

Critic Specification for Domain-Specific Visual Language Tools

Norhayati Mohd Ali

**A thesis submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy in Computer Science,
The University of Auckland, 2010**

Abstract

In recent years we have observed the extensive evolution of tools and techniques that work with the user to achieve a range of computer-mediated tasks. One of these support techniques is the use of *critics*. Critics have evolved over the last few years as specific tool features to support users in computer-mediated tasks by providing guidelines or suggestions for improvement to designs, code and other digital artefacts. Some critics may also facilitate semi-automatically improving a design for the tool user. Although critics have been used widely in very diverse domains, such as education, programming and product design tools, critic authoring continues to be a challenge. In addition critic approaches have not been applied within meta-modelling tools that implement domain-specific visual language (DSVL) tools.

The main research question in this research project is “Can critic specification and implementation for domain specific visual languages be made accessible to tool end-users?” Hence, the aim of this research is to design and develop a prototype for a critic specification tool that allows the end user tool developers to readily express and construct critics for a DSVL tool. The research involved several steps to attain the research aim. The initial phase of this research has produced a taxonomy of computer-supported critic approaches and led to the identification of key requirements of a critic specification tool. The intermediate phase involved the incremental development of prototypes demonstrating a proof of concept for a critic specification tool. A notational representation and a template-based approach were developed for the final prototype of the critic specification tool and demonstrated via three different domains of DSVL exemplar tools. The final phase of this research addresses the evaluation of the critic specification approach via an end user evaluation which took into account usability aspects and the Cognitive Dimensions framework.

This research has contributed to the development of a critic specification approach for DSVL tools based on a notational representation and a critic authoring template-based approach to support tool end users in specifying critics.

Dedication

I dedicate this thesis to

My late father, who made this entire journey possible

My husband Akramin, son Irfan, and daughters Insyirah and Iffah

My mother, brothers and sisters

“Your love and support are the greatest gift of all”

Acknowledgements

In the Name of Allah, the Most Benificent, the Most Merciful

All the praises and thanks be to Allah, with Whose blessings are completed the righteous deeds. Peace, Blessings and Graces of Allah be upon our Prophet Muhamad (pbuh), his family and his companions.

I would like to acknowledge some of the many persons and groups who supported me for the past four years of my PhD study. Without them, this thesis would never have been possible. Please accept my apologies if I fail to mention your contribution; I can assure you that it is only an omission in writing and not in feeling.

Firstly, a special thank to my late-father who has taught me the value of a good education. I also would like to thank my mother, brothers and sisters for their love, their kind words of encouragement and advice. I would also like to thank my in-laws family for their unconditional love and support during these challenging years.

I would like to thank both of my main supervisors, Prof. John Hosking and Prof. John Grundy (Swinburne University of Technology) for their constant support and insightful guidance over the past four years. Both of them are tremendous supervisors, and I feel incredibly lucky to have been their student. I thank both of them, who generously offered their precious time in helping me and cheering me when I got stuck. Working with John & John has indeed been a great pleasure. Their guidance, patience and support during our discussions have taught me how to enjoy researching though it is a stressful process. Without their numerous proof reading of my rough drafts and invaluable suggestions for rewriting, this thesis work would be far worse. I owe limitless gratitude to them. Thank you for making my graduate research experience one of the most rewarding and defining moments in my life. To John & John -Thanks for everything.

I would also like to acknowledge the generous financial support and assistance provided for me by the Ministry of Higher Education (Malaysia), Universiti Putra Malaysia, and Software Process and Product Improvement Funding (University of Auckland). I am very grateful to the Postgraduate Research Student Support Account (University of Auckland), CS Graduate Student Travel fund, U.S National Science Foundation grant and Build IT travel fund for supporting my conferences travel.

I want to thank my colleagues in John's research group, Karen, Jun, Richard, Rainbow, Chris, Masila, Su, Farid, Emily, James, Brian and Rick for listening and discussing my research progress. A special thank to Jun who helped me a lot in the coding task and to Karen who proof read my draft and always have a time when I need her to discuss about my research work. I am very greatful to the group members for participating in the tool's evaluation.

I would like to thank the Computer Science Department staff for helping me when I need their assistance. I would like to thank Robyn, Sithra, Anita, Heather, Keith, Pan, Lei and many others. Thank you for everything.

I would also like to thank Dr. Susan and her colleagues from the Student Learning Centre for handling the doctoral skills programme which I found it very helpful for my research study. I am very grateful to the Doctoral EAL support group that helps me with my thesis writing.

In general, I would like to thank all my friends around the world who endlessly encouraged and supported me throughout these years. A huge thanks to Sarah and Eyad for all the helps they offered to my family. Thank you for everything. A special thanks to Muhammad and Mehwish for helping me with the thesis formatting work. I would also like to thank Zakiah, Ema, Zaidah, Dila, and all MAPSA members for helping and supporting me in whatever occasions. Thank you all for the friendship that made my life enjoyable.

Last, but certainly not least, I want to thank my lovely and wonderful husband, Zainul Akramin for his love, support, encouragement, understanding and patience throughout these years. I will never be able to thank him enough for that. I will not even be able to express my gratitude to and love for him. I would also like to thank my son, Irfan, my daughters, Insyirah and Iffah for their love and patience. They are indispensable for me to accomplish this work. I could never have enough words to thank my family for what they did for me.

Table of Contents

Abstract.....	i
Dedication.....	ii
Acknowledgement.....	iii
List of Figures.....	xi
List of Tables.....	xv
Chapter 1. Introduction.....	1
1.1 Research Background	1
1.2 Research Motivation	3
1.3 Research Questions.....	4
1.4 Research Objectives.....	5
1.5 Research Methodology	6
1.6 Research Contributions.....	6
1.7 Thesis Organization.....	8
Chapter 2. Related Research.....	12
2.1 Introduction.....	12
2.2 Critic as a Supporting Tool.....	12
2.2.1 Critics in Information Systems	13
2.2.2 Critics in Software Engineering.....	16
2.2.3 Critics in Education Environment	19
2.2.4 Critics in Recommender Systems	21
2.2.5 Benefits from Critics Application.....	23
2.3 Constraint Specification in a Meta-Modelling Tool.....	27
2.3.1 MetaEdit+	30
2.3.2 Pounamu	32
2.3.3 Marama	33

2.3.4 DECS	35
2.4 Discussion and Conclusion.....	36
Chapter 3. Research Methodology.....	38
3.1 Introduction.....	38
3.2 Methodology.....	39
3.2.1 Literature Review of Critic Tools.....	39
3.2.2 Identify a Set of Requirements for Our Critic Specification Tool.....	39
3.2.3 Develop Prototype to Explore Issues in Designing Critic Specification Tool	40
3.2.4 Identify a Set of Building Blocks Needed for a Critic Specification Tool ...	42
3.2.5 Proof of concept for the critic specification approach.....	42
3.2.6 Perform user evaluation of our critic specification approach	42
3.2.7 Draw conclusions from our survey, design, prototyping and evaluation work	43
3.3 Conclusions.....	43
Chapter 4. A Critic Taxonomy.....	44
4.1 What is Taxonomy?.....	44
4.2 Critic Definitions and Examples.....	45
4.3 A Critic Taxonomy	47
4.3.1 Critic Domain	49
4.3.2 Critiquing Approach	50
4.3.3 Modes of Critic Feedback.....	53
4.3.4 Critic's Rule Authoring	54
4.3.5 Critic Realisation Approach	56
4.3.6 Critic Dimension.....	61
4.3.7 Types of Critic Feedback.....	63
4.3.8 Critic Types	66

4.4 Applying the Taxonomy	67
4.4.1 ArgoUML (Robbins and Redmiles, 2000)	67
4.4.2 ABCDE-Critic (de Souza et al., 2000)	69
4.4.3 IDEA (Bergenti & Poggi, 2000)	71
4.4.4 RevJava (Florijn, 2002)	73
4.4.5 DAISY (de Souza et al., 2003)	75
4.4.6 Java Critiquer (Qiu and Riesbeck, 2003).....	77
4.4.7 Design Evaluator (Oh et al., 2004)	79
4.4.8 ClassCompass (Coelho & Murphy, 2007).....	80
4.4.9 FFDC (Oh et al., 2009)	83
4.4.10 HeRA (Knauss et al., 2009)	85
4.4.11 Summaries of Comparison	86
4.5 Conclusion	95
Chapter 5. A Visual and Template-based Approach for Critic Specification.....	97
5.1 Introduction.....	97
5.2 Visual Specification Approach	97
5.2.1 Visual Notations Used by the Critic Specification Editor	101
5.3 Template-Based Approach	104
5.3.1 Introduction to Business Rule Templates	106
5.3.2 Critic Authoring Templates	109
5.4 Visual and Template-based Critic Specification for DSVL tools	112
5.5 Analysis of Critic Specification Tool using Physics of Notations	113
5.6 Conclusion	122
Chapter 6. Initial Prototype for Critic Specification.....	123
6.1 Introduction.....	123
6.2 Initial Prototype: Specifying Critic in a Marama Metamodel Definer views	
124	

6.2.1 Background and Motivation	124
6.2.2 Approach.....	128
6.2.3 Initial Critic Authoring Template	130
6.2.4 Implementation	131
6.2.5 Example Usage	134
6.2.6 Preliminary Results for the Initial Prototype	139
6.4 Conclusion	141
Chapter 7. Final Prototype for Critic Specification.....	142
7.1 Background and Motivation	142
7.2 Final Prototype: the Marama Critic Definer Editor	144
7.2.1 Approach.....	144
7.2.2 Visual Critic Definer Editor.....	145
7.2.2.1 CriticShape with Extended Critic Authoring Templates	147
7.2.2.2 Critic Feedback Specification	149
7.2.2.3 Critic dependency, Operator shape, Operator and OperatorCriticFeedback connectors	152
7.2.2.4 Simple and Complex Critics.....	153
7.2.2.5 Critic Template Editor	156
7.2.2.6 Critic Authoring Guideline	158
7.2.2.6 Critic and Feedback Repository.....	162
7.3 Summary of the Implementation	162
7.4 Conclusion	164
Chapter 8. Case Studies.....	165
8.1 Introduction.....	165
8.2 Case Study I: A Visual Care Plan Modelling Language (VCPML) Tool	166
8.2.1 Case Study Description.....	166
8.2.2 Example Usage	168

8.3 Case Study II: A Simplified Marama EML Tool	174
8.3.1 Case Study Description.....	175
8.3.2 Example Usage	177
8.4 Case Study III: MaramaUML Tool	181
8.4.1 Case Study Description.....	181
8.4.2 Example Usage	184
8.5 Discussions and Conclusions.....	190
Chapter 9. Evaluation.....	193
9.1 Introduction.....	193
9.2 Cognitive Dimensions of Notations framework (CDs)	195
9.3 The Four Criteria to Evaluate Usability.....	196
9.4 Design of the Survey	198
9.4.1 The Observation Design	198
9.4.2 The Questionnaire Design	199
9.4.3 Survey Method.....	200
9.5 Survey Result and Analysis	201
9.5.1 Analysis of Task List and Observation.....	201
9.5.2 Analysis of Questionnaire Responses.....	205
9.7 Conclusion	212
Chapter 10. Conclusions and Future Work.....	214
10.1 Research Summary	214
10.2 Research Contributions.....	219
10.3 Future Work.....	220
10.4 Summary.....	221
Appendix A – Participation Information Sheet (Head of Department).....	222
Appendix B – Participation Information Sheet (Student).....	224
Appendix C – Consent Form (Head of Department).....	226

Appendix D – Consent Form (Student).....	227
Appendix E - Survey : Evaluation of Template-based Critic Authoring for Domain Specific Visual Language Tools.....	229
References.....	236

List of Figures

Figure 2.1: Screen shot of an expert critiquing system	14
Figure 2.2: Example of DSS that employed critics	15
Figure 2.4; Example of recommender system	22
Figure 2.5 Example of group recommender system	23
Figure 2.6: Examples of constraints expressed in OCL	29
Figure 2.7: Constraint expression using OCL expression	30
Figure 2.8: MetaEdit+ architecture	31
Figure 2.9: Constraints definer editor and Graph constraints tool	31
Figure 2.10 Structure of Pounamu specification	32
Figure 2.11: Example of code-based event handler for model constraints	33
Figure 2.12: Constraint specification via MaramaTatau using OCL formula	34
Figure 2.13 Architecture of DECS	36
Figure 3.1: Prototype development for critic specification tool	41
Figure 4.1: Our critic taxonomy.	48
Figure 4.2: Critic rule using pattern-matching approach	58
Figure 4.3: Rules for architectural floor plans using predicate style	59
Figure 4.4: Critics written in OCL expressions	60
Figure 4.5: The ArgoUML user interface.	67
Figure 4.6: The mapping of the ArgoUML tool to the critic taxonomy	69
Figure 4.7: The mapping of the ABCDE-Critic tool to the critic taxonomy	71
Figure 4.8: The mapping of the IDEA tool to the critic taxonomy	73
Figure 4.9: RevJava Critics	74
Figure 4.10: The mapping of the RevJava tool to the critic taxonomy.	75
Figure 4.11: The mapping of the DAISY tool to the critic taxonomy.....	76
Figure 4.12: Java Critiquer interface	78
Figure 4.13: The mapping of the Java Critiquer tool to the critic taxonomy.	78
Figure 4.14: The mapping of the Design Evaluator tool to the critic taxonomy.....	80
Figure 4.15: ClassCompass user interface	82
Figure 4.16: The mapping of the ClassCompass tool to the critic taxonomy.....	82
Figure 4.17: The mapping of the FFDC tool to the critic taxonomy.....	84

Figure 4.18: The mapping of the HeRA tool to the critic taxonomy.....	86
Figure 5.1: Visual notations of the visual critic specification editor:.....	102
Figure 5.2: Meta-model defined for a critic specification tool.....	103
Figure 5.3: Icons for the critic specification editor.....	104
Figure 5.4: A form-based interface to represent the critic authoring templates	112
Figure 5.5: The mapping between (a) metamodel of the visual critic definer and (b) graphical symbols.	115
Figure 5.6: Element types in the visual critic specification editor	116
Figure 5.7: Critic specification diagram	118
Figure 5.8: Integration between critic definer view and critic construction editor, and integration between critic construction editor and meta elements.....	119
Figure 5.9: Textual encoding	120
Figure 6.1: UML class diagramming tool metamodel.....	124
Figure 6.2: Simple critic (same named classes) violation in MaramaTatau	125
Figure 6.3: Simple critic (class with no name) violation in MaramaTatau	126
Figure 6.4: Critic development approach	128
Figure 6.5: Critics specified in the meta-model definer editor	129
Figure 6.6: New function added in the Marama meta-model editor.....	131
Figure 6.7: CriticShape (orange colour) linked with a critic authoring template....	132
Figure 6.8: Critics store in critictypes folder.....	133
Figure 6.9: Architecture of critic processing	134
Figure 6.10: MaramaMTE metamodel definer view	134
Figure 6.11: Visual CriticShape function associate with the critic authoring templates.....	136
Figure 6.12: Critics for MaramaMTE are stored in critictypes folder.....	137
Figure 6.13: Critic statement: remote object must have a unique name. Attribute Constraint template: <entity>must have may have [unique] <attributeTerm>.....	138
Figure 6.14: Critic statement: remote object must have many services. Relationship constraint template: <entity1> must have may have [<cardinality>]<entity2>.....	138
Figure 7.2: A new specification tool, Marama Critic Definer	145
Figure 7.3: A visual critic definer editor: (a) Initial notation, (b) Improved notation	146

Figure 7.4: <i>CriticShape</i> (top) associated with <i>Critic Construction View</i> interface (bottom)	149
Figure 7.5: CriticFeedbackShape associated with Critic feedback view interface..	150
Figure 7.6: An example of passive critic	151
Figure 7.7: A CriticFeedbackConn connector links the critic and feedback.	152
Figure 7.8: A CriticDependencyLink connects two critics	152
Figure 7.9: Examples of unit/ simple critics	153
Figure 7.10: Critics specified in the critic definer editor (bottom) based on the meta-model of SimplifiedMaramaEML tool defined in the meta-model editor (top).....	155
Figure 7.11: A critic specified using an action assertion template.	156
Figure 7.12: A new critic template created in the Critic Template editor.	157
Figure 7.13: A guideline for the critic authoring template style.....	158
Figure 7.14: Critic (critictypes folder) and feedback (feedbacktypes) repository browser.	162
Figure 7.15: Architecture view of the Marama meta-tools and the extension of Marama Critic Definer view	163
Figure 8.1: The VCPML meta model	168
Figure 8.2: An example of the VCPML model	168
Figure 8.3: A CriticFeedbackConn connector links the critic and feedback.	170
Figure 8.4: A uniqueness name critic via the attribute constraint template.....	170
Figure 8.5: A critic on cardinality constraint using the relationship constraint template.....	171
Figure 8.6: Critic feedback for the uniqueness name critic	172
Figure 8.7: A critique message is displayed when a uniqueness name critic is violated	173
Figure 8.8: A critic feedback with a brief explanation and suggestion	173
Figure 8.9: The fix action for the uniqueness name critic.	173
Figure 8.10: Meta model for the simplified MaramaEML	175
Figure 8.11: University enrollment service model using a simplified MaramaEML (modified from (Li, et al., 2007b)).....	177
Figure 8.12: Critics specified in the critic definer editor based on the meta-model of SimplifiedMaramaEML tool.	178

Figure 8.13: A critic specified using an action assertion template.....	179
Figure 8.14: Action assertion critic execution after the trigger event occurs: a critique is displayed to warn the user	179
Figure 8.15: Feedback of a complex critic using the logical operator OR (top) and fix action for this critic (bottom).	180
Figure 8.15: Metamodel for MaramaUML tool.	182
Figure 8.16: Class diagram example (left) and Collaboration diagram example (right)	182
Figure 8.17: Graphical representation of a consistency rule between collaboration diagram (bottom) and class diagram (top).....	185
Figure 8.18: A new critic template created in the Critic Template editor.	187
Figure 8.19: New critic authoring template: <entity1><attributeTerm><relationalOperator><entity2><attributeTerm1> (bottom critic).....	187
Figure 8.20 A critique is displayed when a consistency critic rule is violated.....	188
Figure 8.21: A critic feedback displays a brief explanation and suggestion.	188
Figure 8.22: A fix action to resolve the consistency critic rule	189
Figure 9.2: Usability responses.....	207
Figure 9.3: CD questionnaire responses.	208

List of Tables

Table 4.1: Critic definitions.....	46
Table 4.2: Examples of critic tools and their application domain.	46
Table 4.3: Critics applied to various domains.	49
Table 4.5: Critic dimensions.....	61
Table 4.6: Critic types.....	66
Table 5.1: Definition of constraint, action assertion and derivation.....	107
Table 5.2: Business rule templates	108
Table 5.3. Critic Authoring Templates-constraint and action assertion templates..	111
Table 5.4: Association of critic template properties with the tool meta-model.....	111
Table 5.5: Association of metamodel elements and graphical symbol	114
Table 6.1. Critic statement and OCL expression.....	125
Table 6.2: Attribute and relationship constraint templates.....	130
Table 6.3: Association of tool meta-model with the critic phrase type	130
Table 6.4: Lists of critic statements and critic authoring templates for MaramaMTE.	
.....	135
Table 7.1: Critic Authoring Template	148
Table 8.1: Attribute and relationship constraint templates.....	169
Table 8.2: Examples of critics and feedbacks for VCPML tool.....	169
Table 8.3: Basic rules of EML tree structure (adopted from (Li, 2010))	176
Table 8.4: Action assertion template	178
Table 8.5. Attribute constraint template	180
Table 9.1: The meaning of each dimensions (Blackwell, et al., 2001).....	196
Table 9.2: Section 1- Background information	205
Table 9.3: Usability responses	207
Table 9.4: Cognitive dimension responses	208
Table 9.5: Participants' Comment	212
Table A: Critic Domain.....	87
Table B: Critic approach.....	88
Table C: Modes of critic feedback.....	89

Table D: Critic rules authoring.....	90
Table E: Critic's realisation approach.....	91
Table F: Critic dimensions.....	92
Table G: Types of critic feedback.....	93
Table H: Types of critics.....	94

Chapter 1

Introduction

This chapter presents an overview of the research in this thesis. It describes the background of the research area and introduces the motivation for this research. The research questions for this research are summarised as well as the research objectives. Our methodology to perform this research is outlined, followed by our expected research contributions. Finally we end this chapter with the outline of our thesis structure.

1.1 Research Background

In recent years we have observed the extensive evolution of tools and techniques that work with the user to achieve a range of computer-mediated tasks. One of these support techniques is the use of *critics*. The term “critic” was initially used by Miller (1986) to describe a software program that critiques human-generated solutions. Critics have evolved in the last several years as specific tool features to support users in computer-mediated tasks by providing guidelines or suggestions for improvement to designs, code and other digital artefacts.

The concept of a critic is one that has been adopted in various domains, including medical systems (Gertner & Webber, 1998; Miller, 1986), programming (Fischer, 1987; Florijn, 2002), design (Fischer, Lemke, & Mastaglio, 1991; Oh, Do, & Gross, 2004), education (Coelho & Murphy, 2007; Qiu & Riesbeck, 2004), expert systems (Hagglund, 1993; Silverman, 1992), and decision support systems (Gertner & Webber, 1998; Irandoost, 2006). Research work and efforts from (Fischer, 1987, 1989; Fischer, Lemke, & Mastaglio, 1991; Fischer, Lemke, Mastaglio, & Morsch, 1991; Fischer & Mastaglio, 1990; Miller, 1986; Silverman, 1992; Silverman & Mehzer, 1992) and others have created a wider audience on the use of a critic-based approach. Furthermore, many studies have found evidence that critiquing tools are an efficient feedback-providing mechanism. These tools offer several benefits

including a proactive design improvement, early error detection, and heuristic-based guidance and context-sensitive feedback.

As a simple example consider a software designer manipulating a design artefact in an editing tool. The tool's critics analyze the design artefact as it changes and reveal to the designer some potential problems/errors with the design artefacts e.g. wrong naming convention, over-complex design relationships, and potential misuse of design domain concepts. The critic tool will offer feedback, or "critique" the design, usually proactively as the design evolves. The tool may also suggest alternative design decisions to the designer to resolve potential problems. The interaction between designer and critic tool is iterative until the designer is satisfied with the design artefacts. Typically critic feedback is kept "unobtrusive" to the designer so as not to overly interfere with the design process.

One of the most significant examples of a critic tool in the software engineering domain is ArgoUML (Robbins & Redmiles, 2000) an open source Unified Modeling Language (UML) CASE tool that supports the editing of UML notation diagrams. Its critics offer suggestions to designers when a software architecture diagram violates various UML rules (Robbins & Redmiles, 2000). The LISP-Critic (Fischer, 1987), Argo (Robbins & Redmiles, 1998), ABCDE-Critic (Bergenti & Poggi, 2000; de Souza, Jr., & Goncalves, 2000), IDEA (Bergenti & Poggi, 2000) and RevJava (Florijn, 2002), are further examples of critic-based tools in the software design domain. These tools were developed for the domains of LISP programming, software architecture, object-oriented analysis and design, design patterns and Java object-oriented software respectively. Oh et al. (Oh, Gross, & Do, 2008) point out that most rules for critic tools are written in advance and that their customisation is not easy.

Extending the use of critics into meta-tool environments that implement domain-specific visual language (DSVL) tools and targeting to support end user tool developers makes it possible to improve critic specification in DSVL tools.

1.2 Research Motivation

While many studies have reported that critic tools provide an efficient mechanism for feedback, critic authoring continues to be a challenge i.e. allowing end user tool developers to customise critic rules. There are various approaches (e.g. rule-based, knowledge-based, code and predicate logic) that can be applied for specifying critics, however these approaches are mostly used by skilled developers. A few of the critic tools (e.g. ArgoUML (Robbins & Redmiles, 2000), ABCDE-Critic (de Souza, et al., 2000) and IDEA (Bergenti & Poggi, 2000)) allow for critic customisation but the process of authoring or customising the critics is not easy. The users have to understand both the tool domain and the critic approach used before designing and realising critics.

In addition, the use of the critic concept had not to date been applied within meta-modelling tools that implement DSVL tools. The application of a critic approach is mostly discussed in application domains as stated in above section. Meta-modelling-based DSVL specification tools often employ a constraint definition/specification approach (e.g. MetaEdit+ (Kelly, Lyytinen, & Rossi, 1996), Pounamu (Zhu et al., 2007), and Marama (Grundy, Hosking, Huh, & Li, 2008)). The process of specifying constraints for meta-modelling tools is more complex as it requires good knowledge in programming skills, it uses formal approach and it involves deep cognitive load. This makes it hard for non-skilled users to understand and use the constraint approach.

Inspired by the existing critic tools work, we have made an attempt to apply similar ideas to our meta-modelling tools i.e. Marama (Grundy, et al., 2008). Marama is a meta-tool implemented as set of Eclipse-plugins and includes meta-tools as well as modelling tools (Grundy, et al., 2008). Our meta-tools are used to generate complex visual modelling tools, and these modelling tools could benefit from the addition of various critics. Thus, we wanted to extend our Marama meta-tools by embedding a critic specification component. Furthermore, we wanted to assist end-user tool developers to specify and generate critics efficiently and easily for DSVL tools.

The overall motivation of our research is to be able to provide a critic specification approach that is accessible to end user tool developers for specifying critics for DSVL tools. The focus of this research has led to the design and development of a notational representation and a critic authoring template-based approach for critic specification approach for DSVL tools.

1.3 Research Questions

The main research question in this research in relation to our research motivation can be framed as:

“Can critic specification and implementation for domain specific visual languages be made accessible to end-user tool developers?”

