

Example 6.3 Consider a Java code class example's abstract graph is given as Figure 6.7. Assuming value similarity recommender is checking elements inside attribute of Java abstraction graph and UML class diagram graph of Figure 6.6 against each other (see Figure 6.8). Since type element of Java graph has a string value, and type element of UML class graph has two string values, it will return with the score of 2 for the two pairs. Similarly, since access elements of both have two "public" values there would be four similar pairs and hence score of 4. Note that value similarity is not case-sensitive. The rest of elements will return 0.

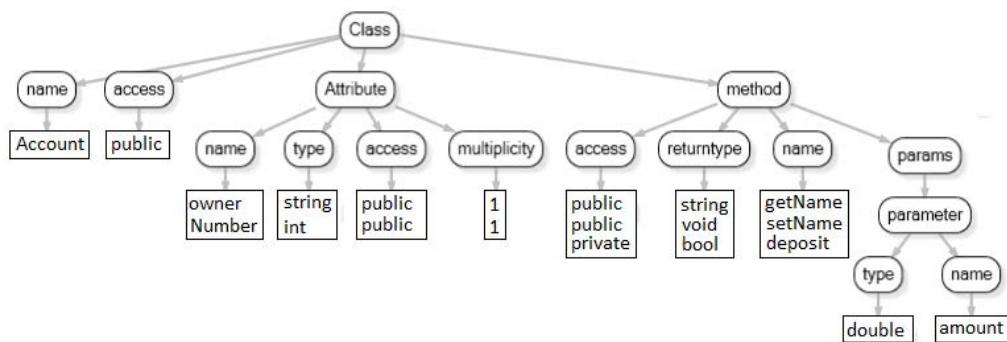


Figure 6.7 Abstraction graph of a Java class.

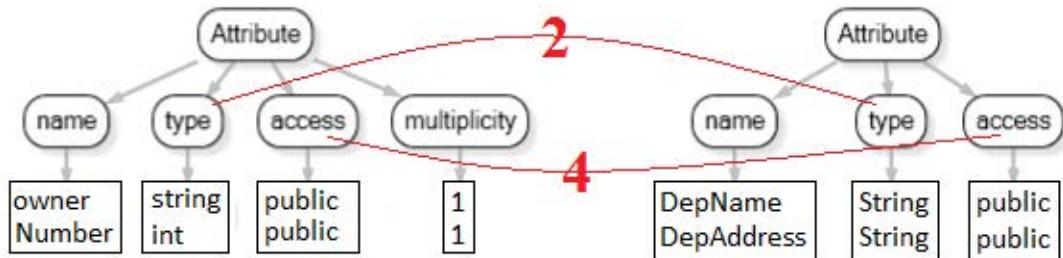


Figure 6.8 Correspondences returned by value similarity suggester and their similarity scores.

The reverse engineered abstraction also provides an estimate of value types for each construct. A type similarity recommender analyses these types and accordingly provides possible type matches as correspondences. It analyses value types for numerical, string and date types and returns score of one if the type of the values inside construct pairs are similar and zero otherwise.

Example 6.4 Figure 6.9 shows the sub-graphs of Figure 6.8 and the correspondences returned by type similarity. As is shown in the figure all similar types are returned as possible correspondences.

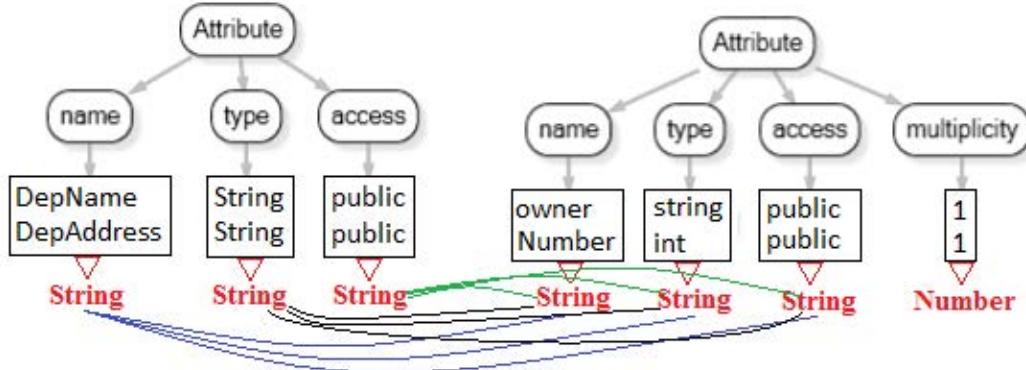


Figure 6.9 Correspondences returned by type similarity suggester.

6.2.1.2 Structural similarity recommenders

Structural similarity recommenders judge similarity according to the structure of construct pairs being analysed. These recommenders consider abstractions as graphs and include neighbourhood similarity and graph similarity.

Neighbourhood similarity recommender checks neighbours of each node pairs to see if they are similar. If the neighbours of a node n_s are similar to neighbours of another node n_t , then the two nodes are probably similar. To calculate similarity of the neighbours, neighbourhood similarity recommender checks the similarity scores returned by name tag similarity. It calculates the similarity score by cumulative similarity scores of the neighbour pairs as follows:

$$Score_{(n_s, n_t)} = \sum_{i \in N(n_s)} \sum_{j \in N(n_t)} Sim(i, j) \quad 6.1$$

Where $N(n_s)$ is the set of neighbours of node n_s and $N(n_t)$ is the set of neighbours of node n_t . $\text{Sim}(i, j)$ indicates the normalised name tag similarity of nodes i and j . Higher values of these scores represent higher similarity.

Graph similarity considers outgoing and incoming links of node pairs in both source and target graphs. Considering n_s to be a node in source graph G_s and n_t to be a node in target graph G_t , graph similarity recommender calculates similarity score for node pair (n_s, n_t) using following formula:

$$\text{Inbound}_{(n_s, n_t)} = \frac{\text{Min}(n_{s_{in}}, n_{t_{in}})}{\text{Max}(n_{s_{in}}, n_{t_{in}})} \quad 6.2$$

$$\text{Outbound}_{(n_s, n_t)} = \frac{\text{Min}(n_{s_{out}}, n_{t_{out}})}{\text{Max}(n_{s_{out}}, n_{t_{out}})} \quad 6.3$$

$$\text{Score}_{(n_s - n_t)} = \frac{(\text{Outbound} + \text{Inbound})}{2} \quad 6.4$$

Where $n_{s_{in}}$ is the number of in-links and $n_{s_{out}}$ is the number of out-links of node n_s . Same is true for node n_t . This graph similarity returns 1 if the number of in-link and out-links of the two nodes in the node pair are similar.

Example 6.2 Consider a UML class diagram and Java code example to have graph structures similar to Figure 6.10. Assume the pair to be analysed is the UML Attribute – Java Attribute. Analysing this pair using neighbourhood similarity recommender, results in similarity score of 10. It is calculated based on three exactly similar neighbour pairs (name, type and access) and a neighbour pair (class-UMLClass) that is somehow similar. Therefore, the score returned by neighbourhood similarity for this pair is $3+3+3+1 = 10$. For graph similarity recommender, using equations 6.2 and 6.3 we have inbound = 1 and outbound = $\frac{3}{4} = 0.75$. Given these values, the score for graph similarity will be calculated as $\frac{1+0.75}{2}$ which equals 0.875.

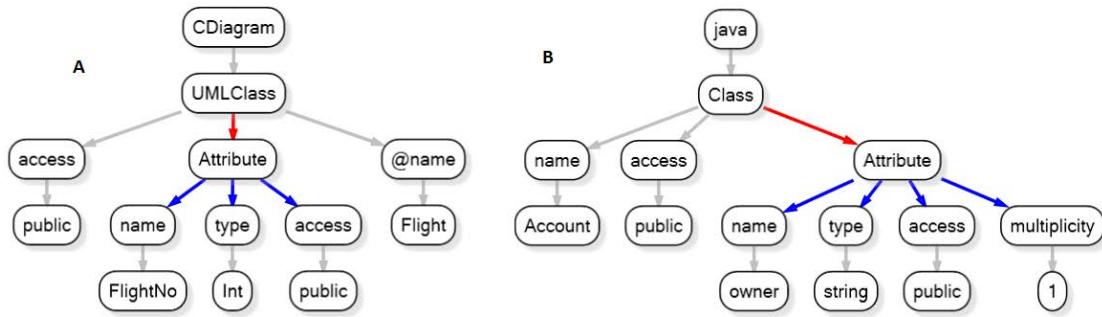


Figure 6.10 Sample graph of UML class diagram (A) and Java code (B).

Example 6.3 Using same input as Example 6.2, assume the pair to be analysed is the UML Class – Java Class node pair. Neighbourhood similarity recommender in this case would return 7 based on two exactly similar neighbour pairs (type-type, access-access) and a somehow similar neighbourhood pair (name-@name). Note that since the name in UML class is an attribute, framework prepends an '@' character in front of its name to differentiate it from other elements. Graph similarity recommender, will return 1 since the number of inbound and outbound edges are the same, $\text{inbound} = 1$ and $\text{outbound} = \frac{3}{3} = 1$ and $\text{score} = \frac{1+1}{2}$ which equals 1.

6.2.1.3 Propagated similarity recommender

Propagated similarity also considers input models to be graphs and calculates similarity of elements according to recursive analysis of their neighbouring elements. With this similarity function, similarity of two nodes in a graph is defined by similarity of their neighbourhood topology. As a result, using propagated similarity, two nodes are similar if their neighbours are similar and the neighbours of their neighbours are similar and so on.

To calculate correspondence using this similarity, our Suggester system adopts IsoRank approach used in biology for alignment of Protein-Protein Interaction (PPI) networks [5]. A PPI network is a graph in which each node corresponds to a protein and an edge indicates a direct physical interaction between proteins. PPI network alignment is a

required step to analyse and understand sequencing of genomes. IsoRank considers a protein in a PPI network to be a good match for a protein in another network if their respective sequences and neighbourhood topologies are a good match. It seeks two objectives to satisfy best network (graph) alignment: 1) maximising the size of common graph implied by linking similar proteins, and 2) aggregate sequence similarity between nodes linked to each other.

IsoRank uses similar approach to Google's PageRank by encoding propagated alignment similarity as an eigenvalue problem [171]. To achieve the two objectives, IsoRank works in two stages. It first associates a similarity score with each node pair of the two graphs using Basic Local Alignment Search Tool (BLAST) similarity [172]. Then it constructs the mapping for global network alignment by extracting a set of high scoring and mutually consistent matches.

Our adoption of IsoRank associates similarity scores return by name tag similarity with node pairs of the source and target model graphs. It is possible to alter the approach to adopt any or combination of other similarity recommenders.

Let R_{ij} be the IsoRank similarity score for node pair (i, j) , where i is from source graph G_s and j is from target graph G_t . Given the name tag similarity score $Sim(i, j)$ and the source and target graphs G_s and G_t , an eigenvalue problem is constructed and solved to calculate the vector R of all R_{ij} as follows. For all possible node pairs (i, j) a similarity score of their respective neighbours should be computed recursively. Therefore, Equation 6.5 should hold for all possible node pairs. Where $N(i)$ is the set of neighbours of node i , $N(j)$ is the set of neighbours of node j , V_s is the set of vertexes of graph G_s and V_t is vertexes of target graph G_t . Considering edge weights, the score propagated to each node is in proportion to edge weights in Equation 6.6 where $w(i, j)$ is the weight of the edge between vertices i and j . Equation 6.5 is special version of 6.6 where weights are equal to one.

$$R_{ij} = \sum_{u \in N(i)} \sum_{v \in N(j)} \frac{1}{|N(u)| |N(v)|} R_{uv} \quad i \in V_s, j \in V_t \quad 6.5$$

$$R_{ij} = \sum_{u \in N(i)} \sum_{v \in N(j)} \frac{w(i, u) w(i, v)}{\sum_{r \in N(u)} w(r, u) \sum_{q \in N(v)} w(q, v)} R_{uv} \quad i \in V_s, j \in V_t \quad 6.6$$

Example 6.4 This example demonstrates how vector R can be calculated for sample graph of Figure 6.11 (this example is adopted from [5]). For example, for calculating $R_{aa'}$ for node pair (a, a') , since node b is neighbour of a , and node b' is neighbour of b , $R_{bb'}$ should be considered. Since b and b' each have 2 neighbours, $R_{aa'}$ will be calculate according to $\frac{1}{2*2}$ proportion of $R_{bb'}$ which equals $\frac{1}{4}$. Calculation of R for other pairs will continue accordingly.

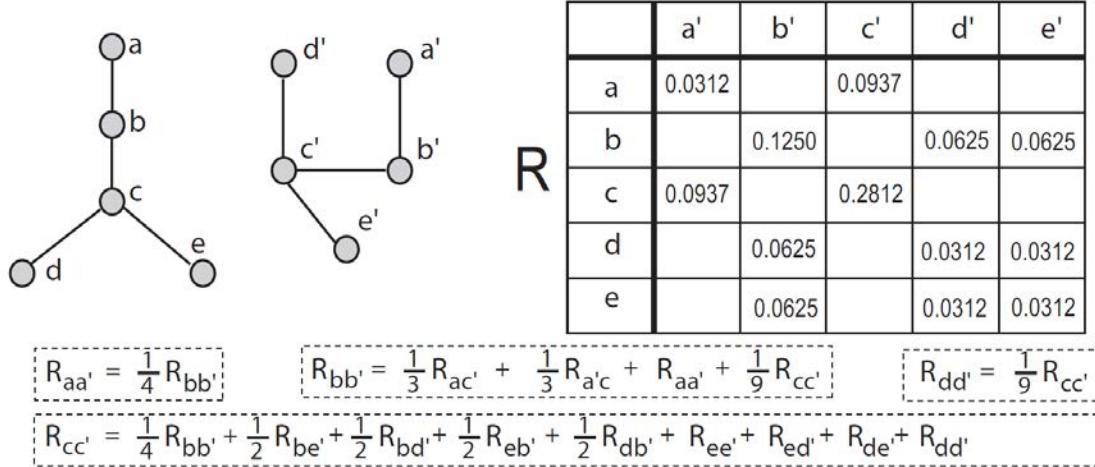


Figure 6.11 Calculating IsoRank similarity for sample graphs [5].

To calculate R a doubly indexed matrix A is adopted from Equation 6.5 using the matrix form provided by Equation 6.7. A is a $|V_s||V_t| \times |V_s||V_t|$ matrix and can get large depending on the size of input model abstraction graph.

$$R = AR, \text{ where}$$

$$A[i,j][u,v] = \begin{cases} \frac{1}{|N(u)||N(v)|} & \text{if } (i,u) \in E_s, (j,v) \in E_t \\ 0 & \text{Otherwise} \end{cases}$$

6.7

Where E_s is the neighbourhood matrix of graph G_s and E_t is the neighbourhood matrix of target graph G_t . A is a stochastic matrix (i.e. each of its columns sum to 1), therefore

its principal eigenvalue is 1. The principal eigenvector of A is the vector R and its values define possible source model to target model correspondences. This vector is then analysed and returned to $m*n$ matrix format to be considered as a similarity matrix for source and target model abstractions.

6.2.2 Suggester system ensemble

Each recommender returns a similarity matrix containing normalised calculated scores of the source-target pairs. Size of these matrices depends on the size of input model abstractions and is identical for all recommenders. To generate final list of recommendations, the returned similarity scores of recommenders need to be analysed and summed up.

Using classifier ensembles and composing a final data classification based on a collection of weak classifiers has been previously practiced in machine learning and data mining applications [163], [173], [174]. Classifier ensembles and boosting application are based on a collection of weak classifiers that are rated using a training set. This rating is then used when preparing final classifier. These approaches have been practiced for recommender systems as well [121], [175], [176].

The approach adopted here to calculate final recommendation list is inspired by classifier ensembles. Each recommender calculates a similarity matrix. Similarity matrices are normalised and sent to Suggester system. Suggester calculates final similarity based on the confidence scores assigned to each recommender. Similarity scores returned by recommenders are multiplied by their confidence score. The resulted scores are summed up in a final similarity matrix. The final similarity matrix is the basis for calculation of recommended correspondences.

Unlike most classifier ensembles, the rating or confidence scores are not calculated using predefined training sets. Instead, the scores are assigned (and updated) by continuous user evaluation of the recommender system. If a user accepts a recommendation, the recommender(s) that came-up with that recommendation get promoted. This promotion is achieved by increasing the confidence associated to those recommenders. The feedback analyser subsystem of the Suggester looks in similarity

matrices returned by recommenders to identify awarding recommender(s). If otherwise the user rejects a suggested correspondence, the feedback system penalises responsible recommender(s) accordingly by reducing the associated confidence weight.

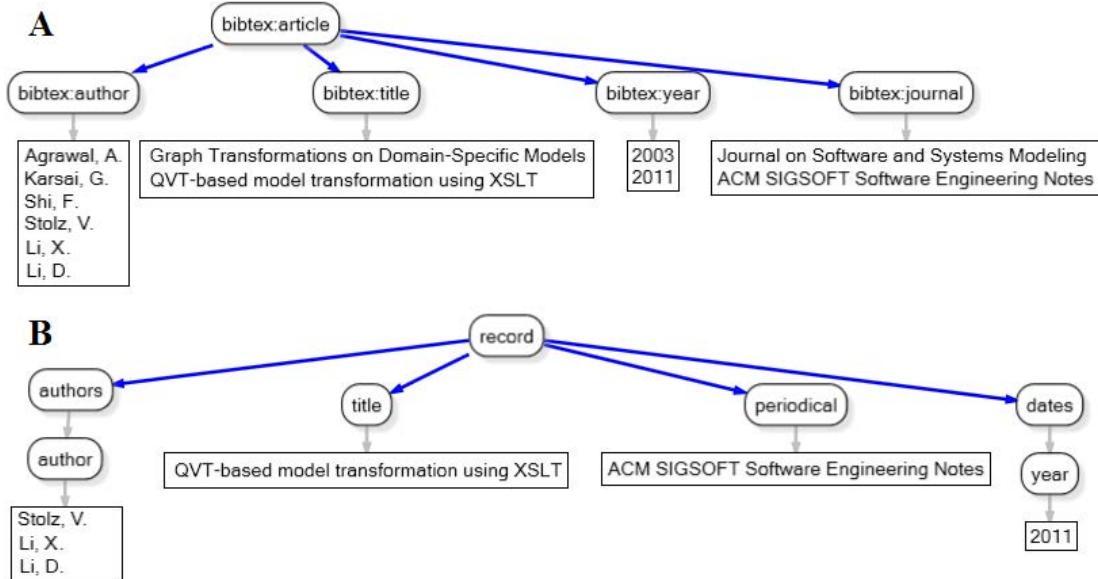


Figure 6.12 Abstraction graph examples of two citation formats.

Example 6.4 Assume abstractions of two citation index formats are given as Figure 6.12. Format A is the source and is to be transformed to format B. Given that format A has five elements and format B has seven elements, their similarity matrix would be a 5×7 matrix. The normalized similarity scores returned by recommenders for this example source and target are calculated as matrices M1 to M6.

		Format B						
Format A	bibtex:article	record	authors	author	title	periodical	dates	year
	bibtex:author	0	0	0.51	0	0	0	0
	bibtex:title	0	0	0	0.17	0	0	0
	bibtex:year	0	0	0	0	0	0	0.17
	bibtex:journal	0	0	0	0	0.17	0	0
		M1: Value similarity suggester results						

		Format B						
Format A	bibtex:article	record	authors	author	title	periodical	dates	year
	bibtex:author	0	0	0.333	0	0	0	0
	bibtex:title	0	0	0	0.333	0	0	0
	bibtex:year	0	0	0	0	0	0	0.333
	bibtex:journal	0	0	0	0	0	0	0
		M2: Name similarity Suggester results						

		Format B						
Format A	bibtex:article	0.333	0.333	0	0	0	0.333	0
	bibtex:author	0	0	0	0	0	0	0
	bibtex:title	0	0	0	0	0	0	0
	bibtex:year	0	0	0	0	0	0	0
	bibtex:journal	0	0	0	0	0	0	0
		M3: Neighbourhood similarity Suggester results						

		Format B						
Format A	record	authors	author	title	periodical	dates	year	
	bibtex:article	0.038	0.023	0.019	0.019	0.019	0.023	0.019
	bibtex:author	0.019	0.019	0.038	0.038	0.038	0.019	0.038
	bibtex:title	0.019	0.019	0.038	0.038	0.038	0.019	0.038
	bibtex:year	0.019	0.019	0.038	0.038	0.038	0.019	0.038
	bibtex:journal	0.019	0.019	0.038	0.038	0.038	0.019	0.038

M4: Graph similarity Sugester results

		Format B						
Format A	record	authors	author	title	periodical	dates	year	
	bibtex:article	0	0	0	0	0	0	0
	bibtex:author	0	0	0.1	0.1	0.1	0	0
	bibtex:title	0	0	0.1	0.1	0.1	0	0
	bibtex:year	0	0	0	0	0	0	0.1
	bibtex:journal	0	0	0.1	0.1	0.1	0	0

M5: Type similarity Sugester results

		Format B						
Format A	record	authors	author	title	periodical	dates	year	
	bibtex:article	0.103	0.103	0.003	0.007	0.007	0.103	0.003
	bibtex:author	0.014	0.003	0.173	0.003	0.003	0.003	0.006
	bibtex:title	0.014	0.003	0.006	0.169	0.003	0.003	0.006
	bibtex:year	0.014	0.003	0.006	0.003	0.003	0.003	0.173
	bibtex:journal	0.014	0.003	0.006	0.003	0.003	0.003	0.006

M6: IsoRank similarity Sugester results

Assuming all confidence scores are one, the final similarity matrix will be calculated based on sum of the values similar to matrix MF. Returned correspondence recommendations are highlighted in the matrix.

Format B							
	record	authors	author	title	periodical	dates	year
Format A	bibtex:article	0.474	0.459	0.022	0.026	0.026	0.459
	bibtex:author	0.033	0.022	1.154	0.141	0.141	0.022
	bibtex:title	0.033	0.022	0.144	0.81	0.141	0.022
	bibtex:year	0.033	0.022	0.044	0.041	0.041	0.022
	bibtex:journal	0.033	0.022	0.144	0.141	0.311	0.022

MF: Value similarity suggester results

Depending on the application and user preference, it is possible to select how many recommendations to be presented to users per pair in the suggestion list. By default only one recommendation per pair is provided. These are calculated using stable marriage algorithm and are optimised to provide best overall recommendation list [177]. For instance in Example 6.4, bibtex:article-authors pair with score of 0.459 has not been selected while the pair bibtex:journal-periodical with score of 0.311 has been selected as a recommended correspondence. This is due to the fact that stable marriage algorithm assigns bibtex:article to record which shows the highest score for bibtex:article element and ignores the rest of the pairs involving this element. In case more than one recommendation per pair is desired, users can alter Suggester system preferences accordingly.

The following section provides our approach to representing these recommendations to users and hence our response to research question 3.3.

6.2.3 Recommendation representation

The recommendations provided by our Suggester system are presented to users by default using a list of recommendations that can be accepted or rejected. A sample of this list is provided by Figure 6.13. These recommendations are provided for both model-to-visualisation and visualisation-to-visualisation steps and are considered as an alternative for drag and drop of notations.

<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Map ClassDiagram/Classes/UMLClass To javasource/class-declarations/class-declaration
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Map ClassDiagram/Classes/UMLClass/Name To javasource/class-declarations/class-declaration/name
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Map ClassDiagram/Classes/UMLClass/Access To javasource/class-declarations/class-declaration/properties/Field/Access
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Map ClassDiagram/Classes/UMLClass/Attributes To javasource/class-declarations/class-declaration/properties
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Map ClassDiagram/Classes/UMLClass/Attributes/UMLAttribute To javasource/class-declarations/class-declaration/properties/Field

Figure 6.13 Sample recommendation list.

A recommendation session is started by each recommendation list update. For example loading a source and a target visualisation triggers a recommendation list update. Or dropping a notation on the designer canvas in the visualisation procedure also triggers a recommendation list update.

Accepting or rejecting recommendations will disable their selection button to prevent users from selecting a recommendation multiple times in a recommendation sessions. Figure 6.14 shows the result of accepting and rejecting some recommendations of Figure 6.13. In this example the recommendation for class diagram's Access has been rejected and the recommendation for UML attribute has been accepted.

<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Map ClassDiagram/Classes/UMLClass To javasource/class-declarations/class-declaration
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Map ClassDiagram/Classes/UMLClass/Name To javasource/class-declarations/class-declaration/name
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Map ClassDiagram/Classes/UMLClass/Access To javasource/class-declarations/class-declaration/properties/Field/Access
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Map ClassDiagram/Classes/UMLClass/Attributes To javasource/class-declarations/class-declaration/properties
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Map ClassDiagram/Classes/UMLClass/Attributes/UMLAttribute To javasource/class-declarations/class-declaration/properties/Field

Figure 6.14 Result of accepting and rejecting recommendations.

To provide users with more useful recommenders, we have adopted the Guide and Filter mechanism proposed by Hernández del Olmo and Gaudioso [178]. In their proposal, a Guide provides answer to when and how each recommendation must be shown to the user, while the Filter must answer which of the items are useful/interesting candidates to become recommended items. A schematic view of this approach is shown in Figure 6.15.

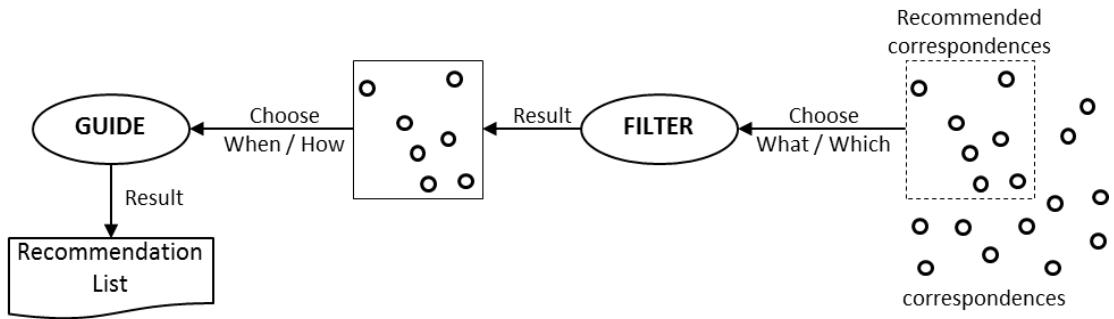


Figure 6.15 Guide and Filter system for representing correspondence recommendations.

Once all recommendations are available, our ensemble configuration filters the recommendation list by the stable marriage algorithm [177]. This will result in a selection of recommendations that possess the highest overall recommendation score per pair. The filtered results are then sent to guide system for representation. The Guide system chooses among recommendations according to the task that the user is about to perform, e.g. when user provides source and target visualisation to perform mappings, the Guide system first represents the recommendations that will result in transformation rules (i.e. parent correspondences). That is because a rule must be defined first, and its internal rule correspondences are to be defined later (i.e. child correspondences). Figure 6.16 provides an example where the parent correspondence recommendations are represented for a Class diagram to Java code example.

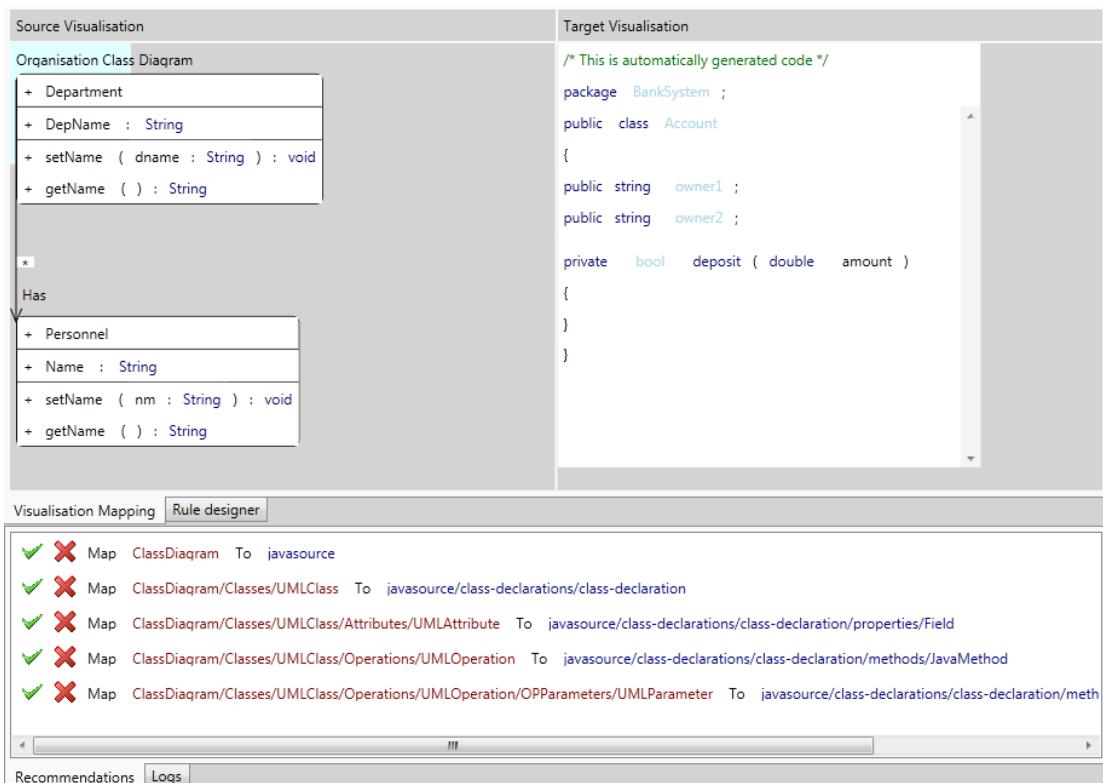


Figure 6.16 Example of recommendations that result in transformation rules for UML class diagram to Java code mapping.

If user accepts a recommendation from these recommendations (or alternatively performs a drag and drop of notations), the Guide system will update the list of recommendations to represent possible child correspondences which can be specified according to source and target model constructs of the selected parent correspondence. For example Figure 6.17 shows the result of accepting recommendation for UML attribute to Java field. As can be seen, the recommendation list has been updated to reflect internal correspondences of the two visual notations.

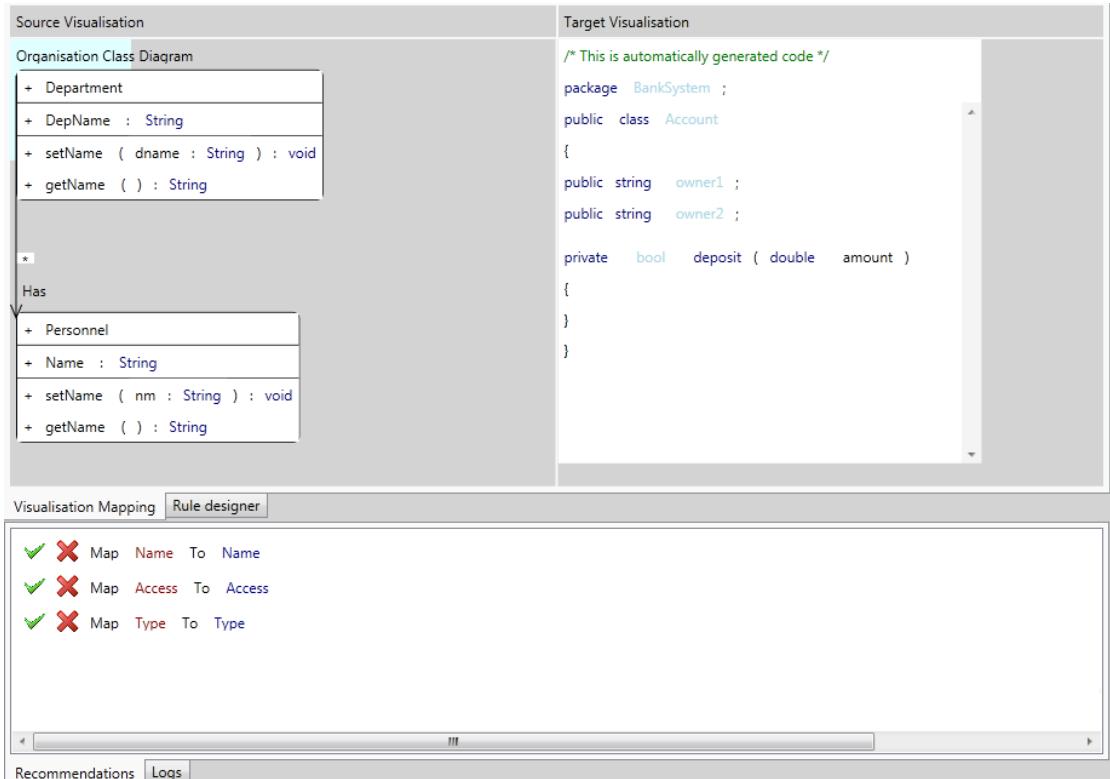


Figure 6.17 Updated list of recommendations after selecting a parent correspondence.

The guide and filter configuration will help users find targeted recommendations and reduce the amount of time spent on exploring long recommendation lists.

6.3 Summary

This chapter described our Suggester system which is designed to guide users in finding possible and likely correspondences between source and target models. A group of similarity recommenders check source and target examples and prepare a list of recommendations. The similarity heuristics used in these recommenders range from static value similarity to structural similarity. Each recommender provides a similarity matrix to the suggester system and the system uses an ensemble mechanism to calculate final recommendation list.

Users can use the provided recommendations as guidance or select them to define model transformation correspondences. It is possible to select or reject any recommendation provided in the list. By selecting or rejecting these recommendations a feedback analyser updates the Suggester system to improve its learning and recommendation capability.

Chapter 7

Tool support: Concrete visual assisted transformation (CONVErT)

7.1 Introduction

This chapter describes tool support and a proof of concept prototype of our approach, called Concrete visual assisted transformation (CONVErT). CONVErT provides facilities for specification and use of familiar concrete visualisations of source and target models. With CONVErT, users can specify complex model element mappings between concrete visual notational elements using interactive drag-and-drop and reusable, spread sheet-like mapping formulae.

This chapter specifically provides our answer to the fourth research question on whether the approach presented by this thesis can be implemented in a usable, scalable and user friendly tool. The following sections describe CONVErT's Architecture, Implementation, User Interface (UI), and key design features.

7.2 Overview of CONVErT

This section provides an overview on key components of CONVErT. These components are Reverse engineering, Transformation code generator, Correspondence recommender (Suggerer), Visual notations, and Renderer. Figure 7.1 shows these components and how they are inter-related.

Reverse engineering component provides automatic generation of abstractions from model examples. These abstractions are then used for calculation of recommendations in the Suggester system and as transformation templates. Transformation code generator uses these abstractions and set of correspondences to generate transformation scripts.

The Suggester component uses the abstractions and model data to recommend possible correspondences between source and target model examples to users. It uses set of similarity heuristics to calculate the similarity of elements in source and target model examples.

Visual notations are the centre piece of visualisations and enable user interaction (drag and drop). Each visual notation embeds a model data and a controller transformation. Using this model data and the controller the notation's view can be generated. The Renderer component provides a mechanism for rendering visualisations using the controller transformation of notations. It can also render full visualisations by checking the visualisation input files against available notations in the notation repository. The following paragraphs are dedicated to describing these components in more details.

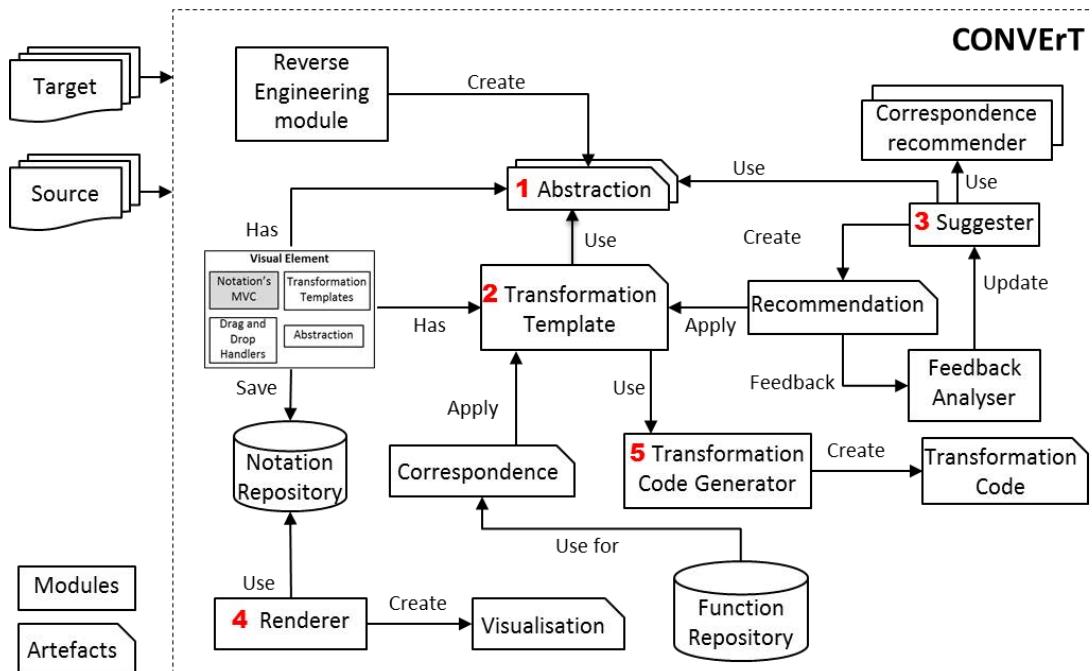


Figure 7.1 Components of CONVERT.

The reverse engineering and model abstraction mechanism of CONVErT (Figure 7.1 (1)) uses a graph lattice as meta-model and for transformation rule templates. The abstraction graph is also used for transformation rule scheduling. Once source and target examples are provided, a visitor pattern creates an empty graph lattice. It then traverses the examples and fills the lattice with new element structures that it faces. This way the structure of source and target are known to the system. Each transformation rule template (Figure 7.1 (2)) will be initially created from such structure retrieved from the data part of the element being dragged or dropped. During transformation script generation for transforming source visualisation to target visualisation, the transformation code generator checks for the position of each rule's source structure in the abstraction to identify transformation rules to be called first.

Concrete model representation in CONVErT uses Model View Controller architecture [2]. Each notation has a view created and provided by XAML. A model data is provided as an XML that describes internal elements of that notation. The controller in this configuration is a transformation that transforms notation's model XML to the view's XAML.

Since the generated concrete visual notations need to provide interaction (drag and drop) capabilities and host transformation templates, the Renderer mechanism of CONVErT (Figure 7.1 (4)) wraps each notation in interaction logic provided by an instance of a Visual Element (VE) class. A VE provides a container for the notations and other VEs and is implemented using XAML and C#. This architecture allows our framework to let users interact with composing elements of a model visualisation regardless of the embedding hierarchy of the notation.

Analysing large input models and visualisations is costly for the Suggester system. Therefore, it uses abstract lattices as input to calculate similarities. Suggester uses a group of mapping correspondence recommenders (Figure 7.1 (3)) that analyse these abstractions according to a similarity heuristic. Then the resulting similar source and target elements of each recommender are returned to the Suggester as possible correspondences. A confidence score is associated with each correspondence recommender. Based on the scores given to the recommended correspondences and the confidence weight of the recommenders, a final score is calculated for each

recommended correspondence. Suggester selects from the recommendations and prepares a recommendation list. If users select from the recommended correspondences or reject them, a feedback analyser updates the confidence weights associated with the recommenders and thus improves Suggester's learning mechanism.

The transformation code generator in CONVErT (Figure 7.1 (5)) works with the transformation templates embedded in each notation. These templates are initially defined by reverse engineering notation's model data. Code generator uses correspondences defined by dragging and dropping of elements to this notation and its internal elements, and forms correspondence snippets that will be inserted in the template. Then once the template is filled with these snippets, the transformation code generator creates a full XSLT template and transformation rule. Using these transformation rules, the transformation code generator generates a complete model transformation specification in XSLT.

Since the transformation code generator is based on the templates, it is possible to generate transformation code for alternative transformation languages. To do so, additional transformation code generator components can be integrated to CONVErT to parse these templates to desired transformation scripts.

In the following sections we describe elements of CONVErT in more detail including its User Interface, how transformations and recommendations are generated, and details of the visualisation and rendering.

7.3 CONVErT's User Interface

This section provides an overview of CONVErT's UI. This has been divided into three parts: 1) Visualiser, for specifying correspondences between model elements and predefined visual notations. 2) Mapper, which provides facilities for transformation generation between visualisations, and 3) Notation designer (Skin++), which allows users to define and add new visual notations. In the following sub sections, these parts are described in more detail.

7.3.1 Visualiser

The approach presented in this thesis for model transformation uses concrete visualisations of source and target models. The specification of these concrete visualisations needs to be done in the framework and can be reused for multiple model examples. This visualisation specification requires users to drag and drop elements of their model examples to provided visual notation in the frame work to generate model-to-visualisation mappings. These model-to-visualisation mapping notations are then composed to generate full visualisation specifications and the model to visualisation transformations. Examples of this procedure are provided and discussed previously in Chapter 4. This section describes the facilities provided in CONVeRt framework (Visualiser) to enable this visualisation approach.

Visualiser in CONVeRt allows users to view multiple input models in a default tree-like representation (Figure 7.2(1)). These inputs could be CSV or XML files. Once these models are loaded, CONVeRt generates their tree view representation. This is to allow use of interaction with each element or value of the input examples.

The predefined visual notations are provided in a separate panel (Figure 7.2(2)). These notations can be dragged and dropped on the designer canvas (Figure 7.2(3)). The required elements of the input model can then be dragged and dropped on them or onto their internal elements. Notations' model elements are accessible via a popup window which list internal elements according to notation's model. This popup provides elements by showing the name of corresponding model element. This is somewhat a limitation as these names may not be unique or not provide an easy to understand specifier. In an ideal implementation, element would be dropped on actual graphics of notations. For example in bar chart visualisation, one could drag the name of their input model to the actual label representing the name of a bar chart. The popup approach however was chosen due to simpler implementation.

Mapping functions are available to be used at this stage for specifying more complex model-to-visual notation correspondences and are provided in a separate panel (see Figure 7.4(9)). These functions and conditions can also be dropped on visualisation designer canvas and are used similar to notations. In example of Figure 7.2, a visual notation for a building tree structure and an “IF” condition are dropped on visual

notation designer canvas. Elements of input model can be dragged and dropped on the notation to start a transformation rule template. Internal elements of notations, functions and conditions can be accessed by right clicking on them. Elements of input model can then be mapped to these internal elements by drag and drop. In example of Figure 7.2(3) a right click has been performed on the condition and its internal elements are provided as a result (“arg1”, “arg2”, “arg1 = arg2”, and “otherwise”). These internal elements constitute arguments and condition expressions of the condition. Elements of input model can be dragged and dropped on these values and expression to specify the values that should be passed when this condition is true or otherwise false.

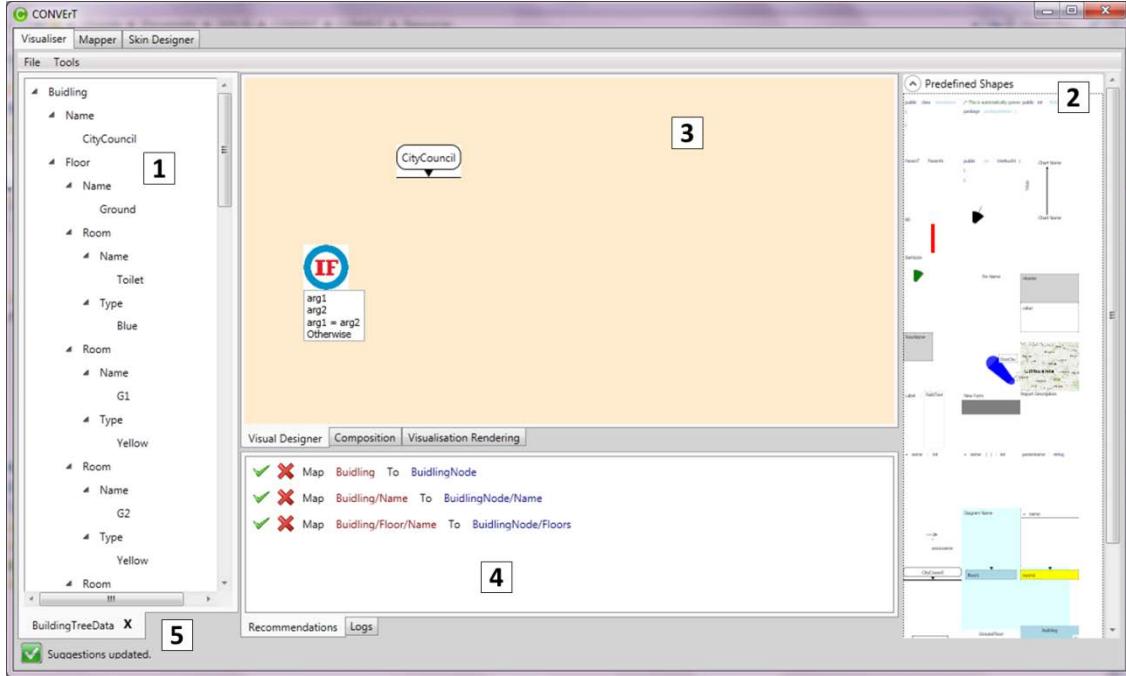


Figure 7.2 Using CONVERT’s visualiser UI for mapping input model elements to visual notations. 1) Input model, 2) Predefined notation, 3) Designer canvas, 4) Recommendations, 5) Status panel.

Once a notation is dropped on the designer canvas, CONVERT’s suggester analyses its underlying model and the input model being selected by user and provides a list of recommended correspondences (Figure 7.2(4)). Users can interact with these recommendations using accept and reject buttons provided beside each recommendation item. For example, to define a customised notation for input models and thus a model-

to-visual notation transformation rule, users can drag and drop elements from input model tree to the notation on the designer canvas, or select from suggested correspondences.

An alternative to this recommendation representation would be to highlight recommendations on the notations as the user hovers mouse pointer on model elements in the tree visualisation. However, the intractable list metaphor was preferred since it provided better separation of concerns between recommendation mechanism of the approach (Suggester) and the visualisation. This way to represent recommendations, the framework does not need to traverse visual notations' visual tree to find corresponding elements of the recommended items in their visual representation.

An interactive status bar has been designed in CONVERT that shows the status of the system and provides feedback to users if the tasks are being performed correctly or if an error is made (Figure 7.2(5)). These status reports are provided with appropriate icons to report or alert users of any problems associated with the task being performed. Similarly, interactions of users with visualiser are recorded by the Logs panel in Figure 7.3(8). These logs can be used to keep track of user interaction and system automated tasks. For example if an element is dropped on model element of a notation, the status bar shows whether the task has been accomplished and an event will be logged. To keep track of these drag and drops, users can check the provided logs. These logs provide a history of user performed and automated tasks and work similarly for all notations, functions and conditions. When requesting to generate transformation code for the composition, if any errors or exception is occurred, they will be shown in the working status bar and logged. Upon completion of transformation code generation, the completion of the task will be logged as well.

Once the user defines correspondences between input model elements and the notation, saving the customised notation will result in the notation being added to notation repository. CONVERT monitors this repository and provides its containing notations in “Designed Visual Elements” panel (Figure 7.3(6)) for later (re)use. For example customised visual notations previously generated for bar chart area, bars, troops movement, map, Java code and UML class diagram can be seen in Figure 7.3(6).

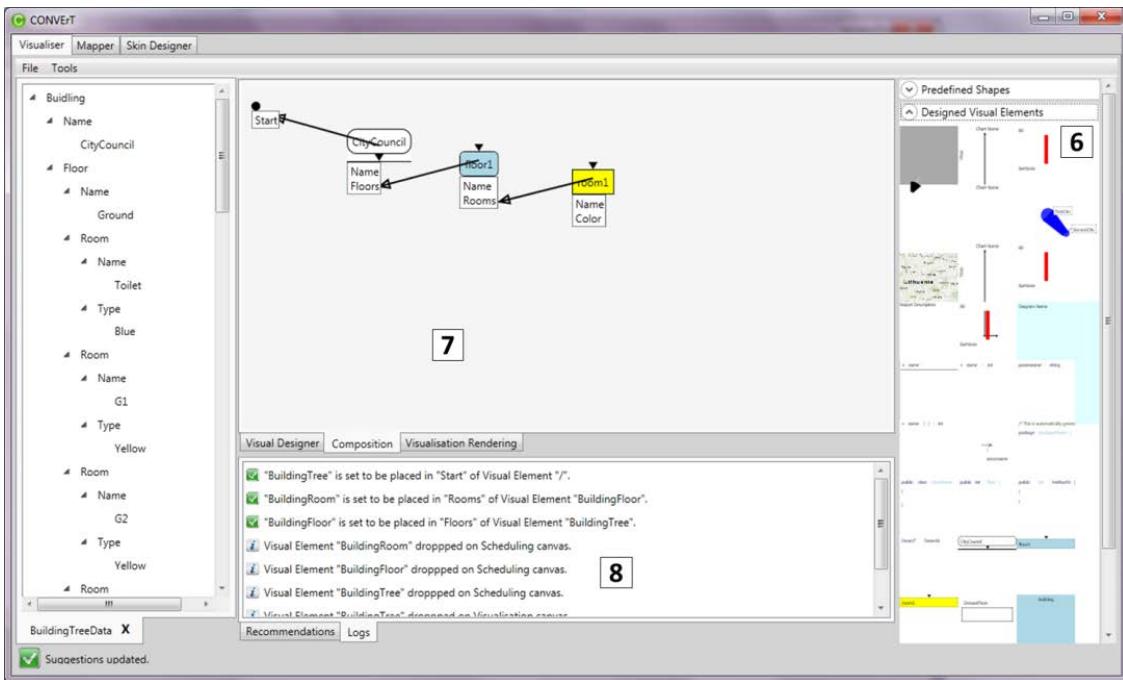


Figure 7.3 Using CONVERT's visualiser UI for composing visual notations. 6) Customised notations' panel, 7) Notation composition Canvas, 8) Usage logs.

To compose notations, a specialised composition canvas is provided (Figure 7.3(7)). The defined customised notations can be dragged and dropped on this canvas and linked according to their placeholder elements to create a complete visualisation and model-to-visualisation transformation. In example of Figure 7.3, required customised visual notations for generating a tree visualisation of building structure example are being composed. This composition will result in scheduling of the transformation rules embedded in the notations and a complete transformation script to transform the input model used in defining customised notation to the tree structure visualisation. Examples of these compositions and their visualisation results were provided in Chapter 4.

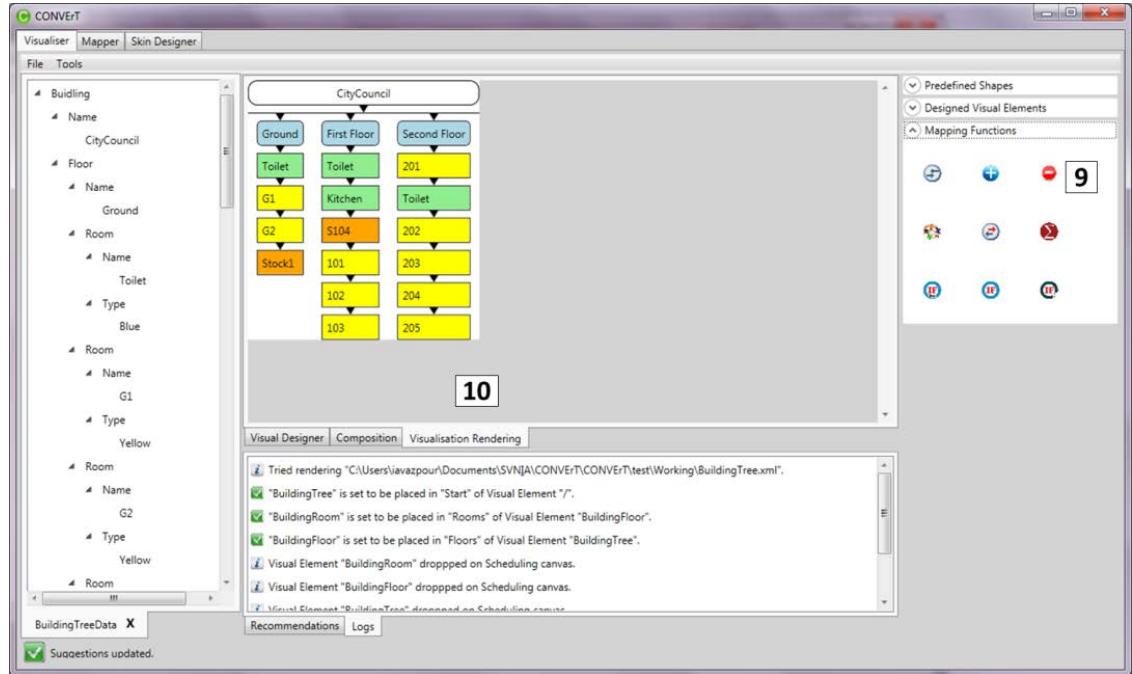


Figure 7.4 CONVERT's visual functions and conditions panel (9) and visualiser renderer (10).

The resulted model-to-visual transformation will be applied on the input model. The resulting visualisation will be automatically rendered in the visualisation rendering panel. Figure 7.4(10) demonstrates the rendering panel and the resulted visualisation of the composition of Figure 7.3. Users can also use this panel to view previously generated visualisations.

7.3.2 Mapper

A significant contribution of this thesis is on using concrete visualisations to generate transformation between source and target models. This way, users can view and specify correspondences using the more familiar concrete visualisations of their example models.

As discussed previously in Chapter 5, our approach enables generation of forward and (if possible) reverse transformation rules by drag and dropping visual elements of source and target visualisations. Low level model transformation scripts are then

generated from these drag and drop interactions. This section describes use of CONVErT's Mapper for generating these transformations.

Mapper window in CONVErT provides facilities for generating transformations between visualisations. It embeds two visualisation renderer panels that allow viewing source and target visualisations side by side (Figure 7.5 (1) and (2)). Once source and target model visualisations are loaded, their notations and internal notation elements can be dragged and dropped to create visual notation-to-visual notation transformation rules. For example, in Figure 7.5 UML class diagram is loaded as source and the target is Java code visualisation. Elements of this UML class diagram can be dragged and dropped on visual notations of the Java code.

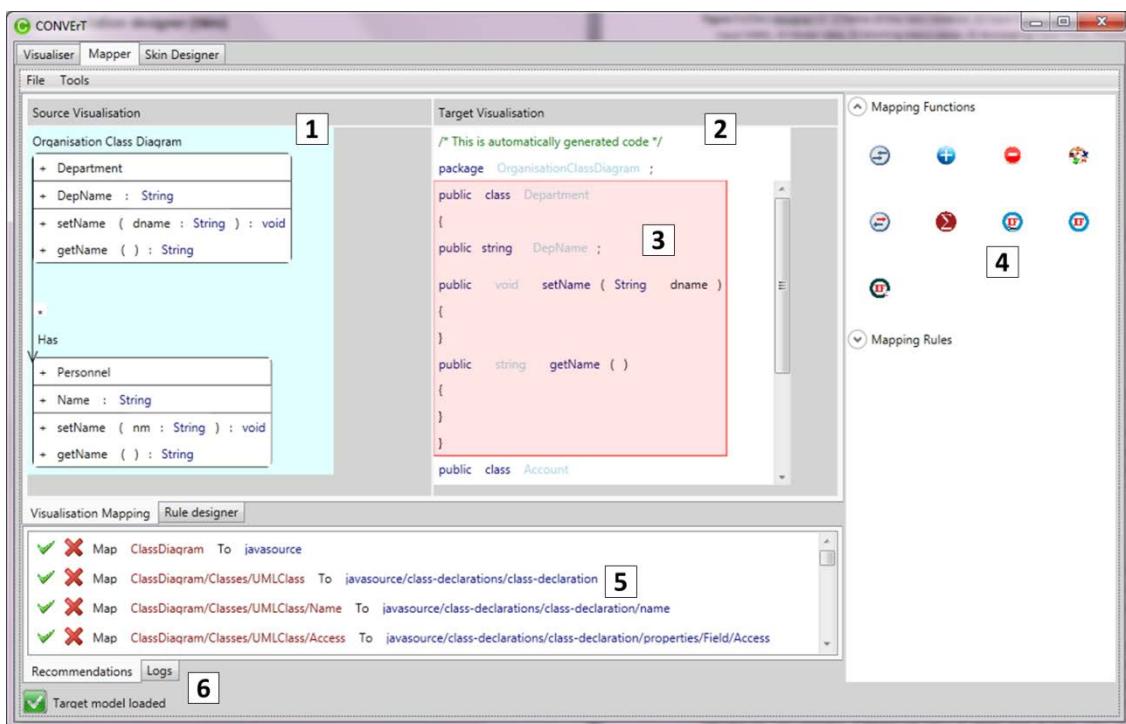


Figure 7.5 CONVErT's mapper UI. 1) Source visualisation, 2) Target visualisation, 3) Highlighting elements, 4) Functions and conditions, 5) Recommendations and 6) Status panel.

CONVErT facilitates a highlighting mechanism to guide users on choosing the intended notation to drop on. When a source notation is being dragged on top of target visualisation, this highlighting mechanism highlights the element under mouse cursor.

An example of highlighting a Java class is provided in Figure 7.5(3). In this example a UML class diagram notation is being dragged (dragging element is not shown) on top of a Java class notation. As a result, the Java class notation is highlighted in red.

Having source and target visualisations loaded, CONVERT’s suggester provides the list of recommended correspondences. In example of Figure 7.5 the suggester is recommending mapping correspondences between UML class diagram and Java code (Figure 7.5(5)). When a source notation is dropped on a target notation, this recommendation list is automatically updated to represent the correspondence recommendation related to those notations. For example, when a UML class is dropped on a Java class, the list will be updated to show the internal correspondences of UML class and Java class (see Figure 7.6(10)).

Similar to visualiser, mapping functions and conditions are also available in mapper window (Figure 7.5(4) and 7.6(4)). These functions are dragged and dropped on the provided canvas to be used for specifying more complex model transformation correspondences. In case more complex rules are to be specified, the default notation views of both source and target notations will be automatically rendered in rule designer panel when source notation is dropped on target notation (see Figure 7.6(7))). This rule designer canvas allows definition and use of constant values, strings, functions and conditions. For example if name of an UML class is to be altered by adding a “_Class” string to its name, a merging function can be used (Figure 7.6(9)). The name of the UML class will be dropped on first argument of the function and the “_Class” can be provided using the available value specification UI (Figure 7.6(8)). The defined “_Class” can be dragged and dropped on second argument of the function and the function output will be dragged and dropped on name element of Java code.

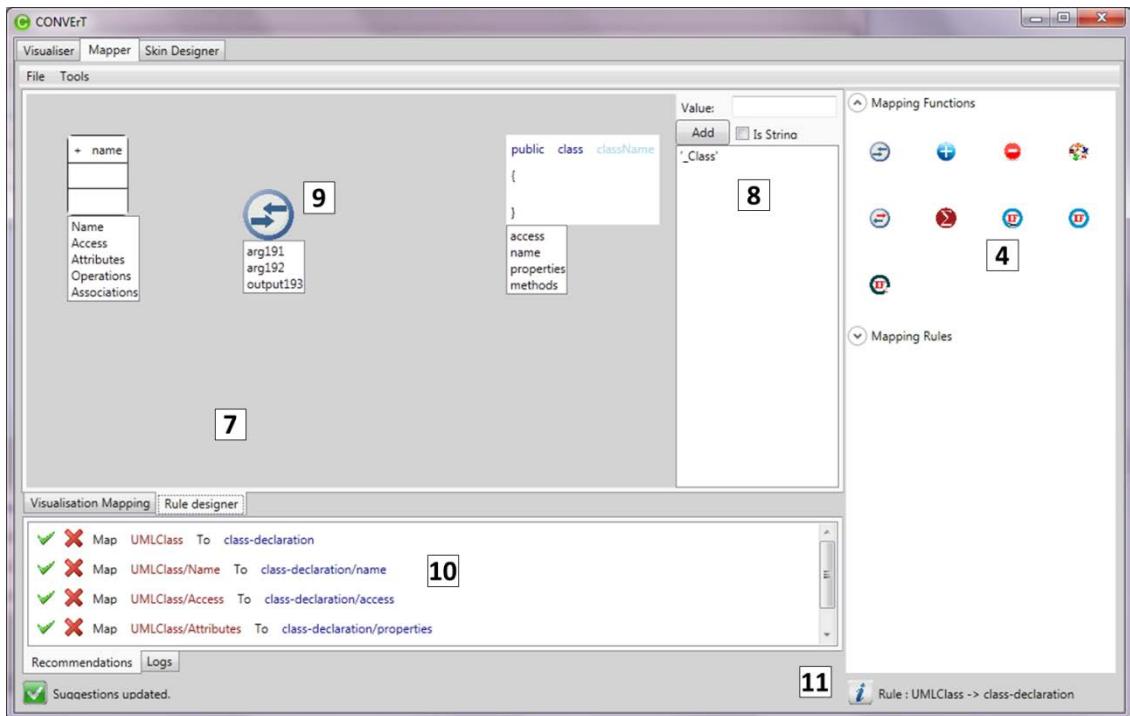


Figure 7.6 CONVErT’s rule designer. 4) Functions and conditions, 7) Rule designer canvas, 8) Panel for adding values, 9) A merge function, 10) Updated recommendations, 11) Rule designer status.

Mapper also features a status bar for reflecting system status (see Figure 7.5(6)) and logs panel to log user interaction (see Figure 7.7(13)). This log works similar to visualiser’s log and records users’ interaction with the system and automated tasks.

A dedicated panel has been provided in Mapper UI to display the transformation rule currently being created (Figure 7.6(11)). This panel helps users identify the source to target rule being created and whether the rule has been saved or not. For example in Figure 7.6, the rule being created is for transforming a UML class to Java class notation (Figure 7.6(11)). Once the rule is saved, this status bar will be cleared.

If user drag and drops a notation of source visualisation to a notation in target visualisation, underlying model of the notations are used for generation of a visual representation for the resulting rule. Since notations in our approach include the controller transformation for transforming their model to their view representation, it is possible to regenerate their view at any time using the embedded controller transformation. As a result, once a transformation rule is saved, a visual representation of the rule can be generated using underlying model of source and target notations and

their controller transformations. These visual rules will be placed on “Mapping Rules” panel and will depict the source and target notations corresponding to that rule. For example, Figure 7.7(12) demonstrates a transformation rule for transforming UML class to Java class. Note that the values provided to these rule notation representations are provided by their default models, therefore regardless of which UML class or Java class notation is used for rule generation, the representation is always the same. For example in Figure 7.7(12) the default UML class diagram notation has “name” for its class name and is a public class. Similarly Java class notation is a public class and has “className” as its default name.

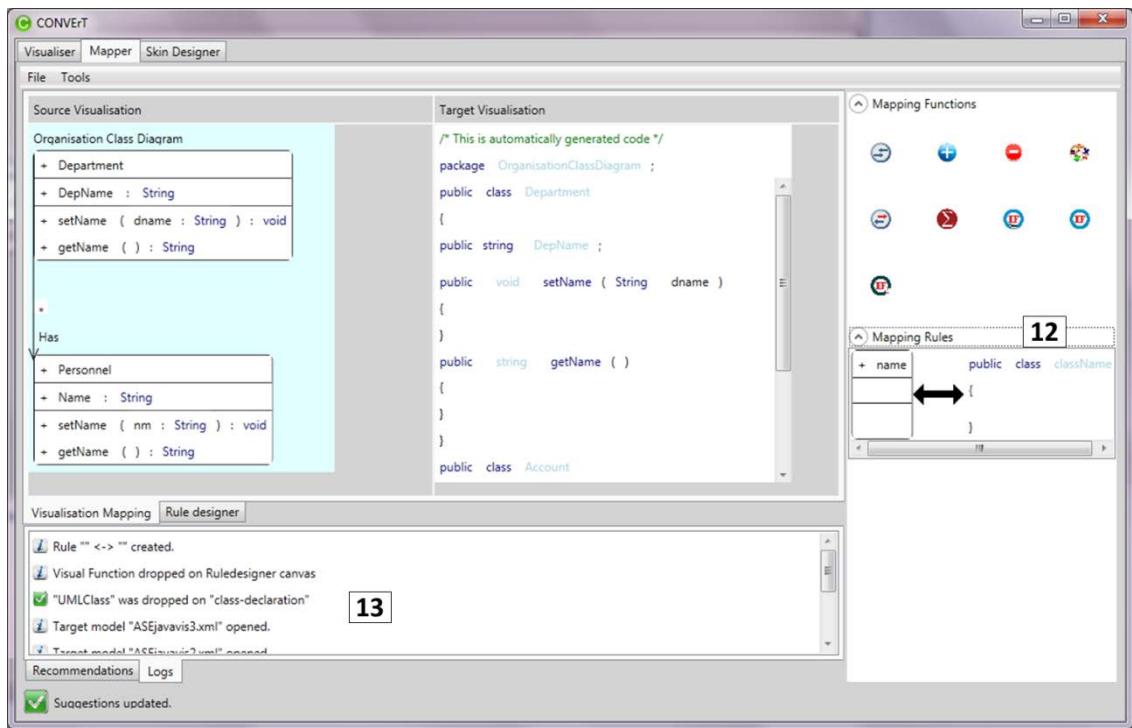


Figure 7.7 CONVERT’s UI for visual representation of transformation rules (12) and 13) User logs.

7.3.3 Notation designer (Skin++)

To increase applicability of our approach, it should be possible to define new notations depending on application domain of the models being used for transformation and the required visualisations. As a result, notation designer or Skin++ provides facilities to define, alter and add notations to CONVERT.

As previously described (Chapter 4) notations in CONVErT provide a model, a view and a controller transformation to generate view from the model. Views in our implementation are generated using Extensible Application Markup Language (XAML) [179]. XAML is a declarative mark-up language based on XML and is used for designing UIs and visual applications. Since XAML is XML-based, it is a suitable option for integrating visualisations in various tools and to be used in model transformations generated by XSLT. Visualisations generated by XAML are renderable in most browsers like Internet Explorer and Mozilla Firefox. This gives CONVErT the capability of exporting generated visualisations across different platforms.

To generate notations, CONVErT provides a separate UI for users to define new visual notations and save them in notation repository for reuse. This UI (called Skin Designer) allows importing graphical XAML views and provides facilities to render and edit them. Figure 7.8 shows an example where a XAML view for a bar to be used in bar chart visualisation is imported to Skin designer (Figure 7.8(2)). The designer provides rendering of the imported View in a separate UI compartment (Figure 7.8(3)). Users can alter the imported XAML view and see the results accordingly.

According to the desired visualisation, provided visual notation views are linked to model data to generate complete notations. A data XML should be specified to represent the data part of the visual notation. For example, Figure 7.8(4) shows the data provided for a bar's notation. It includes a name, a value and a colour. The data XML can be imported or generated from scratch in the provided section of the UI.