To be able to tackle this question, we divide it into smaller research questions that enable us to identify possible solutions:

- *Can a notation for critic specification be designed that is accessible?* This question aims to address the main topic of our research, so that by reviewing existing critic approaches, it is possible to understand key critic elements and how these elements can be supported in designing a notation for our critic specification tool. In addition, the designed notation for critic specification should be accessible to end-user tool developers. To answer this question, we reviewed related research on critics, adapted business rule templates and used a visual notation-based approach. This question is addressed in Chapter FOUR (Critic Taxonomy) and Chapter FIVE (A Visual and Template-Based Approach for Critic Specification).
- *Can such a notation be realised as a tool?* To answer this question, we developed a prototype critic specification tool and used an iterative-incremental approach to allow improvement in the prototype. This question is addressed in Chapter THREE (Research Methodology), Chapter SIX (Initial Prototype for Critic Specification Tool) and Chapter SEVEN (Final Prototype for Critic Specification Tool).
- *How can such a tool be integrated with existing tools for domain-specific visual language design and implementation?* This question is to be answered

through a proof-of-concept system that identifies technical dependencies among the tool components. This question is addressed in Chapter SIX (Initial Prototype for Critic Specification Tool), Chapter SEVEN (Final Prototype for Critic Specification Tool) and Chapter EIGHT (Case Studies).

- *How can such an integrated tool set be evaluated?* To answer this question, we designed a survey to perform an end-user evaluation for the critic specification tool. We gained ethics approval from the University of Auckland Human Participants Ethics Committee before conducting an end-user evaluation for the developed critic specification tool with several target end users. This question is addressed in Chapter NINE (Evaluation).

In short, to answer these research questions, we adopted a visual language and template-based approach as our notation for the design and implementation of a critic specification tool to be accessible by end-user tool developers. We measured the accessibility issue by performing an end-user evaluation to assess whether our critic specification approach supports end-user tool developers in the critic-specification task.

1.4 Research Objectives

The main objective of our research is to provide a critic specification capability that allows the end-user tool developers to specify and generate critics for domain-specific visual language tools effectively and easily. In particular, the research aims:

1. To review existing critic approaches used for critic specification and implementation. These would assist us in identifying key critic elements and to recognise techniques or methods applied in critics.
2. To design and develop a simple critic specification approach that is accessible to end-user tool developers.
3. To embed the critic specification approach within a meta-tool environment that implements domain-specific visual language tools.
4. To provide proof concept of the critic specification approach by applying it to three different domains of DSVL exemplar tools.

5. To assess how well the critic specification approach supports the end-user tool developer by performing an end-user evaluation.

1.5 Research Methodology

Our approach to responding to our research question and achieving our objectives was based on the following methodology:

- We conducted a literature review of critic tools, comparing and analysing their approaches for critic specification and implementation;
- We then identified a set of key requirements for a critic specification tool for domain-specific visual language tools;
- We developed a prototype to explore the problems and issues in designing a critic specification tool. We applied an iterative-incremental approach that supports refinement and improvement for our prototype development;
- We identified from our prototyping experience a core set of building blocks required for a generic critic specification editor and design notation;
- We proved our concept of a visual critic specification approach by applying it to three different domains of DSVL exemplar tools: health care planning domain, business process domain and UML design domain;
- We performed an end-user evaluation of our critic specification approach to assess its usability and effectiveness;
- Finally, we derived conclusions from our review, design, prototyping and evaluation work. These are discussed in the final chapter of this thesis i.e. Chapter TEN.

1.6 Research Contributions

The research discussed in this thesis contributes to the field of software engineering particularly in the area of critic tools and critiquing systems development. Main contributions from this research are as follows:

1. This research provides a taxonomy of critics that can assist other users/designers or developers in obtaining relevant information about

critics. Our critic taxonomy identified eight groups: critic domain, critiquing approach, modes of critic feedback, critic rule authoring, critic realisation approach, critic dimension, types of critic feedback, and types of critic. We believe that our critic taxonomy will be useful to critic developers in providing a meaningful way of describing and reasoning about critics. A conference paper describing this taxonomy and titled “A Taxonomy of Computer-supported Critics” was published in Proceedings of the 2010 IEEE International Symposium on Information Technology.

2. This research invented a visual way of expressing and constructing critics for domain-specific visual language (DSVL) tools. Notational representation of critic authoring facilities is offered to end-user designers to express critics for their DSVL tools. Furthermore, this research provides a space for end-user tool developers who want to express critics for their specific tool without the need to have a comprehensive technical knowledge on expressing and constructing critics. A conference paper titled “A Generic Visual Critic Authoring Tool” presented our research proposal in Proceedings of the 2007 IEEE Symposium on Visual Languages and Human-Centric Computing. Papers supporting this work were co-authored and these include:
 - A conference paper titled “Critic Authoring Templates for Specifying Domain-Specific Visual Language Tool Critics”, which was published in Proceedings of the 20th Australian Software Engineering Conference, 2009.
 - A conference paper titled “Template-based Critic Authoring for Domain-Specific Visual Language Tools”, which was published in Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing.
3. This research invented a template-based critic authoring approach which is much easier and quicker to author critics compared to other approaches for designing and realising the critics. An end-user tool developer uses the

critic authoring template to generate critic rule templates. The critic rule templates (CR) adapt the business rule (BR) templates which are currently applied in the business process domain. We attempt to apply the critic rule templates in the software tool domain. By using the critic authoring templates, it is fairly easy for end-user tool developers to introduce new critic templates or modify existing critics in the tool. Papers supporting this work were co-authored and these include:

- A conference paper titled “Critic Authoring Templates for Specifying Domain-Specific Visual Language Tool Critics”, which was published in Proceedings of the 20th Australian Software Engineering Conference, 2009.
 - A conference paper titled “Template-based Critic Authoring for Domain-Specific Visual Language Tools”, which was published in Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing.
4. This research developed a prototype of a visual critic authoring tool which was embedded in the existing Marama meta-tool; which acts as a proof-of-concept of our approach. We evaluated the prototype using an end user study conforming to the Cognitive Dimensions (CD) approach (Green & Blackwell, 1998) and Physics of Notations (PON) principles (Moody, 2008). A conference paper describing this approach titled “End-User Oriented Critic Specification for Domain-Specific Visual Language Tool”, will appear in Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, 2010.

1.7 Thesis Organization

The following chapters are organized as:

Chapter 2: Related Research

- This chapter discusses key related research on critic tools (or critiquing systems) and several meta-modelling tools that have constraint evaluation for

static semantic conformance. Review of these research areas has made it feasible to develop a generic critic specification editor for domain-specific visual language tools. This chapter also led us to develop the critic taxonomy described in Chapter 4.

Chapter 3: Research Methodology

- This chapter describes our approach to designing and prototyping a critic specification tool for domain-specific visual language tools.

Chapter 4: Critic Taxonomy

- This chapter describes a new taxonomy for computer-supported critics. We start with an introduction of what is a taxonomy and then explain the concept of a computer-supported critic. We then present our surveyed literature information in terms of our new critic taxonomy. We also describe each of the elements in the taxonomy using various examples from the surveyed literature on critics. We then apply the taxonomy to characterise several exemplar critic tools.

Chapter 5: A Visual and Template-Based Approach for Critic Specification

- This chapter explains our visual and template-based approach for the critic-authoring task of a domain-specific visual language (DSVL) tool. This chapter begins by introducing the concepts and approaches used for our critic specification research. We introduce the visualization concept followed by the visual notations designed for our critic specification tool. Then we describe the template-based approach, followed by the business rule templates and critic templates. We also explain the concept of authoring and the approach of template-based authoring for critics. In the last section, we present an analysis of the design of our critic specification editor using Moody's Physics of Notations principles (Moody, 2008).

Chapter 6: Initial Prototype for Critic Specification Tool

- This chapter introduces and explains the development steps of the visual and template-based approach for our critic specification tool. We explain our first

attempt to employ MaramaTatau (N. Liu, Hosking, & Grundy, 2007) in specifying critics for Marama-based tools which became our motivation to develop another prototype for the critic specification tool. We then describe the second prototype, which specifies critics in the meta-model editor using a similar visual approach to MaramaTatau however tailored to the critic specification rather than the constraints domain.

Chapter 7: Final Prototype for Critic Specification Tool

- This chapter describes our third prototype for our critic specification tool. We describe the improvements that we made on the previous prototype that we have developed for the critic specification tool as a proof-of-concept of our critic specification approach.

Chapter 8: Case Studies

- This chapter describes three case studies that we used to demonstrate and evaluate the utility of the critic specification editor for Marama DSVL tools. We begin by introducing and describing the first case study - Marama VCPM that explains the use of constraint templates provided by our critic specification editor. We then describe the second case study - MaramaEML that demonstrates the action assertion templates of our critic specification editor. We then describe our third case study - MaramaUML that illustrates the customizing of a critic authoring template via our critic template editor. The chapter ends with some conclusions based on the results from these case studies.

Chapter 9: Evaluation

- This chapter presents the evaluation of our final critic specification prototype for domain-specific visual language tools. We begin by introducing the concepts of evaluations and usability evaluations. Then we introduce the Cognitive Dimensions of Notations framework (CDs) and describe the criteria to evaluate a tool's usability. We then explain the design/method of our survey carried out to assess whether the visual and template-based critic

authoring tool effectively supports end-user developers in specifying critics for DSVL tools. We analyse the survey results and present our findings.

Chapter 10: Conclusions and Future Work

- This chapter concludes this thesis. It discusses the overall research results and limitations of the research. This chapter also suggests some future work that can be performed to extend this body of research work.

Chapter 2

Related Research

This chapter discusses key related research on critic tools (or critiquing systems) and several meta-modelling tools that have constraint evaluation for static semantic conformance. The review of these research areas has made it feasible to develop a generic critic specification editor for domain-specific visual language tools. This chapter also leads us to develop the critic taxonomy which is described in Chapter FOUR.

2.1 Introduction

The value of having integrated support tools (e.g. ArgoUML, Rational Rose, Visible Analyst) to assist developers in software development activities has received significant attention. Some of these integrated support tools have components in the form of *critics*, *recommenders*, or *constraint evaluation facilities* that can support the developers while performing their software development tasks. Many researchers have investigated and developed these support tools. This chapter however, is focused on reviewing the research concerning the evaluations of critics and constraints which is explained in the following sections.

2.2 Critic as a Supporting Tool

The term “critic” was initially used by Miller (1986) to describe a software program that critiques human-generated solutions (Miller, 1986). A considerable amount of literature has been published on critic tools, also known as critiquing systems. Motivations from many efforts such as Miller’s work (1986), Fischer’s endeavour (1987, 1989-1991), Silverman’s study (1992) and others have attracted a wider audience on critic-based approaches. Critic tools/systems have been recognized as an essential support tool in a range of domains. The types of support offered by these critic tools are certainly in various ways. The following sections discuss the

purpose and support provided by the critic tools in diverse domains/systems/environments.

2.2.1 Critics in Information Systems

The critic concept or critic-based approach was initially introduced in Information Systems (IS) mainly in the medical domain from the work by (Miller, 1986). Critics are widely used in expert systems, decision support systems, knowledge-based systems and other IS applications. We explain a few of these applications below.

According to Silverman and Mehzer (1992) expert critiquing systems are “a class of program that receive as input the statement of the problem and the user-proposed solution. They produce as output a critique of the user’s judgement and knowledge in terms of what the program thinks is wrong with the user-proposed solution.” In addition, Silverman (1992) reports an illustrative survey on the development of expert critiquing systems. The survey paper (Silverman, 1992) illustrates several applications that were developed using the expert critiquing approach (e.g., ONCONCIN, ATTENDING, CRITTER, COPE). In 1993, Hägglund published a paper that introduces the approach of expert critiquing systems. Hägglund (1993) explains several characteristics that apply to expert critiquing systems and also distinguishes the use of critics and critiquing based on the work from Fickas (1988), Fischer (1993) and Rankin (1993). Critics functioned as a mechanism for reasoning and problem solving, whereas critiquing as a way of offering non-intrusive recommendations to a user and also as the basis for providing arguments and explanations in an effective way (Hägglund, 1993). An example of an expert critiquing system was illustrated by Mehzer et al (1998) in a decision making problem that was implemented in an automated environment as shown in Figure 2.1. The application of expert critiquing systems in decision making problems can reduce human errors (Mehzer, Abdul-Malak, & Maarouf, 1998; Silverman & Mehzer, 1992).

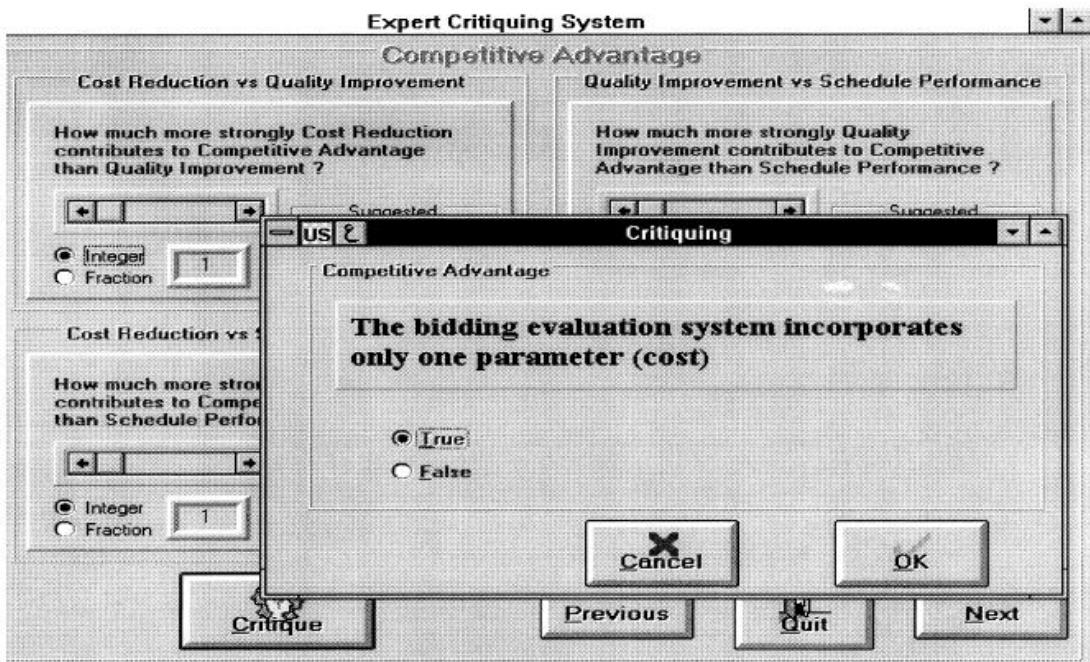


Figure 2.1: Screen shot of an expert critiquing system (Mehzer, et al., 1998).

While critics have been used in expert system applications which are known as expert critiquing systems, they have also been applied in several decision support systems (DSS). For instance, Gertner and Webber (1998) developed an online decision support system for trauma management, TraumaTIQ. TraumaTIQ can help a physician with treatment planning. It interprets the goal of the physician's treatment plan, evaluates the inferred plan structure by comparing it with the system's recommended treatment plan, and finally generates a critique that addresses the potential problem (Gertner & Webber, 1998). Vahidov and Elrod (1999) introduce a framework for an active DSS based on critiques and argumentations (Vahidov & Elrod, 1999). They describe the use of positive ('angel') and negative ('devil') critiquing agents in a DSS to allow active participations in decision making processes. Figure 2.2 shows an example of the critiquing DSS from an investment problem (Vahidov & Elrod, 1999).

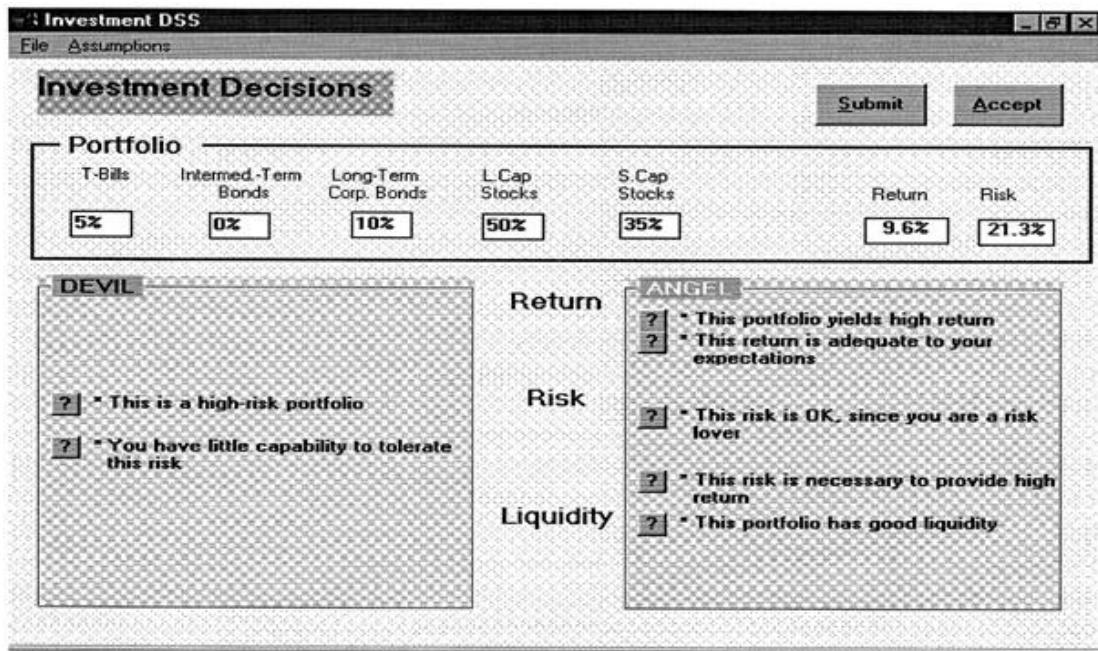


Figure 2.2: Example of DSS that employed critics (Vahidov & Elrod, 1999)

Irandoost (2006) published a technical report that discusses the critiquing systems for decision support (Irandoost, 2006). The objective of the report was to explain critiquing systems and discusses their details as decision support tools (Irandoost, 2006). In fact we used the Irandoost (2006) report as our basis in creating our critic taxonomy which is explained in Chapter FOUR.

Critics are also employed in knowledge-based systems. Furthermore a conceptual framework for knowledge-based critic systems has been established by (Fischer & Mastaglio, 1990) with the aims to support the collaboration between a computer and a user and to improve problem solving and learning by users. In 1990, Lemke and Fischer have published an article that describes FRAMER, a knowledge-based system for windows user interface design using high-level constructs. The purpose of FRAMER is to ease the knowledge required to design (Lemke & Fischer, 1990) and help less skilled designers in applying a high-level abstraction-program frameworks (Fischer, Lemke, Mastaglio, et al., 1991; Lemke & Fischer, 1990). Critics are a formal knowledge source in FRAMER. Lemke and Fischer (1990) claim that FRAMER offered mandatory and optional critiques. Mandatory critiques reflect the system requirements which must be fulfilled for the construction of a program framework. Optional critiques suggest typical design choices which

designers can ignore if necessary (Lemke & Fischer, 1990). In addition, designers can browse the explanation repository for explanations about the critiques. The descriptions of FRAMER can be found in (Lemke & Fischer, 1990), (Fischer, Lemke, Mastaglio, et al., 1991) and (Fischer, Lemke, & Mastaglio, 1991).

In the research by Liu et al (1995), they illustrated a knowledge-based engineering design system that adopted critics. The system offers a set of critics: expertise completion, correctness and consistency checking, and alternative solution critics (H. Liu, Rowles, & Wen, 1995). The critic system is basically to assist the knowledge engineers in acquiring sufficient knowledge for building a desired system and employing appropriate knowledge to generating designs (H. Liu, et al., 1995).

2.2.2 Critics in Software Engineering

The critic-based approach which was well-accepted in Information Systems (IS) then received a significant attention from the software engineering (SE) community. Critics are now in wide-spread use in the field of SE.

We identified three examples of critic tools that are recognized to be useful in software requirement engineering: AIR, Prefer and HeRA. Maiden and Sutcliffe (1994) describe an Advisor for Intelligent Reuse (**AIR**), a tool to assist the requirement engineer during requirements critiquing. They claim that it is essential to use a critic for intelligent assistance during requirements engineering (Maiden & Sutcliffe, 1994). They proposed requirements critiquing using domain abstractions that represent the fundamental behaviour, structure and functions of a domain class (Maiden & Sutcliffe, 1994). The AIR tool consists of three components known as capture, match and critic requirements. The capture component performs the acquiring process of new facts and requirements from the requirement engineer. The matcher component performs the mappings between abstractions and the new domain to detect problem situations. The requirements critic supports domain understanding and critiquing by explaining retrieved domain abstractions and detected problem situations to the requirement engineer. The good thing about AIR is that it provides ‘rollback’ buttons which allows the requirement engineer to undo

matching if mistakes are noticed. Therefore it is responsive to requirement engineers' needs and can support situated reasoning during requirements engineering.

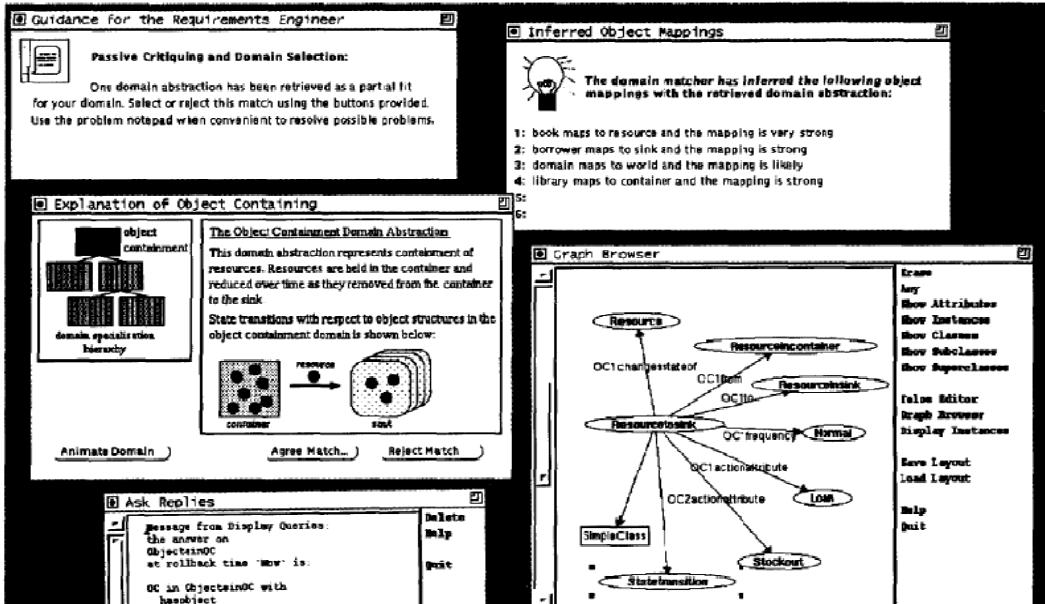


Figure 2.3: Screen shot of the AIR tool (Maiden & Sutcliffe, 1994)

In another research work, Redmiles (1998) argues the need for cognitive support by requirements engineers to produce a better requirements design. Requirements engineers need knowledge of requirements specification method, problem domain, and other relevant knowledge to create a good requirements design (Redmiles, 1998). Applying design critics to software requirements helps designers to improve the quality of requirements design. The *Prefer* tool is used to model state-based requirements design in the CoRE notation (Redmiles, 1998). *Prefer* adopted the common tool infrastructure of Argo/UML. *Prefer* includes design critics and a dynamic “to do” list that presents feedback from critics in a systematic way.

Knauss et al's (2009) recent research also involves critics in requirements engineering. It illustrates the Heuristic Requirements Assistant (HeRA) editor, which offers a heuristic feedback to the requirements analyst on incomplete requirements specification. The functions of the HeRA editor are to: 1) capture high-quality requirements at the user goal level; 2) identify contradictions to other user's requirements; and 3) align user goals with the planned business process quickly

(Knauss, Luebke, & Meyer, 2009). The HeRA descriptions are explained in Chapter FOUR.

While critics have been used in the requirements engineering area, critics for object-oriented modelling heuristics, as well as the UML semantics have also been offered by several software architecture modelling tools. For example, Robbins and Redmiles (2000) describe Argo/UML, a tool for object-oriented modeling. This tool supports the editing of diagrams according to the Unified Modeling Language (UML) notation and detects common errors made by software designers. Argo/UML supports the designer with online critics about the design model under construction (Robbins & Redmiles, 2000). They describe Argo/UML, based on cognitive theories, to support the development of software architecture models. Argo/UML is a software architecture design environment that helps architects by focusing on cognitive challenges of design that introduced by three theories : i) theory of reflection-in action, ii) theory of opportunistic design and iii) theory of comprehension and problem solving (Robbins & Redmiles, 2000). More on Argo/UML descriptions are explained in Chapter 4.

An environment called **Annotation Based Cooperative Diagram Editor** (ABCDE)-Critic which was developed by (de Souza, et al., 2000) uses a critic-based approach to check UML class diagrams. ABCDE-Critic is a Domain Oriented Design Environment (DODE) for object-oriented analysis and design, which implements a group critic system. In ABCDE-Critic, feedback is presented as annotations attached to the diagram elements that trigger the critic to fire (de Souza, et al., 2000). These annotations are also displayed to all other designers who are owners of these diagram elements. Developing software systems is a complex task and most of the software development activities are handled by a group of people. Thus, a team effort is essential in producing solutions to a complex software problem and also to ensure it will succeed. Due to this reason, ABCDE-Critic is a useful system because it supports cooperation among designers as a means of annotation and warns designers of the occurrence of problems. More detailed description of ABCDE-Critic is given in Chapter 4.

Another research project (Dashofy, Hoek, & Taylor, 2002) demonstrates the ArchStudio3 tool that uses design critics for architecture analysis. The design critics

monitor changes performed in architecture modelling. The design critics check any potential problems that may exist due to the changes and then report to a central issue database (Dashofy, et al., 2002). Dashofy et al (2002) claims self-healing systems can be applied by using critics to do ‘what-if’ analysis on the affect of a possible repair.