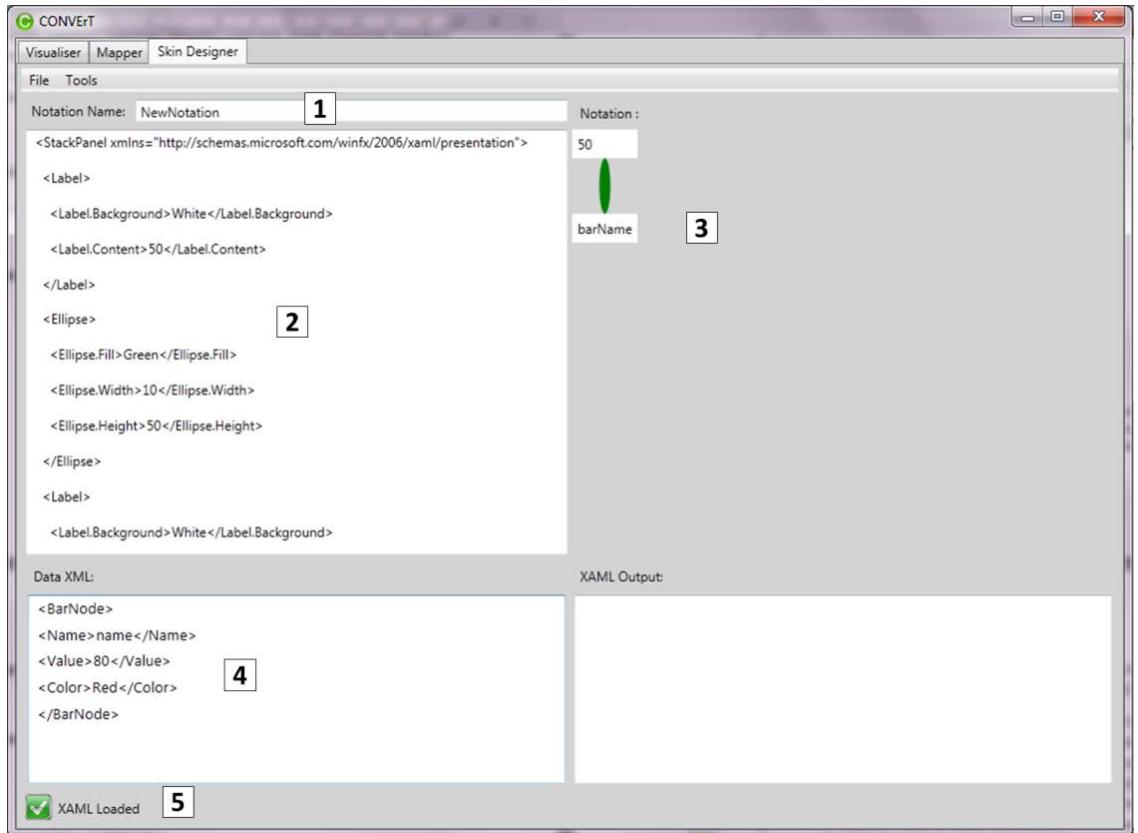


Figure 7.8 Notation designer user interface, 1) Name of the new notation, 2) Input XAML view, 3) Rendering of the input XAML, 4) Model data, 5) Status panel.

The provided data is mapped to the view using a controller transformation. To link data to the view, an annotation script is designed to specify one-to-one and one-to-many mapping correspondences between data and the view. These mapping correspondences are specified by “linkto” and “callfor” annotations accordingly. The same panel used to load and edit XAML, is used to annotate the view. Figure 7.9(6) shows an example where a user is specifying annotations for linking elements of a bar’s data model to the provided bar’s view.

Once annotations are specified, a controller transformation code is generated that transforms the values provided by the data to the XAML view (Figure 7.9(7)). Using this controller transformation, the resulting notation is generated and rendered in notation panel (Figure 7.9(8)). Saving a notation generated here will automatically insert it to the notation repository. For example, in Figure 7.9 the new bar’s notation

will be inserted in the notation repository and can be used to generate bar chart visualisation.

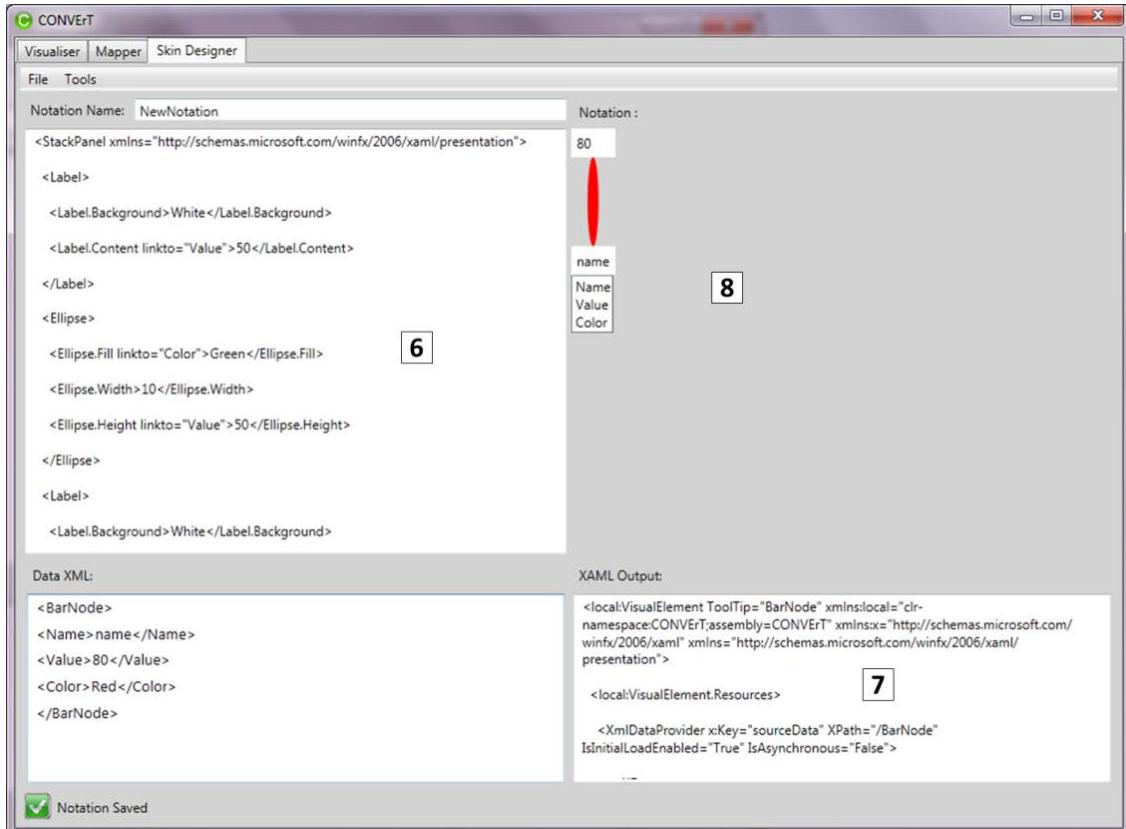


Figure 7.9 Using notation designer UI to annotate input view, 6) Annotated view 7) Controller transformation, and 8) Generated notation.

7.4 Implementation

To implement CONVERT, Microsoft Visual Studio was chosen as the implementation framework and IDE since it provides seamless integration of XAML graphics. As a result, the graphics used for notation design and visualisations could be natively designed, rendered and altered. Also, Visual Studio generally provides good backward compatibility which helps the implementations and the project to still be runnable and maintainable over the course of time.

Given that visual studio is not a freely available IDE, our decision was to implement CONVErT as a standalone application rather than an IDE integrated application. This would allow users to use CONVErT as a desktop application without the need to have visual studio installed.

CONVErT's code has been implemented using C# and latest available versions of .Net framework (version 4.5). This decision was partly due to availability of the required expertise, and partly due to use of XAML graphics. XAML graphics can be controlled nicely with the code behind written in C# or Visual Basic (VB). As a result, the layout controls of the graphics could be easily implemented using the programming language of the framework.

The technologies used for implementation of CONVErT are focused on Microsoft Windows operating system and there are certain concerns with regards to cross platform execution of the toolset, i.e. the tool cannot be executed on Macintosh or Linux based operating systems. Although it is possible to use CONVErT using virtual application environments, we are investigating possibilities for implementing a lighter version of the toolset as web based application where users can work with the tool using their browsers. Web based and possible mobile versions of the tool are parts of our future work.

Following sections briefly describe implementation of user interfaces, visual rendering mechanism, abstraction, Suggester and transformation subsystems.

7.4.1 User interface design

Components of CONVErT's UI are implemented in Microsoft Visual Studio and are designed in Windows Presentation Foundation (WPF). WPF allows implementation of UI using the simple XML-based representation of XAML and is natively available as part of the visual studio. Using WPF the logic behind each element of the UI like windows, menus, or interactions is separately implemented by C# code. Therefore, it provides a good separation of concerns between elements of the UI and the subsystems and components of the tool.

For example, when users request transformation code to be generated, the implemented C# code behind the pressed menu button will call the appropriate code generation components. Or when users drop a visual notation on a visual designer, the code behind the UI will call the Suggester system and passes the notation's model and the loaded input model to the Suggester.

Special considerations were made to make different windows of CONVErT's UI consistent. For example the menus of all three parts of CONVErT provide only two categories designated by File and Tools. Tasks like loading models and visualisations, or importing XAML are grouped under File menu. Specific tasks related to each window is grouped under Tools menu. For example, saving a customised notation or generating a transformation code based on a set of composed notations are provided in Tools menu of visualiser, while the tools menu of Mapper has dedicated buttons for saving a mapping rule, generating transformation between visualisations or clearing defined rules.

7.4.2 Visual Rendering

The rendering of visual elements in CONVErT reuses the controller transformation of notations available in notation repository. Its components are depicted in Figure 7.10. When a visualisation is to be rendered, convert checks the visualisation file against the controllers of the notations in the notation repository. It uses a visitor pattern to search the visualisation file for constructs similar to data part of notations' MVC where those controller transformations could be applied. Controller of the matching notations will be returned by this visitor.

From retrieved controller transformations, the renderer generates a transformation script to transform the file to be visualised to XAML representations. Transformation engine of CONVErT then executes this transformation code on input file and renders the result on visualisation canvas.

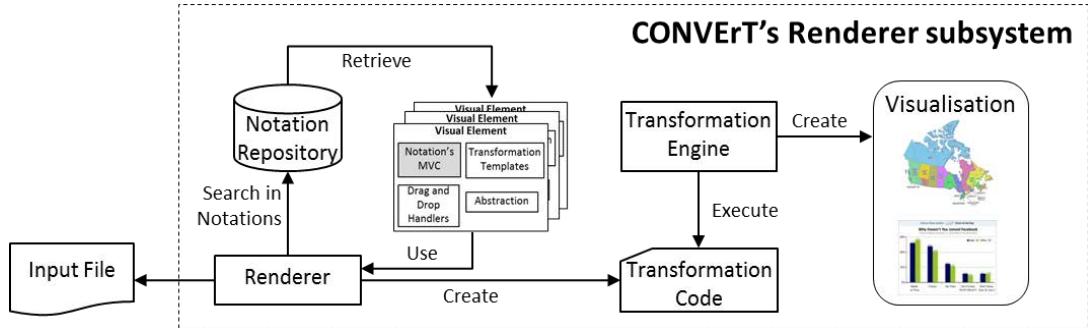


Figure 7.10 Components of renderer subsystem.

Example 7.3 Assume a file to be visualised is given similar to Figure 7.11. Renderer's visitor checks the notations available in the repository and finds two notations with similar models as Figure 7.12a and 7.12b. Figures 7.12c and 7.12d show the model parts of these notations accordingly.

```

<HorizontalBarchart>
  <Name>Question 13</Name>
  <ScaleName>Group 2</ScaleName>
  <HBars>
    <HorizontalBar>
      <Value>50</Value>
      <Name>5</Name>
      <Colour>Green</Colour>
    </HorizontalBar>
    <HorizontalBar>
      <Value>60</Value>
      <Name>4</Name>
      <Colour>LightGreen</Colour>
    </HorizontalBar>
  </HBars>
</HorizontalBarchart>

```

Figure 7.11 Sample visualisation file to be rendered by Renderer.

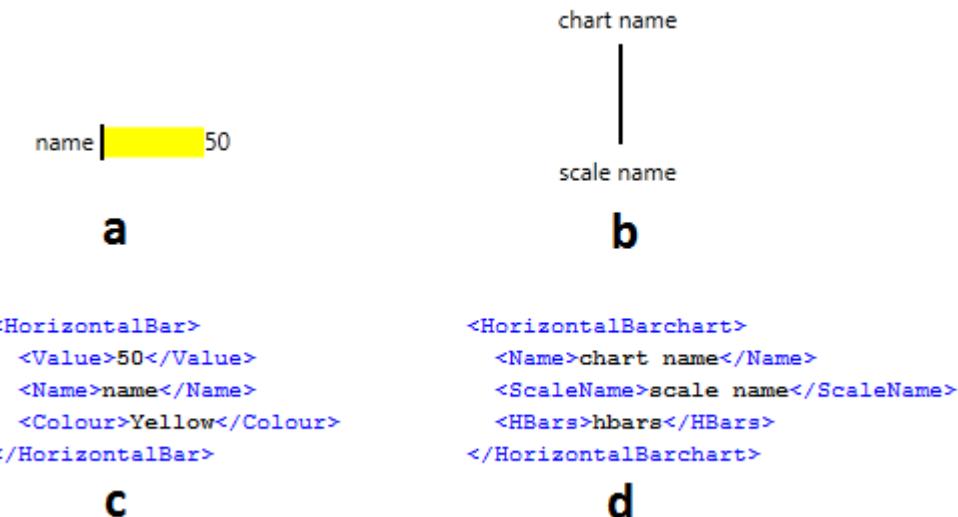


Figure 7.12 Example of notations retrieved from notation repository. a) Horizontal bar, b) Horizontal bar chart, c) Horizontal bar's model, and d) Horizontal bar chart's model.

Once the visitor completes checking of the input visualisations file, the returned controller transformations are treated as transformation rules and a transformation script is generated from those rules. These notation controllers are called declaratively since appropriate call to other rules are already specified in the controllers during notation design phase (recall Chapter 4 where the “linkto” and “callFor” annotations were used to generate controller transformations for notations). The result of the final transformation script would be the rendered visualisation similar to Figure 7.13.

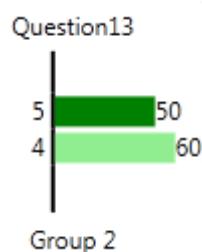


Figure 7.13 Final rendered visualisation.

Renderer's visualisation capability is limited to the notations available in the notation repository. Also, if duplicated notations with different controllers are available in repository, the rendering might not result in the desired visualisations. As a result, CONVErT provides facilities for importing and exporting notations into notation

repository. The generated notations available in the repository can be exported in an importable format, i.e. the defined new notations will be saved in XML files and can be imported at later time. This will allow users to be able to expand and control visual rendering capability. The exported XML files include model data and the controller transformation of each notation. Once imported, CONVErT uses the controller transformation on provided default model data to generate the default visual representation of the notation in repository.

7.4.3 Abstraction

The Abstraction subsystem of CONVErT uses a graph lattice to keep structural information of input models and data. Its components are depicted in Figure 7.14. It uses a visitor which traverses input model or data and sends graph constructs (nodes, links) to the abstraction system. The abstraction checks these constructs in a graph lattice. The elements returned by the visitor get added to the lattice if constructs similar to them has not been added previously. Each graph node of this lattice has a collection for keeping values seen in that element position of the model. These collections are used by value similarity recommender of the Suggester subsystem for calculating value similarity score. Also, through use of a type checker, Abstraction updates the type of the visited graph nodes of the input file. These types will be used in type similarity recommender.

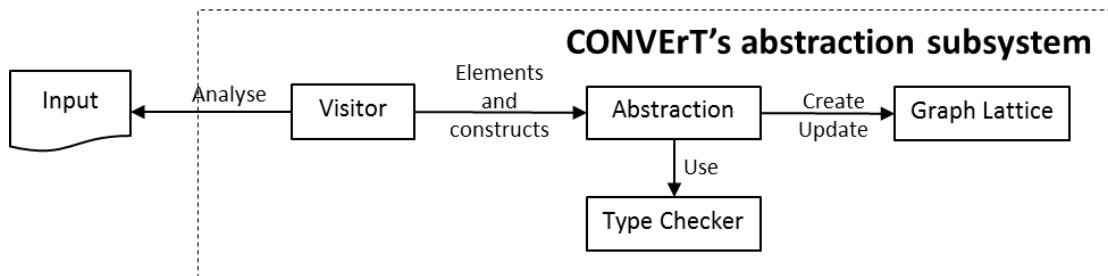


Figure 7.14 Components of CONVErT's abstraction subsystem.

Example 7.2 Assume the input model is similar to Figure 7.15. The visitor of abstraction subsystem traverses the input file and records every structural

construct it faces. These constructs will look like the graph of Figure 7.16. Note that the first “Sales” element in the example does not include an “Amount” element, but since the visitor checks all elements of the input file it has provided the structure in its position in the abstraction. Also the reverse engineered types and the values retrieved from the input are provided in the abstraction graph.

```

<Spreadsheet>
  <name>Sales Records</name>
  <sales Region="Europe">
  </sales>
  <sales Region="Asia">
    <Amount>50</Amount>
  </sales>
  <sales Region="America">
    <Amount>40</Amount>
  </sales>
</Spreadsheet>

```

Figure 7.15 Example input model representing Sales elements.

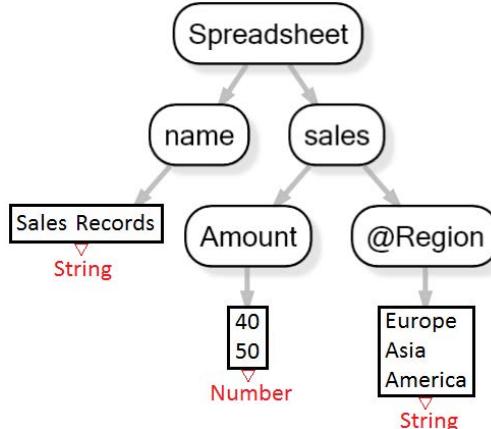


Figure 7.16 Abstraction graph of input file in Figure 7.15.

7.4.4 Suggester System

The implementation of Suggester system follows an ensemble learning strategy. Figure 7.17 shows the components of this system. Each correspondence recommender considers source and target inputs as graphs and calculates the similarities of pairwise elements using defined similarity heuristics. The results of these calculations are then normalised and returned to Suggester as a similarity matrix.

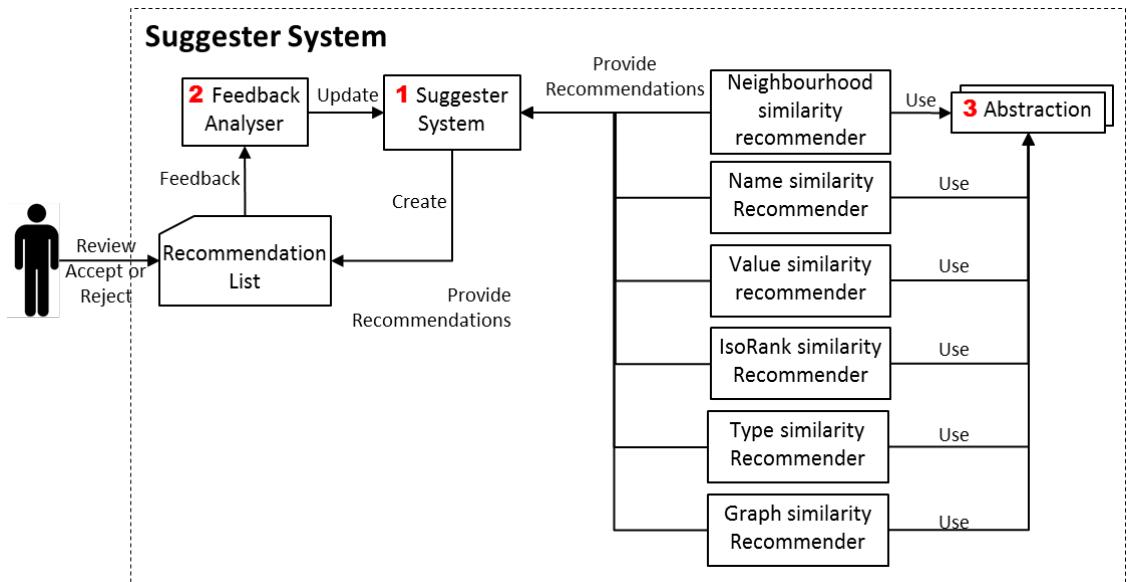


Figure 7.17 Components of Suggester system.

Suggester maintains a set of confidence values for the recommenders. These confidence values are initially set to 1. Based on the feedback collected from users in terms of accepting or rejecting recommended correspondences, the Suggester updates these weights to increase or decrease confidence. These confidence values are then multiplied to the similarity matrices. The resulting similarity values are summed up to calculate the final similarity matrix of pairwise element correspondences.

An option is provided to users to limit or increase the number of suggestions presented to users per pair. By default Suggester returns only one recommendation per pair. It considers the stable marriage approach to return only the one option that results in better overall recommendation list [177]. Users can modify this option to return the desired number of recommendations per pair.

Each suggestion is presented to users with a reject or accept button. Hitting accept button not only updates the weights of the recommenders that came up with the recommendation, it also sends the correspondence to the control unit of the active window. This control unit checks the task that was being performed and the returned recommendation. If for instance the task is to map input model to visual notations in the

Visualiser, the control unit looks to see whether it can find the source and target elements of the selected correspondence in the source input and target visual notation's model. It then applies the correspondence accordingly.

Spotting correspondences in the visualisations for mapping from a source visualisation to a target visualisation was a challenging task due to complexity of some visualisations in XAML. As a result, we have modified the Renderer to always return a list of visual elements that are available in each visualisation as well. With this list the controller unit in mapper can simply look for the source and target of a correspondence in the list rather than the visual tree of the visualisation, and apply the correspondence accordingly.

7.4.5 Transformation

The transformation subsystem of CONVErT is depicted by Figure 7.18. Transformation engine used in CONVErT is the embedded transformation engine of Microsoft Visual Studio. It is currently limited to XSLT version 1.0. However, this limitation has not affected the applicability of the approach since the transformation code generator is tuned for generating XSLT 1.0 transformation scripts.

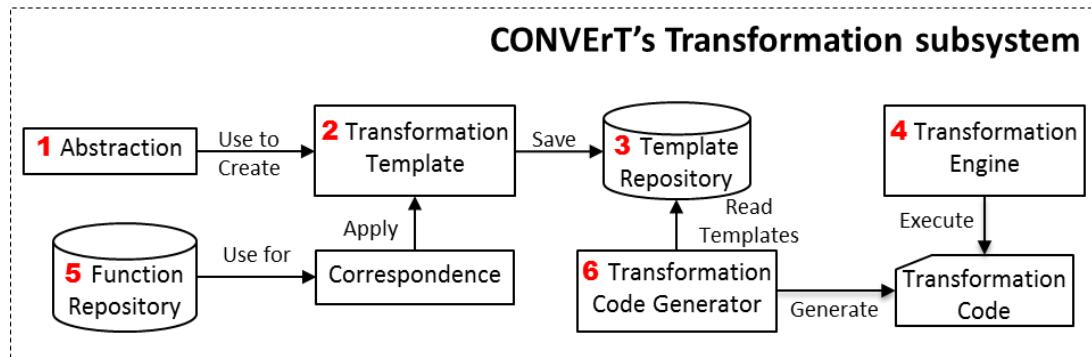


Figure 7.18 Components of CONVErT's transformation subsystem.

The transformation code generator uses the transformation templates embedded and defined in notations to generate XSLT code scripts. Once defined, these templates are available in template repository (Figure 7.18(3)). Depending on where the

transformation code is being generated (in visualiser or mapper), this template repository could be provided by customised notations, or mapping rules. Code generator reads these templates and depending on correspondences specified inside each template, generates required value fetches or function codes.

Example 7.3 Assume the transformation template specified in a customised notation is similar to Figure 7.19. It describes a transformation template for transforming sales elements to bar's visual notation.

```
Transform Sales to {
    BarNode
    Internal model:
    {
        Map @Region To BarNode.Name
        Function1 Add Amount1 and Amount2 -> put results in
        Output1
        Map Output To BarNode.Value
    }
}
```

Figure 7.19 Transformation rule template for transforming a sales record to bar notation.

Once users drag and drop elements, system inserts the address of the elements being dragged and dropped into forward and reverse templates accordingly. The transformation template of Figure 7.19 is the result of interactions shown in Figure 7.20.

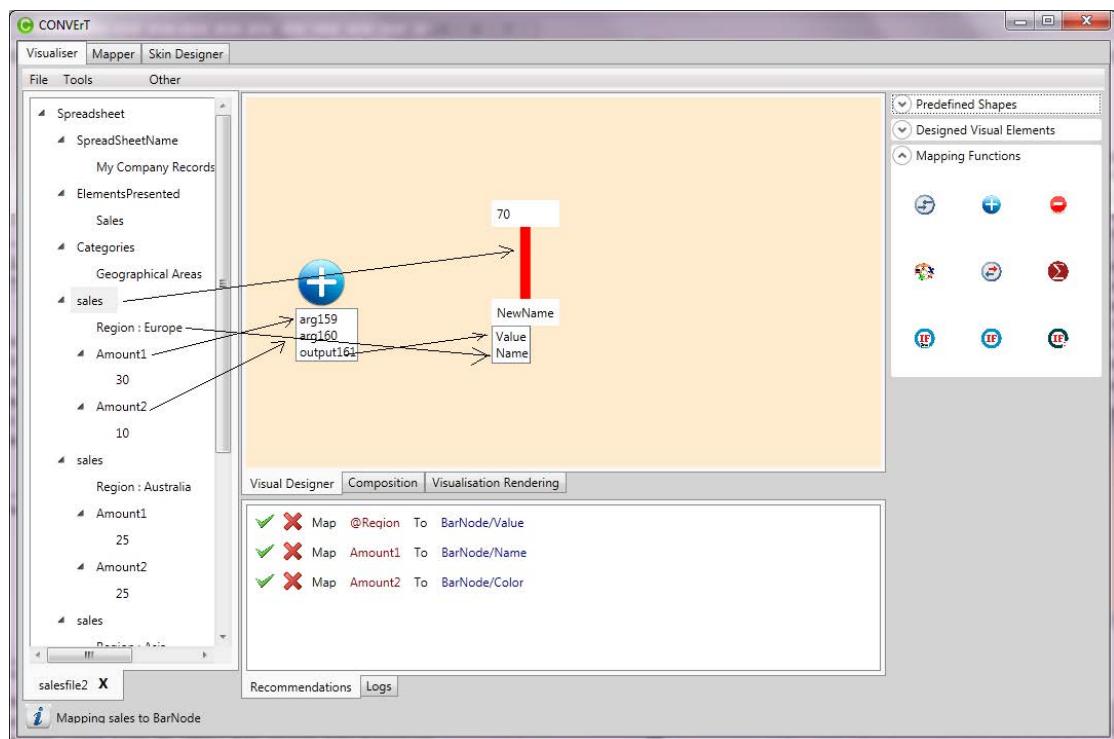


Figure 7.20 Defining the transformation between sales element and bar's notation. Arrows depict drag and drop.

By requesting to generate transformation scripts, the code generator reads the templates and generates the transformation script. For example the transformation script generated from template of Figure 7.19 is shown in Figure 7.21. A collection of these transformation scripts are used to generate final transformation code. The transformation code will be used in the transformation engine. In this case, the XSLT transformation engine of Visual Studio runs the generated XSLT transformation code.

```

<xsl:template match="sales" xmlns:xsl="...">
  <BarNode>
    <Name>
      <xsl:value-of select="@Region" />
    </Name>
    <xsl:variable name="arg225" >
      <xsl:value-of select="Amount1" />
    </xsl:variable>
    <xsl:variable name="arg226" >
      <xsl:value-of select="Amount2" />
    </xsl:variable>
    <xsl:variable name="output227">
      <xsl:value-of select="$arg225 + $arg226 " />
    </xsl:variable>
    <Value>
      <xsl:copy-of select="$output227" />
    </Value>
  </BarNode>
</xsl:template>

```

Figure 7.21 Resulting transformation code.

Since templates are the bases for transformation code generation, it is possible to provide alternative transformation language support, for example ATL. To do so, code generator components can be integrated to CONVErT to generate scripts from transformation templates similar to our XSLT code generator.

7.5 Usage scenarios

This section provides two usage scenarios in the form of Sequence Diagrams to show how user interaction with CONVErT's UI affects the internal components. Two major scenarios are described. The First scenario is where a user is generating a transformation rule to transform portion of input model to a visual notation. The Second scenario is when a user is defining a transformation rule between notations of two visualisations.

The first scenario is depicted by Figure 7.22. In this figure, Lars (as a MDE user) has loaded an input file representing a class diagram and has dropped a UML class notation

on the designer canvas. Given this situation, the visualisation window creates an instance of the notation and initiates Suggester system to provide list of correspondence recommendations. The corresponding sequence diagram for this scenario is depicted in Figure 7.23. This sequence diagram shows how visualiser window creates an instance of the visual notation and triggers Suggester for recommendations.

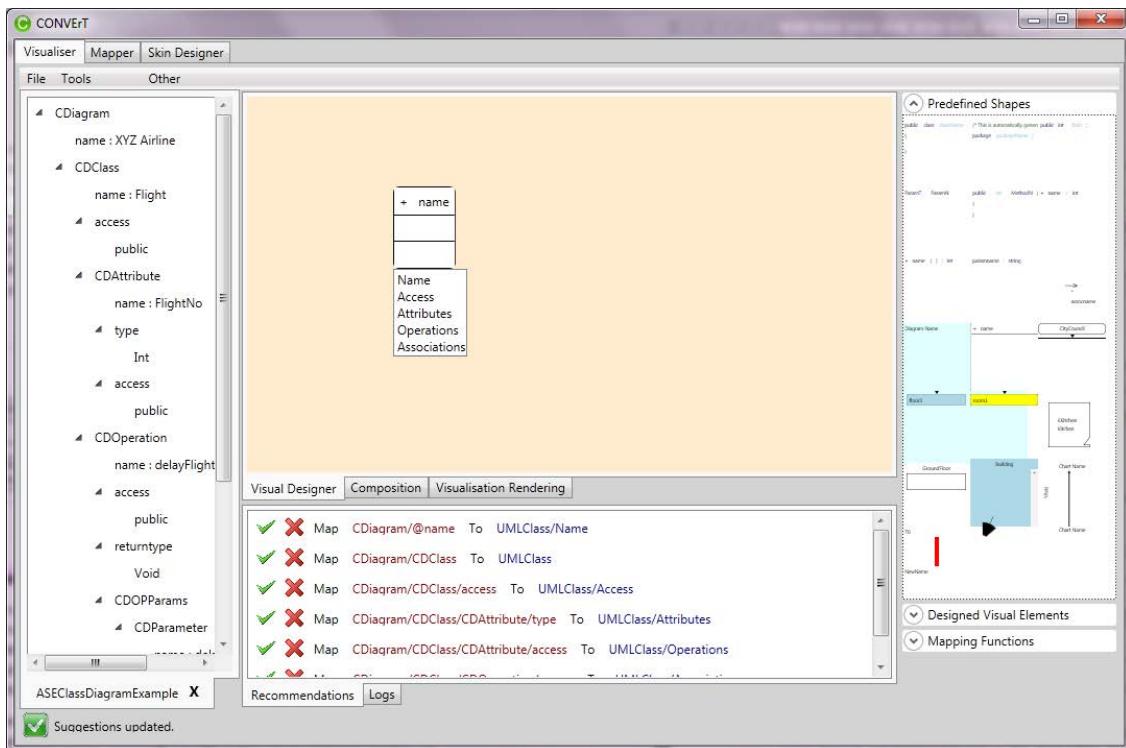


Figure 7.22 Transformation generation between a portion of input model and a visual notation.

Lars's first task is to drag and drop an element of the input model to be visualised on the notation to define which portion of the input model is to be transformed. An alternative is to select the same element from provided recommendation list. Lars can accept or reject recommendations. If he chooses from recommendations, the visualiser triggers Suggester to promote correspondence recommenders. Optional rejection of recommendations at each point will also trigger Suggester to update confidence weights of correspondence recommenders.

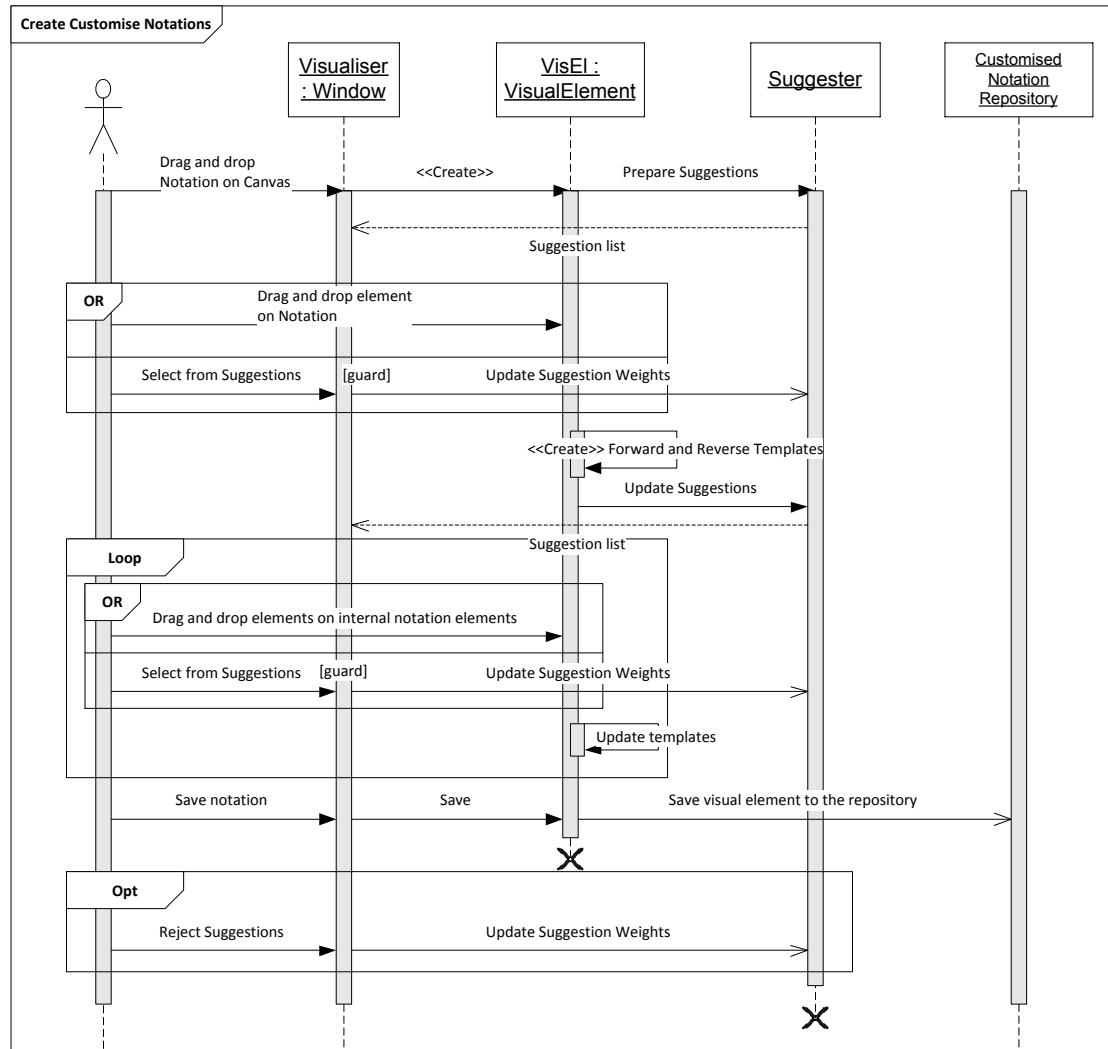


Figure 7.23 Usage scenario of creating a transformation rule between portion of input model and a visual notation.

Once the element to be visualised is specified to the system (by drag and drop or selecting from recommendations), transformation templates inside each visual element are created based on abstraction of the dragging element and the abstraction of the notation's data. The Sugester also updates the recommendation list accordingly to provide element specific recommendations.

At this point, Lars would select and define internal element correspondences with drag and drop or selecting from recommendations. Once all internal correspondences are defined, saving the notation will put the customised notation in notation repository for

future (re)use. This customised notation carries the required transformation templates to perform this model to visual notation transformation task. The transformation script from this notation however will be generated once Lars requests the transformation code to be generated. Note that if he makes a mistake during drag and drops, he can perform the drag and drop again to overwrite the previously specified correspondence. Once notation is saved, editing the customised notation is not possible that is a limitation of current version of the framework.

Figure 7.24 depicts the second scenario for specifying a transformation rule between notations of source and target visualisations. For this scenario, Lars has loaded source and target visualisations in Mapper window. Figure 7.24 shows the UML class diagram as source visualisation and Java code as the target.

Loading source and target visualisations will trigger the Suggester to provide a list of likely correspondences. The corresponding sequence diagram of this scenario is depicted in Figure 7.25.

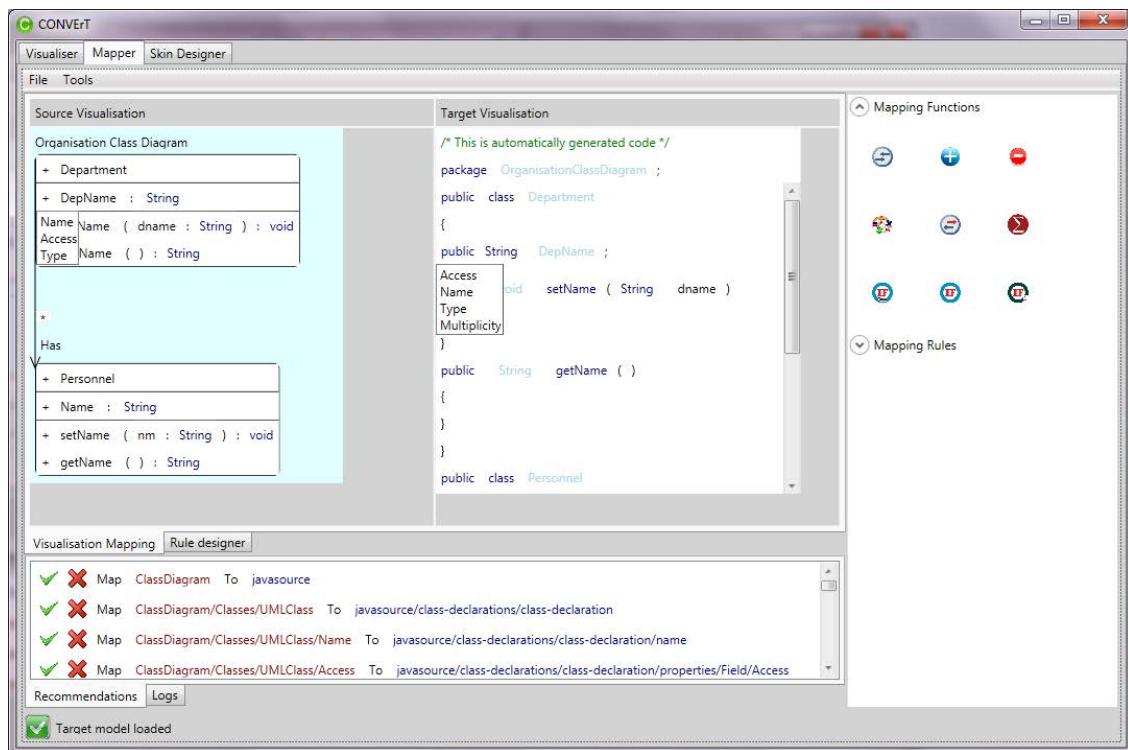


Figure 7.24 Transformation generation between notations scenario.

Loading source and target Visualisations results in the mapper window requesting Suggester system to update (or provide) recommendation list according to the source and target visualisations. Similar to the first scenario, Lars can choose the notation to be transformed from recommendations list or drag and drop the notations on each other. This will trigger initiation of transformation templates inside both notations and a request will be sent to Suggester to update recommendation list accordingly if a recommendation has been selected. Both forward and reverse transformation templates are triggered as a result of this interaction.

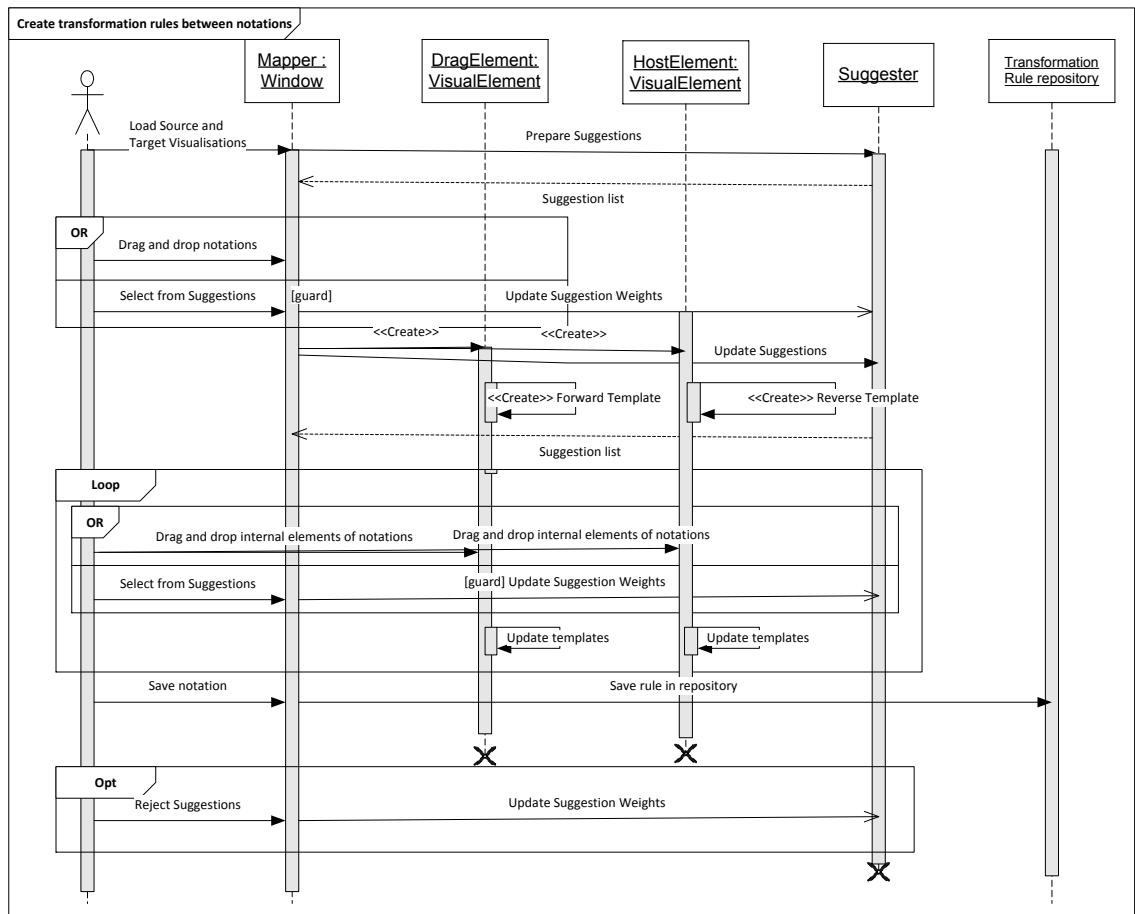


Figure 7.25 Usage scenario of creating a transformation rule between source and target notations.

Similar to previous scenario, Lars specifies internal correspondences by drag and dropping internal elements or selecting from recommendation list. Selecting or optional

rejecting of recommendations in the list will send a request to the Suggester to update confidence weights.

Once Lars has finished defining internal correspondences, he can save the transformation rule generated as a result of his interactions. Saving the rule will put the generated rule in the transformation rule repository. Visual representation of this transformation rule will be provided in the Mapping Rules section which is the reflection of this transformation rule in repository. Again a complete transformation script from these templates will be generated upon receiving a request for generating the transformation script.

7.6 Summary

This chapter described concrete visual assisted transformation (CONVErT) framework. CONVErT is the proof of concept tool for realisation of our approach presented by this thesis. It provides facilities for specifying correspondences on concrete and familiar visualisations of source and target models. It reverse engineers required abstraction from model examples and provides recommendations for possible source and target correspondences.

CONVErT is implemented in Microsoft Visual Studio and with C#. The visualisations and notation views benefit WPF and XAML, natively available in Visual Studio. Although current version facilitates XML and CSV as default formats for input models and examples, and uses XSLT as the transformation language of choice, the implementation can be extended to use alternative formats and transformation languages.

Chapter 8

Evaluation

8.1 Introduction

This chapter describes several evaluations of our CONVErT approach and toolset. The evaluation strategies adopted for this research consist of:

- A comparative study of tool support features;
- Quantitative evaluation of the Suggester mechanism and transformation code quality; and
- A user study of the tool's usability and functionality.

Comparative analysis was chosen in order to compare features and capability of the approach against other available approaches and toolsets, both research and commercial. It involves analysis of user modelling and interaction issues, and feature sets of the various toolsets. The comparison study also provides a list of features available in current approaches and toolsets, and that available in our approach and realised in our CONVErT prototype toolset. Among available toolsets, the approach provided in ALTOVA MapForce was the most comparable to the work embodied in this thesis [47]. Therefore, our detailed comparison uses MapForce as an alternative toolset to our CONVErT.

The use of quantitative analysis was targeted to evaluate the recommenders of CONVErT's Suggester system, by using a group of quantitative measures (e.g.

precision, recall and f-measure). This quantitative analysis also includes analysis of the quality of the generated transformation code using a set of code quality attributes and metrics.

The user study was designed to assesses a typical user's experience with the proof of concept tool implementation of CONVErT. It captures the users' perspective of the approach, the design and detailed visualisation and model mapping specification, and tool interaction issues. The following sections provide details of these three separate yet complimentary evaluations.

8.2 Comparative Study

This section provides a comparative study of our approach against a set of available transformation approaches discussed previously in Chapter 2. This comparison is divided into three parts. First, the most commonly used transformation languages are compared against the transformation specification language embodied in CONVErT. Second, available transformation tools are briefly compared to our CONVErT toolset. And finally, a discussion is provided to compare CONVErT and the commercial ALTOVA MapForce tool, which provides the most comprehensive comparison of current state-of-the-art commercial data mapping approaches and our novel approach.

Table 8.1 summarises how the transformation language of CONVErT compares to the most commonly used model transformation languages. In terms of technical space and the visualisation capability, CONVErT is not limited to a specific technical space. Instead, a variety of technical spaces can be visualised in CONVErT and used for transformation generation. However, since their visualisations need to be generated first, we have shown this option as partial support. Indeed if the visualisations are generated once, they can be reused many times.

Transformations in CONVErT are performed on visual notations. These visual notations may represent any technical space or be represented in a wide variety of shapes and formats. Correspondences are specified using drag and drop of these notational

elements and therefore no textual or graph-based coding is required. In terms of specification syntax, CONVErT is the only approach that provides visual specification for transformation generation that is not limited to textual coding or specific visualisations. Mapping functions, also expressed visually and added and linked by drag-and-drop metaphor, provide higher level complex model mapping and transformation constructs.

Table 8.1 Comparison of most used transformation languages and CONVErT's language. ✓ indicates support, (+) shows partial support and – shows no support.

Transformation language comparison	TGG [24]	ATL [68]	XSLT [69]	QVT [70]	MOLA [71]	ETL [72]	CONVErT [80]
Technical Space							
Model	✓	✓	-	✓	✓	✓	(+)
Text	-	✓	-	✓	-	-	(+)
XML	✓	✓	✓	✓	✓	-	(+)
Specification Syntax							
Text	✓	✓	✓	✓	-	✓	-
Graph	✓	-	-	✓	✓	-	-
Visualisation	-	-	-	-	-	-	✓
Input Artefact Syntax							
Abstract	✓	✓	✓	✓	✓	✓	-
Concrete	-	-	-	-	-	-	✓
Rule application control							
Imperative	-	(+)	✓	✓	✓	✓	(+)
Declarative	✓	✓	✓	✓	✓	✓	(+)
Transformation Scenario							
Vertical transformation	✓	✓	✓	✓	✓	✓	✓
Horizontal transformation	✓	✓	✓	✓	✓	✓	✓
Transformation Engineering							
Exogenous	✓	✓	✓	✓	✓	✓	✓
Endogenous	✓	✓	✓	✓	✓	✓	✓
Support for directionality							
Unidirectional	✓	✓	✓	✓	✓	✓	✓
Bidirectional	(+)	-	-	(+)	-	-	(+)
Multidirectional	-	-	-	-	-	-	-

Input models in CONVErT are at the concrete representation level. No abstraction is required for input examples and users can generate visualisations and transformations using their example models. Consequently, CONVErT is the only approach that allows input artefacts to be at concrete visual representation level. All other approaches require some form of meta-model or abstracted meta-model from model instance elements.

Similar to most approaches, CONVErT provides both imperative and declarative rule application control. The transformation of models to visualisations follows an imperative approach while transformation between visualisations uses a declarative approach. However, since the user has no control over which approach to choose at each step, we have indicated support for Imperative and declarative rule application control as partial in Table 8.1. We should note here that in design of CONVErT, our intentions were to provide a proof of concept prototype and therefore we have not included alternatives. Inclusion of alternative rule application controls or technical space is therefore part of our future work.

Visualisations of source and target models can be at any abstraction level. For example, source and target could be UML class diagram and Java code, or a UML 1.0 compatible and a UML 2.0 compatible class diagrams. As a result our CONVErT approach supports both vertical and horizontal transformations.

Similar to transformation scenarios, exogenous and endogenous transformations can be both implemented in CONVErT with regards to transformation engineering. For example, if there is a need to refactor a class diagram to remove a construct, it can be easily done using source and target visualisations. If the resulting target does not introduce new visualisation constructs, the exact same visualisation specification can be used for both source and target inputs.

Bijective correspondences, i.e. when elements of source and target represent one-to-one correspondence relations, are defined bidirectionally in CONVErT. As a result CONVErT tends to support bidirectional transformations. For non-bijective transformation correspondences (for example one-to-many, many-to-one and many-to-many correspondences), the reverse direction is not guaranteed. An example is vertical transformation in which information may need to be added or removed. As a result,

CONVeRt provides partial support for bidirectionality. Given that a capable transformation language is used in the transformation code generator, it is also possible to generate multidirectional transformations.

For the second part of our comparison, Table 8.2 provides a comparison of the tooling aspect of CONVeRt against other available transformation tools. It summarises how this tooling aspect of CONVeRt compares with previously realised transformation tools.

Table 8.2 Comparison of model transformation tools and CONVeRt. ✓ indicates support, (+) shows partial support and – shows no support.

Tool comparison	ATOM3 [96]	VIATRA2 [97]	ALTOVA [47]	CLIO [101]	GReAT [76]	ATL [68]	UMLx [98]	BOTL [100]	Form-based [32]	CONVeRt [80]
Application domain										
Model to model	✓	✓	✓	✓	✓	✓	✓	✓	(+)	(+)
Model to text	-	✓	(+)	-	-	✓	-	✓	-	(+)
Text to text	-	-	-	-	-	✓	-	-	-	(+)
Specification Syntax										
Text	✓	✓	-	-	✓	✓	-	-	-	-
Graph	(+)	-	-	-	✓	-	✓	✓	-	-
Visualisation	-	-	(+)	(+)	-	-	-	-	(+)	✓
Transformation Cardinality										
1-to-1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1-to-M	✓	✓	✓	✓	✓	✓	✓	(+)	✓	✓
N-to-M	✓	✓	✓	✓	✓	✓	-	(+)	✓	✓
Input Artefact Syntax										
Abstract	✓	✓	✓	✓	✓	✓	✓	✓	-	-
Concrete	-	-	-	-	-	-	-	-	(+)	✓
User interaction										
Textual	✓	✓	-	✓	✓	✓	✓	✓	-	-
Interactive	-	-	✓	✓	-	-	-	-	✓	✓
Drag-and-drop	-	-	✓	-	-	-	-	-	✓	✓
Support for directionality										
Unidirectional	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Bidirectional	✓	-	-	-	-	-	(+)	✓	-	(+)
Multidirectional	-	-	-	-	-	-	-	-	-	-
User support mechanism										
Interactive guidance	-	-	-	✓	-	-	-	-	-	✓
Visualisation	-	-	-	-	-	-	-	-	(+)	✓

CONVErT's application domain is not limited to a specific domain, instead it can be used for varieties of domains provided that the visualisations for those domains are specified in CONVErT. Therefore, Table 8.2 demonstrates all application domains to be partially supported by CONVErT. Similar to the languages aspect of CONVErT in Table 8.1, Specification syntax is based on model visualisations and the input artefact syntax is a concrete representation.

In terms of transformation cardinality, one-to-one transformations are specified by simple drag and drop. One-to-many and many-to-many transformations can be specified using transformation functions. Functions can be added, removed, altered according to the desired transformation specific tasks, again using drag-and-drop. New, complex transformation functions can be defined visually from simpler functions and parameterised, then reused by drag-and-drop.

CONVErT provides a very high level, highly interactive transformation specification approach through use of visualisations and drag and drop. It also helps users decide and explore which correspondences are possible. For example, when a notation is dropped on another notation, CONVErT tries to recommend correspondences according to internal elements of those notations. As a result, it provides an interactive approach for transformation specification.

User support in CONVErT provides both an interactive guidance mechanism (implemented as "Suggerer") and use of familiar, concrete model visualisations. Accordingly, these options have been selected as having full support in Table 8.2.

To provide a more in-depth comparison of our approach's features, we have selected ALTOVA MapForce as the most comparable tool to CONVErT. This selection has been based on the availability of the toolsets and the fact that MapForce is the only toolset that also fully provides correspondence specification using an interactive, drag and drop approach.

Figure 8.1 provides a screen capture of MapForce. It shows an example of EDI message mapping being implemented. Elements of the source and target are visualised by default using a tree-based representation (marked by 1 in Figure 8.1). To generate mapping,

these elements can be mapped using drag and drop. Similar to CONVErT, functions can be used and defined. For example a NAND function is marked by 2 in Figure 8.1.

To be able to generate a mapping between source and target input files, users need to provide these files to the toolset. If schemas are available, they should be provided. Otherwise, MapForce will try and reverse engineer a suitable schema from provided input examples.

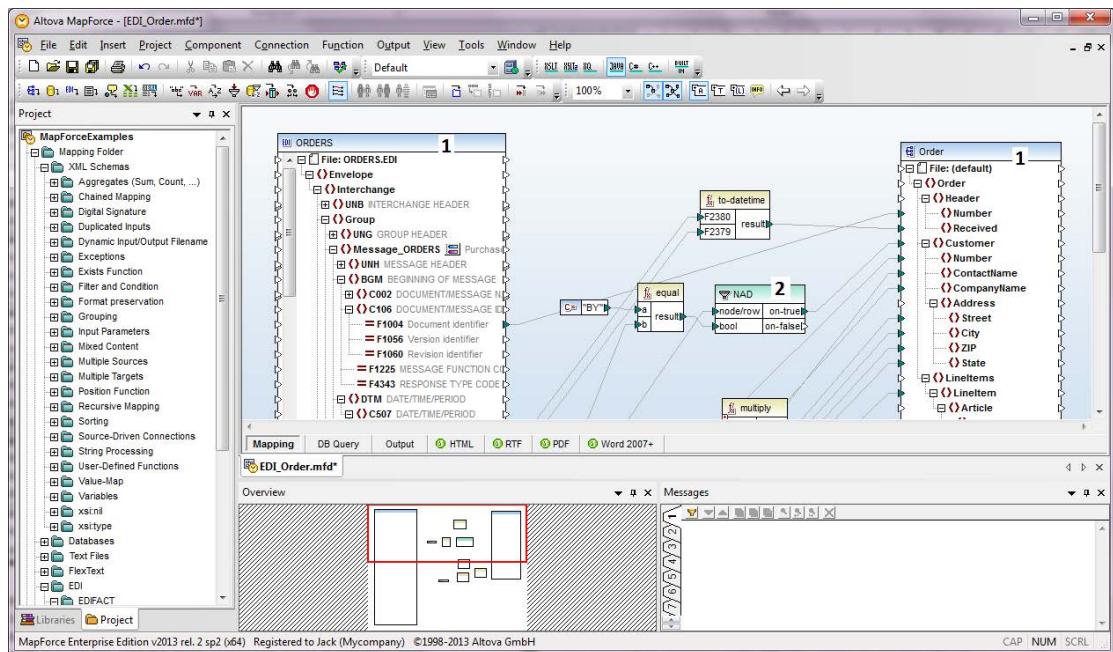


Figure 8.1 Screen capture of ALTOVA MapForce. 1) Default tree-based representations, 2) Using a NAND function.

Given that MapForce is an industry standard tool, it provides data importers from a wide variety of sources, e.g. database and Excel files. Also it generates mapping specification code in XSLT, XQuery, C#, C++ and Java to make it a more widely acceptable transformation and mapping tool for its target users.

MapForce allows text files to be used as source or target of the transformation. However, text support is limited to structured files, e.g. Comma Separated, and does not provide mapping between models and source code. It also provides mappings only in one direction. Although mappings of a reverse direction can be generated as a separate

transformation, our CONVErT approach does provide limited support for generating the reverse directions, where possible, while the forward transformation is being specified.

The biggest difference between MapForce and CONVErT's approach is in the way source and target are visually represented. Figures 8.2 and 8.3 show example of class diagram and Java XML input being used for transformation specification in CONVErT and MapForce respectively. As can be seen in these figures, our CONVErT approach is very flexible in terms of its use of visualisations and can provide drag and drop capabilities on notational elements as opposed to the fixed tree-based visualisation of MapForce.

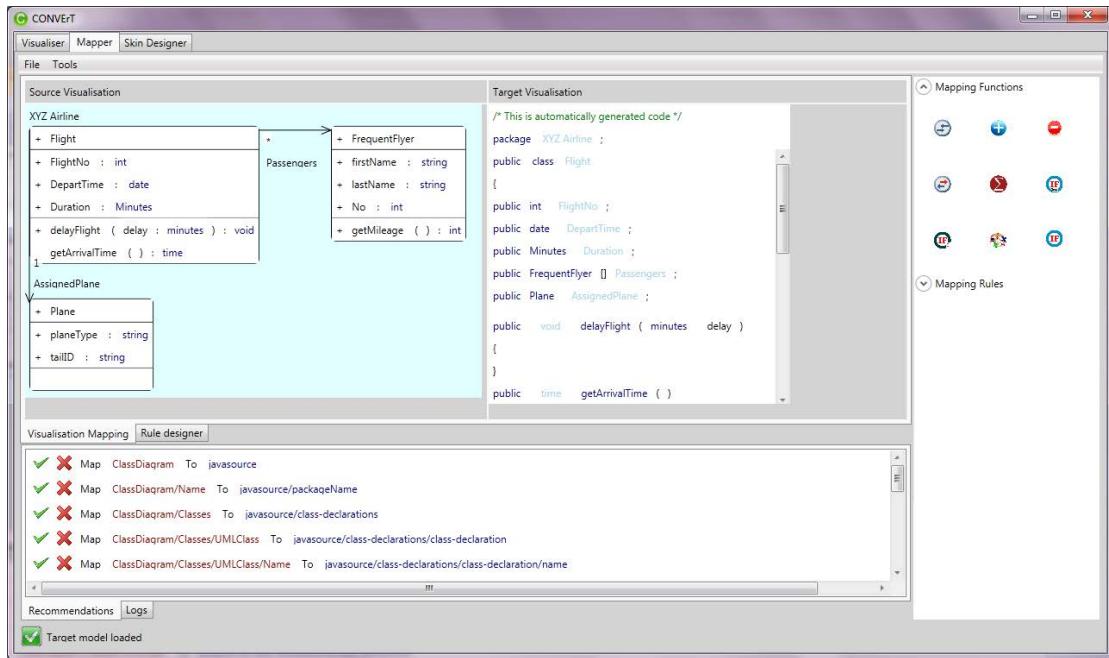


Figure 8.2 Mapping class diagram example to Java code visualisation example in CONVErT.

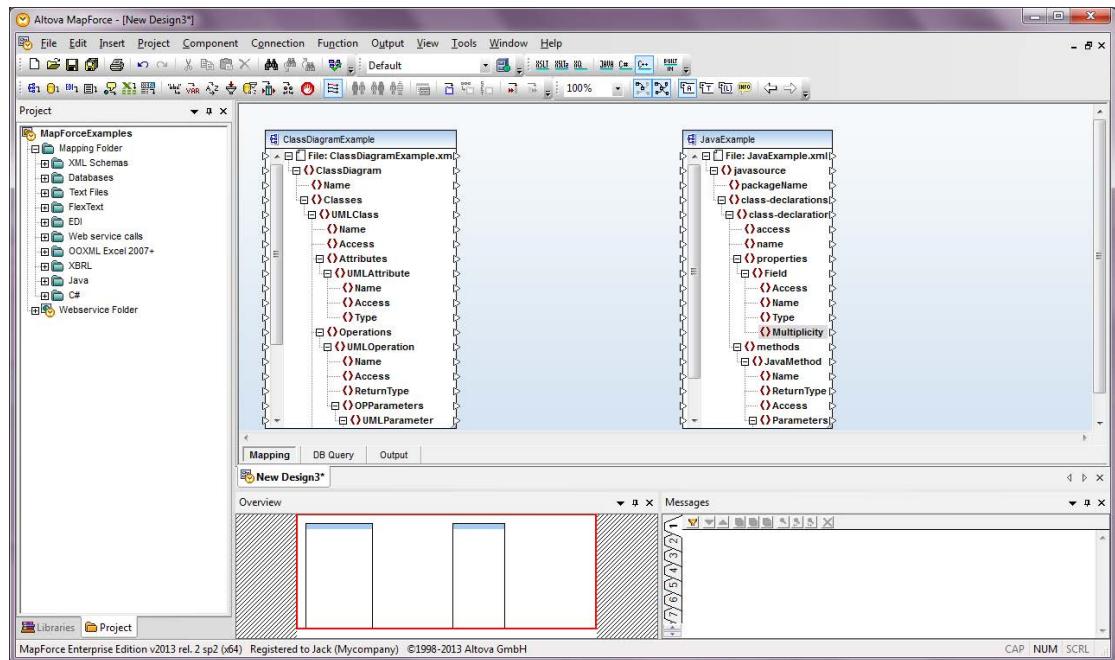


Figure 8.3 Representation of class diagram example and Java code example in MapForce.

Since mapping generation is done on schema elements rather than actual example elements, MapForce provides the reverse engineered schema as the source and target elements. The elements of the reverse engineered schema of both source and target are represented in the tree-based visualisation as depicted in Figure 8.3. Thus while CONVErT uses provided model instance example data in its visualisation, MapForce uses the provided or reverse-engineered schema elements.

Another distinctive difference between MapForce and CONVErT is in their representation of transformation rules and mappings. For example, the generated mapping rules for transforming UML class diagram to Java visualisation in CONVErT are marked by 1 in Figure 8.4. Similarly, the correspondences specified on examples of Figure 8.3 in MapForce are depicted in Figure 8.5.

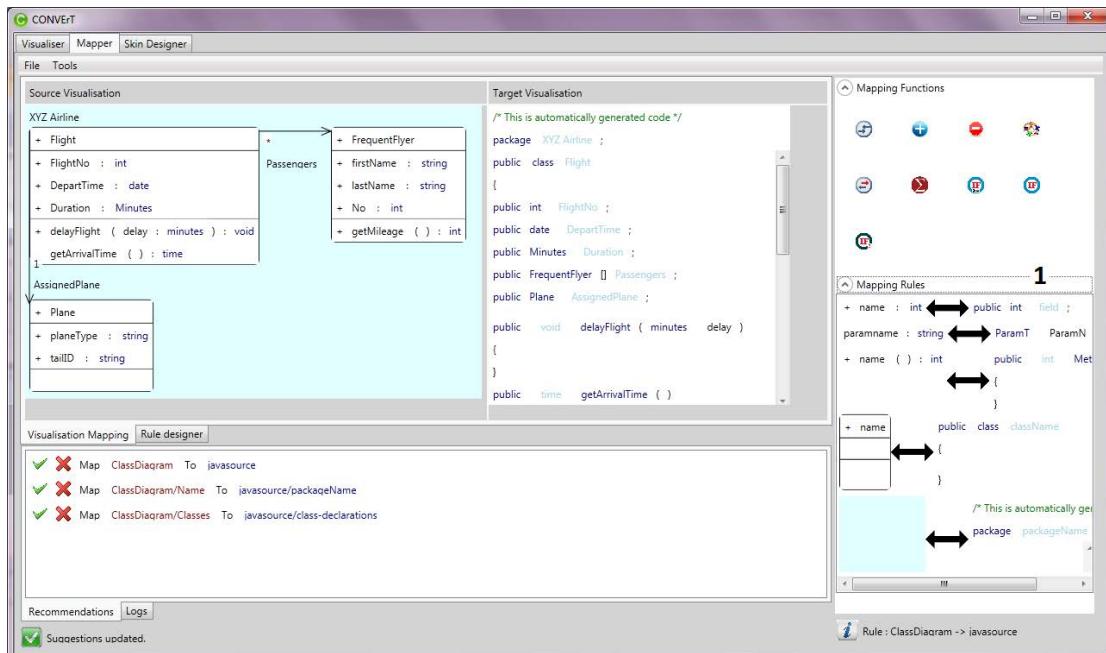


Figure 8.4 Generated transformation rules for transforming UML class diagram to Java visualisation in CONVErT. Transformation rules are marked by 1.

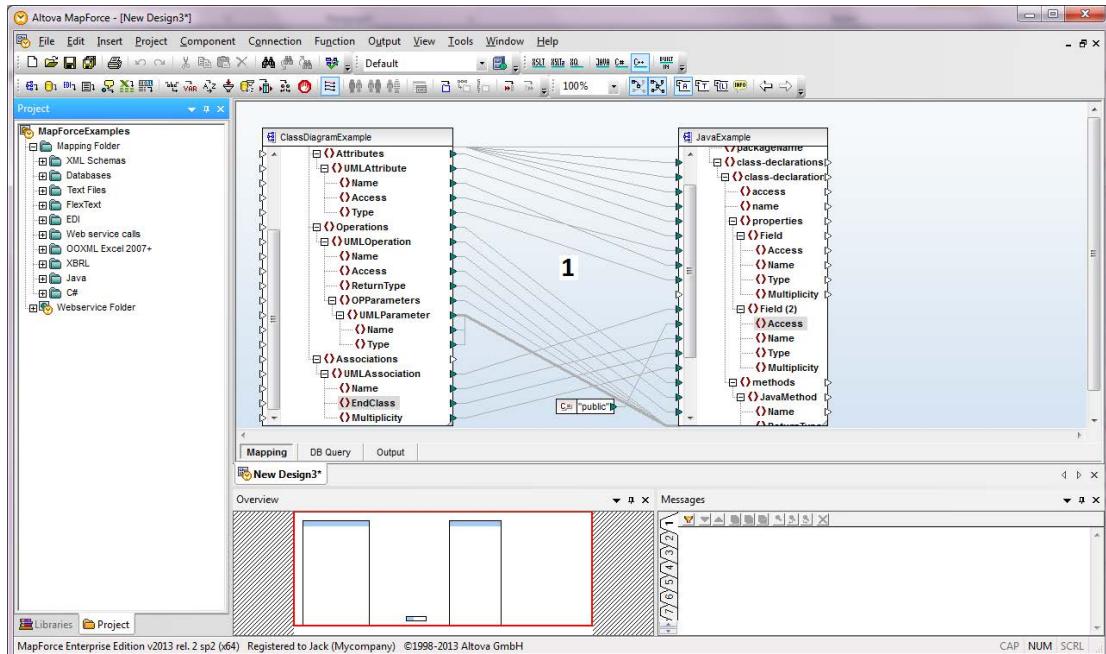


Figure 8.5 Correspondence specification between UML class diagram example and Java code example in MapForce. Mapping correspondences are marked by 1.

As can be seen in the figures, CONVErT follows a more procedural approach to specifying transformation rules and correspondences, i.e. a transformation script is

generated from a set of related transformation rules. These transformation rules are individually visualised and represented using source and target notation visualisations. Therefore, we claim that it is easier to define, explore and maintain transformations (this is tested in our user evaluation described below). In MapForce however, all correspondences are represented on the schemas using mapping lines. For larger examples, finding correspondences and maintenance becomes a challenging task. If functions are to be used, then the situation becomes even more complex. In CONVErT however, correspondences using functions and conditions are specified in a separate window and therefore will not be shown on source and target visualisations. As a result, using multiple functions or conditions will not complicate the visualisations.