Grundy and Hosking (2003) explain the SoftArch tool that assists architects in static validation of their architecture models. SoftArch provides a set of model analysis agents that monitor changes in architecture models and then offer feedback to architects in a form of an immediate error report and ‘error list’ (Grundy & Hosking, 2003). The agents are actually design ‘critics’ where they watch for model changes and add messages (critique) to a critic message dialogue.

2.2.3 Critics in Education Environment

The education community strives to enhance teaching and learning between students and educators (e.g., teachers, instructors, lecturers, mentors and others). One of the most important elements that can improve teaching and learning is by providing learners with effective and timely feedback (Brown, 1988). Thus to address the key teaching and learning concern, a computer-supported learning tool using a critic-based approach has often been adopted in the education area.

For instance, Fischer (1987) implemented the LISP-CRITIC with the aim to support users on how to improve their LISP code. The LISP-CRITIC is used in an introductory LISP course which teaches LISP programming. The user’s code is matched against a large set of critiquing rules that specify how to improve LISP code. Any mistakes in the code will cause the tool to offer modification suggestions (critique) to the user. The user can check the code improvement suggestions and make a decision on whether to agree or disagree with the suggestion. In addition the tool provides explanation and justification on its suggestions.

Other research that supports program critiquing in an education environment includes: Submit! (Pisan, Richards, Sloane, Koncek, & Mitchell, 2003) and Java Critiquer (Qiu & Riesbeck, 2008). Pisan et al (2003) developed *Submit!*, a program critiquing system that provides critical feedback to students about the computer

programs they write. Program critiquing refers to the process by which students obtain critical feedback about their programs. In their approach, students are allowed to use the critiquing tools before final submission of an assignment. Thus students can get a formative assessment that supports self-directed learning. Pisan et al (2003) performs usability evaluations and the results show that Submit! is generally effective. A preliminary study of the impact of Submit! on student results indicates that students who apply the system to get feedback on assignment submissions do better than those who do not apply.

Likewise, Qiu and Riesbeck (2008) demonstrate the development of an educational critic tool, JavaCritiquer. They created a critiquing tool for Java programming. This critic tool not only supports the teachers but also the students. Teachers use the Java Critiquer to critique student java code whereas the students get feedback support from JavaCritiquer before sending their assignments to their teacher. Their conclusions identified two main points: 1) the tool is good at providing individualized feedback to students and 2) the tool is difficult to create and requires significant development effort (Qiu & Riesbeck, 2008).

Another example of an educational critic tool is ClassCompass (Coelho & Murphy, 2007). Coelho and Murphy (2007) demonstrate ClassCompass that assists students and instructors in software design activities. The ClassCompass supports the students by offering an automatic critique that gives suggestion when a potential error on the design is identified. The instructor can view the student design and can provide additional feedback via the tool. The tool supports automatic and manual critiquing of software designs, specifically in UML class diagrams and sequence diagrams. Descriptions on ClassCompass and Java Critiquer are explained in Chapter FOUR.

Oh et al (Oh, Gross, Ishizaki, & Do, 2009) present a tool called Flat-pack Furniture Design Critic (FFDC). The FFDC tool is to support students who are involved in an architecture design course/program. The motivation from the strength of critiquing in architectural design studio (Oh, et al., 2009) has led the development of the FFDC tool which provides students with feedback via five delivery types: interpretation,

introduction, example, demonstration, and evaluation, along with three communication modalities: written comments, graphical annotations, and images. A student's task model is evaluated by the FFDC tool which chooses the delivery type and modality to offer a critique. Description on the FFDC tool is explained in Chapter FOUR.

2.2.4 Critics in Recommender Systems

McGinty, Smyth, McCarthy, and Reilly (K. McCarthy, et al., 2005; K. McCarthy et al., 2006; McGinty & Smyth, 2003; Reilly, McCarthy, McGinty, & Smyth, 2005) employ critiquing-based approaches to improve the efficiency of their recommender systems. Recommender systems are programs that help users by facilitating access to relevant items. For example, if a user wants to buy a desktop PC through an online system, he can specify the essential features of the desktop PC, such as model, price, hard disk capacity, etc. to query a recommender system. Then the recommender system will provide access to the relevant desktop PC configuration based on the user's specification.

McGinty and Smyth (2003) explain the use of a critiquing system as the main technique of feedback on reactive recommender systems. Reactive recommender systems (McGinty & Smyth, 2003) are designed to make recommendations based on a user's query. McGinty and Smyth (2003) made a comprehensive evaluation of three critiquing techniques in a comparison-based recommender. These three critiquing techniques are standard critiquing (STD), critiquing with carrying the preference (CP), and critiquing with adaptive selection (AS) (McGinty & Smyth, 2003). In this evaluation, the performance of these critiquing techniques is compared and the results indicate that AS significantly improves recommendation efficiency. The main idea of AS is to increase the degree of diversity among recommended items to cover more item space in a given cycle and thus increase recommendation efficiency.

Later, McCarthy et al (2005) presented a dynamic critiquing approach which supports users in modifying multiple features concurrently by selecting from compound critics. A live-user evaluation is done and results indicate that users who

apply compound critics obtained shorter recommendation sessions that direct to higher quality purchases (K. McCarthy, et al., 2005).

Following dynamic critiquing, Reilly et al (2005) describe an incremental critiquing approach that considers a user's critiquing history, as well as their current critic, when making new recommendations (Reilly, et al., 2005). An evaluation of incremental critiquing shows that it can deliver significant performance benefits by reducing session lengths by up to 70%, regardless of whether to use unit or compound critics. In fact, the dynamic critiquing is combined with incremental critiquing and it improves the efficiency of critiquing in recommender systems. Figure 2.4 shows a screen shot of such a recommender system.

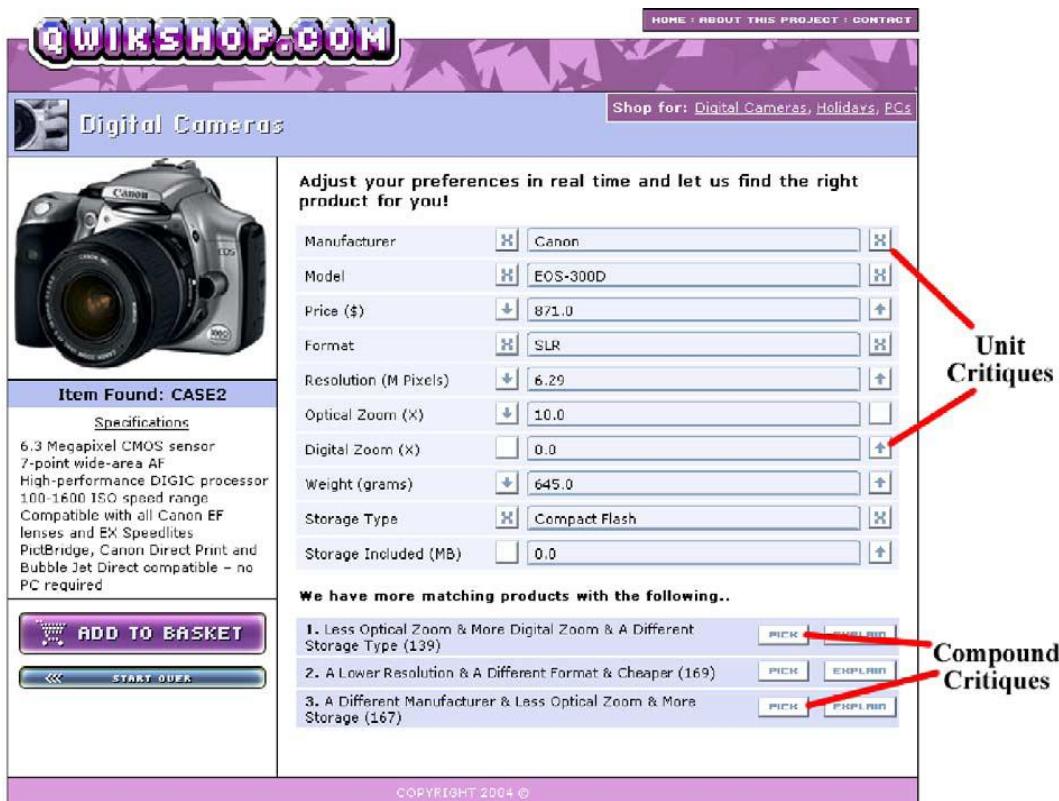


Figure 2.4; Example of recommender system (Reilly, et al., 2005)

In another research work, McCarthy et al (2006) describe the use of a critiquing-based approach for group recommender systems. A group recommender system called **Collaborative Advisory Travel System (CATS)** is designed to assist a group of users in making decision for a vacation (K. McCarthy, et al., 2005). A DiamondTouch tabletop device is used to showcase the CATS. The CATS approach

is based on collaborative recommendation framework. There is an interaction component in CATS that consists of an individual or group interaction. There is also a recommendation component that consists of two parts: 1) an individual recommendation (system reactively recommends cases to the user), 2) a group recommendation (system proactively pushes recommendations to the group of users). Critics made by users are stored in a group user model and this is used as a basis for recommendations. The contribution of CATS is to enable the user as an individual or a group to interact simultaneously through recommendation dialogs and to achieve consensus in their decision making about vacation planning.

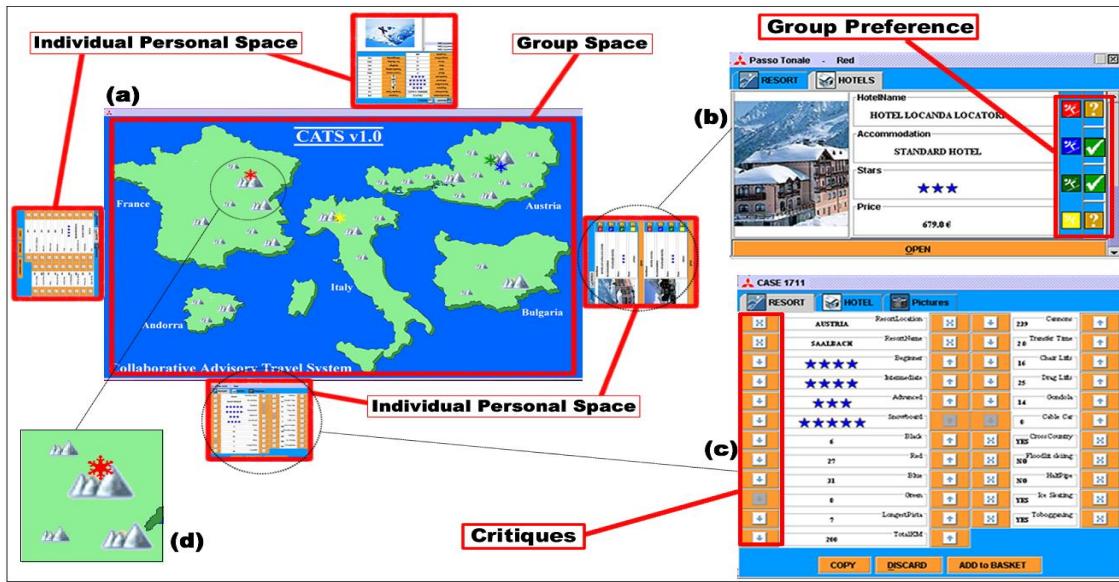


Figure 2.5 Example of group recommender system (K. McCarthy, et al., 2006)

2.2.5 Benefits from Critics Application

All of the applications stated above have shown that critics are an efficient feedback-providing mechanism and offered huge advantages/benefits. Furthermore critics are applicable to various domains as described above. While critics in these application domains have its own deficient, the application of critics in diverse domains has contributed several benefits/advantages including:

- Problem solving and learning improvement;

Critics provide problem solving and learning improvement (Fischer & Mastaglio, 1990; Fischer, Nakakoji, Ostwald, Stahl, & Sumner, 1993; Robbins, 1998; Tianfield & Wang, 2004) to users. The critic is like an

assistant to the user in the problem solving task because the user is the one that deals with the problem solving activities. With feedback or critiques generated by the critic tool it would eventually improve the user's skill in problem solving process in an incremental way. Furthermore, the user's problem learning of domain knowledge would also be improved during the problem solving process (Robbins, 1998). For instance, Java Critiquer (Qiu & Riesbeck, 2008) is a critiquing system for educational purposes. The Java Critiquer helps students to learn and improve their Java programming skills. Java Critiquer uses passive critiquing so as to allow the students to make mistakes during the problem solving task without intrusion. It then critiques any bad programming code made by the students, and offers a suggestion to improve the code. Thus, the students' problem solving and learning skills will be improved incrementally. Other critic tools including LISP-Critic, Argo/UML, FFDC, and ClassCompass also contribute to the problem solving and learning improvement.

- Human errors reduction;

Human errors in whatever task have a variety of causes (Mehzer, et al., 1998). However, with critics support that provides explanations and alternative solutions it would help the users to make fewer errors in their action or decision making tasks. For example, TIME critic (Silverman, 1991) is an organizational support system that helps generate a document that is part of the system acquisition milestone or decision process. The application assists headquarters decision makers to communicate best-practice information to the field, and also minimizes the number of field-created errors and biases that headquarters must deal with (Silverman, 1991). Other examples that contribute to this benefit are expert critiquing systems (Mehzer, et al., 1998; Silverman & Mehzer, 1992), critics in decision support systems (Vahidov & Elrod, 1999) and critics in recommender systems (K. McCarthy, et al., 2006; Reilly, et al., 2005).

- Human-computer interaction enhancement;

Critics can effectively facilitate human-computer interactive problem solving (Fischer, 1989; Fischer, Lemke, & Mastaglio, 1991; Fischer, Lemke, Mastaglio, et al., 1991; Fischer, et al., 1993; Tianfield & Wang, 2004). It is almost impossible for a human to have the complete knowledge about a domain (Robbins, 1998; Tianfield & Wang, 2004). Therefore it is necessary to have an interaction with a supporting tool, such as critics to assist the user in activities that the user cannot perform well (Terveen, 1995). Critics can augment the ability of human to assess their actions/solutions. However, it is still up to the human to make a decision whether to follow the critic suggestions or not. For instance, JANUS (Fischer, Lemke, & Mastaglio, 1991) is an integrated design environment for residential kitchen layout configuration. It allows a designer to construct residential kitchen floor layout plans and to learn general principles underlying such constructions (Fischer, Lemke, & Mastaglio, 1991). The knowledge stored in JANUS includes building codes, safety standards and functional preferences which improve the interaction of the designers with the system as well as their learning in design construction.

- Proactive design improvement;

A good sign of how well a design tool/system is developed and knowledge is used is based on ‘designs’ that produced by users (i.e. designers/developers) (H. Liu, et al., 1995). Poor designs or erroneous designs are normally caused because the users lack specific knowledge about the design problems or solution domains (Robbins & Redmiles, 2000). With critics that provide knowledge support in terms of guidelines or suggestions, the users would be assisted in achieving improvements in their design tasks and artifacts. For example, ArgoUML (Robbins & Redmiles, 2000) is a design critiquing tool. It is an open source UML modelling tool that supports all standard UML 1.4 diagrams (<http://argouml.tigris.org/>). The ArgoUML critics constantly check the current model and if the conditions for triggering a critic are met, the critic will generate a list of items (i.e. critiques) in a dynamic ‘To do’ list. The presentation of a short description of the problem, with the guidelines to

resolve the problem, and a wizard helps ArgoUML users to improve the design and solve the problem automatically. Examples of other tools that contribute to this benefit include JANUS, IDEA and HeRA.

- Proactive inconsistency and incompleteness detection;

Critics can help users to detect any inconsistency and incompleteness in analysis and design situations (de Souza, et al., 2000; de Souza, Oliveira, da Rocha, Goncalves, & Redmiles, 2003). Critics can offer proactive design feedback to a user's action if the action violates the inconsistency and incompleteness rules of a design. For instance, DAISY(de Souza, et al., 2003) is an environment that supports the construction of domain engineering and application engineering models. It provides consistency checking of the models via critics. The ArgoUML (Robbins & Redmiles, 2000) advises designers when an inconsistency and incomplete UML models is detected and feedback is given to resolve the problem. Likewise, the HeRA (Knauss, et al., 2009) detects any incomplete and inconsistency of requirements specifications.

- Heuristic-based guidance;

Knowledge support offered by critics is usually in a form of guidelines or recommendations that are based on certain general principles (e.g., design principles), standards (e.g., UML standards) or relevant source documents. However, critics also provide heuristic-based guidance to their users. For example, HeRA (Knauss, et al., 2009) provides heuristic feedback to requirement engineers for: 1) capturing high-quality requirements on user goal level; 2) identifying contradictions to other user's requirements; and 3) aligning user goals to the intended business process (Knauss, et al., 2009). Similarly ABCDE-Critic (de Souza, et al., 2000) provides critics on UML class diagrams based on experiences and object-oriented design heuristics.

- Context-sensitive feedback;

The feedback provided by critics is often context-sensitive which depends on a task or situation of the problem domain. This is necessary as to ensure the human-computer (critic) interaction is achieved and the users can obtain good feedback for resolving a problem or improve a solution. For instance, the FFDC (Oh, et al., 2009) is a critic tool that helps the architecture students to be familiar with the design problem-solving tasks. The FFDC offers student feedback via five delivery types (interpretation, introduction, example, demonstration, and evaluation) and three communication modalities (written comments, graphical annotations, and images). For example, painting parts that violate a constraint are coloured in red, with graphic icons such as arrows to represent load placed on a furniture part (Oh, et al., 2009). Likewise, the ArgoUML offers a constructive and instant feedback in a non-intrusive manner to designers in solving a problem. Furthermore, the cognitive features of ArgoUML provide designers with support for decision-making, decision ordering, and task-specific design understanding which are believed to be useful in designing contexts and tools (Robbins & Redmiles, 2000).

2.3 Constraint Specification in a Meta-Modelling Tool

Many meta-tool environments and toolkits have been developed to support the development of visual language environments. Examples of these tools are: MetaEdit+ ((Kelly, et al., 1996), ATOM (Lara & Vangheluwe, 2002), KOGGE (Ebert, Suttenbach, & Uhe, 1997), Pounamu(Zhu, et al., 2007) and Marama (Grundy, et al., 2008). Meta-tools provide an integrated environment for developing other tools and often these tools also offer constraints evaluation/checking which are similar to the critic concepts.

We introduce here some of the views on constraints in software tools. According to Borning (1986), a constraint “specifies a relation that must be maintained, for example, that a line be horizontal, that a resistor obey Ohm’s law...” In contrast, Balarin et al. (2001) propose performance constraints specification at higher levels of abstraction, thereby limiting their constraints definition scope to “a representation

that is more natural to the designer and that is more computationally tractable.” In the work by Qattaous (2009), constraints are used for “governing the syntax and semantics of model elements and the values of their attributes” in meta-CASE tools.

A meta-CASE tool is often concerned with metamodeling processes and techniques. According to Qattaous (2009), meta-modelling techniques rely on two elements to identify the domain specific language syntax and semantics: 1) a meta-model and 2) constraints. The constraints are viewed as indications to lead users to a good design solution (Qattous, 2009). Cook et al. (2007) also provide similar view about constraints which are seen as “a way for humans to evaluate the current state of a model with respect to some criteria; for example, whether all of the web server configurations are compliant with the corporate standards.” There are various views on the definition of constraints as mentioned above. One common aspect from the various views about constraints is that they involve specifying or defining constraints using some kind of representation/approach with the intention to establish a set of rules with respect to some criteria that should be compliant to a particular product/item (e.g., model, design, standard, document, etc).

We are more interested in constraint specification within a meta-tool environment as our research work also deals with a meta-tool (i.e. the Marama meta tools (Grundy, et al., 2008)). Specifying or expressing constraints is often applied in the metamodeling tools (Jaramillo, Vangheluwe, & Moreno, 2003). A constraint language is added to a meta-model to constrain the structure of a model. Sourrouille and Caplat (2002) classify constraints as syntactic constraints and semantic constraints. Syntactic constraints are specified in a formal language, such as OCL, and can be verified automatically (cf. Sourrouille and Caplat (2002)). Semantic constraints are specified in natural language and have to be checked manually (cf. Sourrouille and Caplat (2002)). In another research by Bezivin and Jouault (2006), constraints can be labelled as a warning, error, or critic. The three labels are used to describe the severity of a constraint (Bezivin & Jouault, 2006). Examples of these constraints with their severity labels are shown in Figure 2.6.

```

-- (C1) Error: the name of a Classifier must
-- be unique within its package.
context Classifier inv:
    not self.package.contents->exists(e |
        (e <> self) and (e.name = self.name))

-- (C2) Error: the name of a StructuralFeature must
-- be unique within its Class and its supertypes.
context StructuralFeature inv:
    not self.owner.allStructuralFeatures()->exists(e |
        (e <> self) and (e.name = self.name))

-- (C3) Warning: an abstract class should have children.
context Class inv:
    not (self.isAbstract and
        (Class.allInstances()->select(e |
            e.supertypes->includes(self)
        )->size() = 0))

-- (C4) Critic: the name of a Classifier should
-- begin with an upper case letter.
context Classifier inv:
    not (let firstChar : String =
        self.name.substring(1, 1) in
        firstChar <> firstChar.toUpperCase())

```

Figure 2.6: Examples of constraints expressed in OCL (Bezivin & Jouault, 2006)

Constraints also can be categorised as: operational constraints (used to restrict design space alternatives based upon the operations of a model), composability constraints (express compatibility between different alternatives), resource constraints (indicate specific hardware resources that are needed by software modules) and performance constraints (indicate an end-to-end latency, throughput, power consumption, and bit precision) (Gray, Bapty, & Neema, 2000). A screen shot of this is shown in Figure 2.7.

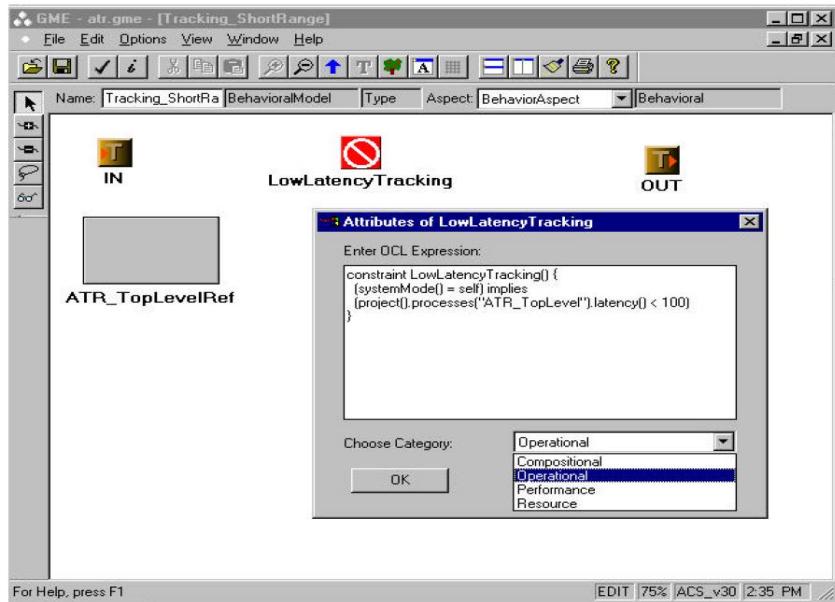


Figure 2.7: Constraint expression using OCL expression (Gray, et al., 2000)

All of these constraints regardless of their classifications have to be specified using some kind of formal representation. There are various approaches that can be used to express the constraints. Constraints can be expressed or specified using the Object Constraint Language (OCL) expressions (Gray, et al., 2000; Karsai, Nordstrom, Ledeczi, & Sztipanovits, 2000) executable scripting language Python (Jaramillo, et al., 2003), ATL language (Bezivin & Jouault, 2006), programming by example (Qattous, 2009) and other approaches. Figure 2.6 and Figure 2.7 are examples of constraints using OCL expressions. We describe a few of the meta-modelling tools regarding their constraints specification/evaluation in the following sections.

2.3.1 MetaEdit+

MetaEdit+ is a fully configurable multi-user and multi-tool computer-aided system and method engineering environment (Kelly, et al., 1996). The tool architecture for MetaEdit+ is shown in Figure 2.8. The tool architecture comprises of five main tools: 1) environment management tools; 2) model editing tools; 3) model retrieval tools; 4) model linking and annotation tools; and 5) method management tools. MetaEdit+ provides a metamodeling language and tool suite for defining the method concepts, their properties, associated rules, symbols, checking reports, and generators. MetaEdit+ is based on an implementation of the Graph, Object, Port,

Property, Relationship and Role (GOPPRR) metamodeling language and is written in Smalltalk (Pohjonen, 2005). The main advantage of MetaEdit+ tool is the ability to quickly specify a tool for a given modeling language (Pohjonen, 2005; Tolvanen, 2004; Tolvanen, Pohjonen, & Kelly, 2007).

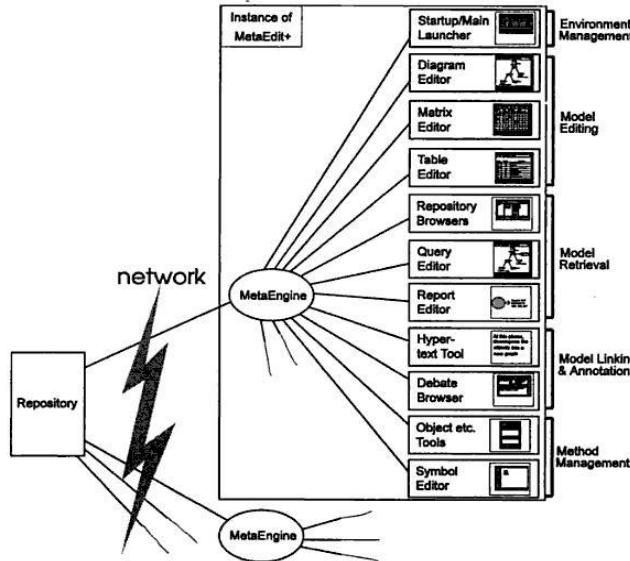


Figure 2.8: MetaEdit+ architecture (Kelly, et al., 1996)

According to Tolvanen et al. (2007), rules and constraints are the main components of a meta model that guide the application of a modeling language. MetaEdit+ provides a Constraints Definer tool as shown in Figure 2.9 (left) to support the definition of rules that refine and constrain the behaviour and the use of language (Tolvanen, 2004). Furthermore, to set constraints on design elements' occurrence, connectivity and uniqueness can be defined via the Graph Constraints Tool as shown in Figure 2.9 (right) (Tolvanen, et al., 2007). The defined rules and constraints are enforced at run-time to ensure the correctness of the models.