CONVErT's approach will not be so affected by the number of transformation rules or correspondences since the defined transformation rules are saved in a separate UI compartment (marked by 1 in Figure 8.4). Considering the familiarity of users with the models being transformed, the source and target notations used to visualise transformation rules provide a good reference to track recently created rules. However, there are possible shortcomings for scalability, i.e. if the number of transformation rules increases, the list will be longer and users have to scroll down to find already generated rules. Also, during our test trials, it was suggested that it would be beneficial to provide facilities to highlight notations in the source and target visualisations which represent transformation rules, as the user hovers over the defined transformation rules in the list. This has been assigned to our future work.

When large examples are being used, the Suggester mechanism will also help users define and explore various possible correspondences. It recommends correspondences that can form both transformation rules and internal rule correspondences. MapForce provides a very basic help in this regard, i.e. when an element of source schema is mapped to an element on the target, it can automatically map their internal elements that represent same name and type. However, it is not capable of providing mappings for more complex situations.

MapForce's approach to defining correspondences has an advantage over CONVErT's in editing transformations. For example if users mistakenly link a source and target element and notice it at the end, they can alter the correspondence by unlinking and

linking the correspondence to the correct target element. In CONVErT's approach however, once the rules are defined and saved, they cannot be altered. To alter the transformation, new transformation rule should be defined and added and the faulty rule should be removed.

Both tools allow model checking of the resulting targets. MapForce uses the provided schema to check the target. CONVErT by default uses the reverse engineered abstraction to check the results. If schemas are available, they can be imported to CONVErT for model checking.

MapForce always generates the transformation script as a whole for transforming the whole source file to target at once. In CONVErT however, the transformation code is generated using the individually defined transformation rule scripts providing a more procedural and structured transformation script.

Table 8.3 summarises the above mentioned comparison between CONVErT and ALTOVA MapForce.

8.3 Quantitative evaluation

This section provides a quantitative evaluation of the Suggester system using selection of recommender system evaluation metrics, namely precision, recall and F-measure. These metrics are calculated based on correspondence recommendations generated for transformation examples of bar chart to pie chart, Minard map to pie chart, UML class diagram to Java code, CAD visualisation to alternative tree visualisation, and two transformation examples of academic citations.

This section also provides an evaluation of the transformation code generated from our approach against the transformation codes of an XSLT expert and that generated by ALTOVA MapForce. A set of transformation code quality attributes and metrics are used from the literature to examine code quality for two example transformations of UML class diagram to Java code and bar chart to pie chart.

Table 8.3 Summary comparison of ALTOVA MapForce and CONVErT

Comparison Summary	MapForce [47]	CONVErT [80]
Source and target representation	Default tree based	Visualisations of both models
Transformation specification	Using drag and drop	Using drag and drop, Selecting from recommendations
Correspondence specification	Using drag and drop	Using drag and drop, Selecting from recommendations
Use of functions	Drag and drop functions and link corresponding elements	Drag and drop functions and link corresponding elements
Transformation rule representation	Correspondence lines connecting source and target elements	Visual representation using source and target notations
User guidance	Limited, automatically mapping same name and typed elements	Recommendations provide guidance for transformation rules and their internal correspondences
Editing transformation	Possible	Limited, rule has to be defined again
Model checking results	Possible	Possible
Transformation script design	As a whole for transforming source to target	Procedural, generated using collection of individual transformation rules
Possible transformation scripts	XSLT, XQuery, C#, C++, Java	XSLT, other languages need additional code generator component

8.3.1 Suggerer evaluation

This section evaluates the recommender system of CONVErT's Suggerer mechanism to partially address research question RQ3.2, on whether acceptable recommendations are produced. It should be noted here that the acceptability of recommendations are not purely based on their correctness. However, it is customary in research community to

evaluate correctness of recommendation systems using series of metrics and against a benchmark [162], [167], [180].

To perform this evaluation, the recommender system of Suggester was separately applied on multiple examples. The Suggester was configured to use all available recommenders and a set of evaluation metrics was considered. Selected metrics include precision, recall, and f-measure. These metrics are calculated using categorisation of recommendations into four distinctive groups. This categorisation is based on relevant or irrelevant correspondences being recommended or not recommended. This categorisation is depicted by Table 8.4.

Table 8.4 Categories of all possible recommendations.

	Recommended	Not recommended
Relevant	True-Positive (TP)	False-Negative (FN)
Irrelevant	False-Positive (FP)	True-Negative (TN)

Being relevant in this context means that there is a relation between source and target item element and therefore, a correspondence exists. An irrelevant recommendation on the other hand, is a false correspondence relation. A brief description of each metric is provided bellow.

Precision is defined as the proportion of true positive recommendations against all recommended correspondences and is calculated using the following formula:

$$Precision = \frac{TP}{TP + FP}$$

Given this formula, precision is seen as the measure of purity in retrieval performance or the measure of effectiveness of the recommender in excluding non-relevant items [181]. *Recall* on the other hand is defined as the proportion of the true positive recommendations against all relevant (assumed correct) recommendations and is calculated using following formula:

$$Recall = \frac{TP}{TP + FN}$$

Recall is therefore the number of retrieved relevant items as a proportion of all relevant items. In general, higher precision and recall are desirable [181].

It is often the case that precision and recall are inversely related, i.e. improving precision will result in worse recall, and improving recall will also result in worse precision. As a result, *F-Measure* is introduced to capture harmonic mean of the both metrics [162], [182]. F-Measure is calculated using following formula:

$$F - Measure = 2 * \frac{Precision * Recall}{Precision + Recall}$$

F-Measure tries to capture the behaviour of precision and recall metrics in a single value [178]. Accordingly, higher F-Measure indicates higher quality of recommendations. In the following, we describe how these metrics are used to evaluate correctness of the recommender used in our Suggester system.

To evaluate Suggester mechanism, a set of transformation examples was selected. These examples include the mapping examples of chapter 5 (bar chart to pie chart, Minard map to pie chart, UML class diagram to Java code, and CAD visualisation to alternative tree visualisation) and a more complex mapping example of academic citations. For the academic citation example, 420 citations were used in two different formats (EndNote XML, and DocBook XML) and the suggester was applied on both directions.

For evaluation purposes, a user-defined benchmark was produced to evaluate the recommended correspondences against it. This benchmark included a list of correct correspondences between source and target model abstractions. For example, if transformation task is to transform a class diagram to Java XML, the benchmark list includes class diagram's class element to Java class's class element, class diagram's class name to Java class's name and so on. All correct correspondences between source and target elements are considered for the benchmark and the resulting correspondences of recommender system in Suggester are compared against this benchmark.

Since the motivation of using the Suggester system was to guide users in exploration of possible correspondences, certain considerations have been made to allow more possibilities to be included in the benchmark. For example, in calculation of relevant recommendations in our benchmarks, correct correspondences have been considered regardless of being direct or indirect. For instance, consider the example where the type of a CAD design's room defines colour of a tree node element in a tree visualisation. Although this correspondence is not a direct correspondence and needs to be performed using a condition, recommending room type and node colour as a correspondence does represent a correct recommendation. Also if multiple elements of source or target are possible to be matched, all possibilities are considered in the benchmark. For example, a bar chart's name, YAxis label, or XAxis label could correctly define the name of a pie chart depending on user's interest. These considerations have impacted the selected metrics. For example the former adversely affects precision since Name recommender does not consider type and colour to be correct correspondences. The latter however, allows more possibilities in the benchmark and therefore affects recall accordingly.

Table 8.5 shows the results calculated for the suggester mechanism using the examples. It also provides the time consumed by Suggester to produce the suggestion list (in milliseconds) and provides the size of source and target examples being used (in kilobytes).

Table 8.5 Resulted values calculated from the evaluation metrics.

Example	Metric				Example size (KB)
	Precision	Recall	F-Measure	Calculation Time (m. sec.)	
Bar chart – Pie chart	1	0.77	0.87	< 1	1 - 1
Minard – Pie chart	0.85	0.66	0.75	1.01	9 - 1
CAD design – Tree	0.77	0.77	0.77	< 1	2 - 2
Class diagram – Java	1	0.84	0.91	< 1	3 - 3
DocBook – EndNote	0.46	0.5	0.48	37.52	313 - 661
EndNote – DocBook	0.38	0.55	0.45	43.83	661 - 313

Based on Table 8.5, apart from citation mapping example, our Suggester has performed acceptable (more than 0.85) for the majority of first four examples with regards to precision (1, 0.85, 0.77 and 1 respectively). This indicates that the majority of true recommendations have been produced. With regards to recall, the calculated results for the four examples are less acceptable (0.77, 0.66, 0.77, 0.84). However, the calculated F-Measure indicates good overall acceptance range (0.87, 0.75, 0.77 and 0.91).

The models used for this evaluation each have example-specific naming convention, typing, structure and sizes. As a consequence, the returned recommendations of the system have resulted in different evaluation results for the examples. Given that the Suggester was used for a prolonged period on similar examples, its learning mechanism would improve its recommendation accuracy. However, as mentioned before, this was not the intention of this evaluation and the first encounter of the Suggester with the testing examples was considered.

The resulted precision, recall and F-Measure for the two larger examples are significantly less than other examples (0.46, 0.38 and 0.5, 0.55 and 0.48, 0.45 accordingly). As a result, we conclude that our research question 3.2 on generating acceptable recommendations is satisfied partially. However, continues use of recommender on similar examples improved the recommender performance, but as stated above, the purpose of this evaluation was to check recommender's performance on its first encounter with the examples. We have put evaluation of possible performance tuning approaches and experimenting with large examples as part of our future work.

The relatively poor performance of the Suggester for citation mapping problem is due to extensively different structural and name conventions used for both citation formats. This structural difference is also reflected in the size of two formats given the same amount of citations (313KB vs. 661KB). For example, the “authors” field of the DocBook format has separate fields for first name and last name of the authors. The EndNote format however, uses a single field for author names. Our specified benchmark uses author name to first name, author name to last name, and author name to author field as possible accepted correspondences. Samples of these examples can be seen in Appendix 6. Given that the Suggester system provides only one suggestion per

pair by default, lots of such multi possibilities are not considered. This has resulted in lower number of true-positive choices and higher number of false-negatives and consecutively lower precision and recall.

In terms of calculation time, for smaller examples (like bar chart to pie chart example) full recommendation list is prepared almost real time. For larger examples (like the citation examples) the calculation time increases as example size is becoming larger. For example in citation mapping example, it took Suggester system 37.52 milliseconds to produce recommendation list for DocBook format to EndNote and 43.83 milliseconds for the reverse. These times were measured based on using a PC with dual core CPU and 3 gigabytes of RAM.

The combination of recommenders being used in our Suggester mechanism is very much dependant on the accuracy of name similarity recommender since it has been used in neighbourhood similarity and IsoRank as the seeding similarity measure. It has a direct effect on accuracy of recommendations when source and target examples do not represent similar element names. An example is the citation mapping example above.

Also, the combination of recommenders can be altered according to the examples being used. For example if the examples do not represent the same underlying data, the value recommender can be disabled to prevent it from producing a large number of outlier recommendations. It is possible to alter value similarity recommender to look for non-trivial values only like Boolean, dates, IDs, etc. Another alternative is to use filtering or chain of recommenders, i.e. use the recommendations returned by a set of recommenders (e.g. name and structure) and apply other recommenders (e.g. value similarity recommender) on the results returned by these recommenders.

If prior knowledge or a benchmark for the examples is available, users can alter the seeding similarity or the combination of recommendations to match their needs. It is also possible to train Suggester mechanism for certain examples by using optimisation techniques. For example, it is easy to calculate which combination of recommenders provides better results for certain examples if their respective benchmark is available.

It is worth mentioning here that the Suggester system updates its recommendations according to the tasks that user is performing to produce a more interactive mapping

support environment. For example when user is mapping a bar to a pie piece of the pie chart, it updates the recommendations list to reflect recommendations according to the internal elements of bar and the pie piece. This evaluation was based on the recommendations list produced for the whole visualisation examples and smaller samples are not considered. Also if the suggester is being used continuously by users for similar examples, it updates its weights to produce more acceptable recommendations. For this evaluation, we have tried to reflect the case where the Suggester is being used for the first time for examples and have reset the weights to defaults for each calculation (default weight is 1). Indeed given more time and use this would increase the accuracy of recommended correspondences.

8.3.2 Transformation code quality

To have an assessment of quality of the automatically generated transformation code in CONVErT, this section provides a quantitative comparison of CONVErT's transformation script against transformation scripts generated by an XSLT expert and ALTOVA MapForce. From the six transformation examples of previous section, two examples of transforming UML class to Java and bar chart to pie chart were selected for this comparison. This selection is based on similarity of the source and target model examples in terms of their structural complexity to other examples and availability of their visualisations in CONVErT.

We asked an XSLT expert with more than three years experience using XSLT transformations to write two XSLT scripts for transforming example UML class diagrams to Java and example bar charts to pie chart. These transformation scripts were then compared with the automated XSLT transformation code generated by CONVErT and ALTOVA MapForce.

To evaluate the generated transformation codes, we have adopted a set of quality attributes and metrics proposed by van Amstel et al. [6]. In total, they introduced eight quality attributes. A brief description of these quality attributes and metrics is provided below:

- Understandability: The amount of effort required to understand a model transformation.
- Modifiability: Whether a model transformation can be adopted to possess different or additional functionality.
- Reusability: Whether (a part of) model transformation can be reused as-is by other model transformations.
- Reuse: Whether a model transformation reuses parts of other model transformations.
- Modularity: Is a model transformation systematically structured?
- Completeness: Is a model transformation fully developed and does it result in a complete target?
- Consistency: Does model transformation include conflicting information?
- Conciseness: Whether a model transformation does not include superfluous information like code clones.

The metrics provided by van Amstel et al. for evaluating these quality attributes are grouped into four categories of Size, Function, Module and Consistency [6]. However, those metrics were designed primarily for ASF+SDF transformation language and had functional languages in mind. Although XSLT can be considered as a functional transformation language, certain metrics proposed by van Amstel et al. do not apply to our context here and we had to make adjustments for the metrics. For example, transformations in ASF+SDF are defined in form of functions and modules. The functions refer to algebraic function in the context of ASF+SDF and not transformations functions as defined in this thesis; whereas the transformations in our context are defined in form of correspondences and rules. As a result, function metrics defined in van Amstel et al. are not applicable to our context. Also, due to nature of our tested transformation examples, certain metrics were not applicable. For example, size of domain specific and domain independent parts, or number of code clones. As a consequence, the metrics evaluating consistency and conciseness quality attributes were not included in our comparison.

Similarly, the metrics we have included in our comparison do not have effects on completeness of model transformations. This is due to the fact that ASF+SDF guarantees syntax-safety, i.e. every syntactically correct source model is transformed into a syntactically correct target model. Van Amstel et al. had defined metrics to ensure this quality attribute is satisfied. In our comparison, to check completeness of transformation is satisfied, we check the resulted target of each model transformation

script individually and imported them to CONVErT’s renderer to see how they are being rendered. Table 8.6 shows the list of metrics and how they affect the quality attributes.

Table 8.6 Metrics and quality attributes to evaluate model transformations adopted from [6]. + indicates direct affect while – indicates adverse effects.

#	Metric	Quality Attribute				
		Understandability	Modifiability	Reusability	Reuse	Modularity
1	Lines of code	-	-			
2	Number of correspondences	-	-			
3	Number of transformation rules					+
4	Number of equations	-	-			
5	Rule fan-in			+		
6	Rule fan-out				+	
7	Rule information flow complexity	-				

Metrics of Table 8.6 include lines of code of the transformation, number of correspondences per transformation script, number of individual transformation rules, and number of equations used in the transformation script. Transformation rules can be used by other rules. Accordingly, fan-in defines the number of times a transformation rule is used in other rules and fan-out defines the number of times a transformation rule uses other rules. Rule information flow complexity is measure of complexity which is calculated by squared product of fan-in and fan-out of a transformation rule:

$$\text{Information flow complexity} = (\text{fan.in} \times \text{fan.out})^2$$

These metrics are calculated individually for each transformation script and the results are provided in table 8.7.

Table 8.7 Comparison of transformation codes generated by CONVErT, ALTOVA MapForce and XSLT expert.

#	Metric	Experiments					
		UML to Java			Bar chart to Pie		
		CONVErT	MapForce	Expert	CONVErT	MapForce	Expert
1	Lines of code	98	93	86	29	22	27
2	Number of correspondences	25	25	20	7	7	6
3	Number of transformation rules	7	1	1	3	1	1
4	Number of equations or conditions	1	1	1	0	0	0
5	Rule fan-in	7	0	0	3	0	0
6	Rule fan-out	6	0	0	2	0	0
7	Rule information flow complexity	1764	0	0	36	0	0
8	Execution time (milliseconds)	0.7	0.8	0.6	0.2	0.3	0.2

The results of Table 8.7 demonstrate higher number of lines of code for the generated transformation script of CONVErT. This is due to the fact that CONVErT's code generates a procedural transformation script and uses the individual transformation rules which have significant effects on number of lines of code. Although this decreases the understandability of the code, it increased the modularity and hence the possibility of reusing certain parts of the code.

The transformation script used by our expert used fewer correspondences. This is due use of XPath addressing strings. For example when multiple bars are available in a “bars” element of a bar chart, two separate correspondences should be specified for “bars” to “Pieces” and bar node to pie piece. Figure 8.6 demonstrates these correspondences for ALTOVA MapForce and Figure 8.7 show similar correspondences in CONVErT. However, the expert used XPath constructs similar to "Bars/BarNode" and as a result used fewer correspondences. It is noteworthy here that when using low level coding in model transformation languages, transformation designers can use powerful constraint languages like OCL to define more complex transformations. Our approach in CONVErT uses local notation to notation transformation specifications and relies on facilities provided by the transformation

language and the user defined functions in XSLT. If for alternative transformation languages (e.g. ATL) are used in toolset, these constraint specification languages (e.g. OCL) can be provided in functions.

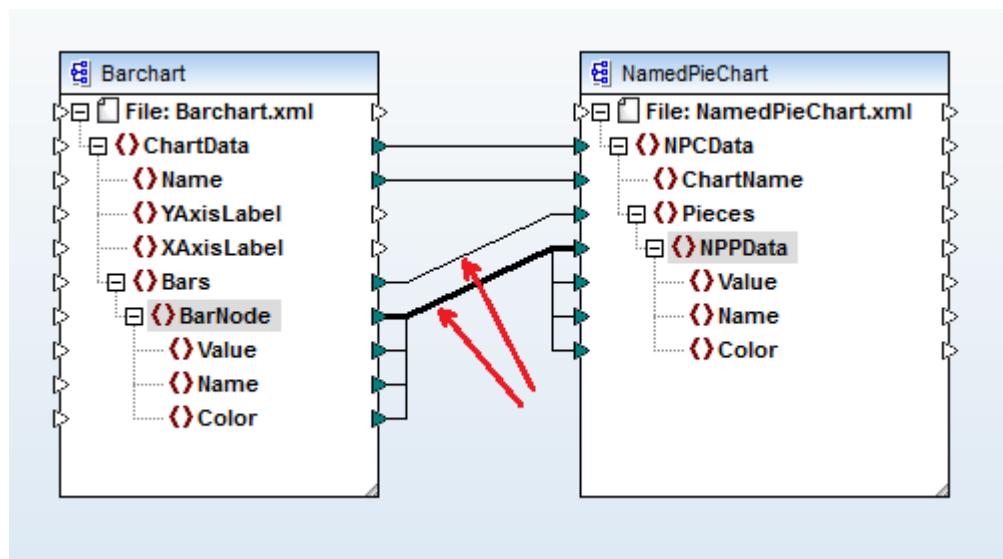


Figure 8.6 Example correspondences for bar chart in ALTOVA MapForce. Arrows mark correspondences.

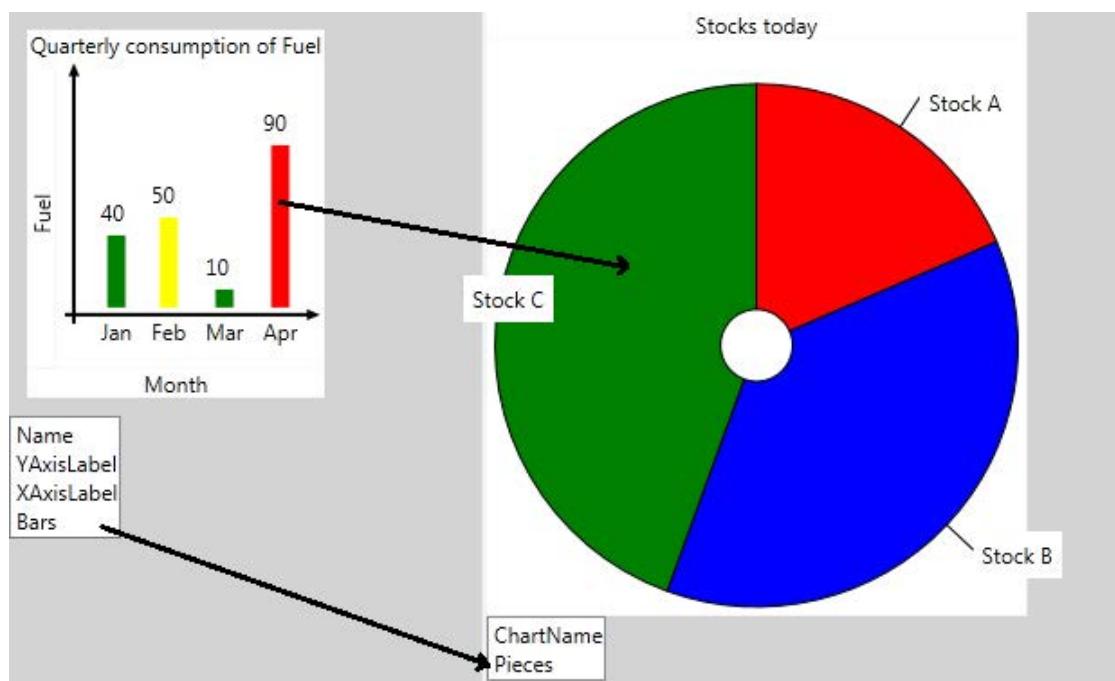


Figure 8.7 Example correspondences for bar chart in CONVERT. Arrows demonstrate drag and drop directions to specify correspondences.

Given that CONVErT's code was the only code that included individual transformation rules, fan-in, fan-out and rule information flow complexity was only calculated for CONVErT's scripts. These numbers were calculated based on the number of rule calls and arrangement of the rules available in the script. For example, class rule in CONVErT's transformation code calls three other rules (for creating associations, attributes, and operations) while transformation rule for creating attributes does not call any other rules.

It should be noted here that the generated transformation code of CONVErT is not meant to be edited. Although users save the transformation in a separate XSLT file and can edit them, our intention was otherwise in separating users from interacting with the code. Also, given the power of coding transformation by an expert, there are certain cases where CONVErT does not provide similar capabilities to its users. One such limited capability is in the way correspondences are specified. Correspondences in CONVErT are specified between a pair of visual notations. As a result, modularity of the generated transformation code depends on the pair of notations being mapped to each other. For example, when mapping a bar chart to pie chart of Figure 8.7, once user drags bar chart area to pie chart area, he cannot specify correspondences between their embedded notations i.e. elements of bars and pie pieces. Correspondences between bars and pie pieces need to be specified when a bar is dragged to a pie piece. Similarly, correspondences involving more than one notation as source or target cannot be easily specified with current version of CONVErT's Mapper. These more complex transformation specifications are part of our future work.

Table 8.7 also provides execution time of each transformation script using a PC with dual core CPU and three gigabytes of RAM. To measure execution time, all transformation scripts were used in the XSLT parser and engine of Microsoft Visual Studio and applied on two different examples of source model for each transformation script. Class diagram XML examples were 3kb and 4kb while bar chart examples were 1kb and 2kb in size. We repeated the transformation execution five times on each example and recorded execution time. Repeating the experiments did not alter average times. The measured time does not demonstrate significant differences between transformation scripts.

All transformation scripts of this comparison produced renderable target models. However, the generated target of CONVErT was more complete with regards to the missing values in UML class diagram to Java code visualisation transformation as Figure 8.8 demonstrates. This is because the transformation code generator of convert uses the reverse engineered abstraction of the notations in each transformation rule and automatically fills in the missing values by default values. For example, when mapping UML associations to Java properties, since associations do not possess access value, CONVErT’s code generator assigns “public” as their default value from the abstraction of Java property notation. ALTOVA MapForce uses the reverse engineered abstraction of the full source and target and did not produce any default value in case any value is missing. To have similar capability in MapForce, such default values should be indicated in the schema and provided to the tool.

```

/* This is automatically generated code */
package XYZ_Airline ;
public class Flight
{
    public int FlightNo ;
    public date DepartTime ;
    public Minutes Duration ;
    public FrequentFlyer [100] Passengers ;
    public Plane AssignedPlane ;

    public void delayFlight ( minutes delay )
    {
    }
    public time getArrivalTime ( )
}

```

(A)


```

/* This is automatically generated code */
package XYZ_Airline ;
public class Flight
{
    public int FlightNo ;
    public date DepartTime ;
    public Minutes Duration ;
    public FrequentFlyer [100] Passengers ;
    public Plane AssignedPlane ;

    public void delayFlight ( minutes delay )
    {
    }
    public time getArrivalTime ( )
}

```

(B)

Figure 8.8 Rendering the generated target model as a result of running transformation scripts of (A) CONVErT’s and (B) ALTOVA MapForce.

Although we did not ask our expert to generate transformation code for transforming the other examples of previous section (CAD to tree visualisation, and Minard’s map to Pie chart), our experiments with ALTOVA MapForce and CONVErT on those examples showed similar results to the two examples being discussed here and in Table 8.7. As a result, we conclude that the evaluation results of table 8.7 are generalizable to other examples as well.

8.4 User evaluation

A user evaluation of our CONVErT tool and its approach focuses on the user's perception of the approach and associated toolset for generating visualisations and using them for a transformation task. This user study has been approved by Swinburne University Human Research Ethics Committee (SUHREC Project 2013/010). A copy of ethics clearance letter is provided in Appendix 2. The following subsections provide details of this user evaluation and experiments.

8.4.1 Experimental setup

Our experimental setup comprised a laptop with an attached mouse. Prior to starting the experiment, participants were asked to sign a consent form. They were then introduced to the toolset (CONVErT) through a ten-minute screencast. This screencast described the purpose of the toolset, its user interface, visualisation, and transformation generation procedures. The screencast also described basic functionalities of the toolset like where and how to drag and drop, how to use functions, how to compose visual elements and how to view visualisations. Using a screen cast would limit bias compare to presentation and ensure repeatability for multiple participants.

Participants were asked to perform a set of model visualisation and mapping tasks following think-aloud approach. They could ask questions and an instructor was available during the experiment. They were reminded that there was no time limit for performing tasks and that they could leave at any point during the experiment. Given these reminders, all our participants were able to finish assigned tasks.

Two sets of tasks were defined for participants to carry out, each consisting of a visualisation specification and a transformation specification. The first task used a visualisation and transformation from a business domain and the second task used software engineering domain examples. The rationale behind having two sets was to test our approach for different application domains. We hoped that it would help us to investigate if interactive concrete visualisations better support using users' domain knowledge in general. It was designed to investigate if concrete visualisations have similar effects on users of different domains (software engineering and business

analysis in this case). Using this strategy, we are in a stronger position to generalize that our concrete interactive visualisations can help wider domains of model transformation specification and therefore we can make assertions about our answers to research questions RQ1 and RQ2.

To record participant's interactions, screencasts were captured during the process and each participant's voice was recorded. Upon completion of each task, a matching questionnaire was given to each participant. The survey questionnaire was designed in four sections (the questionnaire is provided in Appendix 5). Section one was targeted to visualisation approach and included questions evaluating usefulness, cognitive dimensions, ease of use of the tool, ease of learning the tool and finally questions to capture user satisfaction. Section two was targeted at transformation generation using concrete visualisations and included questions for evaluating usefulness of the approach, cognitive dimensions, ease of use of the tool, ease of learning, and satisfaction. Section three of the questionnaire was designed to evaluate the guidance and recommender system integrated to the toolset. It included questions capturing usefulness, presentation and user satisfaction. The final section of the questionnaire was dedicated to the participant demographic questions.

The first three sections of the questionnaire featured 5-point Likert scale responses and dedicated spaces to leave optional comments and feedback for each Likert item. The Likert scale parameters range from Strongly disagree, Disagree, Undecided, Agree, and Strongly agree. Additional comments section was also provided at the end of each questionnaire section in case users would like to leave further comments.

The tasks assigned to each group were to create a visualisation with CONVERT and then use it as source and generate a transformation from the model to another model with a provided CONVERT visualisation as target model visualisation. Input models and visualisations were the same for participants of each group.

The first group were given a model representing business sales data and were asked to create a bar chart visualisation of their sales data (task 1). They were then asked to alter the bar chart visualisation for a different input model (task 2) and finally transform that bar chart visualisation to a pie chart visualisation (task 3). The second group were given

a class diagram data (XML) and asked to generate a UML class diagram visualisation (task 1). For Task 2 they were asked to transform that class diagram visualisation to a provided Java code visualisation.

Each participant was handed a hard copy of task descriptions. These task descriptions did not describe instructional steps. Instead, they included the input file names and their locations, and a snapshot of the desired final visualisation and the transformation result. Task descriptions are available in Appendix 3 and Appendix 4. Users had to come up with steps required to get similar results. They were allowed to ask questions from the instructor if they had trouble understanding those steps. The following subsection provides the results gathered from this study.

8.4.2 Experiment results

For this user study, 19 users (including 4 controls for instrument testing) were recruited from staff and students at Swinburne University of Technology. No age or restrictions were applied for recruiting participants. To capture user demographics, participants were required to complete a demographic questionnaire. The summary of demographic data of participants is provided in Table 8.8.

Ten participants (8 male, 2 female) chose to use the business analysis domain tasks and five participants (3 male, 2 female) used the tasks from software engineering domain. Participants of both groups had basic understanding of domain models used for experiments but with very little or no experience in model transformation.

The demographics information of Table 8.8 demonstrates that only 17 percent of users had experience with at least one modelling and transformation approach and 47 percent were aware of them. 13 percent of users had experience with at least one visualisation approach, and 60 percent were somewhat familiar with visualisations. It can be concluded that about two third of the users were familiar with visualisations to some degree and around two third were also familiar with modelling and transformations.

Table 8.8 Demographic questions of the user study questionnaire and participant's responses.

	Question	Options	Participants (%)
D.1	Gender?	Male Female Prefer not to say	67 33 0
D.2	Age rang?	23-30 31-40 41-50 51-60 61+	60 40 0 0 0
D.3	How familiar are you with model transformation and modelling in general?	Very familiar Somewhat familiar I had heard about it Not familiar at all	17 47 33 13
D.4	How familiar are you with data visualisation?	Very familiar Somewhat familiar I had heard about it Not familiar at all	13 60 13 13
D.5	What best describes your area?	Software engineering Computer Science/IT Economics Management Other	47 40 0 0 13

To analyse participants' responses, Likert scale scores were collected from questionnaires. We have assigned scores of 1 (for perfect negative) to 5 (perfect positive) to each Likert item response. Based on the scores, Median, Mode and frequency of responses for each Likert item is calculated for comparison. Results are separately presented for visualisation, transformation, and evaluation of the Suggester in the following subsections.

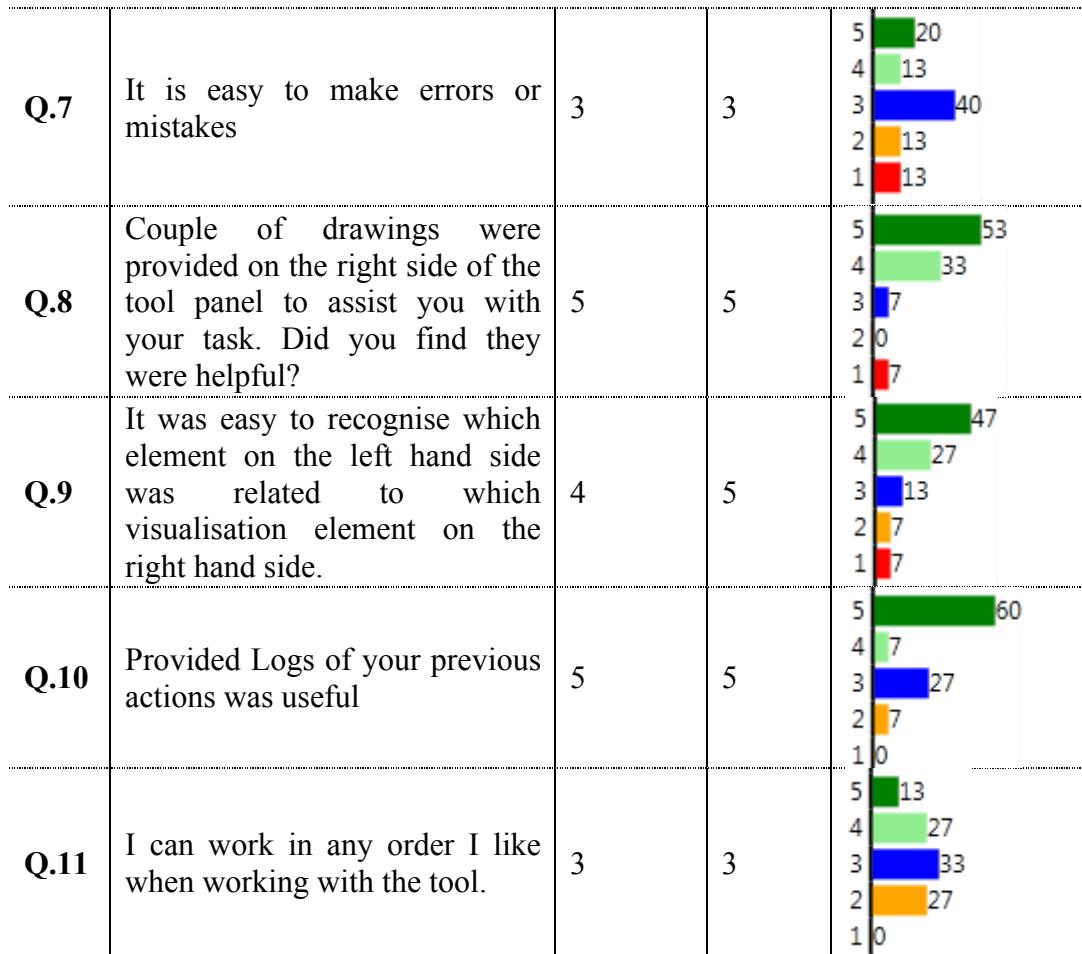
8.4.2.1 Visualisation

Tasks 1 and 2 for first group and task 1 for the second group were designed to check the efficiency of our approach and the tool for visualising input model data. After the tasks were completed by participants, first part of the questionnaire was handed to each

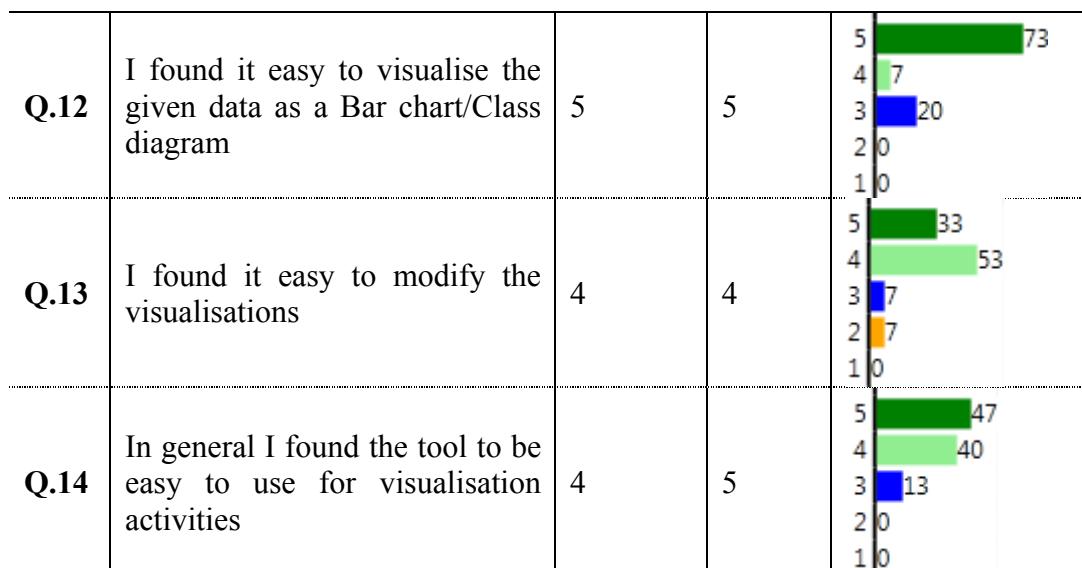
participant. The questions for the evaluation of these tasks and the results collected from it are provided in Table 8.9. Note that the frequency charts in the table demonstrate rounded values of the frequencies and are generated using CONVERT.