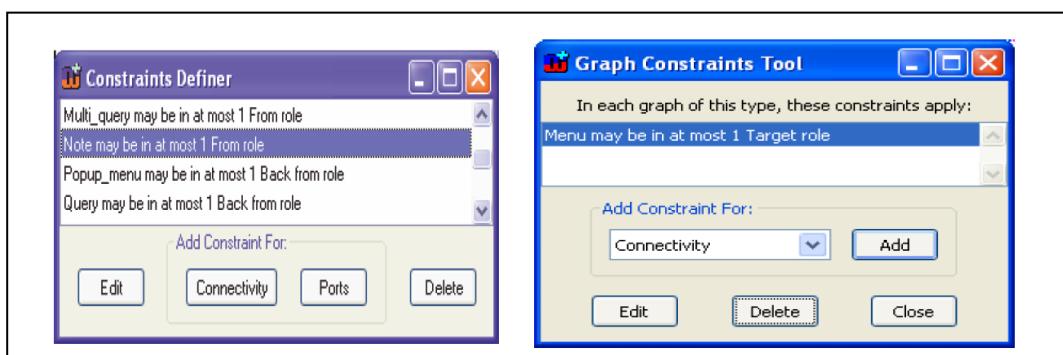


Figure 2.9: Constraints definer editor ((Tolvanen, 2004)) and Graph constraints tool (Tolvanen, et al., 2007)

2.3.2 Pounamu

Pounamu (Zhu, et al., 2007) is a meta-tool developed for building visual design tools. Pounamu allows users to specify the meta-model, shapes and diagrams for tools using a variety of visual languages. These elements are shown in Figure 2.10 that represents the structure of Pounamu tool specifications.

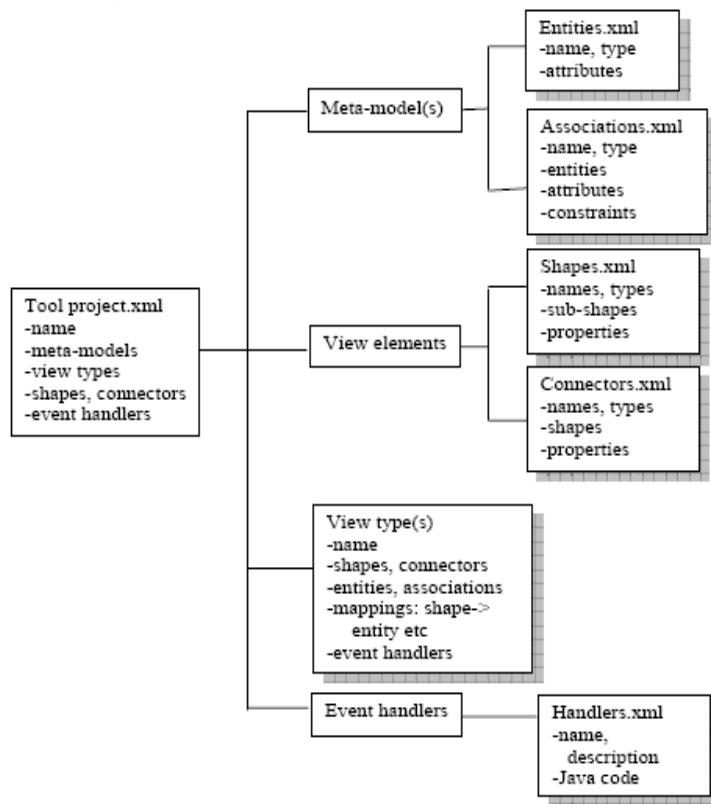


Figure 2.10 Structure of Pounamu specification(Grundy, Hosking, Zhu, & Liu, 2006)

The Pounamu meta-tool also includes a visual language to represent events and associated actions. In Pounamu, the definition and insertion of constraints is performed via an event handling approach (Zhu, et al., 2007). A visual event handler definer is used to build both simple and complex event handling functionality for Pounamu tools. Some of the constraints that can be defined via the event handler definer are: type checking, model constraints, layout constraints and behaviour, mapping constraints, and back-end functionality constraints (Zhu, et al., 2007). These constraints are implemented via hard-coded approach using Java code scripts.

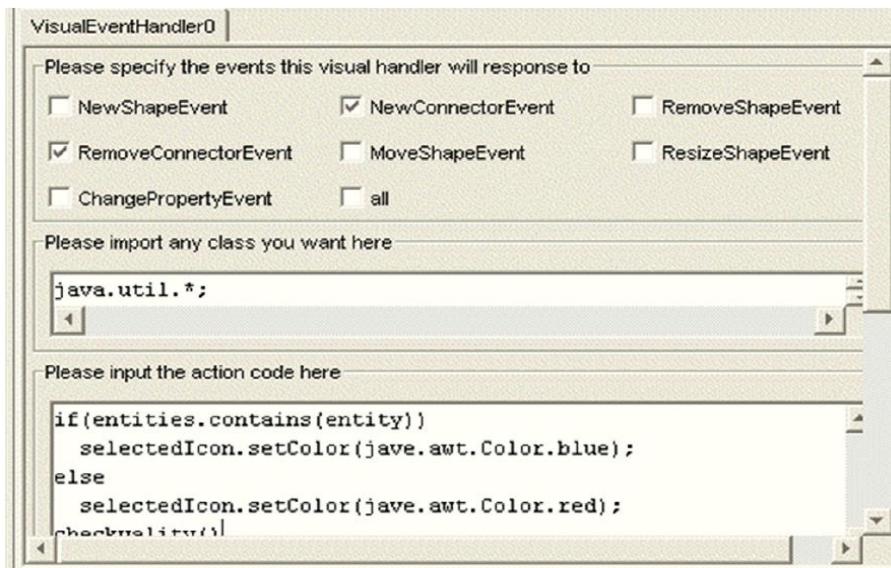


Figure 2.11: Example of code-based event handler for model constraints (Zhu, et al., 2007)

Constraints specified using the event handler approach in Pounamu requires users to be familiar with the Java code scripts and the Pounamu API (Zhu, et al., 2007). This provides a difficulty or barrier to the less experienced users in defining constraints via the event handler definer. Even for expert uses, maintaining complex event handler code can be time consuming and error-prone. Constraints can be reused by packaging them as parameterised Java classes in script code files.

2.3.3 Marama

Marama (Grundy, et al., 2008; Grundy, et al., 2006) is an Eclipse based meta-toolset which was initially generated from the Pounamu (Zhu, et al., 2007) meta-tool specifications. According to (Grundy, et al., 2008) the goal for the Marama toolset is to support easy implementation of diagrammatic modelling/MD tools for experienced modellers with basic modelling concepts. These concepts consist of Extended Entity Relationship (EER) models, OCL, and the meta-models notion. In this section we focus our explanation on MaramaTatau which is an extension to the locally developed Marama metatool set. MaramaTatau offers the ability to specify behavioural extensions to Marama metamodel (N. Liu, et al., 2007).

MaramaTatau (N. Liu, et al., 2007) provides a declarative constraint/dependency specification mechanism which focuses on structural constraints for a DSVL

metatool. The main notation for constraint representation used by MaramaTatau is declarative OCL expressions. MaramaTatau allows tool developers to specify constraints over metamodels using the OCL formula. Figure 2.12 shows the Marama metamodel editor with its MaramaTatau extensions. A combination of OCL expressions and a visual notation is used in MaramaTatau. A green coloured circle represents the OCL formula for specifying a constraint (refer to Figure 2.12). The green colour circle shape is associated with an interface known as *Formula Construction View* which is used to define the required constraints based on the metamodel elements and OCL expressions that are listed in the view (N. Liu, et al., 2007). The constraints definition will take effect when a user runs the modeling tool.

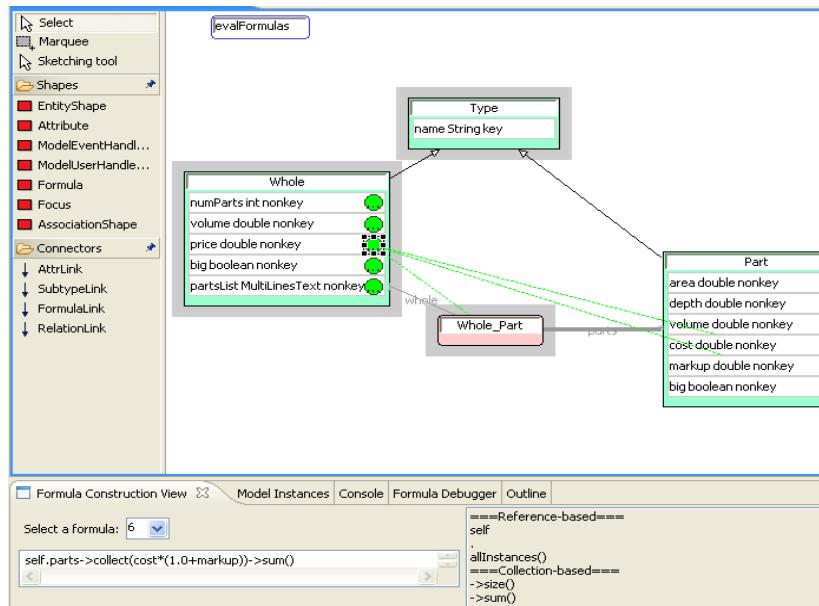


Figure 2.12: Constraint specification via MaramaTatau using OCL formula (N. Liu, et al., 2007)

The MaramaTatau approach was aimed to better support target end users who are programming literate and familiar with modeling concepts (N. Liu, et al., 2007) who would be able to specify the model level constraints using the OCL expressions. While this is a more declarative, high-level approach than Java event handlers used in Pounamu, we have found OCL constraints are still complex and challenging to use for many Marama tool developers.

2.3.4 DECS

Diagram Editor Constraints System (DECS) is an Eclipse-based meta-tool prototype developed with the purpose to generate constraint-based domain specific diagram editors(Qattous, 2009). The research work by (Qattous, 2009) is aim to support and simplify the process of constraints definition as part of domain specific CASE tool specification in a meta CASE tool. The work applies a programming by example approach for a constraint definition. According to Qattous (2009) a DECS user can define a constraint either using a wizard or by example. With a wizard style, a user can use several forms to define values to several constraint properties as necessary. The constraint definition by example approach is performed by allowing a user to create one or more examples of the required constraint. The system should then be able to infer the intended constraint based on the examples (Qattous, 2009).

A constraint manager component which is separated from the DECS holds the XML-based constraint description and expression language. Thus, the constraint manager component will have a list of constraints and uses these constraints as assertions for users' actions in the modelling environment (Qattous, 2009). Whenever a tool user modifies the diagram model, it is checked by the constraint manager. The constraint manager will trigger a warning if there is any violation detected. Figure 2.13 shows the architecture of DECS. According to Qattous (2009) the constraint definition by example approach involves a complex inference process due to the constraints' complex nature and various constraint alternatives that an example could imply. While this approach is more abstract again than Meta-Edit+ or MaramaTatau, the inference process means tool developers need to understand this process to express constraints sensibly. Programming by example-based approaches like this have also been shown to be difficult to describe to end users after specification, making maintaining and reusing the constraints inferred difficult. Complex constraints over collections and relationships can also be very difficult to express with this approach.

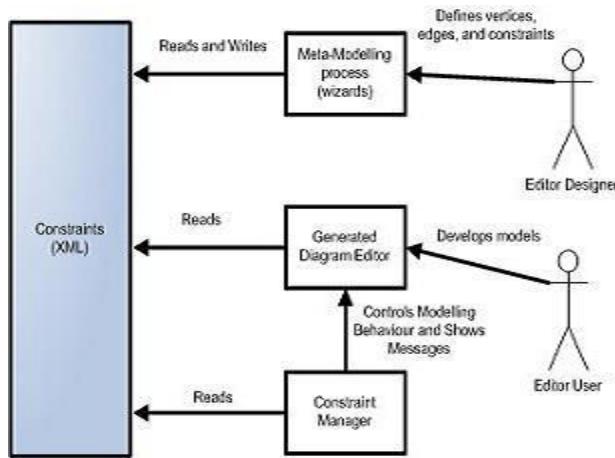


Figure 2.13 Architecture of DECS(Qattous, 2009)

2.4 Discussion and Conclusion

We have introduced and described several related research areas and applications of critics in various domains. These include critics in Information Systems, critics in Software Engineering, critics in education environments and critics in non-software recommender systems. The concept and application of a critic approach is mostly discussed in application domains. We then introduced and briefly explained the concept of constraint definition/specification approaches. Contrasting to the critic approach, the concept and application of a constraint definition is often described in meta-modelling tools environment. A few examples of meta-modelling tools are explained to show the usage of constraint specification approach: MetaEdit+, Pounamu, Marama, and DECS. It seems that a critic-based approach is widely used in application domains whereas the constraint-based approach is often used in meta-modelling tools.

Although critics have been used widely in very diverse domains, to our knowledge, a critic approach has not been applied for meta-modelling tools that implement DSVL tools. While constraint specification/evaluation is common for meta-modelling tools, this is usually at a detailed level, e.g. OCL, scripts or code. The process of specifying and defining constraints for meta-modelling tools is more complex as it requires good knowledge in programming skills, it uses formal approach (e.g. mathematical model), and it involves heavy cognitive load. This

would be difficult for non-programmer users to understand and apply the constraint definition/specification approach.

Therefore, our aim in this research is to extend the capability of our Marama meta-tool set by adding a “critics” mechanism to a meta-tool specification editor. We want to replace the lower-level, complex constraints specification with something more tailored to critic authoring and therefore less general but more user accessible than typical constraint specification techniques. We improve/extend previous work by providing a visual interface for end user developers (specifically less experienced users and, ideally, non-programmers) to author critics for their DSVL tool. Their needs are rather to provide suggestions and modelling tips, to complete and to improve models, rather than hard constraints on model correctness.

The basis of our solution is to associate a critic specification approach (i.e. critic specification editor) with a meta-modelling tool. The critics can be managed in a simple and effective way, while the checking process can be performed according to the modelling process of the target language. Critics within application domains (e.g., FFDC (Oh, et al., 2009), ArgoUML(Robbins & Redmiles, 2000), and FRAMER(Lemke & Fischer, 1990)) do consider constraints as one of the critic specification elements. Since there is no clear difference between *critic* and *constraint*, our critic specification approach will consider constraints as one of the elements that can be defined as a critic. However, our approach will not replace other constraint specification approaches, like code and OCL, but compliment these.

To help us in designing and developing a critic specification editor for our Marama meta-tool set, we reviewed the related research on critic approach and managed to produce a taxonomy of computer-supported critics (Ali, Hosking and Grundy, 2010). The taxonomy is focused on application domains that look at variety of different features and categories of critics against them. The description on the taxonomy is described in Chapter FOUR. The following chapter will describe the steps that we took to achieve our aim that is to develop a critic specification approach in a meta-tool.

Chapter 3

Research Methodology

This chapter describes our approach to designing and prototyping a critic specification tool for domain-specific visual language (DSVL) tools.

3.1 Introduction

Our aim is to design and develop a prototype for a critic specification tool that allows the end user (and other) tool developers to readily express and construct critics. The critic specification tool is embedded within the environment of the Eclipse-based Marama meta-tool (Grundy, et al., 2008) allowing tool developers to concurrently develop visual language environments and critic support for them. Marama is a metatool that is implemented as a set of Eclipse plugins. Our approach to achieving our aim is based on the following methodological steps:

- Conduct a literature review of critic tools, comparing and analyzing their approaches for critic specification and implementation;
- Identify a set of key requirements for a critic specification tool for DSVL tools;
- Develop a prototype to explore the problems and issues in designing a critic specification tool. An iterative-incremental (Robey, Welke, & Turk, 2001) approach has been used for the prototype development to allow for its refinement and improvement;
- Identify from the prototype experience a core set of building blocks needed for a generic critic specification editor and design notation. Design and implement the critic specification tool within a meta-tool (specifically the Marama meta-tool);
- Develop a proof of concept for our critic specification approach by applying it to three DSVL exemplar tools (specifically Marama-based tools) from different domains;
- Perform a user evaluation of the critic specification approach to assess its

- usability and effectiveness;
- Draw conclusions from our survey, design, prototyping and evaluation work.

3.2 Methodology

3.2.1 Literature Review of Critic Tools

The initial step of our research was to review literature concerning critic tools (or critiquing systems). We gathered many articles and reports that described critic tools (or critiquing systems) as a supporting tool for a wide range of computer users in a large variety of domains including education, medicine, CAD and software development. This step allowed us to compare and analyze various critic approaches and identify common properties in critic tools. The aim of this task was to assist us in the development of our own critic specification tool for domain-specific visual language tools. We needed to identify a set of requirements for our critic specification tool and the findings from the literature helped us to obtain these. Furthermore, analysis of the literature led us to generate a taxonomy of computer-supported critics. The review of critic literature is described in Chapter TWO and the taxonomical analysis of the critic tool approaches is described in Chapter FOUR.

3.2.2 Identify a Set of Requirements for Our Critic Specification Tool

Information gathered from the previous stage resulted in the production of a taxonomy of computer-supported critics. Based on this taxonomy we identified properties applied in existing critic tools and these were considered for our critic specification tool. The key critic properties/features are as follows:

- i) *Critic domain*- what domain (s) of discourse is the critic used in (e.g., medical domain, educational domain, and software engineering domain)?
- ii) *Critiquing approach*- does it compare or analyze target domain elements?
- iii) *Critic dimension*- strategies for when a critic should interrupt the user. Is the critic active, passive (invoked on user demand), reactive, proactive etc?

- iv) *Critic type*- does the critic check for completeness, correctness, consistency, alternatives, or a mixture?
- v) *Modes of critic feedback*- how does the tool provide end users with feedback? (e.g., textual representation, graphical representation, 3D-visualization)
- vi) *Types of critic feedback*- suggestions, argument, explanation etc to provide justifications for each identified critic.
- vii) *Critic implementation approach*- how is the critic built or realized in the target tool(s)?
- viii) *Critic rule authoring*- how are the rules embodied in the encoded critic?

The above requirements/properties show the concepts presented in our critic specification tool. These requirements led us to develop a meta-model to describe the valid critic models that the user can build. This meta-model is expressed using an Extended Entity Relationship (EER) diagram which specifies entities and relationships, together with their attributes. The meta-model was then enriched with additional information and constraints. In addition to the above properties, we also defined the following requirements for our critic specification tool to be applied in DSVL tools:

- i. A visual construct/abstraction for specifying critics;
- ii. A visual construct/abstraction for specifying critic feedback;
- iii. A representation for specifying complex critics;
- iv. A representation of visual critic specification notation and environment, embedded within a DSVL tool

The requirements for our critic specification tool are discussed in detail in chapter FIVE.

3.2.3 Develop Prototype to Explore Issues in Designing Critic Specification Tool

We took an iterative-incremental (Robey, Welke, & Turk, 2001) approach to develop prototypes for our critic specification tool. The development of the

prototypes helped us to explore issues and problems in designing the critic specification tool. According to Robey et al. (2001), prototypes are generally produced quickly, and offer appropriate feedback on the feasibility and usefulness of a tool's design and specifications. We had developed several prototypes for our critic specification tool (please refer to Figure 3.1). Our initial attempt was to specify critics using MaramaTatau (Liu, Hosking, & Grundy, 2007), one of the facilities provided in our Marama meta-tool. The critics were specified using the OCL expressions. The difficulties we experienced in the initial attempt had motivated us to develop another prototype. The second prototype was to specify critics at the meta-model level using a similar visual approach to MaramaTatau. We developed a new critic-authoring support extension which provides the ability to specify critics in Marama metamodels. A new functional item, *CriticShape* was added to the Marama meta-model editor and associated with a critic authoring template. Here, critics are specified based on the pre-defined critic authoring template. The limitations we identified from the second prototype had inspired us to improve the critic specification approach. We developed another prototype by creating a new critic specification editor, *Marama Critic Definer*. This critic specification editor is integrated with several form-based interfaces to support the task of specifying critics and feedback. Furthermore, the critic specification editor uses a visual notation approach. These prototypes are described in Chapter SIX and Chapter SEVEN. All of these prototypes were created in the Marama meta-tools through meta-modelling and extended coding, based on which the critic modelling and realisation environments were automatically generated.

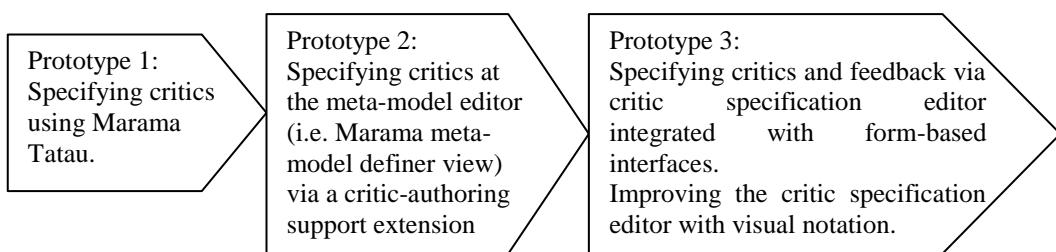


Figure 3.1: Prototype development for critic specification tool

3.2.4 Identify a Set of Building Blocks Needed for a Critic Specification Tool

We identified a core set of building blocks needed for a critic specification tool and designed a notation to represent them. Our use of an iterative-incremental approach led to cyclical refinement of requirements, solutions and prototype development plans. We learned early and efficiently about the building blocks needed in our critic specification tool. Based on the defined building blocks as well as the notation, we had developed the required critic specification tool for DSVL tools as described in Chapter SEVEN.

3.2.5 Proof of concept for the critic specification approach

We proved the effectiveness of our critic specification approach by developing prototypes of visual languages and associated tool support for critic specification for DSVL tools. The critic specification tool prototype was subsequently applied to three exemplars of DSVL tools. These three exemplars were Marama-based tools of different domains: medical (health care planning model, MaramaCPM), business process (enterprise modelling language, MaramaEML) and software design (UML design, MaramaUML). The critic specification tool was integrated into each of these Marama-based tools. The application of the critic specification tool with each of the three exemplars is demonstrated in Chapter EIGHT.

3.2.6 Perform user evaluation of our critic specification approach

We conducted a formal user evaluation to assess the usability and effectiveness of our critic specification approach. The evaluation was carried out with targeted participants who had some basic background knowledge of the Marama meta-tools and who were interested in modelling and the development of modelling tools to support their research work. The methods we employed in our evaluation were: questionnaires, observation and think aloud, and Cognitive Dimensions of Notations framework (CDs). Before the formal user evaluation took place we gained an ethics approval from the University of Auckland Human Participants Ethics Committee. The details of this evaluation are described in Chapter NINE.

3.2.7 Draw conclusions from our survey, design, prototyping and evaluation work

The final step in our research methodology was to draw conclusions from our survey, design, prototyping and evaluation work. These are described in Chapter TEN.

3.3 Conclusions

We have described an overview of our methodological steps in achieving our aim to design and develop a prototype critic specification approach for DSVL tools. Each step in our methodology produced artefacts. These include the critic taxonomy, the prototypes, the evaluation results, and so on. Each of these methodological steps is discussed in details in the following chapters of this thesis.

Chapter 4

A Critic Taxonomy

This chapter describes a new taxonomy for computer-supported critics. We start with an introduction to what a taxonomy is and then explain the concept of a computer-supported critic. We then present our surveyed literature information in terms of our new critic taxonomy. We also describe each of the elements in the taxonomy using various examples from the surveyed literature on critics. We then apply the taxonomy to characterise several exemplar critic tools.

Information gathered from several research efforts on critics (Fischer, Lemke, & Mastaglio, 1991; Fischer, Lemke, Mastaglio, et al., 1991; Irandoost, 2006; Miller, 1986; Oh, et al., 2008; Robbins, 1998; Silverman, 1992) were the initial motivation and basis for the development of our new critic taxonomy. Our contribution is proposing and producing a more comprehensive taxonomy of critics by carrying out an analysis with respect to critics that allows us to better group tools, techniques or formalisms based on their common qualities, features, characteristics and representative elements.

Our intention was for this taxonomy to assist us in designing and developing our own design critics for Marama domain-specific visual language tools. However, it also provides a way to characterise others critics and to compare and contrast a wide variety of computer-supported critic approaches.

4.1 What is Taxonomy?

In the Cambridge dictionary, a taxonomy is “*a system for naming and organizing things ... into groups which share similar qualities*” (<http://dictionary.cambridge.org>). The reason for having a taxonomy is to structure an information repository for browsing. Normally, in a taxonomy, we group properties that share similar values. This chapter presents a new critic taxonomy.

The purposes of this new taxonomy are:

- to provide an overview of the research domain of critics (critiquing systems);
- to capture the features, properties and elements included in the critic domain;
- to characterise concrete critic tools (critiquing systems) and techniques within critic domain;
- to compare critic tools that share the same or similar purpose;
- to identify the differences, strengths and weaknesses of each critic tool.

The following section introduces some critic definitions and examples from various domains. We then describe our taxonomy of critics in the subsequent section.

4.2 Critic Definitions and Examples

Before presenting the critic taxonomy, one should understand some definitions of a critic. The concept of *critic* is one which has been adopted in various domains, including: medical applications (ATTENDING, ONCONCIN), programming (Lisp-Critic, RevJava), design sketching (Design Evaluator), education (Indie, Java Critiquer, Classcompass), software engineering (Argo, ArgoUML), expert and decision support systems (TraumaAID and TraumaTIQ). The term ‘critic’ was initially used by Miller (1986) to describe a software program that critiques human-generated solutions. A “critic” is also often known as a “critiquing system”. However, throughout this thesis we will use the term critic tool instead of critiquing system.

Various critic definitions can be found in the literature. Some of these definitions of a *critic* are shown in Table 4.1. Those definitions normally reflect the type of critics involved in a research effort (Bergenti & Poggi, 2000; de Souza, et al., 2000; Fischer, Lemke, & Mastaglio, 1991; Redmiles, 1998; Robbins, 1998). Each critic tool provides its own definition, but what these critic tools have in common is that they provide knowledge support to users who lack specific pieces of knowledge about their problem or solution domains. These critic tools detect potential problems; give advice and alternative solutions, and possibly automated or semi-automated design improvements to the users. Robbins’s report also lists several definitions of critics or critiquing systems. Thus, critic tools offer an important

approach to facilitating human-computer collaborative problem solving (Tianfield & Wang, 2004). Table 4.2 shows some examples of critic tools and their domain applications.

Table 4.1: Critic definitions.

	Definition	Defined by (year)
1.	“A critic is a system that presents a reasoned opinion about a product or action generated by a human.”	(Fischer, Lemke, & Mastaglio, 1991)
2.	“A design critic is an intelligent user interface mechanism embedded in a design tool that analyzes a design in the context of decision-making and provides feedback to help the designer improve the design.”	(Robbins, 1998)
3.	“Critics are agents that watch for specific conditions in the partial design as it is being constructed and notify the designer when those conditions are detected.”	(Redmiles, 1998)
4.	“Critiquing systems are meant to provide critiques on existing artifacts to improve their realization. They rely on analyzing existing artifacts and on suggesting improvement rules.”	(Bergenti & Poggi, 2000)
5.	“A critiquing system is a software that monitors the user’s action and triggers a signal when any action activates the critic rules of “bad design”.	(de Souza, et al., 2000)

Table 4.2: Examples of critic tools and their application domain.

Tool Name (year-based on published paper)	Description	Application domain
ArgoUML (2000)	“Critiquing is done continuously and designers need not request that critics be applied or even know that any particular critic exists.” (Robbins & Redmiles, 2000)	Software engineering (UML designs)
ABCDE-Critic (2000)	“...implements a construction kit supporting UML class diagrams, an argumentative hypermedia system, and a critic system, where the user is able to define his own critics”(de Souza, et al., 2000)	Software engineering (Class diagram design)
IDEA (2000)	“...is a critiquing system that we developed to work in direct interaction with the software architect to propose pattern-specific critiques” (Bergenti & Poggi, 2000)	Software engineering (design patterns)
RevJava (2002)	“it is used to analyze and critique object oriented software.” (Florijn, 2002)	Software engineering (object-oriented Java)
DAISY (2003)	“...a critiquing system is able to check the consistency of models created during domain and application engineering”(de Souza, et al., 2003)	Software engineering (software modelling)
JavaCritiquer (2003)	“a critiquing system to teach students how to write clean, maintainable and efficient code.” (Qiu & Riesbeck, 2003)	Education (Java programming)

Design Evaluator (2004)	“is a pen-based system that provides designers with critical feedback on their sketches in various visual forms.” (Oh, et al., 2004)	Design engineering (design sketching)
ClassCompass (2007)	“an automated software design critique system with critics that comment on high-level design issues rather than diagram completeness” (Coelho & Murphy, 2007)	Education (software design)
FFDC (2009)	“...as a step toward creating computer-based critics that support design learning in studio setting” (Oh, et al., 2009)	Education (Architecture design)
HeRA (2009)	“a feedback centric requirements editor to help analysts to control the information overload” (Knauss, et al., 2009)	Software engineering (Requirements engineering)

4.3 A Critic Taxonomy

Several articles and reports have been published to explain and discuss critics (or critiquing systems) as a supporting tool for a wide range of computer users. The process of developing our critic taxonomy began by examining the related literature in critics (Fischer, Lemke, & Mastaglio, 1991; Fischer, Lemke, Mastaglio, et al., 1991; Irandoost, 2006; Miller, 1986; Oh, et al., 2008; Robbins, 1998; Silverman, 1992). We classified the information collected from the critic literature in the following groups, which were tailored to meet our specific needs. Figure 4.1 illustrates the groups and elements that make up our critic taxonomy. The groupings and their elements are described in detail in the following subsection.

- Critic domain
- Critiquing approach
- Critic dimension
- Critic type
- Modalities of critiques
- Types of feedback
- Critic realization approach
- Critic rules authoring

Our taxonomy aims to be applicable to critics in general though most of our motivation, applications and examples come from CSE (Critics in Software Engineering).

Critic Groups and Elements						
1. Critic Domain						
2. Critiquing Approach	3. Modes of Critic Feedback	4. Critic Rule Authoring	5. Critic Realisation Approach	6. Critic Dimension	7. Types of Critic Feedback	8. Types of Critic
Comparative critiquing	Textual	Insert new critic rule	Rule-based	Active	Explanation	Correctness critics
Analytical critiquing	Graphical & 3-Dimension Visualization	Modify critic rule	Predicates	Passive	Argumentation	Completeness critics
	Multi-modal	Delete critic rule	Knowledge-based	Reactive	Suggestions	Consistency critics
		Authoring rule facility	Pattern-matching	Proactive	Examples (or precedents)	Optimization critics
		Enable/ disable critic rules	Programming code	Local	Interpretations	Alternative critics
			Object constraint language (OCL)	Global	Positive	Evolvability critics
					Negative	Presentation Critics
					Constructive	Tool critics
					Demonstration	Experiential critics
						Organizational critics
						Pattern critics
						Structure critics
						Naming critics
						Metric critics

Figure 4.1: Our critic taxonomy.

4.3.1 Critic Domain

The first group in the critic taxonomy is the Critic Domain. A domain is defined as a knowledge area characterised by a group of problems with similar techniques, operational and functional specifications. Usually a domain represents a set of well-defined and coherent concepts and functions. Examples of domains are medical, business process, education, software engineering and design environment, among others. Critics are specified based on the domain knowledge of that particular environment/area. In order to define and specify critics, it is required that we understand the domain that we deal with. Only by understanding the domain knowledge will one be able to define and specify meaningful critics for that particular context/domain. The use and context of critics varies from one domain to another. To date, critics have been applied in various domains. Several research efforts (Fischer, Lemke, & Mastaglio, 1991; Fischer, Lemke, Mastaglio, et al., 1991; Irandoost, 2006; Miller, 1986; Oh, et al., 2008; Robbins, 1998; Silverman, 1992) provide either long or short description of critics from different domains. Table 4.3 shows some of the well-known critics from various domains that received much attention in critic research reports and articles (Fischer, Lemke, & Mastaglio, 1991; Fischer, Lemke, Mastaglio, et al., 1991; Irandoost, 2006; Miller, 1986; Oh, et al., 2008; Robbins, 1998; Silverman, 1992). Apart from those domains listed in Table 4.3, critics have also been applied in domains such as education (Indie, JavaCritiquer); design sketching (Design Evaluator); decision making (DecisionLab); architectural design (ICADS) and word processing (COPE).

Table 4.3: Critics applied to various domains.

Domain	Critic system (year)
Medical	ONCOCIN: clinical consultation system (1983) ATTENDING : medical support (1986) TraumaTIQ: treatment of medical trauma cases (1993) AIDA : antibody identification (1995)
Engineering	CRITTER: digital circuit design (1985) Design Advisor: integrated circuit design(1988) CLEER: placement of antennas on military ships (1992) SEDAR: civil engineering (1995)
Design environment	JANUS: kitchen design (1989)

	FRAMER: user interface window layout (1989) KRI/AG: graphical user interface design (1992) VDDE: voice dialog design (1993)
Programming	PROLOG Explaining: explanation of PROLOG code (1984) Lisp-Critic: writing LISP programs (1987) GRACE system: COBOL programming (1990)
Software engineering	KATE : software specifications (1988) Argo family: software development (1996)

We will not describe these critics because the details can be found in (Fischer, Lemke, & Mastaglio, 1991; Fischer, Lemke, Mastaglio, et al., 1991; Irandoost, 2006; Miller, 1986; Oh, et al., 2008; Robbins, 1998; Silverman, 1992). Our objective is to show that critics are applicable to various domains and problems have proved to be one of the effective mechanisms in providing feedback to users. However, in Chapter TWO, we described several key related works in critics.

4.3.2 Critiquing Approach

The Critiquing Approach is the second group our taxonomy. Elements in this group are comparative and analytical critiquing. Critiquing is a way to generate valid reasoning about a product or action (Fischer et al. 1991). Reports and articles from (Fischer, Lemke, & Mastaglio, 1991; Irandoost, 2006; Oh, et al., 2008; Qiu & Riesbeck, 2008; Robbins, 1998; Silverman, 1992) have identified that critic tools commonly use a comparative critiquing, analytical critiquing or both as their critiquing approaches.

In a comparative critiquing approach (Fischer, Lemke, & Mastaglio, 1991; Robbins, 1998), complete and extensive domain knowledge is essential to generate good solutions. When a user recognizes potential problems in a design, the critic tool will then produce an optimal result from the predefined solutions in the system. The user-proposed design is then compared with the system's solution. The comparison will result in a report of the differences between the two solutions. Robbins (1998) points out that a comparative approach can cause difficulties when several good solutions exist and each of the solutions are different from each other. Furthermore, certain domains allow radically different but equally valid solutions (Fischer et al.,

1991). A user also can be discouraged if the system generates its solution without recognizing the user's solution approach. As Fischer et al (1991) point out the critic can only declare that the system solution accomplishes good results if the user and system's solutions differ in a fundamental way. However, it cannot clarify why the user's solution is less than optimal. In a way, it hinders the exploration of different alternatives that may be good enough. In addition, Robbins (1998) also states that a comparative approach can direct users to make their work like the one that the system proposed (Robbins, 1998). Hence, this approach guides the user to a known solution (Robbins, 1998). Besides, the critics authoring is relatively intuitive and straightforward for this approach because it allows authors to write down problems and answers and the system will take care of comparison and feedback generation (Qiu & Riesbeck, 2008). For example, TraumaTIQ (Gertner & Webber, 1998) supports a physician's treatment planning. TraumaTIQ interprets the physician's goal treatment plan, evaluates the inferred plan structure by comparing it to the system's recommended treatment plan, and finally generates a critique that addresses potential problems (Gertner & Webber, 1998).

In an analytical approach (Fischer, Lemke, & Mastaglio, 1991; Robbins, 1998), as long as the domain knowledge is sufficient then solutions can be generated. Hence, this approach can be applied in domains where knowledge is incomplete. In general, this approach uses rules to detect potential problems in the design and change them into *assistance opportunities* (Robbins, 1998). Thus, in a way it guides the user away from recognised problems (Robbins, 1998). Unlike comparative critiquing, this approach does not generate solutions on its own but instead analyses the user-proposed solution to identify any potential problems via set of rules.

It is not easy to author critics in an analytical approach though it is applicable in a broad range of domains. This is because one needs to write rules for all the problems in all situations (Qiu and Riesbeck, 2008). Thus, as Fischer et al. (1991) state, analytical critics can be built incrementally and applied throughout the design process. According to Oh et al. (2008), analytical critiquing supports exploratory problem solving better than comparative critiquing does because design problems rarely have one right answer. For instance, Argo is an analytical critic tool that uses

analysis predicates, goal and decision type attributes to identify undesirable designs and then generates feedback items with more kinds of design context, such as providing contact information for relevant experts and stakeholders (Robbins & Redmiles, 1998).

One critic tool that applied both of these critiquing approaches is UIDA (User Interface Design Assistant). UIDA is a system that critiques user interface window layouts (Bolcer, 1995). UIDA performs analytical critiquing by applying 72 style rules written in an OPS5-like language and comparative critiquing via recording and comparing the particular set of rules satisfied by each layout (Bolcer, 1995).

According to Irandoost (2006), the choice of a critiquing approach depends largely on application domain, the characteristics of the task it supports and the cognitive support needs of the user. The differences of these two approaches are summarised in Table 4.4.

Table 4.4: Differences between comparative and analytical critiquing.

Comparative critiquing	Analytical critiquing
• Requires a complete and extensive domain knowledge to generate a solution	• Does not require a complete domain knowledge to generate a solution
• Uses a differential analyzer (Silverman, 1992)	• Uses rules to detect potential problems in user-proposed solution (Robbins, 1998)
• Generates its own optimal solution, then compares it with the user-proposed solution (Fischer et al, 1991)	• Critiques the user-proposed solution with respect to predefined features and effects (Fischer et al.1991)
• Guides the user to known solution (Robbins, 1998)	• Guides the user away from the recognized problems (Robbins, 1998)
• More suitable for well-structured domains (Oh et al, 2008)	• Can be applied to a broader range of domains (Robbins, 1998)
• Less intrusive	• More intrusive
• Easy to author critics	• It is not easy to author critics
• Example of tools: ATTENDING, TraumaTIQ	• Example of tools: JANUS, Argo

4.3.3 Modes of Critic Feedback

The third group in our taxonomy is the Modes of Critic Feedback. Elements in this group consist of textual, graphical and 3D visualisation, and multi-modal. Presenting critic feedback (Irandoost, 2006) (also known as feedback or critiques) is another element to be considered in the design of a critic tool. Most critics provide critic feedbacks in textual messages. However, graphics can be used as well for presenting critic feedback. Silverman and Mehzer (1992) point out that critic feedback should be textual and visual because it usually provides the most effective results. Thus, critic designers/developers should use visual wherever possible to deliver critique instead of text. Oh et al. (2008) recognise three modes used for presenting critic feedback in existing critic tools: text messages, graphic annotations and three dimension (3D) visualizations. Text message refers to a critique that is presented in a written form. Graphic annotation refers to a critique that is presented in a graphical form. 3D visualizations involve critiques that are presented via images, or diagrams in a three dimension format. We add another element in this group i.e. multi-modal mode to include animation, sound, and maybe movies to represent critiques.

Several researchers have explored the combination of textual, graphic and 3D visualizations for critique presentation in their critic tool. For instance, Oh et al. (2004) develop Design Evaluator; a pen-based critic tool that generates critiques and displays them in textual and visual format. The Design Evaluator involves two design domains: architectural floor plans and Web page layout design. These two design domains have different methods of displaying critiques. The Architectural Design Evaluator display critiques in three ways: as text messages, annotated drawings and texture-mapped 3D models. When a designer selects a text message critique, the tool shows the critiques in two other forms, such as graphic annotation on a designer's floor plan diagram and generates a 3D texture-mapped VRML (Virtual Reality Model Language) model that shows the path via the floor plan. The Web page Design Evaluator also generates text critiques which are linked to visual critiques via sketch annotation and design examples or cases. Similarly, de Souza et al., (2000) present the **Annotation Based Cooperative Diagram Editor (ABCDE)-Critic**, a system that has a construction kit to support UML class diagrams, a

hypermedia system, and a critic system. Apart from the textual critiques, ABCDE-Critic provides graphic annotation on a UML class diagrams, such as mark (and unmark) in a different colour on the diagram elements that are detected as error/problem (de Souza et al., 2000). Stove (1994) developed the PetriNED (**Petri Net EDitor**) prototype to prove that visual critiques are possible. PetriNED (**Petri Net EDitor**) is a design environment supporting the design of Petri Nets. For example, a user constructs a Petri net model of a communication protocol. During the model construction, the user violates the ‘alignment critics’. Thus, the tool will notify the user about the error by drawing lines between the involved objects in the model.

A number of critic tool researchers argue that communicating design information in a mixture of graphical critiques with text critiques is likely to be more effective than selecting one mode (Oh et al., 2004, Silverman & Mehzer, 1991).

4.3.4 Critic’s Rule Authoring

The fourth group in the taxonomy is the Critic Rule Authoring. Elements in this group are: insert new critic rule, modify critic rule, delete critic rule, enable and disable critic rule, and critic rule authoring facility. Critic rules are one of the important components in building critics. In general, critics are composed of a single rule or groups of rules (or procedures) to evaluate different aspects of a product or design in a domain (Fischer, Lemke, & Mastaglio, 1991). Thus, critic rules have to be written for an individual product or design as well as for the critic system as a whole. According to (Oh, et al., 2008), critic rules are normally written in advance by the system designers to develop a critic system. It is often hard or impossible for a user to modify the existing rules or add new critic rules after the critic system is deployed (Oh, et al., 2008; Qiu & Riesbeck, 2004). However, as Irandoost, (2006) and Oh et al., (2008) pointed out, critiquing capacity and issues may need to be adjusted from time to time in various situations. Furthermore, (Fischer, Lemke, Mastaglio, et al., 1991) emphasis that users should not be required to have comprehensive programming knowledge in order to perform the modification of critic rules. For these reasons it is important to allow users to understand the critic

rules and be able to modify and expand the rules by authoring new rules to incorporate in a critic system.

Riesbeck and Dobson (1998) and Qiu and Riesbeck (2003, 2004, and 2008) have explored the issue of authoring critic rules for educational critic system. Riesbeck and Dobson (1998) developed INDIE (Investigate and Decide) systems, an authoring tool for intelligent interactive education and training environments. It allows users (teachers) to author and control the critic rules (Riesbeck and Dobson, 1998). Qiu and Riesbeck (2004) developed an educational critic tool for Java programming, called Java Critiquer. They explored the question of how users can author critic rules. Their Java Critiquer system provides authoring capability, so that users (teacher) can check or modify the critiques in addition to the feedback that Java Critiquer generates (Qiu & Riesbeck, 2004). The tool also allows teachers to gradually enter and update critic knowledge during real use of the system.

Some of the tools that allow for customization of critic rules include ArgoUML, IDEA, Design Evaluator, and ABCDE-Critic. For instance, ArgoUML (Robbins and Redmiles, 1998) provides a class framework, source code templates and examples to support critic implementers. Authoring a new critic requires selecting a starting template, filling in relevance and timeliness attributes, coding an analysis predicates and writing a headline and brief description (Robbins & Redmiles, 1998). In IDEA (Bergenti & Poggi, 2000), the engineer can provide new patterns and new rules to select and fire new critics. Similarly, the Design Evaluator (Oh et al., 2004) allows an end-user (designer) to inspect and edit the rule expressions which are stored in a list. ABCDE-Critic (de Souza, et al., 2000) also allows the user themselves to add critics to the critic system, through its first-order production system.

The capability of rule authoring is to enable end-user designers to construct and store their own critic rules (Oh, et al., 2008). A rule authoring facility will allow critics to deal with various conditions and authorises end-user designers to add to the system's feedback process (Oh, et al., 2008).

4.3.5 Critic Realisation Approach

The Critic Realisation Approach is the fifth group in our taxonomy. This group is about implementing critics by using specific approaches. In order to support critic development, several approaches have been applied to designing and realising critics. Critics implementation in various domains uses a variety of approaches as outlined below.

- **Rule-based approach.**

Critics implemented with a rule-based approach consist of a condition and an action. Rules are defined using the IF-THEN format. The **IF** part of a rule is a *condition* (also called a premise or an antecedent), which tests the truth value of a set of facts. If the *condition* is true, then the **THEN** part of the rule (also called the *action*) is performed. Actions can include suggestions, explanations, argumentations, messages or precedents of problems. Rules in a rule-based approach are also known as *production rules*. They tends to be easy to use and to understand once implemented(Tyugu, 2007).

For instance, ABCDE-Critic (de Souza, et al., 2000) uses rule-based expression to specify critics that comment on UML class diagram-based designs. The critic tool invokes critics when a condition clause is found to be true in the current design parts warning a user that the design possibly have error (de Souza, et al., 2000). It was stated that the rules can be coded in Java, JEOPS (*Java Embedded Object Production System*), or Prolog, according to the critic type (de Souza, et al., 2000).

- **Knowledge-based approach.**

In general, a knowledge base contains set of rules and associations of compiled data which most often take the form of IF-THEN rules (production rules). The knowledge base represents the most important component of a knowledge-based system. The format of the knowledge refers to how this knowledge is represented internally within the knowledge base system so that it can be used in problem-solving. Several knowledge representation schemes that are commonly used: predicate, rules, frames, associative networks and object.

For instance, FRAMER (Robbins, 1998) enables designers to develop window-based user interfaces on Symbolics Lisp machines. FRAMER's knowledge base contains design rules for evaluating the completeness and syntactic correctness of the design as well as its consistency with interface style guidelines. In another example, the IDEA (Interactive Design Assistant) tool (Bergenti & Poggi, 2000) produces design pattern critics implemented with Prolog rules that are directly integrated with a knowledge base. Bergenti and Poggi (2000) stated that the knowledge base of IDEA is comprised of a set of design rules, corresponding critics, and a set of consolidation rules. However, the rules for creating the pattern-specific critics are not easy as it requires a high-level of understanding of design patterns and detailed knowledge of the Prolog and knowledge base structures. Furthermore, Robbins and Redmiles (1998) point out that a knowledge-based approach is more appropriate for design support where the user may lack needed knowledge.

- **Pattern-matching approach.**

According to (Trochim, 1989), a pattern “is any arrangement of objects or entities.” A pattern matching process often involves an attempt to relate two patterns where one is a theoretical pattern and the other is an operational one (Trochim, 1989) or it can consist of left-hand side and right-hand side rules. The most common form of pattern matching involves strings of characters. In many programming languages, a particular syntax of string is used to represent regular expressions, which are patterns describing string characters. For instance, the Java Critiquer tool performs automatic critiquing using a pattern matching approach (Qiu & Riesbeck, 2008). When a pattern is matched, its corresponding critique is inserted right below the problematic Java source code. Two types of patterns are supported in this tool: general regular expressions and JavaML patterns. Regular expression patterns are practical for short text segments and can be used directly to the Java source code. However, according to (Qiu & Riesbeck, 2008), regular expressions can become quite difficult. Thus, a built-in pattern editor is provided to support teachers in the incremental authoring of patterns. The authoring of

JavaML patterns can be more direct and simpler compared to regular expression. Qiu and Riesbeck (2008) claim that the critic rules in the Java Critiquer are written in a type of XML format called LMX (language for Mapping XML). The left-hand side of a rule is a LMX pattern and the right-hand side of a rule is a critique. The “pattern matcher” matches the patterns in the rules against the JavaML code, and returns a list of triggered critiques (Qiu & Riesbeck, 2008).

Figure 4.2 shows an example of a critic rule written using this pattern matching approach.

```

<lmx:pattern>
  <lmx:lhs>
    <if srcEnd="$srcEnd1;">
      <test srcBegin="$srcBegin;" srcEnd="$srcEnd;">
        <lmx:extension class="lmx.extension.SegmentMatch"/>
      </test>
      <true-case>
        <return><literal-boolean value= "true"/></return>
      <true-case>
      <false-case>
        <return><literal-boolean value= "false"/></return>
      <false-case>
    </if>
  <lmx:lhs>
  <lmx:rhs>
    <critique pos= "$srcEnd1;">
      <text>
        There is more code than you need to write. You already have a boolean value. Just write
        <code>return <srcCode srcBegin= "$srcBegin;" srcEnd= "$srcEnd;"/>;</code> instead. You never
        need to write an IF to return true in one case and false in the other.
      <text>
    </critique>
  </lmx:rhs>
</lmx:pattern>

```

Figure 4.2: Critic rule using pattern-matching approach (Qiu&Riesbeck 2008).

- **Predicate Logic.**

According to Tyugu (2007), predicate logic is based on the idea that “sentences (propositions) really express relationships between objects as well as qualities and attributes of such objects (can be people, other physical objects, or concepts).” Such relationships or attributes are called predicates. The objects are called the arguments or terms of the predicate. The use of terms allows a predicate to express a relationship about many different

objects rather than just a simple object (Tyugu, 2007). By using predicates we can express more complex statements about the world than we could with propositions. Predicates can also be used to represent an action or an action relationship between two objects (Tyugu, 2007).

One example of critic tools that applies predicates approach is the Design Evaluator (Oh et al., 2004). The Design Evaluator contains three layers known as *Description*, *Evaluation*, and *Visualization*. The *Evaluation* layer evaluates sketches with predicates that embody design rules. The tool compares the recognized spatial information with each rule. If it finds a rule violation, it generates a design critique to be displayed in the *Visualization* layer (Oh et al., 2004).

In the *Evaluation* layer, rules are coded as Lisp predicates that apply to the design objects. The rule expressions are stored in a list that the end user (designer) can inspect and edit. Figure 4.3 shows the example of a rule for architectural floor plans domain.

- Rule statement: A ward be no smaller than 10,000 area units
 A minimum area rule: express a minimum area requirement about a specific room.
`(<Minimum-area><room><minimum-size>)`
`(<MINIMUM-AREA WARD 10000)`
- Rule Statement: typical room placement in hospital design that states ER, TRIAGE, CLINICAL-FOR-OUTPATIENT, and DAYWARD should be placed in the CLINICAL-ZONE
 A room placement rule: all rooms in the list inside the inner parentheses should be in (or not in) the given zone.
`(<Placement-rule>`
`<Zone>(<Room><Room><Room>...))`

Figure 4.3: Rules for architectural floor plans using predicate style (Oh, et al., 2004).