Table 8.9 User study questions for visualisation evaluation.

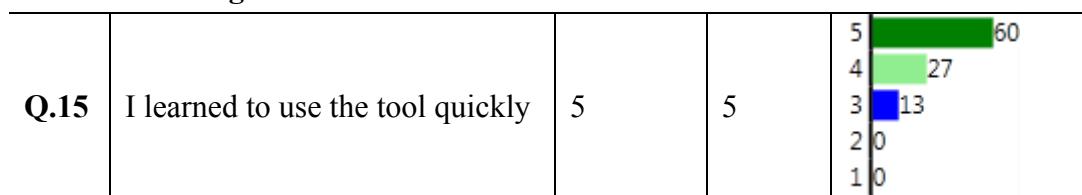
	Question	Participant Responses												
		Median	Mode	Frequency (%)										
Usefulness														
Q.1	It is useful to have a drag and drop approach for visualisation.	5	5	<table> <tr><td>5</td><td>87</td></tr> <tr><td>4</td><td>13</td></tr> <tr><td>3</td><td>30</td></tr> <tr><td>2</td><td>20</td></tr> <tr><td>1</td><td>10</td></tr> </table>	5	87	4	13	3	30	2	20	1	10
5	87													
4	13													
3	30													
2	20													
1	10													
Q.2	Visualisations help me better understand complex data.	5	5	<table> <tr><td>5</td><td>60</td></tr> <tr><td>4</td><td>27</td></tr> <tr><td>3</td><td>13</td></tr> <tr><td>2</td><td>20</td></tr> <tr><td>1</td><td>10</td></tr> </table>	5	60	4	27	3	13	2	20	1	10
5	60													
4	27													
3	13													
2	20													
1	10													
Q.3	It is useful to be able to visualise data tailored to users	5	5	<table> <tr><td>5</td><td>80</td></tr> <tr><td>4</td><td>13</td></tr> <tr><td>3</td><td>7</td></tr> <tr><td>2</td><td>20</td></tr> <tr><td>1</td><td>10</td></tr> </table>	5	80	4	13	3	7	2	20	1	10
5	80													
4	13													
3	7													
2	20													
1	10													
Cognitive dimensions														
Q.4	It is easy to see various parts of the tool such as drawings, functions, etc.	4	4	<table> <tr><td>5</td><td>20</td></tr> <tr><td>4</td><td>67</td></tr> <tr><td>3</td><td>17</td></tr> <tr><td>2</td><td>20</td></tr> <tr><td>1</td><td>7</td></tr> </table>	5	20	4	67	3	17	2	20	1	7
5	20													
4	67													
3	17													
2	20													
1	7													
Q.5	It is easy to make changes to visualisations.	4	5	<table> <tr><td>5</td><td>47</td></tr> <tr><td>4</td><td>40</td></tr> <tr><td>3</td><td>17</td></tr> <tr><td>2</td><td>20</td></tr> <tr><td>1</td><td>7</td></tr> </table>	5	47	4	40	3	17	2	20	1	7
5	47													
4	40													
3	17													
2	20													
1	7													
Q.6	Some things do require a lot of thought.	4	4	<table> <tr><td>5</td><td>7</td></tr> <tr><td>4</td><td>47</td></tr> <tr><td>3</td><td>20</td></tr> <tr><td>2</td><td>20</td></tr> <tr><td>1</td><td>7</td></tr> </table>	5	7	4	47	3	20	2	20	1	7
5	7													
4	47													
3	20													
2	20													
1	7													



Ease of use



Ease of learning



Q.16	I would like to have received further instruction to be able to understand the procedure and perform the task.	3	4	5	13
				4	27
Q.17	I had to redo some parts to be able to understand the procedure	3	5	3	27
				2	20

Satisfaction

Q.18	I easily remember how to use the tool	4	4	5	33
				4	60
Q.19	It is likely that I use the tool for visualisation in my future projects	4	5	3	20
				2	0
Q.20	I had fun using the tool	5	5	1	0
				0	0
Q.21	I would recommend it to a friend	5	5	5	53
				4	27

As can be seen in the table, the results collected for the usefulness and satisfaction of the approach (Q.1 to Q.3 and Q.18 to Q.21) conveys that participants agree on the fact that the drag and drop approach was useful for visualisation tasks. These parts of the questionnaire were designed to capture user experiences on effectiveness of our approach and hence our first main research question on effectiveness of visual by example approach for generating visualisations. Specifically, participants' answer to question Q.1 demonstrates that all our participants have agreed on usefulness of our drag and drop approach for visualisation (87% strongly agree and 13% agree). Similarly their responses to questions Q.2 and Q.3 show general acceptance of the usefulness of

the visualisations. However, the phrasing of these questions might convey that they are generic questions and not targeted to the approach and toolset. We did not notice this in our pilot studies and therefore some participants may have considered questions Q.1 to Q.3 as generic.

Questions Q.4 to Q.17 were primarily focused on tooling aspect of the approach. For example, various cognitive dimension characteristics are being examined like visibility (Q4), viscosity (Q5), hard operation (Q7), and premature commitment (Q11). The results in Table 8.9 indicate that specific parts of the tool need to be improved or redesigned. For example, the version of the tool being used in evaluation (revision 320) did not allow repositioning of notations. If users wanted to reposition the notations, it would copy it to the new location and as a result they had to clear the canvas and redo the drag and drop. This has been reflected in user responses to questions Q.7 and Q.17. Also, some users could not differentiate model elements and placeholders as they were represented similarly for each visual notation by the framework. This resulted in confusion, and a user had to ask the instructor after a mistake was made in notation composition. This confusion by some participants is reflected in responses to questions Q.6 and Q.16. Note that while there was no rational behind using negative questions, the responses to these questions have been reversed to keep the scales consistent and avoid confusion. For example, for question Q.6, 7% of the participants have strongly agreed and 20% have agreed that some things did require lots of though; while 47% have disagreed and 7% have strongly disagreed to that statement.

Given the responses to questions Q.16 and Q.17 were less than what we expected, the majority of participants (60% strongly agree and 27% agree) agree that learning the tool was easy. This has been reflected in their response to question Q.15. We believe that if the implementation problems with the tested version of the tool are fixed in future releases, the difficulties raised from them will be solved and would provide even higher acceptance.

Although the tool had some imperfections, all users agree on ease of use of the tool for generating visualisations. This is reflected in their responses to questions Q.12 and Q.14. The response to question Q.13 in Table 8.9 is calculated using the total responses. It is noteworthy to consider that the second group did not perform the visualisation

alteration task. We have asked them to respond to this question based on their understanding of the tool. Responses to this question for each group are depicted in Figure 8.9. As can be seen in the figure, all participants of the first group agree that the modification of visualisation was easy. However, the participants of the second group have mixed feelings about it. Also, users were provided with predefined notations. Given that users were required to annotate views to generate notations, we would anticipate having slightly different results, since users would have to have basic understanding of XAML representations to understand the elements of visual Views.

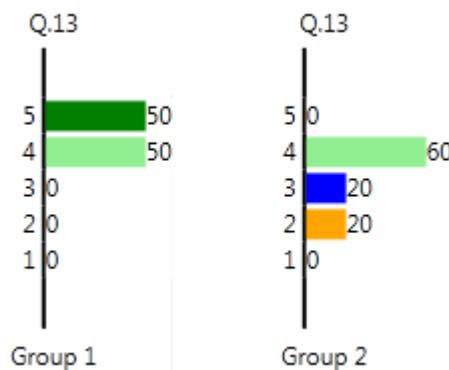


Figure 8.9 Group based responses to question Q.13.

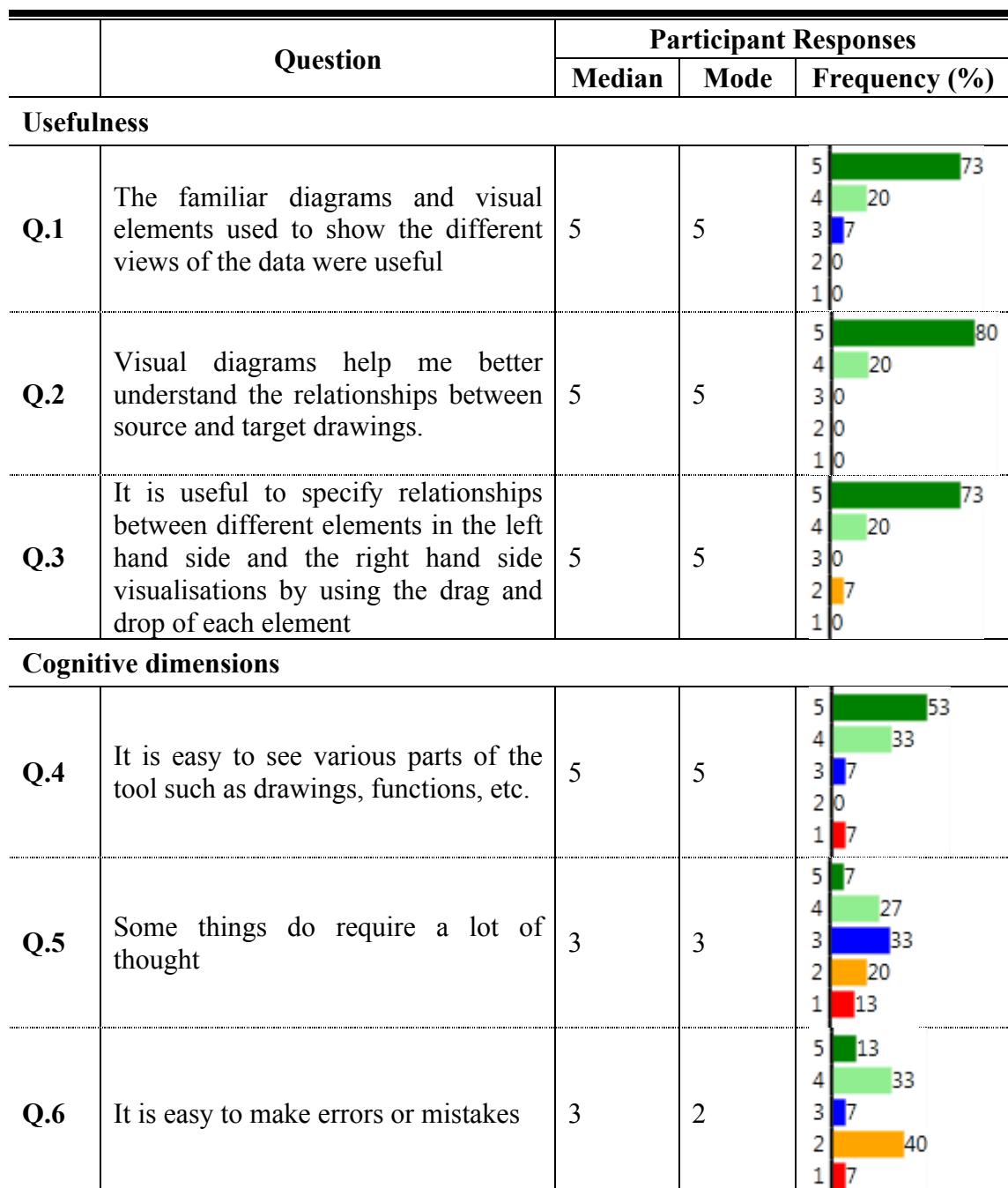
8.4.2.2 Transformation

Final task of our user study included asking users to use the generated visualisation as source model visualisation example and use a provided target model visualisation example to generate a transformation between the two. Users were required to drag and drop elements of both visualisations to create transformation rules and execute them to generate a target visualisation. Alternatively, similar to the visualisation step, they could use the provided recommendations.

Second section of user study questionnaire was dedicated to capturing user experiences with transformation generation using concrete visualised examples. The questions of this user study and analysis of user responses are provided in Table 8.10. Similar to user evaluation of the visualisation step, scores of 1 to 5 were given to responses (from most

negative to most positive) and median, mode and frequency of the scores were calculated accordingly.

Table 8.10 User study questions for transformation evaluation.



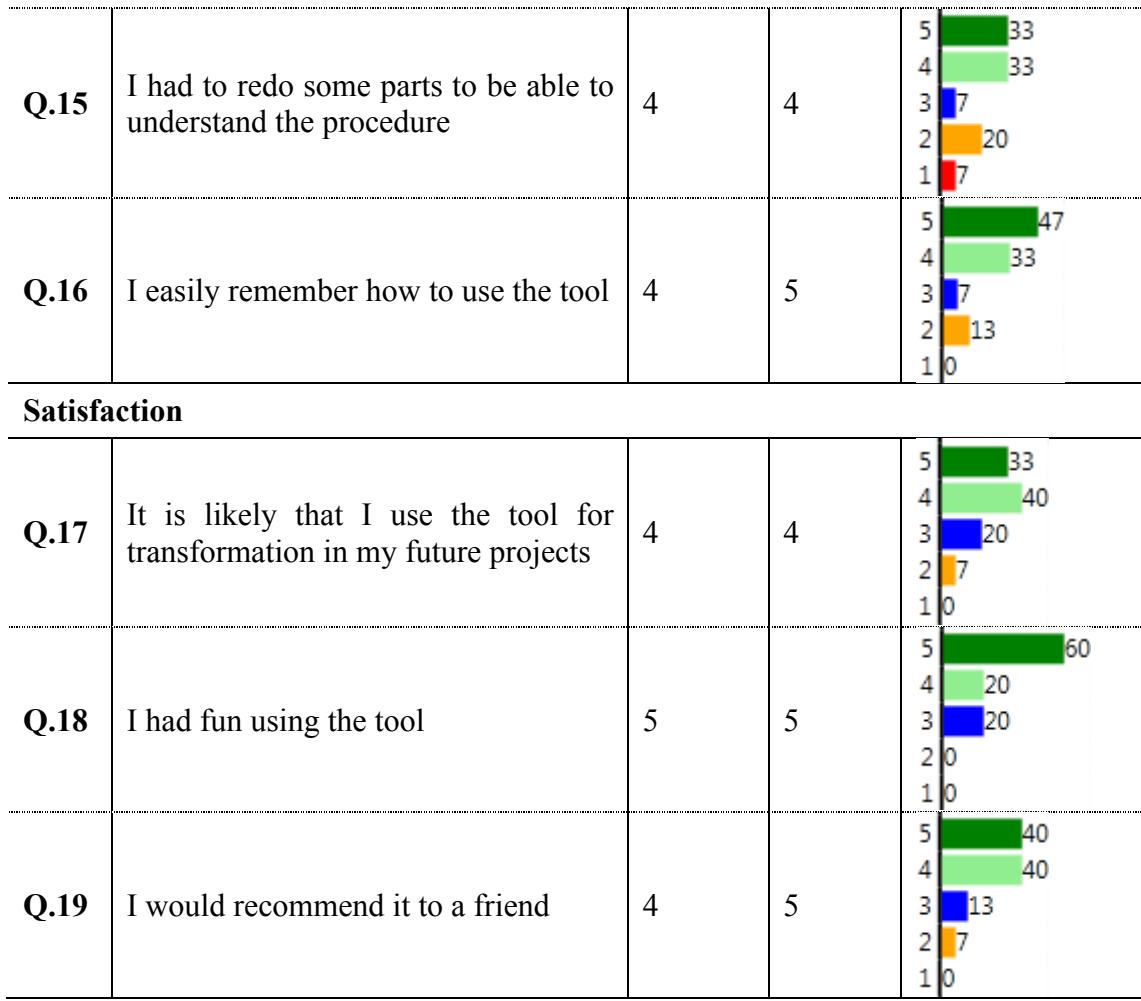
Q.7	It was easy to recognise which visual element on the left hand side was related to which visual element on the right hand side	4	4	5 4 3 2 1	33 47 13 0 7
Q.8	Provided Logs of my previous actions was useful	5	5	5 4 3 2 1	60 13 13 13 0
Q.9	I can work in any order I like when working with the tool	4	4	5 4 3 2 1	27 47 27 0 0

Ease of use

Q.10	I found it easy to specify the relations between left hand side and right hand side visualisations	5	5	5 4 3 2 1	67 20 7 0 7
Q.11	The user interface is very consistent	5	5	5 4 3 2 1	60 27 13 0 0
Q.12	In general I found the tool to be easy for transformation between visualisations	5	5	5 4 3 2 1	53 40 0 0 7

Ease of learning

Q.13	I learned to use the tool quickly	4	5	5 4 3 2 1	40 33 20 0 7
Q.14	I would like to have received further instruction to be able to understand the procedure and perform the task	3	2	5 4 3 2 1	13 27 13 33 13



As can be seen from results of Table 8.10, users positively responded to having visualisations and drag and drop of notations to generate mappings (Q.1 to Q3, and Q.17 to Q.19). These responses demonstrate users' perception of the approach in accordance to our second main research question on usefulness of concrete visual by example approach for generation of mappings. For example, in the responses of Table 8.10, all users agree (80% strongly agree and 20% agree) that the visualisations helped them better understand the relationships between source and target models. Or for example in question Q.3 the majority of participants (93% agree, 7% disagree) agree that the drag and drop method of specifying correspondences between source and target is useful.

In terms of the tooling aspect of the approach, there is a need for further improvements to the tool. For example in response to question Q.5 "Some things do require a lot of

thought” participants have responded with 13% strongly agree, 20% agree, 33% undecided, 27% disagree and 7% strongly disagree. Similarly responses to question Q.6 where 47% of the users have agreed that it is easy to make mistakes, indicate rooms for improvements. Please note that due to negative nature of the statements in question 5 and 6 the responses have been reversed to keep scales consistent. Some users did not use the logs and this has been reflected in their response to question Q.8 “Provided Logs of my previous actions was useful”.

It should be noted here that although recommendations were provided, our user evaluation of the recommender system (provided below) indicates that users did not use the recommendations or did not find the recommended correspondences useful enough. Given that users had used them or the representation of the guidance system was improved, it would have helped users with the tasks and particularly in response to question Q.5.

In terms of ease of use, the responses are fairly consistent and indicate general acceptance of the approach and toolset (see responses to questions Q.10 to Q.12). 87% of the users have agreed that it was “easy to specify the relations between left hand side and right hand side visualisations” which complements responses to question Q.3 on usefulness of drag and drop specification of source and target correspondences. Similarly 93% of the users have agreed that the approach provided by the tool for specifying transformation using visualisations was easy.

In terms of ease of learning, although 73% of the users have agreed on quick learning of the tool, the majority of them were reluctant to drag and drop elements of the visualisations on each other. This is where they asked the instructor for some instructions for performing the task. This has been reflected in their responses to questions Q.14 and Q.15. We believe this is due to the fact that the approach taken for transformation and by the tool was very different to users’ expectations. For example in occasions they were reminded by the instructor that they “can” drag and drop notations on each other and once reminded, were able to perform the given tasks.

Questions Q.17 to Q19 were designed to see users’ satisfaction of the approach and toolset. The responses to these questions suggest that the majority of users perceived the

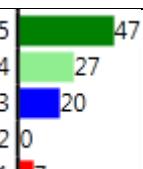
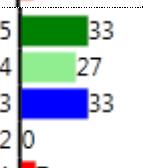
approach positively. For example, 77% of the users agreed that they might use the approach in their future project and some users mentioned that due to the nature of their work, they do not see any need to use transformations in general.

8.4.2.3 Suggester

To have a user evaluation of our Suggester system, third section of questionnaire was dedicated to questions regarding Suggester system and how satisfied users were by the system. The questions of this section and the results collected from this user evaluation are provided in Table 8.11. It should be noted that users were not asked specifically by the tasks to use or follow any of the recommendations. They were however, introduced to the Suggester system in the introduction video and were free to use provided recommendations as they see fit.

Similar to visualisation and transformation steps, the analysis of the collected participant responses are provided in Table 8.11 using median, mode and frequency of the responses.

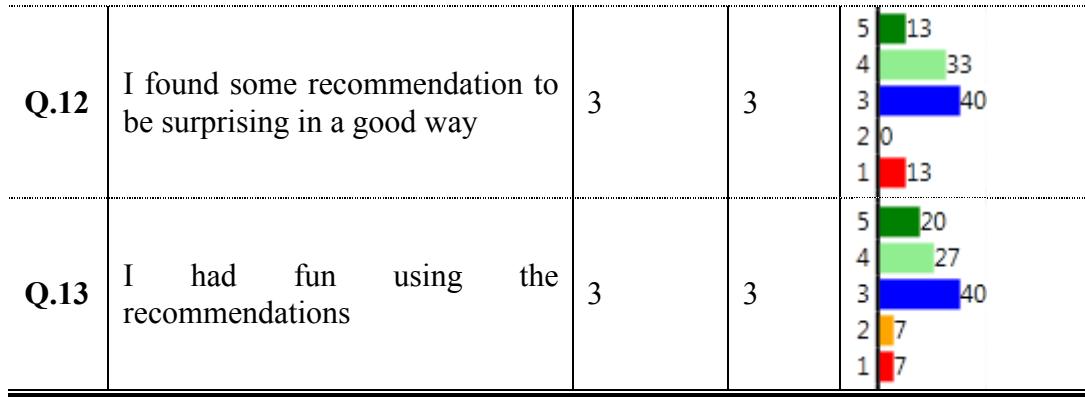
Table 8.11 User study questions of Suggester system and user responses.

	Question	Participant Responses		
		Median	Mode	Frequency (%)
Usefulness				
Q.1	It is useful to have recommendations during the process	4	5	 5: 47 4: 27 3: 20 2: 0 1: 7
Q.2	Recommendations helped me better understand relations between source and target visualisations	4	5	 5: 33 4: 27 3: 33 2: 0 1: 7

Q.3	Recommendations help me discover other possible relations	3	3	 5: 13 4: 27 3: 47 2: 13 1: 0
Q.4	Recommendations seemed to offer a good (correct) solution	4	4	 5: 27 4: 40 3: 27 2: 0 1: 7
Q.5	I was able to trust the recommendations	4	4	 5: 20 4: 47 3: 27 2: 0 1: 7
Q.6	I used recommendations at least once	4	5	 5: 40 4: 20 3: 27 2: 7 1: 7
Q.7	I already knew most of the recommendations	2	3	 5: 7 4: 0 3: 40 2: 27 1: 27

Presentation

Q.8	I was satisfied with the way recommendations were presented	4	5	 5: 47 4: 13 3: 27 2: 7 1: 7
Q.9	When a recommendation said for example “Bar/Name” I was easily able to spot “Name” in source or target visualisations	4	5	 5: 33 4: 27 3: 20 2: 20 1: 0
Q.10	I was able to use recommendations	4	4	 5: 27 4: 33 3: 20 2: 13 1: 7
Q.11	It is likely that I use provided recommendation system in future	4	5	 5: 47 4: 20 3: 27 2: 0 1: 7



The questions and responses of Table 8.11 are grouped into two categories of usefulness and presentation of the recommended correspondences. More specifically, question Q.1 checks the usefulness of having a recommendation system in general. As can be seen from the responses in the table, 74% of participants agree on the usefulness of recommendations. However, when asked whether provided recommendations helped users understand relations between source and target visualisations (Q.2) only 60% of the users have responded agree and strongly agree. This indicates that although majority of users did agree on usefulness of such recommendations, fewer number of them have found it actually helpful. Multiple reasons might have contributed to these results. First, since the users had their first experience with the recommendation system, it is valuable for the recommender to provide higher precision recommendations to spark users' trust [162], [183]. However, user responses to question Q.5 indicate that only 67% of them were able to trust the recommended correspondences.

Second, due to lack of available data in notation's model, the Suggester system is not 100% accurate at visualisation step. Also since few recommendations are provided at that stage, it is easier to notice incorrect recommendations. Given that users' first encounter with the recommendations were at the visualisation step, incorrect recommendations at that stage might have contributed to this loss of trust. This is reflected in users' response to question Q.4 on correctness of the recommendations as well. Only 67% of the users found recommendations to be correct. However, with regards to question Q.6, 60% of participants have agreed that they have used recommendations at least once.

Our observation also indicates that users did not consider the role of recommendations as guidance or means for simplifying the task. This could be due to focus of task handouts on the visualisation and transformation rather than recommendations. For example, only 40% of the participants have acknowledged that recommendations helped them discover new possibilities (Q.3) which indicates that 60% of them were not looking for other possibilities in the recommendations.

The results of Table 8.11 also show interesting findings. Only 7% of the users indicate (in question Q.7) that they did not already know most of the recommendations (note that this is a negative statement). This is in accordance to the calculated precision of one for the tested examples of the tasks (see Table 8.5). Also, the responses to question Q.12 show that less than half of the participants (46%) agree that recommendations were novel and surprising. This indicates the importance of novelty in recommender systems.

In terms of recommendations representation, we have received 60% satisfaction (in response to Q.8) which could be a clue to why users did not use the recommendations and preferred drag and drop to specify correspondences. For example a participant did not realise that by selecting from suggestions, it is possible to specify correspondences and therefore did not use them at all. Similarly only 60% agreed that they could find the correspondences been mentioned by the recommendations in the source and target visualisations (in response to question Q.9). This indicates the inefficiency of representing recommendations as lists. Improved representations could use the augmentation in the visualisations to help users explore possible correspondences.

Overall, the Suggester mechanism achieved lower acceptance than the visualisation and transformation and therefore parts of our third main research question on acceptable recommendations and ways to guide users are not fully satisfied. We believe this is due to the fact that the given examples were simple and the users already knew most of the correspondences, thus did not realise the potential of having a recommendation system for guidance. For example a participant stated that the mapping correspondences were “easy to find and specify” and therefore felt no need to use recommendations. Low level and basic representation of recommendations might have played another role in lower acceptance of the Suggester system. Provided that the visualisations were more complex, it would have evaluated effects of the Suggester system much better.

8.4.3 Threats to validity

We have taken strong considerations to minimise threats to validity and their effects. However, there are set of threats that may have affected the validity of this experiment. In the following, we list these threats according to internal, external, construct and statistical validity classification.

8.4.3.1 Threats to internal validity

Testing: four of our participants mentioned the effect of learning during the experiments. They admitted that since the drag and drop tasks were being repeated for tasks one and two, they could perform second task easier. This might have had effects on better acceptance of the approach for tasks two and three.

Questionnaire: Although the questions of the questionnaire were simplified and an instructor was available during the experiment and when participants were asked to fill out the questionnaire, it is possible that some participants were reluctant to ask questions regarding the items being asked in the questionnaire and therefore responded based on their understanding of the questions.

8.4.3.2 Threats to external validity

Participant affiliation: The users whom participated in the evaluation were mostly chosen among staff and students of Swinburne University of Technology (18 out of 19). This potentially represents a bias and will affect generalisation of our claims.

Participant background: 47 percent of the participants shared common background in Software Engineering and 40 percent shared background in computer science. As a result, their background could have introduced bias in terms of their familiarity with software tools.

8.4.3.3 Threats to construct validity

Task: Due to simplicity of the experiment for one group, performing bar chart to pie chart transformation, five participants did not use the recommendations. These have had

effects on evaluation of the recommendation system. Also, participants that were assigned to second group, did not use the visualisation modification task and their responses to question Q.5 of the visualisation task on making changes to the visualisations was based on their understanding of the ability of the tool and approach.

Experimenter effects: Some instructions made to the participants by the instructor during experiment may have affected the participants' experience. The instructor was asked not to give any instructions unless asked by the participants. However, our observation of the responses and the recordings, demonstrated that the participants who requested more instructions had accordingly mentioned this need in their responses.

8.4.3.4 Threats to statistical validity

Statistical calculations: We have checked and double checked the statistics used for our user study evaluation to confirm their accuracy. As a result and to the best of our knowledge, there is no statistical calculation problem threatening the validity of the results.

Sample size: It is possible that the inferences we have made from our results are due to limited number of participants. The statistics we have used are calculated having non-parametric characteristics of the responses in mind. We do not reject the possibility of changes in the inferences given the number of users increases. The evaluation is therefore an ongoing process and we seek to provide incremental updates to the approach and evaluate accordingly.

8.5 Summary

This chapter provided evaluation of our approach in creating visualisations and transformations. This evaluation was performed using a comparative study of our approach and toolset, a quantitative evaluation of our recommender system and a complimentary user evaluation.

A comparative study of the toolset and approach with other available transformation approaches was provided and a more detailed comparison was discussed with ALTOVA MapForce which is a state of the art mapping tool. This comparison demonstrates how our approach and its tool support (CONVErT) sets apart from state of the art transformation and mapping tools being used today.

The quantitative evaluation was designed to capture correctness of the recommendation system designed in the Suggester. It evaluated the recommendations using a benchmark and couple of examples against quantitative measures of precision, recall, and f-measure. The results demonstrate that for more complex examples the suggester system cannot produce high correctness with regards to precision and recall. It was concluded that given that information on examples being transformed are available, the Suggester can be optimised to perform better recommendations.

Our quantitative evaluation also included a discussion of the quality of the generated code of CONVErT and compared it to the transformation code generated by ALTOVA MapForce and transformation code written by an XSLT expert using set of quantitative metrics. This comparison demonstrated that the automatically generated transformation script of CONVErT is equally effective in comparison to the transformation script written by an expert and generated automatically by ALTOVA MapForce.

A user evaluation was carried out and we described experiment setup, questionnaires and the tasks to be performed by participants. The collected responses of this study were discussed in details. The collective evaluation results demonstrated that the visualisation and transformation approach was perceived positively by users; while the Suggester system did not receive high acceptance. It was concluded that the correctness of recommendations and our approach for their representation might have played key roles in lower acceptance of the Suggester.

Chapter 9

Conclusion and future work

9.1 Conclusion

This thesis introduced an approach and method for performing model transformation on concrete visualisations of models. This approach helps to better incorporate user's domain knowledge by providing familiar example concrete visualisations for transformation generation. Users specify complex model element mappings between concrete visual notational elements using interactive drag-and-drop and reusable, spread sheet-like mapping formulae. Complex, scalable, efficient, accurate and reusable model transformation implementations are then generated from these by-example visual source-to-target mappings. The use of this concrete source-to-target mapping metaphor can be generalised to a wide range of model transformation problems.

This new approach provides support for visualising example models to enable a more user-centric specification of transformation rules using their concrete notations. It allows end users to interactively specify rich, human-centric visualisations of complex data using a visual, drag-and-drop, by-example approach. End users can generate reusable visualisation implementations from these high-level specifications, and use their generated, reusable model visualisations to visualise two (or more) complex data sets (i.e. example models). Model element mappings between their data sets are then generated via drag-and-drop of concrete visualisation elements.

To enable efficient model transformations, the approach automatically creates high-level abstractions for transformation generation from the concrete visualisations. Metamodels of the underlying model of visualisations are reverse engineered to automatically create abstractions from user-provided model examples.

In addition, to better aid users to find correspondences in large model visualisations, an automatic recommender system was introduced which provides suggestions for possible correspondences between source and target model elements. This recommender system uses model characteristics and visual representations to generate guidance for large model mapping problems. These recommendations allow users to cut corners in specification of transformation correspondences by choosing among suggestions. Complex model transformation code is automatically generated from the user's interaction with concrete notations and suggested recommendations.

A proof of concept implementation of this approach, CONcrete Visual assistEd Transformation framework (CONVErT), was introduced to help realisation and evaluation of the approach. It allows generation, design and use of varieties of notations including text, boxes and lines, shapes, etc. It integrates the use and definition of mapping functions and conditions and enables reverse engineering of metamodels. CONVErT generates reverse transformations automatically (when possible) for bijective transformations. It also provides a visual representation of transformation rules using rule's source and target notation visualisations. CONVErT framework is not limited to specific domain and is suitable for a range of large-scale model to model transformation problems, including software tool integration, EDI message transformation, and CAD tool integration among others. In summary, key novel contributions of this thesis research are:

- Producing reusable model visualisation specifications using an interactive, by-example approach.
- Using a concrete, by-example model transformation metaphor.
- Model mapping and transformation specification by drag and drop between concrete visualisations.
- Utilising a set of recommenders using various recommender system techniques and generating mappings from recommendations.
- Supporting fully automated model transformation script generation from specified mappings.