- **Object constraint language (OCL) expressions**

According to (Kleppe & Warmer, 2002), Object Constraint Language (OCL) is a language that offers ways to specify the semantics of an object-oriented model in a very accurate style. The semantics are expressed in invariants and pre-and-post conditions, which are all types of constraints (Kleppe & Warmer, 2002). OCL can be used to construct logical expressions that access attributes, invoke operations, navigate along associations, and manipulate collections (Cook et al., 1999). A research of model checking by (Bezivin & Jouault, 2006) demonstrates the use of OCL to express constraints via a simple domain-specific language (DSL) called Class Diagrams (CD). (Bezivin & Jouault, 2006) argue that OCL needs extensions to support additional elements such as the severity of a constraint attached to constraints. A severity is a representation of a flaw degree in a problem that can be classified either as an *error*, a *warning* or a *critic* (Bezivin & Jouault, 2006). Thus, in their CD example, they show how a critic is expressed using an OCL expression. Figure 4.4 shows the examples of critics using OCL expressions (Bezivin & Jouault, 2006).

- Critic statement: the name of Classifier must be unique within its package
OCL expression:

Context Classifier

Inv: `not self.package.contents->exists (e|(e <> self) and (e.name = self.name))`

- Critic statement: the name of a Classifier should begin with an upper case letter.

OCL expression:

Context Classifier

Inv: `not (let firstChar: String = self.name.substring(1,1) in firstChar <> firstChar.toUpperCase())`

Figure 4.4: Critics written in OCL expressions (Bezivin & Jouault, 2006).

- **Programming code.**

Critics can also be designed and realised through the use of programming code. For instance, critics in Argo/UML (Robbins & Redmiles, 2000) are coded as Java classes sub-classed from class Critic. Class Critic defines several methods that may be overridden to define and customize a new critic. Each critic's constructor specifies the headline, problem description, and relevant decision categories. The central method is a predicate that accepts a design element to be critiqued and returns true if a problem is found (Robbins and Redmiles, 2000). RevJava (Florijn, 2002) is another tool that implements critics via programming code, i.e. Java class files. The tool is used to analyse and critique object oriented software.

4.3.6 Critic Dimension

The sixth group in our critic taxonomy is the Critic Dimension. Critics can be classified by various dimensions. The elements within this group are based on Fischer's suggestion (Fischer, 1989). Report and articles from Qiu and Riesbeck (2008), Oh et al. (2008), Irandoost (2006) and Robbins (1998) support Fischer's suggestions on critic classification dimensions. Our taxonomy's critic classification dimensions are shown in Table 4.5.

Table 4.5: Critic dimensions (Fischer, 1989).

Critic dimension	Brief description
Active critics	Continuously critique the user's design/work
Passive critics	Wait until the user asks for a critique
Reactive critics	Critique the design/work that the user has done
Proactive critics	Guide the user by presenting guidelines before the user makes a decision
Local critics	Critics that evaluate individual design elements
Global critics	Critics that consider interactions between most or all of the elements in a design

In a critic development, a critic designer has to consider using active critics, passive critics or both in their tool. An active critic (Fischer, 1989) usually continuously

monitors user tasks, warns the user as soon as a critic rule is violated and then offers critic feedback (a critique). An active critic makes users aware of their unsatisfactory design/work when the potential problem is easy to correct. However some users may find it a distraction to have something continuously criticise them without giving them a chance to develop their own design/work and corrections.

In contrast to active critics, a passive critic (Fischer, 1989) only works when a user asks for a check of critic rule violation. In this scenario, after the user completes preliminary design/work, the user then asks for evaluation of the design/work. Passive critics are less intrusive compared to active critics because they allow the user to control when to activate the critics. The problem with passive critics is that most of the time, the user does not activate them early enough to prevent potential problems (Qiu&Riesbeck, 2008). Fischer (1989) remarks that active critics are suitable for guiding novice users and passive critics seem to be good for intermediate users.

ArgoUML provides active critics when a user attempts to draw a design diagram. For example, when a user selects a new class to place in the class diagram design, several critics fire to indicate that part of the design has been started, but not yet finished. Java Critiquer uses passive critics because as Qiu and Riesbeck (2008) stated that it is not a requirement to avoid students from making mistakes. Thus, Java Critiquer provides such an opportunity for learning and allows students to concentrate on their programming tasks without interruption (Qiu and Riesbeck, 2008).

Apart from active and passive critics, there are critic tools that use either reactive or proactive critics. A reactive critic (Fischer, 1989) provides critiques on the user's accomplished design/work, whereas a proactive critic attempts to lead the user before the user makes a specific decision. Similar to these two critics are the critic dimensions suggested by Silverman (1992): *before*, *during* and *after*. Silverman's *before critic* is similar to Fischer's proactive critic. *During* and *after critics* can be viewed as Fischer's reactive critics. However, a *during* and *after* critic is different in terms of whether a user's work is completed or not. The SEDAR (Fu et al., 1997)

tool adopts Silverman's dimensions and takes all three strategies: *before* (error prevention), *during* (design review critic, design decision) and *after* (error detection). The HeRA tool (Knauss, et al., 2009) provides proactive support because while a user is typing the requirements, it analyzes the input and warns the user of any ambiguities or incomplete specification detected.

Finally, critics can be classified as either local or global critics. Local critics (Fischer, 1989) are critics that evaluate individual design elements and global critics (Fischer, 1989) involve the interactions between most or all of the elements in a design. For instance, the HeRA tool (Knauss, et al., 2009) provides users with local and global critics. According to ((Knauss, et al., 2009), the local critics of the tool is concerned with the current focus of the requirements editor (i.e. requirements, use cases, and a glossary), while global critics allow users to analyse a global perspective in terms of list of all critiques and inference of global process diagrams (i.e. UML Use Case Diagram, Event-driven Process Chain models, and Use Case Point View).

4.3.7 Types of Critic Feedback

The next group in our taxonomy is the Types of Critic Feedback. There are ten elements in this group: explanation, argumentation, suggestion, example (or precedent), interpretation, simulation, demonstration, positive feedback, negative feedback, and constructive feedback. There are many ways to present critic feedback (Irandoost, 2006) (also known as feedback) in a critic tool. Oh et al., (2008) describes the types of critic feedback as one aspect of the critic's intervention techniques. Critic tools can offer critic feedback to users by choosing the appropriate techniques from the ten elements. However, the most widely used techniques are explanation, suggestion, and argumentation.

The explanations technique is widely used in most critic tools. Explanation as defined in the Cambridge dictionary is “*details or reasons that someone gives to make something clear or easy to understand*”. Thus, critiques provided by a critic tool must produce explanations so that user has the chance to assess the details and reasons before making a decision as whether to accept the critique generated by the

tool. The explanations can be focused on the violations of general guidelines or the differences between the user's design solution and system's solution (Fischer et al, 1991). Having an explanation facility is also needed to show the correctness and usefulness of the critic tool's recommendation (Irandoost, 2006). Furthermore, it is essential to validate a critique via explanation because without valid details or reasons, a user will not accept the critique. In a way, it shows the user acceptance towards the critiques generated by the critic tool.

The explanation provided by a critic tool can be in simple or in-depth explanations. A simple explanation component normally provides pre-stored text explanations. In detailed explanations, hypertext techniques have been shown to be very efficient for providing contextualization explanations (cf. Irandoost, 2006). Fischer and colleagues contribute the incorporation of hypertext into critic's feedback loop and the creation of what they call "minimalist explanation" (Fischer et al., 1990). Via hypertext jumps, the user can obtain more in-depth explanations. Explanations too can be represented textually visually or both.

Argumentation is another option for offering critic feedback. It is also another mechanism for explanation where it can contain issues, answers, and arguments about a product or design domain. A user, who may not understand critiques offered by a critic tool, may want to know more information about the critiques. Thus, via an argumentation component, the user can obtain the required information to justify the critique. Examples of critic tools that provide an argumentation style are Indie (Riesbeck & Dobson, 1998), ABCDE-Critic (de Souza et al., 2000) and HeRA (Knauss et al., 2009). These tools are developed for the domains of education learning, object-oriented analysis and design, and requirements engineering.

Indie (Investigation and Decide) is an authoring tool that provides support for the intelligent interactive education and training environments. The authoring tool helps authors (i.e. teachers) to create knowledge bases for critiquing student arguments. Basically the student's argument is compared against the argument model via the Indie *Critiquer* modules. One of the knowledge bases in the Indie tool has argument models with the purpose of describing what makes good and bad arguments for

every possible decision. The argument contains a claim about a scenario, and a set of evidence which hold scenario facts. The ABCDE-Critic (de Souza et al., 2000) incorporates an *argumentative hypermedia system* to provide in-depth explanation for user that does not understand or wants more information about critics. The argumentation component contains issues, answers and arguments about the design domain (de Souza et al., 2000). Likewise, HeRA (Knauss et al., 2009) facilitates its computer-based critiques via the argumentation component. The argument component allows users to adhere to warnings or to argue against them (Knauss, et al., 2009).

Some critics offer suggestions to change the user's solution. The suggestion style approach is also known as *solution-generating critics* (Fischer, Lemke, & Mastaglio, 1991) which are capable of suggesting alternatives to the user's solution. An example is the JANUS system, where a simple problem detecting critic points out that there is a stove close to a door. Another option is to provide examples (precedents) to support critics. Examples are a way of helping users to understand something by showing them how it is used. For example, the Design Evaluator (Oh et al., 2004) provides an exemplar Web page for the designer to look at when a critique is selected.

Another option for presenting critic feedback is either to provide positive or negative feedback. A positive feedback provides a critique in a praising way when a user produces a good design/solution. A negative feedback is a complaint when a user produces a poor design/solution. Positive and negative feedback is actually related to how humans make decisions because humans tend to judge/evaluate something based on advantages and disadvantages, pros and cons. In PetriNED (Stolze, 1992), positive critiques are delivered in a graphical way and close to the user's focus of attention. This is helpful to those users who are interested in obtaining positive feedback.

Apart from the styles stated above, critic feedback can be presented through the use of a simulation component or demonstration (e.g. JANUS, HeRA), interpretation (Nakakoji et al. 1993), and constructive feedback (ArgoUML). A mixture of styles

in presenting critic feedback (critique) certainly facilitates users/designers to clarify their understanding, as well as improve their knowledge.

4.3.8 Critic Types

Finally, the last group in our taxonomy is the Types of Critic. Critics can be classified according to the type of domain knowledge that they present (Robbins & Redmiles (1998); Robbins (1998)). Thus, the Critic Domain group and this group complement to each other. Table 4.6 shows a list of critic types we define in our taxonomy.

Table 4.6: Critic types

Critic type	Description
Correctness critics	detect syntactic and semantic flaws (Robbins & Redmiles, 1998)
Completeness critics	remind the designer to complete design tasks (Robbins & Redmiles, 1998)
Consistency critics	point out contradictions within the design (Robbins & Redmiles, 1998)
Optimization critics	suggest better values for design parameters (Robbins & Redmiles, 1998)
Alternative critics	prompt the architect to consider alternatives to a given design decision (Robbins & Redmiles, 1998)
Evolvability critics	address issues such as modularization, that affect the effort needed to change the design over time (Robbins & Redmiles, 1998)
Presentation critics	Look for awkward use of notation that reduces readability (Robbins & Redmiles, 1998)
Tool critics	inform the designer of other available design tools at the times when those tools are useful (Robbins & Redmiles, 1998)
Experiential critics	provide reminders of past experiences with similar designs or design elements (Robbins & Redmiles, 1998)
Organization critics	express the interest of other stakeholders in the development organization (Robbins & Redmiles, 1998)
Pattern critics	Improve a design via design patterns (Bergenti & Poggi, 2000)
Structure critics	detect problems that involves structural properties (Coelho & Murphy, 2007)
Naming critics	identify potential sources of confusion introduced by names (Coelho & Murphy, 2007)
Metric critics	Report when the number of occurrences of some aspect of a design is beyond normal values (Coelho & Murphy, 2007)

According to Robbins (1998) critic types are descriptive rather than definitive. In fact, new categories can be defined based on the application domain. For instance, IDEA (de Souza et al., 2000) offers pattern-specific critiques to assist the architects in finding and improving the realisations of design patterns in UML designs. Similarly, (Coelho & Murphy, 2007) define three categories of critics: structure critics, naming critics and metric critics for the ClassCompass tool.

4.4 Applying the Taxonomy

In this section, we apply our new critic taxonomy to position several critic tools within the critic domain. Several systems and tools that adopt or implement the critic concept have been identified and selected randomly regardless of whether they are research prototype tools, commercial tools or open source tools. We apply our taxonomy to this set of tools which has been shown previously in Table 4.2. We briefly explain each of the tools in the following section and characterise them with our taxonomy dimensions.

4.4.1 ArgoUML (Robbins and Redmiles, 2000)

ArgoUML (Robbins&Redmiles, 2000, <http://argouml.tigris.org>) is an object-oriented design tool using the Unified Modeling Language (UML) design notation. It is a *design critic* tool that supports several identified cognitive needs of software designers. Figure 4.5 shows the ArgoUML user interface.

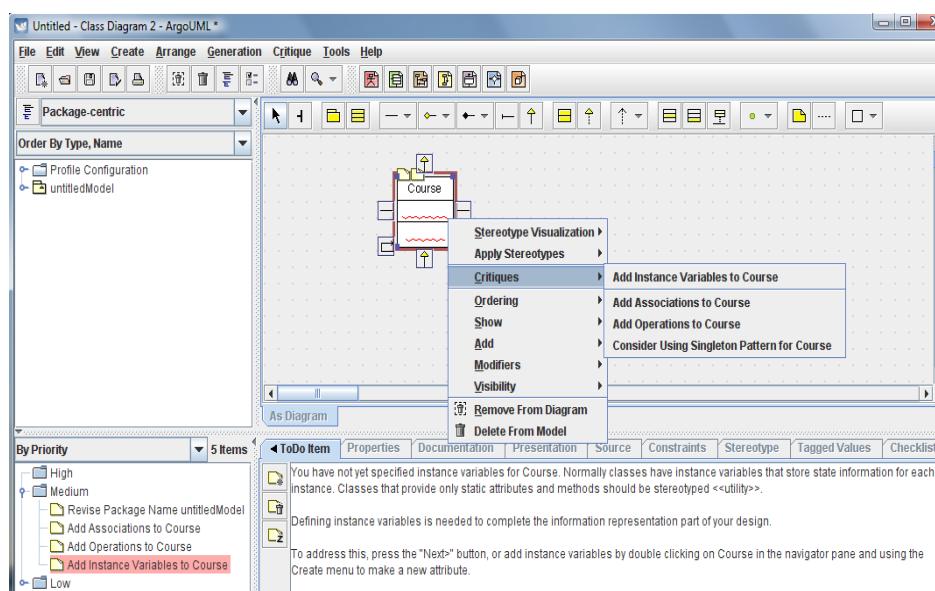


Figure 4.5: The ArgoUML user interface.

As Robbins and Redmiles (2000) state “design critics are agents that check the design for potential problem” (Robbins & Redmiles, 2000). Thus, ArgoUML has predefined agents, called *critics*, that are constantly checking the current model designed by software designers. The critic will generate a *ToDo Item* (as a critic feedback item or a critique) in the *ToDo* list if the conditions for causing a critic occurred. The *ToDo Item* (as shown in Figure 4.5) is presented in a constructive manner and this is very helpful to software designers because it contains an explanation of the problem, some suggestions about how to resolve the problem, and if there exists one, a wizard which assists the designer resolve the problem automatically. In addition, a *ToDo* item generated by a critic will remain in the *ToDo* list until the cause of the problem is removed either manually by the designer or by following the actions suggested by the tool’s wizard. We can say that, Argo/UML’s *ToDo* list is practical because it reduces the designer’s reliance on short-term memory and offers convenient ways to organise and browse items.

The critics in ArgoUML are not intrusive, since the user can disregard them completely or disable one or all of them via the critics’ configuration menu. Critics in ArgoUML are not user defined, since they all are implemented as Java classes and are compiled as part of the tool. However it does provides a class framework, source code templates and examples to facilitate the critic implementation process (Robbins & Redmiles, 2000). Thus, adding new critics is done by modifying the source code and this will require Java expertise. Details of ArgoUML can be found at this link: <http://argouml.tigris.org/>. Figure 4.6 shows the mapping of the ArgoUML tool to the critic taxonomy. Items in blue represent the element supported by the ArgoUML tool.

		Critic Domain:	Software engineering	(UML designs)		
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback	Types of Critic
comparative	textual	insert new critic rule	rule-based	active	explanation	correctness
analytical	graphical & 3-dimension visualisation	modify critic rule	knowledge-based	passive	argumentation	completeness
	multi-modal	delete critic rule	predicates	reactive	suggestion	consistency
		enable/disable critic rule	pattern-matching	proactive	examples	optimization
		critic rule authoring	OCL	local	simulation	alternative
			programming code	global	demonstration	evolvability
					interpretation	presentation
					positive feedback	tool
					negative feedback	experiential
					constructive feedback	organisation
						design pattern
						structural
						naming
						metric

Figure 4.6: The mapping of the ArgoUML tool to the critic taxonomy

4.4.2 ABCDE-Critic (de Souza et al., 2000)

An environment called **Annotation Based Cooperative Diagram Editor (ABCDE)-Critic** (de Souza et al., 2000), adopts critics to check UML class diagrams. ABCDE-Critic is a Domain Oriented Design Environment (DODE) for object-oriented analysis and design, which implements a group critic system. The environment implements a construction kit supporting UML class diagrams, an argumentative hypermedia system, and a critic system. ABCDE-Critic uses rule-based expressions to specify critics that comment on UML class diagram-based designs. The critic system in ABCDE-Critic fires critics when the condition clauses are found to be true in the current design parts warning the designer that the design may possibly have a problem/error. The critic's properties in ABCDE-Critic are: 1) critic's name, 2) critic state (active, passive, disable), 3) a quick critic explanation, 4) an

argumentation which is a critic more in-depth explanation, 5) critic importance, 6) a set of rules, and 7) a set of solutions.

In ABCDE-Critic, critic feedbacks are presented as annotations attached to the diagram elements that trigger the critic to fire. These annotations are also displayed to all other designers who are owners of these diagram elements. The critic feedback in ABCDE-Critic is displayed in two views. The first view is where the “*Things to take care of*” window pops up and display the critic name and its quick explanation in a list box. The second view is where the annotations created for the diagrams being constructed are displayed in the graphics interface component known as *annotation column*. The ABCDE-Critic uses a Design Rationale (DR) model to record the justification behind the design decision made during object-oriented analysis and design activities. Designers can define and control the critic’s state (active, passive, disable) when necessary. ABCDE-Critic allows the designers themselves to add critics to the critic tool via its first-order production system. Critics in ABCDE-Critic are normally defined by the critic’s author or extracted from the object-oriented design heuristics.

ABCDE-Critic is good critic system in the sense that it supports cooperation among designers as a means of annotation and warns designers that are involved in the problem. ABCDE-Critic also allows other designers to just add another alternative to the set solution of one critic. Thus, designers can communicate with the critiquing system as a true partner. Figure 4.7 shows the mapping of the ABCDE-Critic tool to the critic taxonomy. Items in blue represent the element supported by the ABCDE-Critic tool.

			Critic Domain:	Software engineering	(UML class diagrams)	
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback	Types of Critic
comparative	textual	insert new critic rule	rule-based	active	explanation	correctness
analytical	graphical & 3-dimension visualisation	modify critic rule	knowledge-based	passive	argumentation	completeness
	multi-modal	delete critic rule	predicates	reactive	suggestion	consistency
		enable/disable critic rule	pattern-matching	proactive	examples	optimization
		critic rule authoring	OCL	local	simulation	alternative
			programming code	global	demonstration	evolvability
					interpretation	presentation
					positive feedback	tool
					negative feedback	experiential
					constructive feedback	organisation
						design pattern
						structural
						naming
						metric

Figure 4.7: The mapping of the ABCDE-Critic tool to the critic taxonomy

4.4.3 IDEA (Bergenti & Poggi, 2000)

Interactive **D**Esign **A**ssistant (**I**DEA) is a critic system that performs direct communication with the software architect to propose pattern-specific critiques (Bergenti and Poggi, 2000). The development of IDEA is designed for automating the task of finding the realisations of design patterns used in UML diagrams and then improving the diagrams. The improvement of the design is made through critics that are presented to software architects. IDEA produces design pattern-based critics implemented with Prolog rules that are directly integrated with a knowledge base.

The IDEA approach is that the UML design which is under construction is analyzed in XMI format and then class and collaboration diagrams are employed to detect all pattern realisations. If a pattern is detected then it is called *detectable*, otherwise it is called *undetectable* because of incomplete information on the diagrams. When a

pattern realisation is discovered, IDEA then examines pattern-specific rules to select a set of critics to improve the design realisation.

IDEA provides the architect with two lists, the “pattern list” and the “to-do list”. The “pattern list” contains all patterns that IDEA found in the UML model. There are eleven patterns detected by IDEA: Template Method, Proxy, Adapter, Bridge, Composite, Decorator, Factory Method, Abstract Factory, Iterator, Observer and Prototype. The “to-do list” (the critic feedback) is the list of all selected critics organized by their importance (high, medium, and low). IDEA allows architects to control the pattern detection directly through these lists.

As Bergenti and Poggi (2000) point out that the knowledge base of IDEA is comprised with a set of design rules, corresponding critics, and a set of consolidation rules. These are maintained dynamically where patterns and rules can be added and removed when required. However, the rules for creating the pattern-specific critics are not easy to understand or author as this requires a high-level of understanding of a design patterns and detailed knowledge of the Prolog and knowledge base structures. Figure 4.8 shows the mapping of the IDEA tool to the critic taxonomy. Items in blue represent the element supported by the IDEA tool.

Critic Domain:			Software engineering	(Design patterns)		
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback	Types of Critic
comparative	textual	insert new critic rule	rule-based	active	explanation	correctness
analytical	graphical & 3-dimension visualisation	modify critic rule	knowledge-based	passive	argumentation	completeness
	multi-modal	delete critic rule	predicates	reactive	suggestion	consistency
		enable/disable critic rule	pattern-matching	proactive	examples	optimization
		critic rule authoring	OCL	local	simulation	alternative
			programming code	global	demonstration	evolvability
					interpretation	presentation
					positive feedback	tool
					negative feedback	experiential
					constructive feedback	organisation
					design pattern	
					structural	
					naming	
					metric	

Figure 4.8: The mapping of the IDEA tool to the critic taxonomy

4.4.4 RevJava (Florijn, 2002)

RevJava (Florijn, 2002,) is a tool used to analyse and critique object-oriented software. According to Florijn (2002), the Revjava design is quite generic and the implementation operates on compiled Java class files. RevJava acts as an assistant to Java coders by examining critics that can identify potential design and style improvements of the Java code. Figure 4.9 shows the interface of RevJava critics.

RevJava components consist of: a model reader, repository, meta-model, property definitions, critic definitions, property evaluator, metrics database, reporting and visualisation. The model reader reads in the Java code and saves it in a repository. The repository is arranged based on a meta-model that identifies all relevant entities in an OO/Java program. For each meta-model type, information about a model element (property and critic) can be defined and derived. The property and critic definition is then loaded into RevJava and can be obtained on request. For example,

when a user loads a program, some of the properties are treated as “critics” and “metrics”. The information collected via critics and metrics then can be manipulated in different kinds of reporting and visualisations tools. According to Florijn (2002), visualisations that highlight specific violations in large collections of classes have been produced. In addition, RevJava also allows Java users to enable and disable critics by configuring the setting menu. Details of RevJava can be found at this link: <http://www.serc.nl/people/florijn/work/designchecking/RevJava.htm>. Figure 4.10 shows the mapping of the RevJava tool to the critic taxonomy. Items in blue represent the element supported by the RevJava tool.

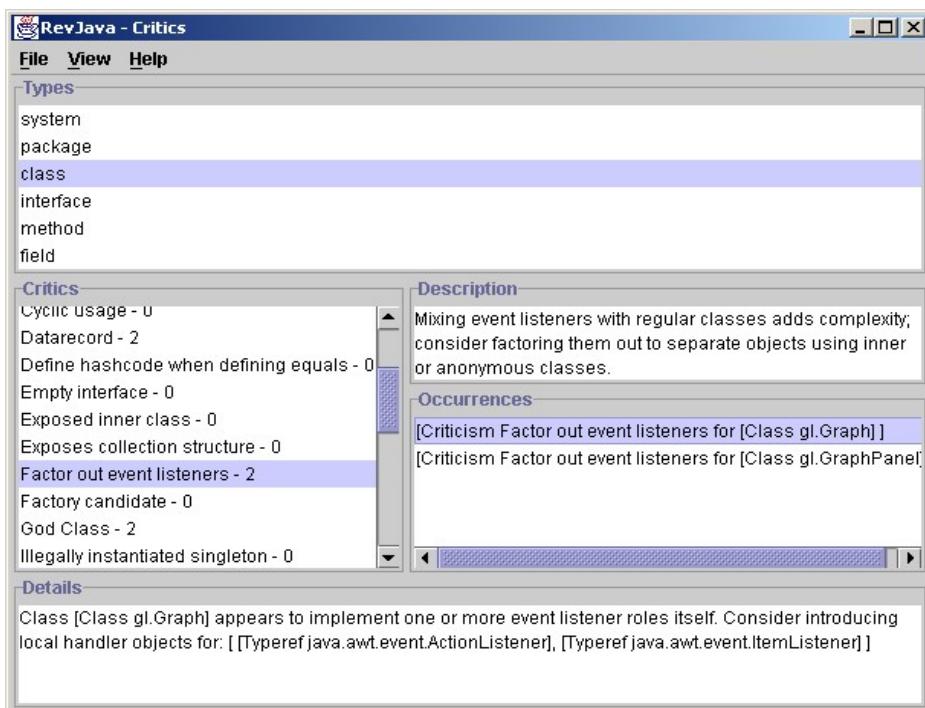


Figure 4.9: RevJava Critics
[\(http://www.serc.nl/people/florijn/work/designchecking/RevJavaScreenShots.htm\)](http://www.serc.nl/people/florijn/work/designchecking/RevJavaScreenShots.htm)

Critic Domain:			Software engineering	(Java coding)		
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback	Types of Critic
comparative	textual	insert new critic rule	rule-based	active	explanation	correctness
analytical	graphical & 3-dimension visualisation	modify critic rule	knowledge-based	passive	argumentation	completeness
	multi-modal	delete critic rule	predicates	reactive	suggestion	consistency
		enable/disable critic rule	pattern-matching	proactive	examples	optimization
		critic rule authoring	OCL	local	simulation	alternative
			programming code	global	demonstration	evolvability
					interpretation	presentation
					positive feedback	tool
					negative feedback	experiential
					constructive feedback	organisation
						design pattern
						structural
						naming
						metric

Figure 4.10: The mapping of the RevJava tool to the critic taxonomy.