- Providing scalable, easy-to-use, robust and extensive tool support for each of these facilities.
- Carrying out an end user evaluation of our prototype toolset and overall approach.

Next section summarises key future work and research directions derived from this thesis.

9.2 Future work

Indeed a major future work is to perform a structured quantitative experiment from which statistically significant statements can be made for example to mitigate against learning effects. Additionally, four key directions have been identified for future work, 1) improved transformation generation and tool support, 2) improved transformation recommender system, 3) dynamic visualisation, and 4) application to other domains. The following subsections describe these future directions in detail.

9.2.1 Transformation generation and tool support

To further expand the scope of the work presented by this thesis, support for complete automatic reverse transformation generation, handling of lossy transformations, composition of transformation rules and model checking for conditional transformations can be also integrated.

The tool support provided in this thesis (CONVErT), was intended as a proof of concept prototype and therefore has number of implementation specific shortcomings to be addressed. For example, it can be improved to accommodate alternative transformation languages like ATL or TGG.

CONVErT uses set of predefined functions to generate more complex transformation rules. Although it is possible to add to the list of functions by using the function template, it is not fully targeted for novice users as it involves understanding of

functions provided by the transformation language. Tool support can be further improved by providing more reusable functions and better function designer interfaces.

Although values to take part in model transformation can be imported from multiple input files, a multi-model to multi-model source to target transformation is not allowed by the framework. Further improvements can cover this limitation.

9.2.2 Transformation recommender system

The Suggester system introduced by this thesis was designed to recommend model element correspondences. It can be further improved to consider links between model and visualisation and suggest correspondences by analysing visual similarity of model elements. For example, if two notations have a box shape, they may probably correspond to each other.

The Suggester system of CONVErT is limited to recommending one to one correspondences and cannot recommend transformation rules. The recommended correspondences (if accepted) will help generating transformation rule templates or internal rule correspondences. It can be extended to provide transformation rules by grouping set of related correspondences. Additionally, the framework can be altered to automatically accept high scoring recommendations. This way it will help further improve efficiency of transformation designers.

Representation of recommendations in our approach is based on list-wise arrangement of suggested recommendations. More advanced visual representations can be included in the framework to augment recommendations in visualisations. Also, further user studies can be conducted to assess the utility of drag and drop actions on the visualisations independently and subsequently examine the additional benefit of the recommendations.

9.2.3 Dynamic and enhanced visualisation

Visualisation examples provided in this thesis were mostly proof of concepts and were provided to show capabilities of our approach. Consequently, very large scale and more

complex examples were not provided. We seek to apply the approach on larger example visualisations including 3D visualisations (e.g. X3D, GXL) or interactive visualisation for web and mobile devices. These visualisations could explore temporal influence on data to show potential multiple linked views of an underlying data set.

Additionally, the approach presented by this thesis is currently limited to visualisations that exhibit clear notation separations. For example, in a class diagram, each class is composed of set of attributes and operations which exhibit clear separation in terms of visual view and their model with the class itself. A visualisation example that does not exhibit this separation is Euler diagrams. In Euler diagrams, each set's notation may contain other sets. With our approach, since the model representing sets are the same (with regards to their abstraction), this will result in an ambiguity for the transformation engine to produce the final visualisations as the transformation rules will have to call themselves recursively. We have not tested these visualisations thoroughly with our approach, and therefore have assigned their support as part of our future work.

The visual notation generation approach (Skin++) introduced in this thesis can be altered to accommodate definition of interaction tasks as well. Currently the interaction embedded in notations is defined for transformation code generation. This can be specified according to users' needs. For example, to show elements of the source model that the target notations are sourced from, by right clicking on target elements; or provide drill-down or hide/show visual elements; or to embed further data relations in the visualisations. An example is where a pie chart has been visualised representing percentage of people who voted for certain product. By clicking on a pie piece in this visualisation, it would be possible to show what percentage of them were male and what percentage were female.

9.2.4 Other Domains

We are investigating possible application domains for our visualisation and transformation approach other than those mentioned in this thesis. These domains can benefit our approach in both areas of visualisation by example and transformation using concrete visualisations by example.

The visualisation by example approach can benefit users that are not expert in visualisations techniques and software engineering in general. Examples of such domains are genealogy and urban traffic monitoring. Users of these domains have the required knowledge to process and understand their data and therefore, can use examples of such data as the basis for visualisation and data to visualisation mapping.

Data and tool integration is among possible domains that can benefit from our concrete visual transformation approach. For example, consider Electronic Data Interchange (EDI) messages that have become standard in e-commerce applications. When a parent company's system is to send or receive data from non-EDI based third parties, a data transformation should be used. Given that users of such e-commerce applications might not be experts in transformation generation, an approach that uses familiar visualisation of both source and target messages and generates transformers by drag and drop can be very helpful.

References / Bibliography

- [1] “ATL Transformation Zoo, A Collection of ATL Transformation Examples,” 2012. [Online]. Available: <http://www.eclipse.org/m2m/atl/atlTransformations/>.
- [2] G. E. Krasner and S. T. Pope, “A Cookbook for Using the Model- View-Controller User Interface Paradigm in Smalltalk-80,” *J. Object Oriented Program.*, vol. 1, no. 3, pp. 26–49, 1988.
- [3] E. R. Tufte, *Beautiful Evidence*, 1St Editio. Cheshire, Connecticut: Graphics Press, 2006.
- [4] M. Humphrey, “Creating reusable visualizations with the relational visualization notation,” in *Proceedings of the conference on Visualization'00*, 2000, pp. 53–60.
- [5] R. Singh, J. Xu, and B. Berger, “Global alignment of multiple protein interaction networks with application to functional orthology detection,” *Proc. Natl. Acad. Sci. U. S. A.*, vol. 105, no. 35, pp. 12763–12768, Sep. 2008.
- [6] M. F. van Amstel, C. F. J. Lange, and M. G. J. van den Brand, “Metrics for analyzing the quality of model transformations,” in *12th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2008)*, 2008, pp. 41–51.
- [7] E. Seidewitz, “What models mean,” *IEEE Softw.*, vol. 20, no. 5, pp. 26–32, Sep. 2003.
- [8] S. Kent, “Model driven engineering,” in *Integrated Formal Methods*, 2002, pp. 286–298.
- [9] R. France and B. Rumpe, “Model-driven Development of Complex Software : A Research Roadmap,” in *Future of Software Engineering (FOSE)*, 2007, pp. 37–54.
- [10] S. Sendall and W. Kozaczynski, “Model Transformation: The Heart and Soul of Model-Driven Software Development,” *IEEE Softw.*, vol. 20, no. 5, pp. 42–45, Sep. 2003.
- [11] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
- [12] L. Kapova, T. Goldschmidt, J. Happe, and R. H. Reussner, “Domain-specific templates for refinement transformations,” in *Proceedings of the First International Workshop on Model-Driven Interoperability*, 2010, pp. 69–78.
- [13] T. Mens, G. Taentzer, and D. Müller, “Challenges in model refactoring,” in *Proc. 1st Workshop on Refactoring Tools, University of Berlin*, 2007, vol. 98, pp. 1–5.

- [14] R. Bull and J. Favre, “Visualization in the Context of Model Driven Engineering,” in *Proceedings of the MoDELS’05 Workshop on Model Driven Development of Advanced User Interfaces*, 2005.
- [15] H. A. Muller and K. Klashinsky, “Rigi: a system for programming-in-the-large,” in *Software Engineering, 1988., Proceedings of the 10th International Conference on*, 1988, pp. 80–86.
- [16] A. M. Ernst, J. Lankes, C. M. Schweda, and E. Denert-stiftungslehrstuhl, “Using Model Transformation for Generating Visualizations from Repository Contents, An Application to Software Cartography,” Munchen, Germany, 2006.
- [17] M.-A. D. Storey and H. A. Muller, “Manipulating and documenting software structures using SHriMP views,” in *Software Maintenance, 1995. Proceedings., International Conference on*, 1995, pp. 275–284.
- [18] J. C. Grundy, J. G. Hosking, R. W. Amor, W. B. Mugridge, and Y. Li, “Domain-specific visual languages for specifying and generating data mapping systems,” *J. Vis. Lang. Comput.*, vol. 15, no. 3–4, pp. 243–263, Jun. 2004.
- [19] J. Grundy, J. Hosking, J. Huh, and K. Li, “Marama : an Eclipse meta-toolset for generating multi-view environments,” in *Proceedings of the 30th international conference on Software engineering ICSE 2008*, 2008, pp. 819–822.
- [20] M. Minas, “Concepts and realization of a diagram editor generator based on hypergraph transformation,” *Sci. Comput. Program.*, vol. 44, no. 2, pp. 157–180, 2002.
- [21] J. Hosking, S. Fenwick, W. Mugridge, and J. Grundy, “Cover yourself with Skin,” Queensland 4072 Australia, 1994.
- [22] D. Steinberg, F. Budinsky, P. Marcelo, and E. Merks, *Eclipse Modeling Framework (The Eclipse Series)*, 2nd ed. Addison-Wesley Professional, 2009, pp. 1–744.
- [23] “Eclipse GMF website.,” <http://www.eclipse.org/modeling/gmp/>. [Online]. Available: <http://www.eclipse.org/modeling/gmp/>.
- [24] A. Königs, “Model Transformation with Triple Graph Grammars,” in *Model Transformations in Practice Satellite Workshop of MODELS*, 2005, pp. 1–16.
- [25] L. Grunske, L. Geiger, and M. Lawley, “A graphical specification of model transformations with triple graph grammars,” in *Model Driven Architecture—Foundations and Applications*, 2005, pp. 284–298.
- [26] J. Bézivin, F. Jouault, and D. Touzet, “An introduction to the ATLAS Model Management Architecture,” 2005.

- [27] J. Bézivin, G. Dupé, F. Jouault, G. Pitette, and J. E. Rougui, “First experiments with the ATL model transformation language: Transforming XSLT into XQuery,” in *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003, p. 50.
- [28] J. Bettin, “Ideas for a concrete visual syntax for model-to-model transformations,” in *Proceedings of the 18th International Conference, OOPSLA 2003, Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.
- [29] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer, “Model Transformation By-Example: A Survey of the First Wave,” *Concept. Model. Its Theor. Found.*, vol. 7260, pp. 197–215, 2012.
- [30] I. García-Magariño, J. Gómez-Sanz, and R. Fuentes-Fernández, “Model transformation by-example: an algorithm for generating many-to-many transformation rules in several model transformation languages,” in *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT '09)*, 2009, pp. 52–66.
- [31] J. Grundy, R. Mugridge, J. Hosking, and P. Kendall, “Generating EDI message translations from visual specifications,” in *Proceedings. 16th Annual International Conference on Automated Software Engineering, (ASE 2001).,* 2001, pp. 35–42.
- [32] Y. Li, J. Grundy, R. Amor, and J. Hosking, “A data mapping specification environment using a concrete business form-based metaphor,” in *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, 2002, pp. 158–166.
- [33] T. Baar and J. Whittle, “On the Usage of Concrete Syntax in Model Transformation Rules,” in *Proceedings of the 6th international Andrei Ershov memorial conference on Perspectives of systems informatics (PSI' 2006)*, 2006, vol. 20, no. 1, pp. 84–97.
- [34] X. Dolques, M. Huchard, C. Nebut, and P. Reitz, “Learning Transformation Rules from Transformation Examples: An Approach Based on Relational Concept Analysis,” in *14th IEEE International Enterprise Distributed Object Computing Conference Workshops*, 2010, pp. 27–32.
- [35] R. Grønmo, “Using Concrete Syntax in Graph-based Model Transformations,” University of Oslo, 2009.
- [36] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler, “Towards Model Transformation Generation By-Example,” in *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 2007, pp. 285–295.
- [37] G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, and M. Wimmer, “Lifting metamodels to ontologies: A step to the

- semantic integration of modeling languages,” in *Model Driven Engineering Languages and Systems*, 2006, pp. 528–542.
- [38] Y. Sun, J. White, and J. Gray, “Model transformation by demonstration,” in *Model Driven Engineering Languages and Systems*, 2009, pp. 712–726.
 - [39] M. Kessentini, H. Sahraoui, M. Boukadoum, and O. Ben Omar, “Search-based model transformation by example,” *Softw. Syst. Model.*, pp. 1–18, Sep. 2010.
 - [40] D. Varró, “Model transformation by example,” in *Model Driven Engineering Languages and Systems*, 2006, pp. 410–424.
 - [41] D. Varró and Z. Balogh, “Automating model transformation by example using inductive logic programming,” *Proc. 2007 ACM Symp. Appl. Comput. - SAC ’07*, p. 978, 2007.
 - [42] X. Dolques, A. Dogui, J. Falleri, M. Huchard, C. Nebut, and F. Pfister, “Easing model transformation learning with automatically aligned examples,” in *7th European Conference on Modelling Foundations and Applications*, 2011, pp. 189–204.
 - [43] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, “An example is worth a thousand words: Composite operation modeling by-example,” in *Model Driven Engineering Languages and Systems*, 2009, pp. 271–285.
 - [44] R. Robbes and M. Lanza, “Example-based program transformation,” in *Model Driven Engineering Languages and Systems*, 2008, pp. 174–188.
 - [45] H. Lieberman, *Your wish is my command: programming by example*. Morgan Kaufmann Publishers, 2001, pp. 1–416.
 - [46] S. Bossung, H. Stoeckle, J. Grundy, R. Amor, and J. Hosking, “Automated data mapping specification via schema heuristics and user interaction,” in *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, 2004, no. c, pp. 208–217.
 - [47] Altova, “MapForce,” 2013. [Online]. Available: <http://www.altova.com/mapforce.html>.
 - [48] I. Kurtev, J. Bézivin, and M. Akcsit, “Technological Spaces: An Initial Appraisal,” in *International Conference on Cooperative Information Systems (CoopIS), DOA’2002 Federated Conferences, Industrial Track, Irvine, USA, 2002*, pp. 1–6.
 - [49] J. Bézivin, “Model driven engineering: an emerging technical space,” in *Proceedings of the 2005 international conference on Generative and Transformational Techniques in Software Engineering*, 2006, pp. 36–64.

- [50] E. Rodriguez-Priego, F. J. García-Izquierdo, and Á. L. Rubio, “Modeling issues: a survival guide for a non-expert modeler,” in *Model Driven Engineering Languages and Systems*, 2010, pp. 361–375.
- [51] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez, “Model-based DSL frameworks,” in *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '06*, 2006, pp. 602–615.
- [52] M. Siikarla, “A Light-weight Approach to Developing Interactive Model Transformations,” Tempere University of Technology, 2011.
- [53] R. Lämmel and E. Meijer, “Mappings make data processing go’round,” in *Generative and Transformational Techniques in Software Engineering*, 2006, pp. 169–218.
- [54] C. Burt, D. Dsouza, K. Duddy, W. El Kaim, W. Frank, S. Iyengar, J. Miller, J. Mischkinsky, J. Mukerji, J. Siegel, R. Soley, S. Tyndal-, A. Uhl, A. Watson, and B. Wood, “Model Driven Architecture (MDA) Document number ormsc / 2001-07-01,” 2001.
- [55] G. Miller, S. Ambler, S. Cook, S. Mellor, K. Frank, and J. Kern, “Model driven architecture,” in *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '04*, 2004, p. 138.
- [56] F. Fondement, “CONCRETE SYNTAX DEFINITION FOR MODELING LANGUAGES,” Ecole Polytechnique Fédérale de Lausanne, 2007.
- [57] T. Baar, “Correctly defined concrete syntax,” *Softw. Syst. Model.*, vol. 7, no. 4, pp. 383–398, Jul. 2008.
- [58] E. Visser, “Meta-programming with concrete object syntax,” in *Generative Programming and Component Engineering ACM SIGPLAN/SIGSOFT Conference, GPCE*, 2002, pp. 299–315.
- [59] H. Krahn, B. Rumpe, and S. Völkel, “Integrated definition of abstract and concrete syntax for textual languages,” in *Proceedings of the 10th international conference on Model Driven Engineering Languages and Systems*, 2007, pp. 286–300.
- [60] J. Bézivin, “In search of a basic principle for model driven engineering,” *Novatica Journal, Spec. Issue*, vol. V, no. 2, pp. 21–24, 2004.
- [61] B. Selic, “The pragmatics of model-driven development,” *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep. 2003.

- [62] K. Czarnecki and S. Helsen, “Classification of model transformation approaches,” in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003, pp. 1–17.
- [63] K. Czarnecki, “Feature-based survey of model transformation approaches,” *IBM Syst. J.*, vol. 45, no. 3, pp. 621–645, 2006.
- [64] M. Amrani, J. Dingel, L. Lambers, L. Lúcio, R. Salay, G. Selim, E. Syriani, and M. Wimmer, “Towards a model transformation intent catalog,” in *Proceedings of the First Workshop on the Analysis of Model Transformations*, 2012, pp. 3–8.
- [65] T. Mens and P. Van Gorp, “A Taxonomy of Model Transformation,” *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, Mar. 2006.
- [66] R. Schaefer, “A survey on transformation tools for model based user interface development,” in *HCI’07 Proceedings of the 12th international conference on Human-computer interaction: interaction design and usability*, 2007, pp. 1178–1187.
- [67] J. Cordy, “The TXL source transformation language,” *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, Aug. 2006.
- [68] F. Jouault and I. Kurtev, “Transforming models with ATL,” in *Satellite Events at the MoDELS 2005 Conference*, 2006, pp. 128–138.
- [69] J. Clark, “XSL Transformations (XSLT) W3C Recommendation,” 1999.
- [70] T. Gardner, C. Griffin, J. Koehler, and R. Hauser, “A review of OMG MOF 2 . 0 Query / Views / Transformations Submissions and Recommendations towards the final Standard.” OMG Document, 2003.
- [71] A. Kalnins, J. Barzdins, and E. Celms, “The model transformation language MOLA,” in *Model Driven Architecture European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004*,, 2005, vol. 68, no. 3, pp. 62–76.
- [72] D. Kolovos, R. Paige, and F. Polack, “The epsilon transformation language,” in *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, 2008, pp. 46–60.
- [73] A. Schürr, “Specification of graph translators with triple graph grammars,” in *Graph-Theoretic Concepts in Computer Science 20th International Workshop, WG ’94 June 16–18, 1994 Proceedings*, 1995, pp. 151–163.
- [74] U. Nickel, J. Niere, and A. Zündorf, “The FUJABA Environment,” in *Proceedings of the 22nd international conference on Software engineering*, 2000, pp. 742–745.

- [75] R. Wagner, “Developing Model Transformations with Fujaba,” in *Proceedings of the 4th International Fujaba Days, 2006*, 2006, pp. 79–82.
- [76] A. Agrawal, “GReAT: a metamodel based model transformation language,” in *18th IEEE International Conference on Automated Software Engineering*, 2003.
- [77] “XML path language (XPath) 2.0,” *World Wide Web Consortium*, 2005. .
- [78] P. Stevens, “Bidirectional model transformations in QVT: semantic issues and open questions,” *Softw. Syst. Model.*, vol. 9, no. 1, pp. 7–20, Dec. 2008.
- [79] A. Kalnins, E. Celms, and A. Sostaks, “Tool support for MOLA,” *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 83–96, Mar. 2006.
- [80] I. Avazpour and J. Grundy, “CONVERT: A framework for complex model visualisation and transformation,” in *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2012, pp. 237–238.
- [81] M. Faunes, H. Sahraoui, and M. Boukadoum, “Generating model transformation rules from examples using an evolutionary algorithm,” in *27th IEEE/ACM International Conference on Automated software Engineering*, 2012, pp. 250–253.
- [82] M. Kessentini, H. Sahraoui, and M. Boukadoum, “Model Transformation as an Optimization Problem,” in *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, 2008, pp. 159–173.
- [83] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut, “Metamodel matching for automatic model transformation generation,” in *Model Driven Engineering Languages and Systems*, 2008, no. Cpre 5326, pp. 326–340.
- [84] K. Voigt and T. Heinze, “Metamodel matching based on planar graph edit distance,” in *Third International Conference on Theory and Practice of Model Transformations, ICMT 2010*, 2010, pp. 245–259.
- [85] L. Lafi, S. Hammoudi, and J. Feki, “Metamodel matching techniques in MDA: challenge, issues and comparison,” in *Model and Data Engineering*, 2011, pp. 278–286.
- [86] M. Strommer, M. Murzek, and M. Wimmer, “Applying model transformation by-example on business process modeling languages,” in *Proceedings of the 2007 conference on Advances in conceptual modeling: foundations and applications*, 2007, pp. 116–125.
- [87] I. Avazpour and J. Grundy, “Using Concrete Visual Notations as First Class Citizens for Model Transformation Specification,” in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2013, pp. 87–90.

- [88] H. Cho, J. Gray, Y. Sun, and J. White, “Modeling Language Creation by Demonstration,” 2010.
- [89] I. Avazpour, J. Grundy, and L. Grunske, “Tool Support for Automatic Model Transformation Specification using Concrete Visualisations,” in *IEEE/ACM International Conference on Automated Software Engineering*, 2013.
- [90] G. G. Robertson, M. P. Czerwinski, and J. E. Churchill, “Visualization of mappings between schemas,” in *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '05*, 2005, p. 431.
- [91] R. Grønmo, B. Møller-Pedersen, and G. K. Olsen, “Comparison of three model transformation languages,” in *ECMDA-FA '09 Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, 2009, pp. 2–17.
- [92] H. Stoeckle, J. Grundy, and J. Hosking, “A framework for visual notation exchange,” *J. Vis. Lang. Comput.*, vol. 16, no. 3, pp. 187–212, Jun. 2005.
- [93] H. Stoeckle, J. Grundy, and J. Hosking, “Approaches to supporting software visual notation exchange,” in *IEEE Symposium on Human Centric Computing Languages and Environments, Proceedings. 2003*, 2003, no. October, pp. 59–66.
- [94] M. Schmidt, “Transformations of UML 2 models using concrete syntax patterns,” in *Rapid Integration of Software Engineering Techniques*, 2006, pp. 130–143.
- [95] E. Visser, “Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5,” in *Rewriting Techniques and Applications*, 2001, pp. 357–361.
- [96] J. de Lara and H. Vangheluwe, “AToM3: A Tool for Multi-formalism and Meta-modelling,” in *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, 2002, pp. 174–188.
- [97] A. Balogh and D. Varró, “Advanced model transformation language constructs in the VIATRA2 framework,” in *Proceedings of the 2006 ACM symposium on Applied computing - SAC '06*, 2006, pp. 1280–1287.
- [98] E. Willink, “UMLX: A graphical transformation language for MDA,” in *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*, 2003, pp. 13–24.
- [99] P. Braun and F. Marschall, “Transforming Object Oriented Models with BOTL,” *Electr. Notes Theor. Comput. Sci.*, vol. 72, no. 3, pp. 103–117, 2003.
- [100] P. Braun and F. Marschall, “Botl—the bidirectional object oriented transformation language,” 2003.

- [101] L. Haas and H. Ho, “Clio grows up: from research prototype to industrial tool,” in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 805–810.
- [102] P. Stevens, “A Landscape of Bidirectional Model Transformation,” in *Generative and Transformational Techniques in Software Engineering II*, R. Lämmel, V. Joost, and J. Saraiva, Eds. Springer-Verlag, 2008, pp. 408–424.
- [103] R. Fagin, L. Haas, M. Hernández, R. Miller, L. Popa, and Y. Velegrakis, “Clio: Schema Mapping Creation and Data Exchange,” in *Conceptual Modeling: Foundations and Applications*, vol. 5600, A. Borgida, V. Chaudhri, P. Giorgini, and E. Yu, Eds. Springer Berlin Heidelberg, 2009, pp. 198–236.
- [104] G. Lindén, H. Tirri, and A. I. Verkamo, “ALCHEMIST: a general purpose transformation generator,” *Softw. Pr. Exper.*, vol. 26, no. 6, pp. 653–675, Jun. 1996.
- [105] A. Van Deursen, J. Heering, and P. Klint, *Language Prototyping: An Algebraic Specification Approach*. World Scientific Publishing Co., Inc., 1996.
- [106] D. Batory, B. Lofaso, and Y. Smaragdakis, “JTS: Tools for Implementing Domain-Specific Languages,” in *Proceedings of the 5th International Conference on Software Reuse*, 1998, p. 143–.
- [107] J. Izquierdo, J. Cuadrado, and J. G. Molina, “Gra2MoL: A domain specific transformation language for bridging grammarware to modelware in software modernization,” in *Workshop on Model-Driven Software Evolution*, 2008, pp. 1–8.
- [108] “Eclipse.” [Online]. Available: <http://www.eclipse.org/>.
- [109] S. Fenwick, J. Hosking, and M. Warwick, “A Visualisation System for Object-Oriented Programs,” in *Technology of object-oriented languages and systems TOOLS 15*, 1994, pp. 93–103.
- [110] L. Li, J. Hosking, and J. Grundy, “MaramaEML: An Integrated Multi-View Business Process Modelling Environment with Tree-Overlays, Zoomable Interfaces and Code Generation,” *2008 23rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pp. 477–478, Sep. 2008.
- [111] M. Kamalrudin, J. Grundy, and J. Hosking, “MaramaAI: tool support for capturing and managing consistency of multi-lingual requirements,” in *Proceedings of the 27th IEEE/ACM Conference on Automated Software Engineering*, 2012, pp. 1–4.
- [112] M. Minas, “Specifying Graph-like Diagrams with DiaGen,” *Electron. Notes Theor. Comput. Sci.*, vol. 72, no. 2, pp. 102–111, 2002.

- [113] A. R. Jansen, K. Marriott, and B. Meyer, “CIDER : A Component-Based Toolkit for Creating Smart Diagram Environments,” in *Proceedings of the Ninth International Conference on Distributed and Multimedia Systems*, 2003, pp. 353–359.
- [114] K. Pantazos and S. Lauesen, “Constructing Visualizations with InfoVis Tools - An Evaluation from a user Perspective,” in *Proceedings of the International Conference on Information Visualization Theory and Applications*, 2012, pp. 731–736.
- [115] K. Pantazos, S. Xu, M. Kuhail, and S. Lauesen, “uVis: A Formula-Based Visualization Tool,” in *IEEE VisWeek 2010 Posters*, 2010.
- [116] I. Herman, G. Melançon, and M. S. Marshall, “Graph Visualization and Navigation in Information Visualization: A Survey,” *IEEE Trans. Vis. Comput. Graph.*, vol. 6, no. 1, pp. 24–43, Jan. 2000.
- [117] C. Hirsch, J. Hosking, J. Grundy, T. Chaffe, D. Macdonald, and Y. Halytskyy, “The Visual Wiki: A New Metaphor for Knowledge Access and Management,” in *System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on*, 2009, pp. 1–10.
- [118] C. Hirsch, J. Hosking, J. Grundy, and T. Chaffe, “ThinkFree: using a visual Wiki for IT knowledge management in a tertiary institution,” in *Proceedings of the 6th International Symposium on Wikis and Open Collaboration*, 2010, pp. 7:1–7:10.
- [119] C. Hirsch, J. Hosking, and J. Grundy, “VikiBuilder: end-user specification and generation of visual wikis,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 13–22.
- [120] M. P. Robillard, R. J. Walker, and T. Zimmermann, “Recommendation Systems for Software Engineering,” *Software, IEEE*, vol. 27, no. 4, pp. 80–86, 2010.
- [121] G. Adomavicius and a. Tuzhilin, “Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions,” *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 6, pp. 734–749, Jun. 2005.
- [122] P. Resnick and H. R. Varian, “Recommender systems,” *Commun. ACM*, vol. 40, no. 3, pp. 56–58, 1997.
- [123] A. Felfernig, G. Friedrich, and L. Schmidt-Thieme, “Recommender systems,” *IEEE Intell. Syst.*, vol. 22, pp. 18–21, 2007.
- [124] C. Drumm, M. Schmitt, H.-H. Do, and E. Rahm, “Quick mig: automatic schema matching for data migration projects,” in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, 2007, pp. 107–116.

- [125] H. Kargl and M. Wimmer, “SmartMatcher-How Examples and a Dedicated Mapping Language can Improve the Quality of Automatic Matching Approaches,” in *Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems CISIS '08*, 2008, pp. 879–885.
- [126] H. Kache, Y. Saitte, and M. Roth, “Transformation rule discovery through data mining,” in *International Workshop on New Trends in Information Integration (NTII)*, 2008, pp. 24–27.
- [127] P. Yeh, B. Porter, and K. Barker, “Mining transformation rules for semantic matching,” in *ECML/PKDD 2nd International Workshop on Mining Graphs, Trees, and Sequences*, 2004, pp. 1–12.
- [128] J. Sousa José, D. Lopes, D. Claro, and Z. Abdelouahab, “A Step Forward in Semi-automatic Metamodel Matching: Algorithms and Tool,” in *Enterprise Information Systems*, vol. 24, J. Filipe and J. Cordeiro, Eds. Springer Berlin Heidelberg, 2009, pp. 137–148.
- [129] S. Melnik, H. Garcia-Molina, and E. Rahm, “Similarity flooding: a versatile graph matching algorithm and its application to schema matching,” in *Proceedings 18th International Conference on Data Engineering*, 2002, pp. 117–128.
- [130] P. Ivanov and K. Voigt, “Schema, ontology and metamodel matching-different, but indeed the same?,” in *Model and Data Engineering, First International Conference*, 2011, pp. 18–30.
- [131] P. A. Bernstein and S. Melnik, “Model management 2.0: manipulating richer mappings,” in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 1–12.
- [132] P. Atzeni, P. Cappellari, and P. A. Bernstein, “Modelgen: Model independent schema translation,” in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, 2005, no. 3, pp. 1111–1112.
- [133] J. Euzenat, “An API for Ontology Alignment,” in *The Semantic Web – ISWC 2004*, vol. 3298, S. McIlraith, D. Plexousakis, and F. Harmelen, Eds. Springer Berlin Heidelberg, 2004, pp. 698–712.
- [134] D. Mladenic, “Text-learning and related intelligent agents: a survey,” *Intell. Syst. their Appl. IEEE*, vol. 14, no. 4, pp. 44–54, 1999.
- [135] J. B. Schafer, D. Frankowski, J. Herlocker, and S. Sen, “Collaborative Filtering Recommender Systems,” in *The Adaptive Web*, vol. 4321, P. Brusilovsky, A. Kobsa, and W. Nejdl, Eds. Springer Berlin Heidelberg, 2007, pp. 291–324.
- [136] R. Burke, “Hybrid Recommender Systems: Survey and Experiments,” *User Model. User-adapt. Interact.*, vol. 12, no. 4, pp. 331–370, 2002.

- [137] R. Burke, “Hybrid Web Recommender Systems,” pp. 377–408, 2007.
- [138] Y. Ye and G. Fischer, “Reuse-Conducive Development Environments,” *Autom. Softw. Engg.*, vol. 12, no. 2, pp. 199–235, Apr. 2005.
- [139] M. P. Robillard, “Topology analysis of software dependencies,” *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 4, pp. 1–36, Aug. 2008.
- [140] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, “Hipikat: a project memory for software development,” *Softw. Eng. IEEE Trans.*, vol. 31, no. 6, pp. 446–465, 2005.
- [141] F. McCarey, M. Ó. Cinnéide, and N. Kushmerick, “Rascal: A Recommender Agent for Agile Reuse,” *Artif. Intell. Rev.*, vol. 24, no. 3–4, pp. 253–276, Nov. 2005.
- [142] A. Ankolekar, K. Sycara, J. Herbsleb, R. Kraut, and C. Welty, “Supporting online problem-solving communities with the semantic web,” in *Proceedings of the 15th international conference on World Wide Web*, 2006, pp. 575–584.
- [143] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala, “DebugAdvisor: a recommender system for debugging,” in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 373–382.
- [144] A. Mockus and J. D. Herbsleb, “Expertise browser: a quantitative approach to identifying expertise,” in *Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 503–512.
- [145] M. Kersten and G. C. Murphy, “Using task context to improve programmer productivity,” in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 1–11.
- [146] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers, “Active code completion,” in *Proceedings of the 2012 International Conference on Software Engineering*, 2012, pp. 859–869.
- [147] R. Robbes and M. Lanza, “Improving code completion with program history,” *Autom. Softw. Eng.*, vol. 17, no. 2, pp. 181–212, 2010.
- [148] M. Bruch, M. Monperrus, and M. Mezini, “Learning from examples to improve code completion systems,” in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 213–222.
- [149] S. Mazanek, C. Rutetzki, and M. Minas, “Sketch-based Diagram Editors with User Assistance based on Graph Transformation and Graph Drawing

- Techniques,” in *Electronic Communications of the EASST Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)*, 2010, vol. 32.
- [150] G. Costagliola, V. Deufemia, and M. Risi, “Using Grammar-Based Recognizers for Symbol Completion in Diagrammatic Sketches,” in *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 02*, 2007, pp. 1078–1082.
 - [151] S. Mazanek and M. Minas, “Business Process Models as a Showcase for Syntax-Based Assistance in Diagram Editors,” in in *Model Driven Engineering Languages and Systems*, vol. 5795, A. Schürr and B. Selic, Eds. Springer Berlin Heidelberg, 2009, pp. 322–336.
 - [152] S. Sen, B. Baudry, and H. Vangheluwe, “Domain-Specific Model Editors with Model Completion,” in in *Models in Software Engineering*, H. Giese, Ed. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 259–270.
 - [153] S. Sen, B. Baudry, and H. Vangheluwe, “Towards Domain-specific Model Editors with Automatic Model Completion,” *Simul. J.*, vol. 3, no. 12, pp. 109–126, 2009.
 - [154] M. Born, C. Brelage, I. Markovic, D. Pfeiffer, and I. Weber, “Auto-completion for Executable Business Process Models,” in in *Business Process Management Workshops*, vol. 17, D. Ardagna, M. Mecella, and J. Yang, Eds. Springer Berlin Heidelberg, 2009, pp. 510–515.
 - [155] N. Mohd Ali, J. Hosking, J. Grundy, and J. Huh, “End-user oriented critic specification for domain-specific visual language tools,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 297–300.
 - [156] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, “Bidirectional Transformations: A Cross-Discipline Perspective GRACE meeting notes, state of the art, and outlook,” *Theory Pract. Model Transform.*, vol. 5563, pp. 260–283, 2009.
 - [157] M. Siikarla, M. Laitkorpi, P. Selonen, and T. Systä, “Transformations have to be developed ReST assured,” in *First International Conference on Theory and Practice of Model Transformations, ICMT 2008*, 2008, pp. 1–15.
 - [158] B. Alexe, L. Chiticariu, R. J. Miller, and W.-C. Tan, “Muse: Mapping Understanding and deSign by Example,” in *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, 2008, pp. 10–19.
 - [159] D. L. Moody, “The ‘Physics’ of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering,” *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 756–779, 2009.