4.4.5 DAISY (de Souza et al., 2003)

Following their work on ABCDE-Critic, de Souza et al., (2003) later developed another environment called **D**omain and **A**pplication engineering using **I**ntegrated critiquing **S**ystems (**DAISY**) that supports the construction of domain engineering and application engineering models. The main goal of their approach is to support consistency management in these models (de Souza et al., 2003). Domain engineering comprises three main activities: 1) domain analysis, 2) domain design, and 3) domain implementation. However their work is more focused on diagrams and models that are created during domain analysis and domain design. Application engineering complements the domain engineering process. It produces software products based on the domain engineering process.

DAISY was built on top of ABCDE-Critic. DAISY supports consistency checking of these models through the use of three different critics systems. The first critic

system assists the development of feature diagrams and defines seven different critics. The feature diagrams show the architectural structure of software features. In this work, DAISY deals with software architecture diagrams and class diagrams. The second critic system is used during application engineering to assess the UML class diagrams using object-oriented design heuristics and has about twenty critics. These two critic systems are used to improve the overall quality of the UML models. The third critic system detects potential inconsistencies and other errors that might occur in the mapping between domain model and application model. There are seven different critics implemented. The contribution of DAISY is the inconsistency detection in a software engineering model through the use of three critic systems. Though the number of critics implemented is small it could potentially be further extended. Figure 4.11 shows the mapping of the DAISY tool to the critic taxonomy. Items in blue represent the element supported by the DAISY tool.

		Critic Domain:	Software engineering	(feature diagrams and class diagrams)		
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback	Types of Critic
comparative	textual	insert new critic rule	rule-based	active	explanation	correctness
analytical	graphical & 3-dimension visualisation	modify critic rule	knowledge-based	passive	argumentation	completeness
	multi-modal	delete critic rule	predicates	reactive	suggestion	consistency
		enable/disable critic rule	pattern-matching	proactive	examples	optimization
		critic rule authoring	OCL	local	simulation	alternative
			programming code	global	demonstration	evolvability
					interpretation	presentation
					positive feedback	tool
					negative feedback	experiential
					constructive feedback	organisation
						design pattern
						structural
						naming
						metric

Figure 4.11: The mapping of the DAISY tool to the critic taxonomy.

4.4.6 Java Critiquer (Qiu and Riesbeck, 2003)

Qiu and Riesbeck (2003-2004, 2008) demonstrate the development of an educational critic tool. They develop a critic tool for Java programming, called Java Critiquer. Java Critiquer is developed by using an incremental authoring approach (Qiu and Riesebeck, 2003-2004, 2008). This critic tool not only supports teachers but also students. Teachers use the Java Critiquer to critique student java code. Student java code is pasted into a textbox and then the Java Critiquer performs automatic critiquing which is done via a pattern matching approach. When a pattern is matched, its corresponding critique is inserted right below the problematic Java source code. The teacher then validates these critiques by modifying or removing inappropriate ones as needed. The teacher can then perform manual critiquing on the code, after reviewing the critiques generated by the tool. The manual critiquing complements the automatic critiquing to ensure the quality of tool critiquing in the early development stage. Java Critiquer allows teachers to add new critique or use the existing critiques in the tool. Critiques are stored in a database and this leads to reusable critiques (Qiu and Riesbeck, 2004).

Java Critiquer is an effective tool because it supports teachers and students. It helps the teacher to perform automatic program critiquing and this would reduce their work in reviewing the student java code manually. Students can get support from Java Critiquer because they get feedback prior to sending their assignments to their teacher. Furthermore, students can do self-learning through Java Critiquer. Figure 4.12 shows part of Java Critiquer interface. Figure 4.13 shows the mapping of the Java Critiquer tool to the critic taxonomy. Items in blue represent the element supported by the Java Critiquer tool.

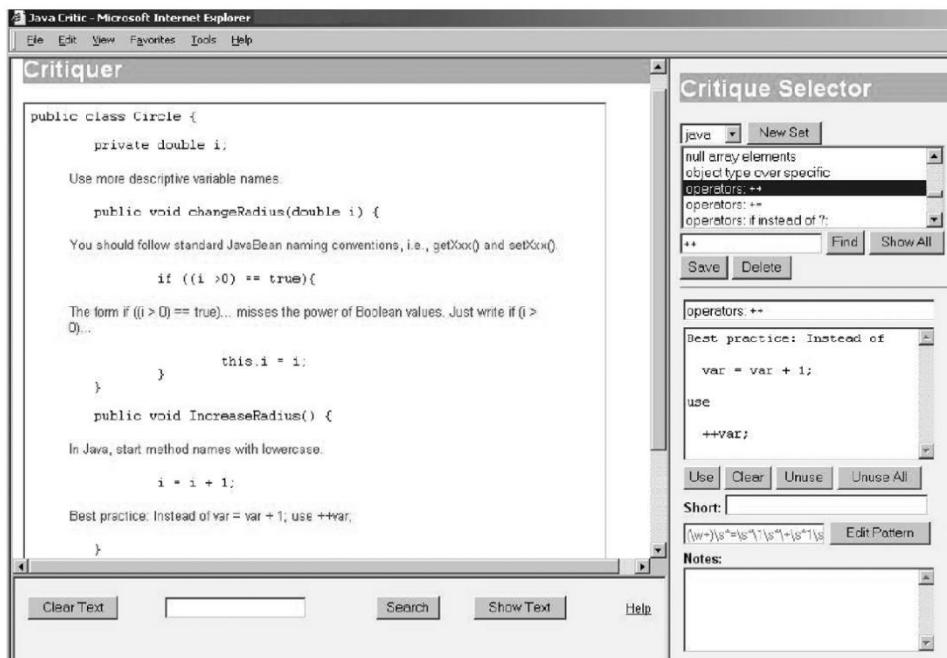


Figure 4.12: Java Critiquer interface (Qiu & Riesbeck, 2008).

Critic Domain:	Education		(teaching Java coding)	
	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback
comparative	textual	insert new critic rule	rule-based	active
analytical	graphical & 3-dimension visualisation	modify critic rule	knowledge-based	passive
	multi-modal	delete critic rule	predicates	reactive
		enable/disable critic rule	pattern-matching	proactive
		critic rule authoring	OCL	local
			programming code	global
				demonstration
				interpretation
				positive feedback
				negative feedback
				constructive feedback
				design pattern
				structural
				naming
				metric

Figure 4.13: The mapping of the Java Critiquer tool to the critic taxonomy.

4.4.7 Design Evaluator (Oh et al., 2004)

The Design Evaluator is a pen-based critic system for design sketching (Oh et al., 2004). The aim of Design Evaluator is to assist designers who draw and then justify their drawings to resolve design problems (Oh et al., 2004). Oh et al. (2004) demonstrated the sketch based critic system with two applications of the Design Evaluator: 1) architectural floor plan, and 2) web page layout. The Design Evaluator has two components where the first component is to allow the system to access the knowledge about the domains and the second component is to make the system be able to present critic feedbacks in a proper way. The Design Evaluator supports designers with critical effective feedback and gives reasoning on their design sketches. A designer receives the feedback in a form of criticism and advice. The way the Design Evaluator presents the critic feedback is excellent because critiques are displayed in various formats: textual, graphical annotation, 3D annotated walk-through models (e.g. architectural floor plan) and case library (e.g. web page layout). It is more helpful by its use of more than one format to communicate information about the design.

The Design Evaluator is composed of three layers: description, evaluation and visualization. These layers offer different activities performed by the designers. The *description layer* captures the sketching data from the designer and applies some preprocessing steps to generate a design representation. The design representation will then be used by the evaluation layer. The *evaluation layer* is composed of rules coded as Lisp predicates that apply to the design objects. These rules are stored in a list that the designer can check and edit. Each rule expression is associated with a text critic, as well as code that specify how to annotate the sketch when the critic is applied. A rule may also carry additional information to be used by auxiliary visualization routines such as the VRML model creator (for architecture) or the URL of a representative example case (for web page layout design evaluator). The *visualization layer* then presents critiques (critic feedback) in a form of textual and visual. The good thing about the Design Evaluator in terms of displaying critiques is that it provides the ability to link the critiques directly on the design sketch and this is very useful because it makes the designer remain focused on the sketches she/he is making. Figure 4.14 shows the mapping of the Design Evaluator tool to the

critic taxonomy. Items in blue represent the element supported by the Design Evaluator tool.

		Critic Domain:	Design engineering	(design sketching)		
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback	Types of Critic
comparative	textual	insert new critic rule	rule-based	active	explanation	correctness
analytical	graphical & 3-dimension visualisation	modify critic rule	knowledge-based	passive	argumentation	completeness
	multi-modal	delete critic rule	predicates	reactive	suggestion	consistency
		enable/disable critic rule	pattern-matching	proactive	examples	optimization
		critic rule authoring	OCL	local	simulation	alternative
			programming code	global	demonstration	evolvability
					interpretation	presentation
					positive feedback	tool
					negative feedback	experiential
					constructive feedback	organisation
						design pattern
						structural
						naming
						metric

Figure 4.14: The mapping of the Design Evaluator tool to the critic taxonomy.

4.4.8 ClassCompass (Coelho & Murphy, 2007)

Coelho and Murphy (2007) develop an educational critic tool for software design, called ClassCompass. They define ClassCompass as “an automated software design critique system with critics that comment on high-level design issues rather than diagram completeness.” ClassCompass is considered as a collaborative software design tool with the purpose to assist the students as well as the instructors in the software design activities.

Students use the system to produce software designs based on a set of requirements. The students can obtain automated feedback (critiques) about typical design problems while they perform their design task. ClassCompass also allows students to manually critique other student’s design task via the menus provided in the

system. Thus, students can see and learn the design styles from the critiques generated by the system as well as from other students (Coelho and Murphy, 2007). Instructors use an extended version that provides additional features for managing instructional sessions. Instructors use a Web application to configure ClassCompass before students take part in the collaborative design tasks. Instructors will specify the design principles that will be used by students to evaluate designs manually. Then, the instructor uses the ClassCompass client to automatically exchange designs between groups of students.

As mentioned above, ClassCompass supports automated critiquing and manual critiquing. The automated critiquing is executed when a user starts creating their design models in the system. When a critic finds a potential design flaw, an entry is added to a list of critiques beside the design diagram. The critics in ClassCompass are not intrusive, since the user can continue their task if they decide to ignore the automated critic feedbacks. Furthermore, ClassCompass lets the user select the item of interest in the critiques box. A detailed explanation of that particular critique is then presented in the Critique Details box. The critique details text in ClassCompass is arranged into three parts: 1) *Critique*-describes the design error, 2) *Rationale*-explains why the identified error can reduce software quality, and 3) *Suggestion*-provides suggestion to correct the identified error. ClassCompass too can highlight the relevant part of the design diagram structure to get user's attention to the detected problem. Critics in ClassCompass are implemented in Java as pluggable classes that check for a particular pattern in an object model representing the design (Coelho & Murphy, 2007). Figure 4.15 shows the user interface of ClassCompass with automated critiquing. Figure 4.16 shows the mapping of the ClassCompass tool to the critic taxonomy. Items in blue represent the element supported by the ClassCompass tool.

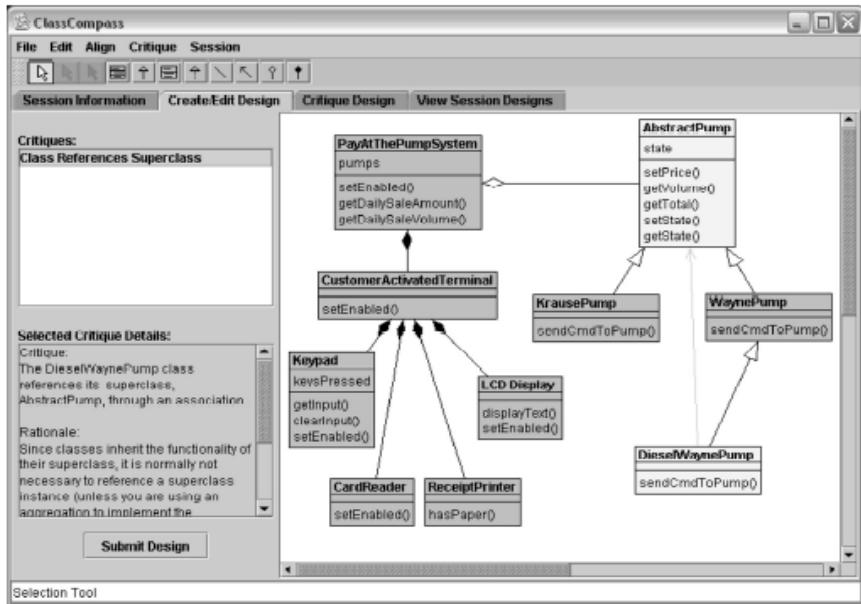


Figure 4.15: ClassCompass user interface (Coelho & Murphy, 2007).

		Critic Domain:	Education	(teaching UML class and sequence diagrams)		
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback	Types of Critic
comparative	textual	insert new critic rule	rule-based	active	explanation	correctness
analytical	graphical & 3-dimension visualisation	modify critic rule	knowledge-based	passive	argumentation	completeness
	multi-modal	delete critic rule	predicates	reactive	suggestion	consistency
		enable/disable critic rule	pattern-matching	proactive	examples	optimization
		critic rule authoring	OCL	local	simulation	alternative
			programming code	global	demonstration	evolvability
					interpretation	presentation
					positive feedback	tool
					negative feedback	experiential
					constructive feedback	organisation
						design pattern
						structural
						naming
						metric

Figure 4.16: The mapping of the ClassCompass tool to the critic taxonomy.

4.4.9 FFDC (Oh et al., 2009)

Flat-pack Furniture Design Critic (FFDC) is a computer-based critic tool that support design learning in studio settings (Oh et al., 2009). Oh et al. (2009) develop FFDC using a constraint based design critic program that provides students feedback with five delivery types and three communication modes. The five delivery types are interpretation, introduction, example, demonstration and evaluation. The three communication modes are written comments, graphical annotations, and images. Oh et al. (2009) points out that their FFDC tool selects specific methods to present feedback by considering a user's knowledge and the critiquing methods that the program has previously used for the user.

The FFDC is written in Macintosh Common Lisp using OpenGL to provide 3D models and the Lisa (Lisp-based Intelligent Software Agent) production rule system to justify a planned furniture design using the stored constraints. FFDC has eight components: *Construction Interface*, *Parser*, *Pattern Matcher*, *Design Constraints*, *Critiquing Rules*, *User Model*, *Pedagogical Module*, and *Critiquer*. The *construction interface* allows a user to perform design sketching via a stylus and digitising tablet. All sketched glyphs are recorded, a Cartesian coordinate system is defined and a 3D model is generated (Oh et al., 2009). The *parser* is used to parse the sketched diagram and the 3D model to generate two kinds of data: 1) parts and their properties and 2) configuration of parts. A symbolic representation of the designed furniture is then saved in text file created by the *parser*. FFDC uses a set of *design constraints* to represent the principles that the designers have to know in designing furniture. FFDC uses 27 structural constraints and 36 functional constraints which are stored in the program. The *pattern matcher* component is used to compare the symbolic representation of the design against the design constraints to detect any critic violations. The *user model* component has of short-term user model and long-term user model. The short-term user model is to store the results of the *pattern matcher* and the long-term user model is to store the history of all violated and satisfied constraints over multiple critiquing sessions. The *pedagogical module* takes input from the short-term and long-term user model. From these user models, it then decides the specific critiquing methods via the *critiquing rules*. The *critiquing rules* determine which delivery types and communication modes are to be used in certain

conditions. For instance, when a designer is recognised as a novice designer, the *pedagogical module* will choose ‘demonstration’ delivery type rather than ‘example’ for the reason that novices normally have trouble to use examples in their designs. After the critiquing method is selected, the *critiquer* component presents the critique to the designer. The *critiquer* component consists of three modules: 1) *text critique*-presents written comments, 2) *example finder*-selects relevant examples, and 3) *graphic critique*- highlights relevant furniture parts and draws graphical annotations. The FFDC tool offers feedback (critiques) in several ways to users based on their knowledge and previous used feedback. Figure 4.17 shows the mapping of the FFDC tool to the critic taxonomy. Items in blue represent the element supported by the FFDC tool.

Critic Domain: Education (teaching furniture design)						
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback	Types of Critic
comparative	textual	insert new critic rule	rule-based	active	explanation	correctness
analytical	graphical & 3-dimension visualisation	modify critic rule	knowledge-based	passive	argumentation	completeness
	multi-modal	delete critic rule	predicates	reactive	suggestion	consistency
		enable/disable critic rule	pattern-matching	proactive	examples	optimization
		critic rule authoring	OCL	local	simulation	alternative
			programming code	global	demonstration	evolvability
					interpretation	presentation
					positive feedback	tool
					negative feedback	experiential
					constructive feedback	organisation
						design pattern
						structural
						naming
						metric

Figure 4.17: The mapping of the FFDC tool to the critic taxonomy.

4.4.10 HeRA (Knauss et al., 2009)

Heuristic Requirements Assistant (HeRA) is a feedback centric requirements editor to support analysts with information based on several feedback facilities (Knauss et al., 2009). HeRA was developed to assist the requirements analyst with heuristic feedback. Requirements analysts receive warnings and hints for any detection of ambiguities or incomplete requirements specification while typing/writing requirements.

The HeRA tool consists of three editors and two components: 1) general purpose requirements editor, 2) use case editor, 3) glossary editor, 4) argumentation component and 5) simulation component. Requirements are constructed using the three editors and produced domain specific artifacts i.e. requirements, use cases and a glossary. The HeRA tool lets users argue with the critiques via the argumentation component. This can help users to clarify their understanding about the requirements problem and also leads to improved heuristics feedback in future. The simulation component provides ‘what-if’ analysis and derives three models while the user case is written: *UML Use Case Diagrams*, *EPC Business Processes*, and *Use Case Points Estimations*. These models can provide extra information (feedback) to the requirements author regarding the requirements being documented (Knauss et al., 2009). In HeRA, heuristics rules are defined in JavaScript and can access the data model of the requirements editor. HeRA also provides wizards that facilitate requirements author to generate Java script code for a rule. Rules can be changed and applied directly. Thus new critiques are shown immediately. HeRA users have the option to fix or ignore the critiques offered to them. In general, HeRA offers different levels of feedback to the requirement analyst using the argumentation and simulation components. Figure 4.18 shows the mapping of the HeRA tool to the critic taxonomy. Items in blue represent the element supported by the HeRA tool.

		Critic Domain:	Software engineering	(requirements engineering)		
Critiquing Approach	Modes of Critic Feedback	Critic Rule Authoring	Critic Realisation Approach	Critic Dimension	Types of Critic Feedback	Types of Critic
comparative	textual	insert new critic rule	rule-based	active	explanation	correctness
analytical	graphical & 3-dimension visualisation	modify critic rule	knowledge-based	passive	argumentation	completeness
	multi-modal	delete critic rule	predicates	reactive	suggestion	consistency
		enable/disable critic rule	pattern-matching	proactive	examples	optimization
		critic rule authoring	OCL	local	simulation	alternative
			programming code	global	demonstration	evolvability
					interpretation	presentation
					positive feedback	tool
					negative feedback	experiential
					constructive feedback	organisation
						design pattern
						structural
						naming
						metric

Figure 4.18: The mapping of the HeRA tool to the critic taxonomy.

4.4.11 Summaries of Comparison

We map the properties and features identified in the above surveyed tools to our critic taxonomy. There are eight elements in the critic taxonomy and we have developed eight tables (Table A -Table H) to correspond to each element. The following eight tables present the application of the critic taxonomy to the ten systems and tools. If a table entry appears to be empty then it means either it is not stated in a paper describing the tool or not available in the tool as we experimented with it.

Table A: Critic domain

Critic tools	Critics domain
ArgoUML (2000)	Software engineering-UML designs
ABCDE-Critic (2000)	Software engineering-UML class diagrams
IDEA (2000)	Software engineering- design patterns
RevJava (2003)	Software engineering-Java coding
DAISY (2003)	Software engineering-feature diagrams and class diagrams
Java Critiquer (2003)	Education-teaching Java coding
Design Evaluator (2004)	Design engineering-design sketching (floor plans and web pages)
ClassCompass (2007)	Education-teaching UML class and sequence diagrams
FFDC (2009)	Education-furniture design
HeRA (2009)	Software engineering -requirements engineering

As we mentioned earlier critics can be applied in various domains. We selected randomly ten systems and tools that employ critics described in papers ranging from the year 2000 to 2009. The year represented for the system/tool is based on the published paper about the system/tool.

Table A shows there are six tools from the software engineering (SE) domain, three tools from the education domain and one tool from the design engineering/architecture domain. The six tools from the SE domain involve critic domains which are either similar or different to each other. Critics domain for this six tools are: 1)ArgoUML is on UML designs, 2) ABCDE-Critic is on UML class diagrams, 3) IDEA is on design patterns, 4) RevJava is on Java software, 5) DAISY is on domain engineering models (feature diagrams) and application engineering models (class diagrams), and 6)HeRA is on requirements engineering. There are three education tools, but two of them focus on the SE domain. The critics domain for these three tools are: 1) Java Critiquer is on Java program/ source code, 2)

ClassCompass is on software design (UML class diagrams and sequence diagrams), and 3) FFDC is on furniture design. These three education tools support the students and instructors in a learning environment. Finally, the critics' domain for the design engineering/architecture tool, Design Evaluator is on design sketching (i.e. architectural floor plan and Web page layout design).

Table B: Critiquing approach

Critic tools/ Critiquing Approach	Comparative	Analytical
ArgoUML (2000)		√
ABCDE-Critic (2000)		√
IDEA (2000)		√
RevJava (2003)		√
DAISY (2003)		√
Java Critiquer (2003)		√
Design Evaluator (2004)		√
ClassCompass (2007)		√
FFDC (2009)	√	
HeRA (2009)		√

The critiquing approach is the way that a system/tool uses to generate valid reasoning to detect any potential problems/mistakes/errors in the user's work or design solution. A finding from Table B is that most tools preferred to use the analytical approach. Thus, by using analytical critiquing, the system/tool designer would not require to incorporate comprehensive/complete domain knowledge into their tools in order to generate a solution for the user's work or design solution.

Table C: Modes of critic feedback

Critic tools/ Modes of Critic Feedback	Textual	Graphical & 3-dimension visualisation	Multi-modal (e.g. sound, animation, & movies)
ArgoUML (2000)	√	√	
ABCDE-Critic (2000)	√	√	
IDEA (2000)	√		
RevJava (2003)	√	√	
DAISY (2003)	√	√	
Java Critiquer (2003)	√		
Design Evaluator (2004)	√	√	
ClassCompass (2007)	√	√	
FFDC (2009)	√	√	
HeRA (2009)	√		

The mode of critic feedback (critiques) concerns the format of feedback to be displayed for users. From Table C we see that most tools applied the textual, and graphical and 3-dimension visualisation format to present the critiques. Three tools only use textual format to present critiques to their users: IDEA, Java Critiquer and HeRA. The Design Evaluator and FFDC have augmented their critiques modes via 3D visualizations. We believe that displaying visual and textual critiques is expected to be more effective than selecting only one single mode. Though all the above tools does not applied the ‘multi-modal element’ but it can be considered for future use/work. Furthermore, it is an advantage to offer critiques in various modes.

Table D: Critic rules authoring

Critic tools/ Authoring	Rule	Insert new rule	Modify rules	Delete rules	Enable/disable rules	Critic rule authoring facility
ArgoUML (2000)		√	√		√	
ABCDE-Critic (2000)		√	√	√	√	√
IDEA (2000)		√	√	√	√	√
RevJava (2003)					√	
DAISY (2003)		√	√	√	√	√
Java Critiquer (2003)		√	√	√	√	√
Design Evaluator (2004)		√	√	√		√
ClassCompass (2007)						
FFDC (2009)						
HeRA (2009)		√	√	√	√	√

The capability of critic rules authoring refers to the ability to allow users to: 1) insert or add new critic rules, 2) modify or edit critic rules, 3) delete or remove critic rules in the system/tool. This also includes a function to allow users to enable and disable the execution of critic rules incorporated in the system/tool. The main part in rules authoring is the facility to let the users author and store their own critic rules in the system/tool. Findings from Table D, shows that **FIVE** tools provide the five functions listed in the table: ABCDE-Critic, IDEA, DAISY, Java Critiquer, and HeRA. ArgoUML allows the end user to enable/disable rules but new rules have to be added using Java by a tool developer. The Design Evaluator allows the end user (designer) to inspect and edit the rules, but a function to enable and disable critics is not provided in the tool. In contrast to RevJava, which allows the users to enable and disable critics via its menu option. However RevJava does not provide the ability for users to add and edit critic rules, as well as facility to author critic rules. Two tools that do not provide these kinds of facilities are ClassCompass and FFDC. This is because the critic rules are written in advance by the system/tool designers to develop the system/tool and the facilities to customize the critic rules are not provided to the users to perform any changes to the critic rules.