- [160] M. Hicks, “Perceptual and design principles for effective interactive visualisations,” in in *Trends in Interactive Visualization*, R. Liere, T. Adriaansen, and E. Zudilova-Seinstra, Eds. London: Springer London, 2009, pp. 155–174.
- [161] J. L. Herlocker, J. a. Konstan, L. G. Terveen, and J. T. Riedl, “Evaluating collaborative filtering recommender systems,” *ACM Trans. Inf. Syst.*, vol. 22, no. 1, pp. 5–53, Jan. 2004.
- [162] I. Avazpour, T. Pitakrat, L. Grunske, and J. Grundy, “Dimensions and Metrics for Evaluating Recommendation Systems,” in in *Recommendation Systems in Software Engineering*, M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds. Springer, pp. 1–29.
- [163] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, “Top 10 algorithms in data mining,” *Knowl. Inf. Syst.*, vol. 14, no. 1, pp. 1–37, Dec. 2008.
- [164] R. Amor, G. Augenbroe, J. Hosking, and W. Rombouts, “Directions in modelling environments,” *Autom. Constr.*, vol. 4, no. 3, pp. 173–187, Oct. 1995.
- [165] K. Swearingen and R. Sinha, “Beyond algorithms: An HCI perspective on recommender systems,” in *ACM SIGIR 2001 Workshop on Recommender Systems*, 2001, pp. 1–11.
- [166] E. Murphy-Hill, R. Jiresal, and G. C. Murphy, “Improving software developers’ fluency by recommending development environment commands,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 42:1–42:11.
- [167] S. McNee, J. Riedl, and J. Konstan, “Being accurate is not enough: how accuracy metrics have hurt recommender systems,” in *Human factors in computing systems*, 2006, pp. 1097–1101.
- [168] Y. Koren, R. Bell, and C. Volinsky, “Matrix Factorization Techniques for Recommender Systems,” *Computer (Long. Beach. Calif.)*, vol. 42, no. 8, pp. 30–37, Aug. 2009.
- [169] R. M. Bell and Y. Koren, “Lessons from the Netflix Prize Challenge,” *SIGKDD Explor. Newsl.*, vol. 9, no. 2, pp. 75–79, 2007.
- [170] J. Bennett and S. Lanning, “The Netflix Prize,” in *Proc. KDD-Cup and Workshop at the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2007.
- [171] S. Brin and L. Page, “The anatomy of a large-scale hypertextual Web search engine,” *Comput. networks ISDN Syst.*, vol. 30, pp. 107–117, 1998.

- [172] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, “Basic local alignment search tool.,” *J. Mol. Biol.*, vol. 215, no. 3, pp. 403–410, Oct. 1990.
- [173] R. E. Schapire, “Theoretical Views of Boosting and Applications,” in *Proceedings of the 10th International Conference on Algorithmic Learning Theory*, 1999, pp. 13–25.
- [174] E. Bauer and R. Kohavi, “An Empirical Comparison of Voting Classification Algorithms: Bagging, Boosting, and Variants,” *Mach. Learn.*, vol. 36, no. 1–2, pp. 105–139, Jul. 1999.
- [175] B. Kille and S. Albayrak, “Modeling Difficulty in Recommender Systems,” in *Workshop on Recommendation Utility Evaluation: Beyond RMSE (RUE 2012)*, 2012, pp. 30–32.
- [176] L. Kuncheva and C. Whitaker, “Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy,” *Mach. Learn.*, no. 51, pp. 181–207, 2003.
- [177] D. Gusfield and R. W. Irving, *The stable marriage problem: structure and algorithms*, vol. 54. MIT press Cambridge, 1989.
- [178] F. Hernández del Olmo and E. Gaudioso, “Evaluation of recommender systems: A new approach,” *Expert Syst. Appl.*, vol. 35, no. 3, pp. 790–804, Oct. 2008.
- [179] L. A. MacVittie, *XAML in a nutshell*. O'Reilly, 2006.
- [180] M. Ge, C. Delgado-Battenfeld, and D. Jannach, “Beyond accuracy: evaluating recommender systems by coverage and serendipity,” in *Proceedings of the fourth ACM conference on Recommender systems, RecSys '10*, 2010, pp. 257–260.
- [181] M. Buckland, “The relationship between recall and precision,” *J. Am. Soc.*, vol. 45, no. 1, pp. 12–19, Jan. 1994.
- [182] C. J. Van Rijsbergen, *Information Retrieval*, 2nd ed. Newton, MA, USA: Butterworth-Heinemann, 1979.
- [183] G. Shani and A. Gunawardana, “Evaluating recommendation systems,” in *Recommender Systems Handbook*, F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, Eds. Springer US, 2011, pp. 257–297.

Appendix 1

ATL transformation script example

```
module UML2JAVA;
create OUT : JAVA from IN : UML;

helper context UML!ModelElement def: isPublic() : Boolean = self.visibility = #vk_public;

helper context UML!Feature def: isStatic() : Boolean = self.ownerScope = #sk_static;

helper context UML!Attribute def: isFinal() : Boolean = self.changeability = #ck_frozen;

helper context UML!Namespace def: getExtendedName() : String = if
self.namespace.oclIsUndefined() then
"
else if self.namespace.oclIsKindOf(UML!Model) then
"
else
    self.namespace.getExtendedName() + '.'
endif endif + self.name;

rule P2P {
    from e : UML!Package (e.oclIsTypeOf(UML!Package))
    to out : JAVA!Package (
        name <- e.getExtendedName()
    )
}

rule C2C {
    from e : UML!Class
    to out : JAVA!JavaClass (
        name <- e.name,
        isAbstract <- e.isAbstract,
        isPublic <- e.isPublic(),
        package <- e.namespace
    )
}

rule D2P {
    from e : UML!DataType
    to out : JAVA!PrimitiveType (
        name <- e.name,
        package <- e.namespace)
}
```

```

rule A2F {
    from e : UML!Attribute
    to out : JAVA!Field (
        name <- e.name,
        isStatic <- e.isStatic(),
        isPublic <- e.isPublic(),
        isFinal <- e.isFinal(),
        owner <- e.owner,
        type <- e.type
    )
}

rule O2M {
    from e : UML!Operation
    to out : JAVA!Method (
        name <- e.name,
        isStatic <- e.isStatic(),
        isPublic <- e.isPublic(),
        owner <- e.owner,
        type <- e.parameter->select(x|x.kind==#pdk_return)->asSequence()-
>first().type,
        parameters <- e.parameter->select(x|x.kind<>#pdk_return)-
>asSequence()
    )
}

rule P2F {
    from e : UML!Parameter (e.kind <> #pdk_return)
    to out : JAVA!FeatureParameter (
        name <- e.name,
        type <- e.type
    )
}

query JAVA2String_query = JAVA!JavaClass.allInstances()->
    select(e | e.oclIsTypeOf(JAVA!JavaClass))->
    collect(x | x.toString().writeTo('C:/test/' + x.package.name.replaceAll('.', '/') + '/'
+ x.name + '.java'));

uses JAVA2String;

library JAVA2String;

helper context JAVA!ClassFeature def: modifierFinal() : String = if self.isFinal then
    'final '

```

```

else
    "
endif;

helper context JAVA!ClassMember def: visibility() : String = if self.isPublic then
    'public '
else
    'private '
endif;

helper context JAVA!JavaClass def: visibility() : String =  if self.isPublic then
    'public '
else
    'private '
endif;

helper context JAVA!ClassMember def: scope() : String = if self.isStatic then
    'static '
else
    "
endif;

helper context JAVA!JavaClass def: scope() : String = if self.isStatic then
    'static '
else
    "
endif;

helper context JAVA!JavaClass def: modifierAbstract() : String = if      self.isAbstract
then
    'abstract '
else
    "
endif;

helper context JAVA!Package def: toString() : String = 'package ' + self.name + ';\n\n';

helper context JAVA!JavaClass def: toString() : String =
    self.package.toString() + self.visibility() +
    self.scope() + self.modifierAbstract() +
    self.modifierFinal() + 'class ' + self.name + ' {\n' +
    self.members->iterate(i; acc : String = " |
        acc + i.toString()
    ) +
    '\n}\n\n';

helper context JAVA!PrimitiveType def: toString() : String =  if self.name = 'Integer'
then
    'int '

```

```

else if self.name = 'Boolean' then
    'boolean'
else if self.name = 'String' then
    'java.lang.String'
else if self.name = 'Long' then
    'long'
else
    'void'
endif endif endif;

helper context JAVA!Field def: toString() : String = '\t' + self.visibility() + self.scope()
+ self.modifierFinal() + self.type.name + '' + self.name + ';\n';

helper context JAVA!Method def: toString() : String = '\t' + self.visibility() +
self.scope() + self.modifierFinal() + self.type.name + '' + self.name + '(' +
self.parameters->iterate(i; acc : String = "" | acc +
if acc = "" then
    ""
else
    ','
endif +
i.toString()
) +
') {\n\t//Your code here\n\t}\n';

```

helper context JAVA!FeatureParameter def: toString() : String = self.type.name + '' +
self.name;

Appendix 2

Ethics approval clearance

To: Prof John Grundy, FICT/ Mr Iman Avazpour

Dear Prof Grundy,

SUHREC Project 2013/010 Evaluation of a model visualisation and transformation tool (CONVERT)

Prof John Grundy, FICT/ Mr Iman Avazpour

Approved Duration: 01/03/2013 To 01/03/2014 [Adjusted]

I refer to the ethical review of the above project protocol undertaken on behalf of Swinburne's Human Research Ethics Committee (SUHREC) by SUHREC Subcommittee (SHESC2) at a meeting held on 8 February 2013. Your response to the review as e-mailed on 22 February was reviewed by a SHESC2 delegate. *Further revision to the protocol was requested by separate e-mail of today's date and the subsequent response from Researcher was approved.*

I am pleased to advise that, as submitted to date, the project may proceed in line with standard on-going ethics clearance conditions here outlined.

- All human research activity undertaken under Swinburne auspices must conform to Swinburne and external regulatory standards, including the National Statement on Ethical Conduct in Human Research and with respect to secure data use, retention and disposal.

- The named Swinburne Chief Investigator/Supervisor remains responsible for any personnel appointed to or associated with the project being made aware of ethics clearance conditions, including research and consent procedures or instruments approved. Any change in chief investigator/supervisor requires timely notification and SUHREC endorsement.

- The above project has been approved as submitted for ethical review by or on behalf of SUHREC. Amendments to approved procedures or instruments ordinarily require prior ethical appraisal/ clearance. SUHREC must be notified immediately or as soon as possible thereafter of (a) any serious or unexpected adverse effects on participants and any redress measures; (b) proposed changes in protocols; and (c) unforeseen events which might affect continued ethical acceptability of the project.

- At a minimum, an annual report on the progress of the project is required as well as at the conclusion (or abandonment) of the project.

- A duly authorised external or internal audit of the project may be undertaken at any time.

Please contact the Research Ethics Office if you have any queries about on-going ethics clearance or you need a signed ethics clearance certificate, citing the SUHREC project number. A copy of this clearance email should be retained as part of project record-keeping.

Best wishes for the project.

Yours sincerely
Kaye Goldenberg
Secretary, SHESC2

Kaye Goldenberg
Administrative Officer (Research Ethics)
Swinburne Research (H68)
Swinburne University of Technology
P O Box 218
HAWTHORN VIC 3122
Tel +61 3 9214 8468

Appendix 3

User study tasks for first group

Task 1

This task is designed to study the user friendliness of CONVErT in generating visualisations from input data. You are required to generate a bar chart visualisation for provided input data “salesfile.xml” from available drawings. The drawings (Chart area and Bar) are available in the predefined shapes section of the tool.



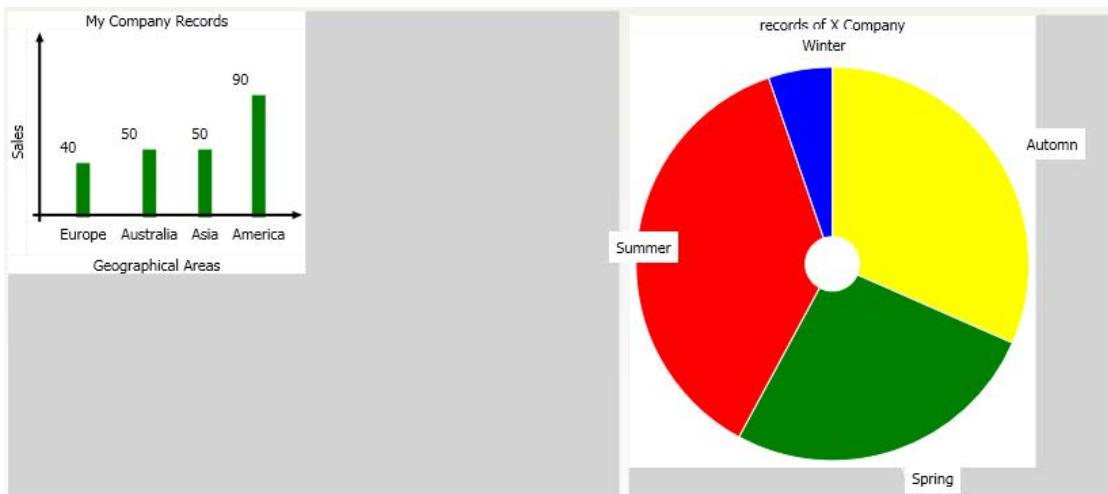
Task 2

This task is designed to check CONVErT in modification of visualisations. You are required to modify the bar chart visualisation by providing a new bar element for provided input data “salesfile2.xml”. We are planning to use provided functions in this task. The required functions can be found in the “Mapping Functions” section.

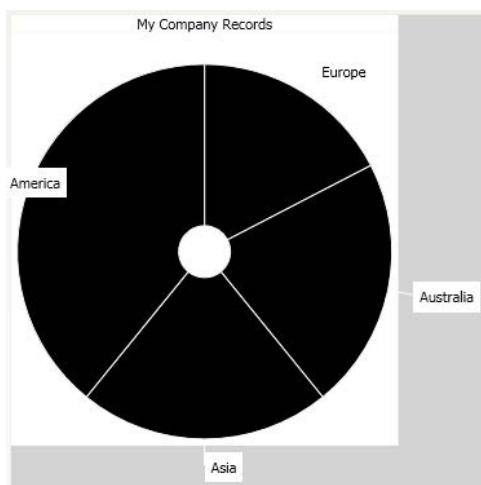


Task 3

This task is designed to study CONVERT in generation of mappings. You are required to create a mapping from a bar chart visualisation to a pie chart visualisation (see figure below). You will be using the bar chart you created in Task 2 and the pie chart provided ("Piechart.xml").



The final visualisation should look like following figure.

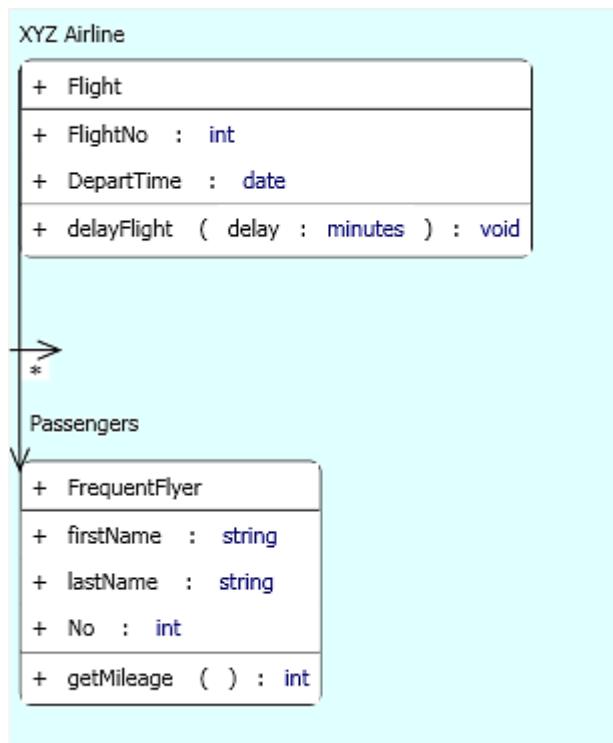


Appendix 4

User study tasks for second group

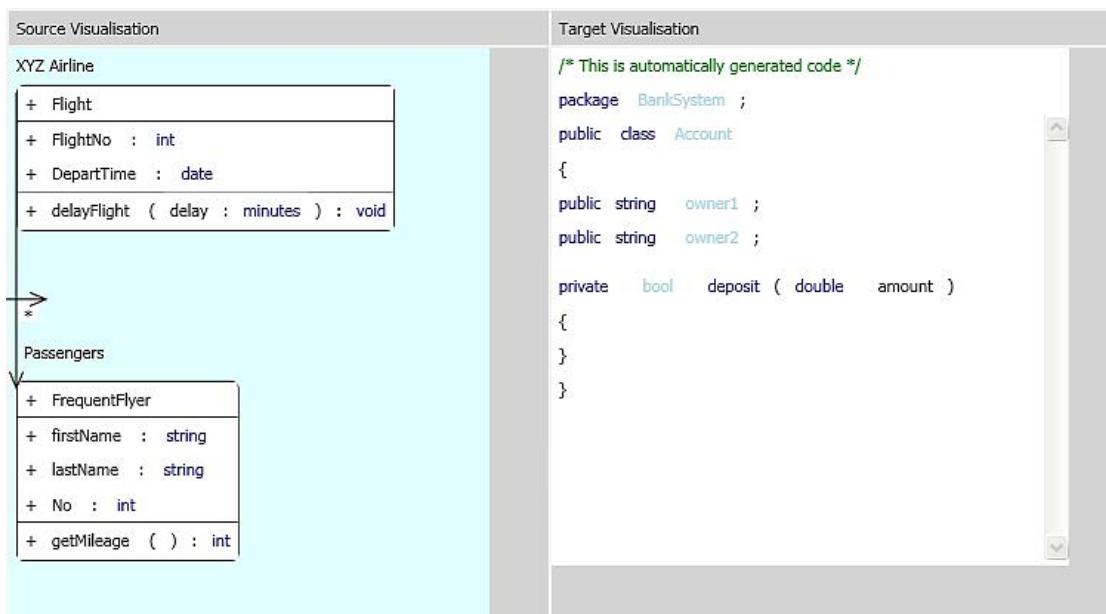
Task 1

This task is designed to study the user friendliness of CONVErT in generating visualisations from input data. You are required to generate a UML class diagram visualisation for provided input data “umldata.xml” from available drawings. The drawings (Diagram, Class, Attribute, Operation, Association and function parameters) are available in the predefined shapes section of the tool.



Task 2

This task is designed to study CONVeRt in generation of mappings. You are required to create a mapping from a UML class diagram visualisation to Java code visualisation (see figure below). You will be using the UML class diagram you just created in task 1 and the Java example (“Javavisual.xml”).



The final visualisation should look like following figure.

```
/* This is automatically generated code */

package XYZ_Airline;

public class Flight
{
    public int FlightNo;
    public date DepartTime;
    public FrequentFlyer [] Passengers;

    public void delayFlight ( minutes delay )
    {
    }
}

public class FrequentFlyer
{
    public string firstName;
}
```

Appendix 5

Survey Questionnaire

Research Supervisor

Prof. John Grundy

Professor in Software Engineering and Head of Academic Group, Computer Science & Software Engineering

FICT, Swinburne University of Technology

Phone: +61 3 9214 8731

Email: jgrundy@swin.edu.au

PhD Student Researcher

Mr. Iman Avazpour

PhD Student

FICT, Swinburne University of Technology

Phone: +61 3 9214 8786

Email: iavazpour@swin.edu.au

Dear Participant,

This questionnaire aims to capture your experience with CONcrete Visual assistEd Transformation framework (CONVErT). CONVErT is proposed as part of PhD research by Iman Avazpour under the supervision of Professor John Grundy. The project 2013/010 is approved by Swinburne University Human Research Ethics Sub Committee (SHESC3) and conducted under the privacy policy followed by Swinburne University of Technology.

This questionnaire has three sections. Section one consists of questions regarding your experience with visualisation and is designed to capture how efficient our visualisation approach was for users. Section two evaluates your experience with CONVErT for generating transformations between two visualisations. Section three is design to capture the effectiveness of CONVErT's recommender system. And finally section four contains some demographic question about you. Please read the questions in each section carefully and put a cross in the box that is closer to your feeling.

Please Note: Individual responses will not be released or shared and individuals will not be identified. The information provided will be kept secure and will be accessible to the researchers only. Aggregate results from analysis of survey responses will be published in peer-reviewed academic journals and conferences.

If you have any complaints or question regarding the approval of the project you can contact Research Ethics Officer, Swinburne Research (H68), Swinburne University of Technology, P O Box 218, HAWTHORN VIC 3122. Tel (03) 9214 5218 or +61 3 9214 5218 or resethics@swin.edu.au

Part 1. Visualisation

Usefulness

1. It is useful to have a drag and drop approach for visualisation.

Strongly Disagree Strongly Agree

Comments _____

2. Visualisations help me better understand complex data.

Strongly Disagree Strongly Agree

Comments _____

3. It is useful to be able to visualise data tailored to users.

Strongly Disagree Strongly Agree

Comments _____

Cognitive dimensions

4. It is easy to see various parts of the tool such as drawings, functions, etc.

Strongly Disagree Strongly Agree

Comments _____

5. It is easy to make changes to visualisations.

Strongly Disagree Strongly Agree

Comments _____

6. Some things do require a lot of thought.

Strongly Disagree Strongly Agree

Comments _____

7. It is easy to make errors or mistakes.

Strongly Disagree Strongly Agree

Comments _____

8. Couple of drawings were provided on the right side of the tool panel to assist you with your task. Did you find they were helpful?

Strongly Disagree Strongly Agree

Comments _____

9. It was easy to recognise which element on the left hand side was related to which visualisation element on the right hand side.

Strongly Disagree Strongly Agree

Comments _____

10. Provided Logs of your previous actions was useful.

Strongly Disagree Strongly Agree

Comments _____

11. I can work in any order I like when working with the tool.

Strongly Disagree Strongly Agree

Comments _____

Ease of use

12. I found it easy to visualise the given data as a Barchart/Class diagram.

Strongly Disagree Strongly Agree

Comments _____

13. I found it easy to modify the visualisations.

Strongly Disagree Strongly Agree

Comments _____

14. In general I found the tool to be easy to use for visualisation activities.

Strongly Disagree Strongly Agree

Comments _____

Ease of learning

15. I learned to use the tool quickly.

Strongly Disagree Strongly Agree

Comments _____

16. I would like to have received further instruction to be able to understand the procedure and perform the task.

Strongly Disagree Strongly Agree

Comments _____

17. I had to redo some parts to be able to understand the procedure.

Strongly Disagree Strongly Agree

Comments _____

18. I easily remember how to use the tool.

Strongly Disagree Strongly Agree

Comments _____

Satisfaction

19. It is likely that I use the tool for visualisation in my future projects.

Strongly Disagree Strongly Agree

Comments _____

20. I had fun using the tool.

Strongly Disagree Strongly Agree

Comments _____

21. I would recommend it to a friend.

Strongly Disagree Strongly Agree

Comments _____

Please put any comments regarding visualisation procedure in the box bellow

Part 2. Transformation

Usefulness

1. The familiar diagrams and visual elements used to show the different views of the data were useful.

Strongly Disagree Strongly Agree

Comments _____

2. Visual diagrams help me better understand the relationships between source and target drawings.

Strongly Disagree Strongly Agree

Comments _____

3. It is useful to specify relationships between different elements in the left hand side and the right hand side visualisations by using the drag and drop of each element.

Strongly Disagree Strongly Agree

Comments _____

Cognitive dimensions

4. It is easy to see various parts of the tool such as drawings, functions, etc.

Strongly Disagree Strongly Agree

Comments _____

5. Some things do require a lot of thought.

Strongly Disagree Strongly Agree

Comments _____

6. It is easy to make errors or mistakes.

Strongly Disagree Strongly Agree

Comments _____

7. It was easy to recognise which visual element on the left hand side was related to which visual element on the right hand side.

Strongly Disagree Strongly Agree

Comments _____

8. Provided Logs of my previous actions was useful.

Strongly Disagree Strongly Agree

Comments _____

9. I can work in any order I like when working with the tool.

Strongly Disagree Strongly Agree

Comments _____

Ease of use

10. I found it easy to specify the relations between left hand side and right hand side visualisations.

Strongly Disagree Strongly Agree

Comments _____

11. The user interface is very consistent.

Strongly Disagree Strongly Agree

Comments _____

12. In general I found the tool to be easy for transformation between visualisations.

Strongly Disagree Strongly Agree

Comments _____

Ease of learning

13. I learned to use the tool quickly.

Strongly Disagree Strongly Agree

Comments _____

14. I would like to have received further instruction to be able to understand the procedure and perform the task.

Strongly Disagree Strongly Agree

Comments _____

15. I had to redo some parts to be able to understand the procedure

Strongly Disagree Strongly Agree

Comments _____

16. I easily remember how to use the tool

Strongly Disagree Strongly Agree

Comments _____

Satisfaction

17. It is likely that I use the tool for transformation in my future projects.

Strongly Disagree Strongly Agree

Comments _____

18. I had fun using the tool.

Strongly Disagree Strongly Agree

Comments _____

19. I would recommend it to a friend.

Strongly Disagree Strongly Agree

Comments _____

Please put further comments regarding transformation procedure in the box bellow

Part 3. Recommendation System

Usefulness

1. It is useful to have recommendations during the process.

Strongly Disagree Strongly Agree

Comments _____

2. Recommendations helped me better understand relations between source and target visualisations.

Strongly Disagree Strongly Agree

Comments _____

3. Recommendations help me discover other possible relations.

Strongly Disagree Strongly Agree

Comments _____

4. Recommendations seemed to offer a good (correct) solution.

Strongly Disagree Strongly Agree

Comments _____

5. I was able to trust the recommendations.

Strongly Disagree Strongly Agree

Comments _____

6. I used recommendations at least once.

Strongly Disagree Strongly Agree

Comments _____

7. I already knew most of the recommendations.

Strongly Disagree Strongly Agree

Comments _____

Presentation

8. I was satisfied with the way recommendations were presented.

Strongly Disagree Strongly Agree

Comments _____

9. When a recommendation said for example “Bar/Name” I was easily able to spot “Name” in source or target visualisations.

Strongly Disagree Strongly Agree

Comments _____

10. I was able to use recommendations.

Strongly Disagree Strongly Agree

Comments _____

Satisfaction

11. It is likely that I use provided recommendation system in future.

Strongly Disagree Strongly Agree

Comments _____

12. I found some recommendation to be surprising in a good way.

Strongly Disagree Strongly Agree

Comments _____

13. I had fun using the recommendations

Strongly Disagree Strongly Agree

Comments _____

Please put any comments regarding recommendation system in the box bellow

Part 4. Demographic Information

Please circle the option that is most applicable to you.

- | | |
|--|---|
| Gender | <input type="radio"/> Male |
| | <input type="radio"/> Female |
| | <input type="radio"/> Prefer not to say |
| Age range | <input type="radio"/> 25-30 |
| | <input type="radio"/> 31-40 |
| | <input type="radio"/> 41-50 |
| | <input type="radio"/> 51-60 |
| | <input type="radio"/> 61+ |
| How familiar are you with
model transformation and
modelling in general? | <input type="radio"/> Very familiar |
| | <input type="radio"/> Somewhat familiar |
| | <input type="radio"/> I had heard about it |
| | <input type="radio"/> Not familiar at all |
| How familiar are you with
data visualisation? | <input type="radio"/> Very familiar |
| | <input type="radio"/> Somewhat familiar |
| | <input type="radio"/> I had heard about it |
| | <input type="radio"/> Not familiar at all |
| What best describes your
area? | <input type="radio"/> Software engineering |
| | <input type="radio"/> Computer Science / IT |
| | <input type="radio"/> Economics |
| | <input type="radio"/> Management |
| | <input type="radio"/> Other _____ |

Appendix 6

Samples of citation formats used for evaluating Suggester

EndNote format

```
<records>
  <record>
    <database name="My Collection.enl" path="My Collection.enl">My Collection.enl</database>
    <ref-type name="Conference Proceedings">3</ref-type>
    <contributors>
      <authors>
        <author>Abramov, Sergei</author>
        <author>Gluck, Robert</author>
      </authors>
      <secondary-authors>
        <author>Mogensen, Torben</author>
        <author>Schmidt, David</author>
        <author>Sudborough, I.</author>
      </secondary-authors>
    </contributors>
    <titles>
      <title>Principles of Inverse Computation and the Universal Resolving Algorithm</title>
      <secondary-title>The Essence of Computation Complexity, Analysis, Transformation</secondary-title>
    </titles>
    <periodical>
      <full-title>The Essence of Computation Complexity, Analysis, Transformation</full-title>
    </periodical>
    <pages>269-295</pages>
    <keywords/>
    <dates>
      <year>2002</year>
    </dates>
    <publisher>Springer Berlin / Heidelberg</publisher>
    <electronic-resource-num>10.1007/3-540-36377-7</electronic-resource-num>
    <urls>
      <pdf-urls>
        <url>internal-pdf://Abramov, Gluck - 2002 - Principles of Inverse Computation and the Universal Resolving Algorithm.pdf</url>
      </pdf-urls>
    </urls>
  </record>
<record>
```

```

<database name="My Collection.enl" path="My Collection.enl">My
Collection.enl</database>
<ref-type name="Journal Article">0</ref-type>
<contributors>
  <authors>
    <author>Adomavicius, G.</author>
    <author>Tuzhilin, a.</author>
  </authors>
</contributors>
<titles>
  <title>Toward the next generation of recommender systems: a survey of the
    state-of-the-art and possible extensions</title>
  <secondary-title>IEEE Transactions on Knowledge and Data
    Engineering</secondary-title>
</titles>
<periodical>
  <full-title>IEEE Transactions on Knowledge and Data Engineering</full-
title>
</periodical>
<pages>734-749</pages>
<volume>17</volume>
<issue>6</issue>
<keywords/>
<dates>
  <year>2005</year>
</dates>
<electronic-resource-num>10.1109/TKDE.2005.99</electronic-resource-num>
<urls>
  <pdf-urls>
    <url>internal-pdf://Adomavicius, Tuzhilin - 2005 - Toward the next
    generation of recommender systems a survey of the state-of-the-art and possible
    extensions.pdf</url>
  </pdf-urls>
  <web-urls>
    <url>http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=14
    23975</url>
  </web-urls>
</urls>
</record>
</records>

```

DocBook format

```
<bibliography>
<biblioentry xreflabel="Abramov2002" id="Abramov2002">
  <authorgroup>
    <author><firstname>Sergei</firstname><surname>Abramov</surname></author>
    <author><firstname>Robert</firstname><surname>Gluck</surname></author>
    <editor><firstname>Torben</firstname><surname>Mogensen</surname></editor>
    <editor><firstname>David</firstname><surname>Schmidt</surname></editor>
    <editor><firstname>I.</firstname><surname>Sudborough</surname></editor>
  </authorgroup>
  <citetitle pubwork="article">Principles of Inverse Computation and the Universal
  Resolving Algorithm</citetitle>
  <publisher>
    <publishername>Springer Berlin / Heidelberg</publishername>
  </publisher>
  <artpgenums>269&#x2013;295</artpgenums>
  <pubdate>2002</pubdate>
</biblioentry>
<biblioentry xreflabel="Adomavicius" id="Adomavicius">
  <authorgroup>
    <author><firstname>Gediminas</firstname><surname>Adomavicius</surname>
  </author>
  </authorgroup>
  <citetitle pubwork="article">Towards More Confident Recommendations : Improving
  Recommender Systems Using Filtering Approach Based on Rating Variance
  Department of Computer Science and Engineering &#44; University of
  Minnesota</citetitle>
  <artpgenums>1&#x2013;6</artpgenums>
</biblioentry>
```

List of publications

Iman Avazpour, Teerat Pitakrat, Lars Grunske, John C. Grundy, Evaluating Recommendation Systems: Quantitative Measures and Features to Consider, Recommendation Systems in Software Engineering, Springer, 2014 (in Press).

Iman Avazpour, John C. Grundy, Lars Grunske, “Tool Support for Automatic Model Transformation Specification Using Concrete Visualisations”, 2013 IEEE/ACM International Conference on Automated Software Engineering, Palo Alto, CA, USA, 11-15 Nov 2013, pp.718-721, IEEE CPS.

Iman Avazpour, John Grundy, “Using Concrete Visual Notations as First Class Citizens for Model Transformation Specification”, 2013 IEEE International Symposium on Visual Languages and Human-Centric Computing, San Jose, CA, USA, Sept 15-19 2013, pp.87-90, IEEE CPS.

Iman Avazpour, John Grundy, “CONVErT: A Framework for Complex Model Visualisation and Transformation”, 2012 IEEE International Symposium on Visual Languages and Human-Centric Computing, Innsbruck, Austria, Sept 30-Oct 4 2012, pp.237-238, IEEE CPS.

Iman Avazpour, “Towards User-Centric Concrete Model Transformation”, 2012 IEEE International Symposium on Visual Languages and Human-Centric Computing (Graduate Consortium), Innsbruck, Austria, Sept 30-Oct 4 2012, pp.215-216, IEEE CPS.