Table E: Critic's realisation approach

Critic tools/ Critic's realisation approach	Rule-based	Knowledge-based	Predicates	Pattern-matching	OCL	Programming code
ArgoUML (2000)			√			√
ABCDE-Critic (2000)	√					√
IDEA (2000)	√	√				
RevJava (2003)						√
DAISY (2003)	√					√
Java Critiquer (2003)				√		√
Design Evaluator (2004)		√	√			√
ClassCompass (2007)				√		√
FFDC (2009)	√			√		√
HeRA (2009)						√

Critics' realisation approach refers to how critics are implemented or specified in a system/tool. Literature regarding critics shows that there are several ways to specify and implement critics. Findings from Table E are that most systems/tools implement critics via programming code. Rule-based and pattern-matching are another widely used approach to specify critics. Furthermore, most systems/tools apply more than one approach to implement critics. OCL is widely used in meta-modelling tools to specify tool constraints and it can be used to specify critics as reported by Bezivin and Jouault (2006). However due to the fact that the selected tools are not a meta-modelling tool, OCL has not been used as an approach to implement critics. A number of software engineering tools do provide OCL-based critic implementations (Grundy et al., 2008).

Table F: Critic dimensions

Critic tools/ Critic dimension	active	passive	reactive	proactive	local	global
ArgoUML (2000)	√	√			√	
ABCDE-Critic (2000)	√	√		√	√	
IDEA (2000)				√	√	
RevJava (2003)	√	√			√	
DAISY (2003)	√	√			√	√
Java Critiquer (2003)		√			√	
Design Evaluator (2004)	√				√	
ClassCompass (2007)	√				√	
FFDC (2009)	√				√	
HeRA (2009)				√	√	√

Critic dimension is one of the aspects that critic designers need to consider when adopting critics in their system/tool. Findings from Table F are that most tools apply active and passive critics. The Design Evaluator only provides active critics when any design sketching activities triggers a critic. Java Critiquer prefers to use passive critics, as they want students to learn from mistakes when they code their Java programs. Three tools provide the proactive critics to their users: ABCDE-Critic, IDEA and HeRA. All tools provide local critics and two tools (i.e. DAISY and HeRA) offer global critics.

Table G: Types of critic feedback

Critic tools/ Types of critic feedback	Explanations	Argumentations	Suggestions	Examples	Interpretations	Positive feedback	Negative feedback	Constructive feedback	Simulation	Demonstration
ArgoUML (2000)	√	√	√					√		
ABCDE-Critic (2000)	√	√	√							
IDEA (2000)	√		√							
RevJava (2003)	√		√							
DAISY (2003)	√	√	√							
Java Critiquer (2003)	√		√							
Design Evaluator (2004)	√		√	√		√	√			
ClassCompass (2007)	√	√	√							
FFDC (2009)	√		√	√	√	√	√			√
HeRA (2009)	√	√							√	

The Type of critic feedback refers to the techniques used to present critic feedback to users. The term critic feedback is also known as feedback or critique. When a critic is triggered to show that there is a potential problem in user's work/solution, critic designer have to consider appropriate techniques to present the critic feedback (critique) to the user. Findings from Table G are that various techniques are employed to present a critic feedback to the user. The most common techniques applied in all tools are explanations, suggestions and argumentations. However, a few tools add the richness/power of critic feedback in the form of constructive feedback, positive and negative feedback, examples, interpretations, simulation and demonstration. Tools that provide multiple critic feedbacks to users are: ArgoUML, Design Evaluator, FFDC, and HeRA.

Table H: Types of critics

Critic tools/ Types of critic feedback	Correctness	Completeness	Consistency	Optimisat ion	Altera tive	Evolvabi lity	Presenta tion	Tool	Experienci al	Organisati on	Design pattern	Structural	Naming	Metric
ArgoUML (2000)	✓	✓						✓						
ABCDE- Critic (2000)	✓	✓												
IDEA (2000)	✓	✓								✓				
RevJava (2003)	✓	✓												✓
DAISY (2003)	✓	✓	✓											
Java Critiquer (2003)	✓	✓												
Design Evaluator (2004)	✓	✓												
ClassCompass (2007)	✓										✓	✓	✓	
FFDC (2009)	✓		✓											
HeRA (2009)		✓	✓											

The types of critics refer to the type of critics that are offered by a system/tool to their users. Findings from Table H are that most tools offered correctness and completeness critics. DAISY, FFDC and HeRA tools provide consistency critics. Tool critics are offered by the ArgoUML and pattern critics are offered by the IDEA tool. The ClassCompass presents structural, naming and metric critics. The types of critics depend on the critic domain defined by a system/tool. For that reason, appropriate and relevant critics have to be designed by the critic designer to incorporate in the system/tool for the user benefit.

4.5 Conclusion

We proposed and illustrated a new critic taxonomy based on several aspects that characterize critics (or critiquing systems). These aspects are gathered widely from the critic literature. Our critic taxonomy identifies eight groups: critic domain, critiquing approach, modes of critic feedback, critic rule authoring, critic realization approach, critic dimension, types of critic feedback, and types of critic.

The utility of our critic taxonomy is manifold: to provide an overview of critic research, to identify and distinguish key critic elements, and to recognize techniques or methods applied in critics. We believe that this taxonomy provides meaningful way of describing and reasoning about critics. We also believe that our critic taxonomy is useful in guiding the critic developer towards realizing robust critic capabilities by comparing and contrasting different critic dimensions. We have applied our taxonomy to ten tools that have critic support. The mapping of the tools to our critic taxonomy shows that the practice of critics is supported by the critic taxonomy.

Providing users with a facility to author or customize critic rules is a useful element to be considered. Realising that critiquing capacity and issues may change from time to time, it is worth allowing users to author or customise (add, delete, modify) their own critic rules for a particular critic domain. However, some kinds of critic tools, critic rule approaches, tool users and domains are more amenable to this than others. Type of critic feedback appears to be another useful element as it shows the range of techniques that can be applied to present critic feedbacks to users. Furthermore, this element is related to the critiques modes. Conventional critics normally provide only a textual critique but realising the benefit of combining several modes in presenting critiques has augmented the visual or graphical critiques and 3D visualizations critiques in critic tools and systems.

The critics' implementation element facilitates tool developers in applying an appropriate approach to realise their critics. Each approach has pros and cons in specifying critics which a critic designer has to take into account. How critics are implemented closely relates to the critiquing approach used in the system. Thus carefully considering the critiquing approach is also a useful dimension that assists

the critic designer in deciding either comparative, analytical or both approaches be used in critic' development.

Critic dimensions are another element that can guide the critic designer in building a critic tool. A critic either provides intervention strategies, activation strategies or timing strategies. The results of mapping the ten tools with this element suggested that there are ranges of critic dimensions that can be used in enhance/improve critic development.

Critic types from the taxonomy are also helpful in guiding what type of critics that can be considered by a critic designer apart from the common critic types i.e. correctness and completeness critics. Thus, critic designers may consider incorporating other types of critics in a system.

Finally, though we selected only ten tools to present in our taxonomy application example, and most of these are from the software engineering domain, our critic taxonomy is applicable to critics in other domains. We showed this through characterising the Design Evaluator, FFDC, ClassCompass and Java Critiquer tools.

Chapter 5

A Visual and Template-Based Approach for Critic Specification

This chapter explains our visual and template-based approach for the critic-authoring task of a domain-specific visual language (DSVL) tool. This chapter begins by introducing the concepts and approaches used for our critic specification research. We introduce the visualization concept followed by the visual notations designed for our critic specification tool. Then we describe the template-based approach, followed by the business rule templates and critic templates. We also explain the concept of authoring and the approach of template-based authoring for critics. In the last section, we present an analysis of the design of our critic specification editor using Moody's Physics of Notations principles (Moody, 2008).

5.1 Introduction

The concepts of critiquing, visual representation, and templates are not new. These three concepts have been applied in various software development activities for various domains. The concept and use of critics (or critiquing) has been explained in the previous chapters (i.e. Chapter TWO and FOUR). The concept and application of a visual approach and a template-based approach is explained in the following section as we describe the design of our critic specification approach for domain-specific visual language (DSVL) tools. These concepts have formed the basis of our visual and template-based critic specification tool.

5.2 Visual Specification Approach

Visualization approaches are increasingly prevalent in modern software engineering. Many visualization research studies have been carried out, such as visual representations (Barton & Barton, 1987; G. L. Lohse, Biolsi, Walker, & Reuter, 1994; J. Lohse, Reuter, Biolsi, & Walker, 1990), diagrammatic representations

(Catarci, Massari, & Santucci, 1991; Gurr & Tourlas, 2000), visual environments for visual languages (Bardohl, 2002), visual notations (Costagliola, Lucia, Ferrucci, Gravino, & Scanniello, 2008; Moody, 2008) and others. However, the details of this body of research are not discussed in this chapter. The key elements that we are concerned for are the application of the visual approach and how it motivates and guides us in our critic specification development.

Before presenting and explaining the chosen visual approach, one should understand a few definitions of visualization in general. McCormick, DeFanti, and Brown (McCormick, DeFanti, & Brown, 1987) define visualization as the study of “*mechanisms in computers and in humans which allow them in concert to perceive, use, and communicate visual representation.*” They suggest that visualization includes the study of both image understanding and image synthesis (McCormick, et al., 1987). Petre and Quincey (2006) view visualization as “*the graphical (or semi-graphical) representation of information in order to assist human comprehension of and reasoning about that information.*” A similar definition of visualization is also provided in Guimaraes et al. (2008) where visualization is termed as “*a process of transforming information into a visual form to help users to understand its meaning.*” Guimaraes et al. (2008) point out that visualization offers a visual interface between two main information processing systems: the computer and human. Their research involves the development of visual approaches to support the information communication between human and computer through direct manipulation (Guimaraes, Neto, & Soares, 2008). Thus, it is very clear from these definitions that the key aspects in visualization are:

- 1) to represent data and information visually;
- 2) to support the interaction between humans (users) and computers via a visual approach; and
- 3) to facilitate human (user) understanding through a visual approach.

There are many aspects that should be considered when developing a system or an application that incorporates visualization. Some of these include: visual techniques/methods; visual representations; visual notations; visual data,

information and knowledge; visual languages; and many others. Some of these aspects are addressed in our critic specification development.

Lohse et al. (1994) consider visual representations as data structures for expressing knowledge. In their research, Lohse et al. (G. L. Lohse, et al., 1994; J. Lohse, et al., 1990) have identified six basic categories of visual representations: graphs, tables, maps, diagrams, networks and icons. According to Lohse et al. (1990, 1994) visual representations contain semantic information that communicates a purpose or graphical message. Visual representations carry no meaning without the translation processes that interprets the visual representation. There have to be rules to interpret features of visual representations (G. L. Lohse, et al., 1994; J. Lohse, et al., 1990). From the six categories, *diagrams* and *icons* are two categories that are related to our research.

A diagram is a sentence in a graphical language (Mackinlay, 1986) that can describe the structure of physical objects, interrelationships and processes associated with them (J. Lohse, et al., 1990) . Lohse et al. (1990) define structure diagrams as a static description of reality and process diagrams that express dynamic interrelationships among components of the diagram. According to Gurr (2001), diagrams are well-accepted, because many users realize that diagrams are more readily accessible compared to other forms of representation. Furthermore, Moody (2006) emphasizes that a good diagram is one which communicates effectively and is believed to be more effective than text for interacting with end users (Moody, 2006). Thus, Gurr (2000) points out that an effective diagram is normally the one that is “well matched” to what it represents. In general, the most effective diagrams are those which are very simple (Barton & Barton, 1987; Gurr & Tourlas, 2000). An example of a diagrammatic form is the popular UML diagram that consists of 13 types of diagrams (or models), all of which are represented in a graphical form.

Another type of visual representation is icons, which can convey a general understanding or meaning for a picture (J. Lohse, et al., 1990). Lohse et al. (1990) suggest that each icon assigns a unique label for a visual representation. Catarci et al. (1991) also gives a similar view about icons. According to Catarci et al. (1991),

icons are mainly used to represent a pictorial symbol of an object or an abstract concept which sometimes can involve an action. Icons that represent objects are easily understood because they are a stylized imitation of the real-world objects. Icons that represent actions and processes are generally harder to understand because they are more abstract (Catarci, et al., 1991). Thus, to present an effective icon, it should be clearly understandable by the majority of the users (Catarci, et al., 1991; J. Lohse, et al., 1990). Examples are the universal set of traffic icons and the icons used to represent several services and locations in an airport. However, as Catarci et al. (1991) point out, users can tailor their own icon shape based on their specific requirements and mental representation of the tasks and methods they want to carry out.

Visual and diagrammatic representations play a central role in several application domains since they are recognised to be important tools for describing and reasoning (Costagliola et al, 2008). Their employment allows us to improve productivity of expert and non-expert users in several application domains. This is because they provide a means to easily capture and model difficult concepts. This visual approach is advantageous due to the reduction of mental load and the immediate availability of descriptions of the computation processes and their interrelationships (Catarci, et al., 1991).

For these reasons, we have been motivated to develop our critic specification approach for DSVL tools with visual and diagrammatic representations. Additionally, we wanted to add our critic specification support to a Domain-Specific Visual Language (DSVL) meta-tool, Marama, which itself extensively employs visual notations to specify DSVL tools. Thus choosing a visual specification approach for critics allows us to leverage benefits of visual approaches to specification and to seamlessly integrate our critic designer into the Marama toolset. The next section explains the visual notation aspects that comprise in the visual and diagrammatic representations of our critic specification tool.

5.2.1 Visual Notations Used by the Critic Specification Editor

We introduce a few definitions of visual notations before explaining the visual notation of our critic specification editor. Visual notations have played a significant role in communicating with end-users, as they are believed to express information more effectively to non-technical users than text (Moody, 2008). There are several definitions of a visual notation. However, we only choose the definition of a visual notation from (Costagliola et al, 2004) and (Moody, 2008). According to Costagliola et al. (2004) a visual notation “*is a visual language, since it is formed by a set of visual symbols from an alphabet and a set of feasible visual sentences over these symbols.*” Whereas Moody (Moody, 2008) describes the visual notation as “*a visual notation (or visual language, graphical notation, diagramming notation) consists of a set of graphical symbols, a set of compositional rules for how to form a valid visual sentences, and definitions of their meanings (visual semantics).*”

We applied an incremental approach towards the development of a new critic specification editor for the Marama meta-tools and this has resulted in several developments/improvements of prototypes (this is explained in the following chapters- Chapter SIX and SEVEN). However, for conciseness and simplicity this section only describes the final prototype of our critic specification editor.

In this section, we describe the visual notation of the critic specification editor that we call the “Marama Critic Definer view”. This new designer has been developed to allow end-user developers to specify and generate Marama DSVL tool critics. The critic specification editor is an extension to our existing Marama meta-tools (Grundy, et al., 2008). Using it, end user tool developers can specify and generate tool critics more efficiently and easily than using Marama’s existing facilities of OCL and/or Java-based event handlers.

There are seven items provided by the editor to support critic specification. The symbols used in the critic specification editor include *CriticShape*, *CriticFeedbackShape*, *Operator*, *CriticFeedbackConn*, *CriticDependencyLink*, *OperatorConn* and *OperatorCriticFeedbackConn* as shown in Figure 1. There are three shapes and four connectors to represent visually the critic specification. The

three shapes are: 1) *CriticShape* represented by an orange rounded square shape, 2) *CriticFeedbackShape* represented by a green oval shape, and 3) *Operator* represented by a grey diamond shape. The four connectors are: 1) *CriticFeedbackConn* represented by a black arrow line that connects critic (s) and feedback, 2) *CriticDependencyLink* represented by an orange arrow line that links two critics, 3) *OperatorConn* represented by a grey line linking two critics and an operator, and 4) *OperatorCriticFeedbackConn* represented by a black arrow line linking operator and feedback. The editor's toolbar comprises seven icons to represent the shapes and the connectors. This is shown in the left side of the diagram in Figure 5.1.

The visual notations of the critic specification editor represent key elements in the meta-model that was defined for our critic specification tool. These critic meta-model entities, attributes and associations were defined based on our initial critic taxonomy creation. However, not all elements in the critic taxonomy are used to define the meta-model. We only selected the necessary elements to describe the critic specification task that we want to incorporate in the Marama meta-tools. The meta-model could be extended in future to incorporate more of our critic taxonomy features. Figure 5.2 shows the new meta-model defined for our Marama critic specification tool.

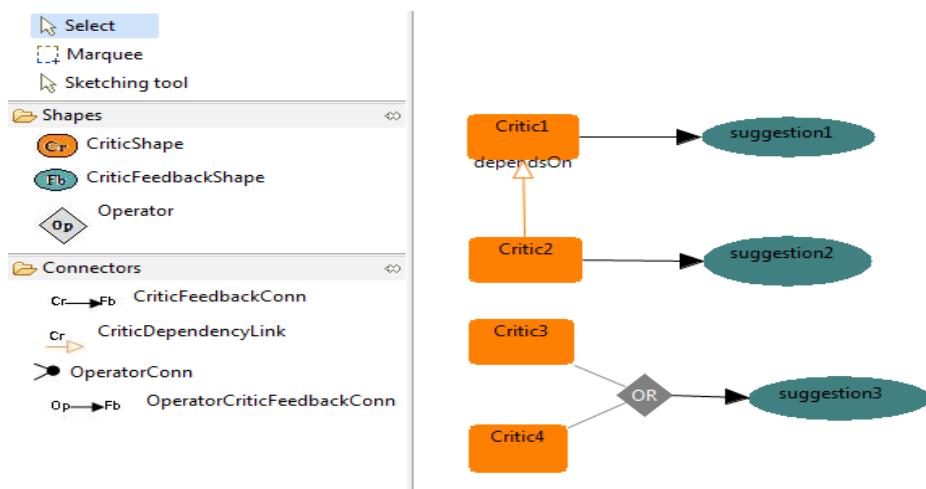


Figure 5.1: Visual notations of the visual critic specification editor: toolbar (left side) and diagram (right side)

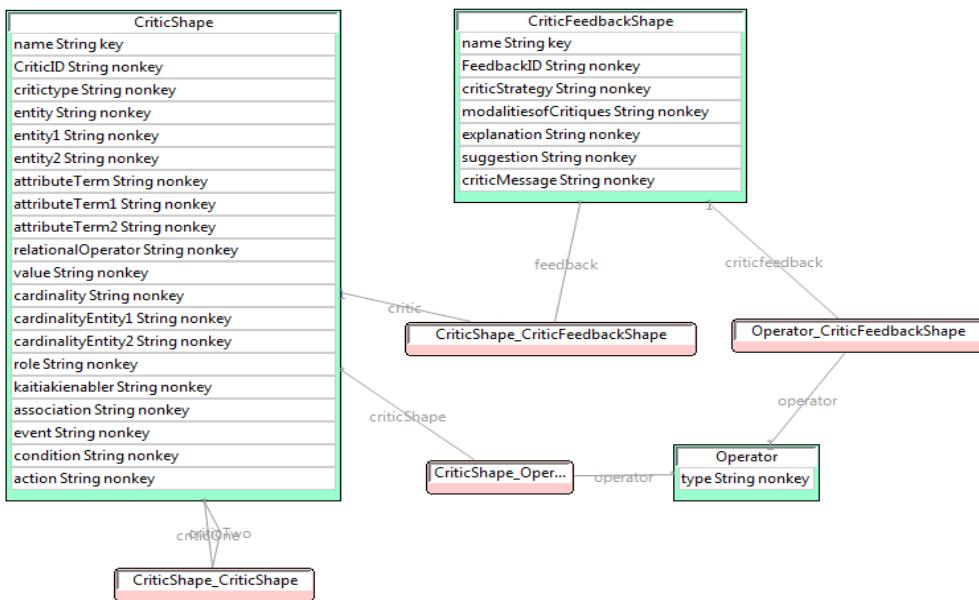


Figure 5.2: Meta-model defined for a critic specification tool

Generally, the core elements that end user tool developers should know when it comes to specifying critics for a DSVL tool are critic and critic feedback (fix action). These two elements are then presented to the tool users. The critic element contains information/statements regarding to detected errors, such as structural incorrectness, structural incompleteness, and constraint violations, whereas the critic feedback is about the suggestion to resolve the detected errors. Thus, end user tool developers should be able to recognise the sort of critics to be defined for their DSVL tool. The critic specification task involves the definition of a critic and also the critic feedback (suggestion to fix the problem identified by the critic). We explain the critic specification task through the icons that were designed for the critic specification editor.

The function of a *CriticShape* icon is to specify and define a critic. We use a mnemonic, **Cr** to mean ‘critic’ and this can help the end user tool developers to remember easily the function of the icon. Similar styles are used for the other icons. Once a critic has been specified and defined, the next step is to define a critic feedback, i.e. a suggestion to fix the critic. Thus, the *CriticFeedbackShape* icon with a mnemonic, **Fb** to denote ‘critic feedback’ is selected to specify the necessary fix action for the particular critic. However, the specification of a critic and a critic feedback involves an association with form-based interfaces that need to be filled in

by the end user tool developers. The critic element is associated with a critic authoring template designed in a form-based interface. The critic authoring template is discussed in the following section. Similarly, the critic feedback element has a link to a form-based interface in order to specify the critic feedback. To support end user tool developers to specify more than just simple critics, we provide a logical operator that consists of *OR*, *AND*, and *XOR* to link between two critics. These two critics possibly have/share the same critic feedback. Thus, we design an operator icon, **Op** to hold these logical operators value-OR AND XOR. These elements are shown in Figure 5.3 (on the first row).

A specified and defined critic should be connected to a critic feedback. The *CriticFeedbackConn* icon is designed to implement this connection. Hence, for every critic that has been specified and defined it will have a solution to fix the problem specified by the critic. In a situation where a critic can be dependent on another critic, in order to show the critic execution sequence we have created a *CriticDependencyLink* icon to represent this. Since we have created an operator icon to link two critics and with a feedback, we need to have connectors that can realize this situation: the *OperatorConn* icon and *OperatorCriticFeedbackConn*. The four connectors are shown in Figure 5.3 (on the second row).

CriticShape	CriticFeedbackShape	Operator	
CriticFeedbackConn	CriticDependencyLi	OperatorCon	OperatorCriticFeedbackCo

Figure 5.3: Icons for the critic specification editor

5.3 Template-Based Approach

A template is defined in the online Cambridge dictionary as “something that is used as a pattern for producing other similar things.” However, the meanings of the term template from a researcher’s point of view are numerous. We quote here, some of the definitions of a template from several studies. In 2003, Czarnecki and Helson published an article in which they described model transformation approaches. They

described template-based approaches as one of the methods to perform the model-to-code transformation (Czarnecki & Helson, 2003). They state a template “usually consists of the target text containing splices of meta-code to access information from the source and to perform code selection and iterative expansion”(Czarnecki & Helson, 2003). In another study by Xiyong and Xingwang (2006), they proposed a template-based approach for the mass customization of service-oriented e-business applications (Xiyong & Xingwang, 2006). They define a template as“ a nearly complete application where the completed parts include an application’s architecture and reusable components.” (Xiyong & Xingwang, 2006). They claimed that the use of a template-based approach has reduced the complex development of the e-business applications. A recent study by Hill et al. (2010) defines a template as “an abstraction that captures the fixed and variable portions of a context,”(Hill, Gokhale, & Schmidt, 2010). They presented four template patterns for improving testing and experimentation (T&E) configurability and scalability for enterprise distributed real-time and embedded (DRE) systems (Hill, et al., 2010).

Although these definitions of a template come from only three studies, a template-based approach is in fact widely employed in numerous research domains. The usage of the template-based approach in various application domains helps users to use their application context in an easy way (Czarnecki & Helson, 2003; Hill, et al., 2010; Xiyong & Xingwang, 2006). This has motivated us to apply a template-based approach for our critic specification task.

This section introduces the template-based approach we used which helps end user tool developers to perform the critic specification task. However, before we describe our critic authoring templates, we first introduce business rule (BR) templates (Loucopoulos & Kadir, 2008) from the business process domain. The introduction of this template is necessary as it has inspired us to adapt its concept to the critic authoring domain.

5.3.1 Introduction to Business Rule Templates

There has been an increasing interest in using business rules modelling in software development environments. Various approaches of business rule modelling exist today (e.g. BROCOM (Herbst, 1997), BRG (Hay & Healy, 2000), BROOD(Loucopoulos & Kadir, 2008), etc). However, the one which has motivated our research in specifying critics for DSVL tools is the Business Rules-driven Object Oriented Design (BROOD) approach (Loucopoulos & Kadir, 2008).

The BROOD approach proposes simple templates for specification of a restricted typology of business rules and a simple object-oriented development process that improves UML by allowing for business rules as an integral part of an object-oriented development (Loucopoulos & Kadir, 2008). The BROOD process is supported by a tool which was developed on top of the Generic Modelling Environment (GME). The BROOD metamodel and business rule (BR) templates were applied to implement the BROOD tool environment. The BROOD metamodel is complemented by a language definition based on the context-free grammar EBNF. EBNF is a meta syntax notation used to express context-free grammar: that is, a formal way to describe computer programming languages and other formal languages (Wang, 2009). The details of the BROOD approach can be found in (Loucopoulos & Kadir, 2008). Loucopoulos and Kadir (2008) described several concerns with the BROOD approach in their published article (Loucopoulos & Kadir, 2008). However, one of the concerns that has motivated our research in specifying critics for DSVL tools is the BR templates.

Business rule templates come from the business rule typology and consist of three main types: constraint, action assertion and derivation (Loucopoulos & Kadir, 2008). The definition and brief description of these three types is shown in Table 5.1 below.

Table 5.1: Definition of constraint, action assertion and derivation (adopted from BROOD approach (Loucopoulos & Kadir, 2008))

Type	Definition and description
Constraints	“... specify the static characteristics of business entities, their attributes, and their relationships. They can be further divided into attribute and relationship constraints. The former specifies the uniqueness, optionality (null), and value check of an entity attribute. The latter asserts the relationship types, as well as the cardinality and roles of each entity participating in a particular relationship.” (Loucopoulos & Kadir, 2008)
Action Assertion	“...concerns a behavioral aspect of the business. Action assertion specifies the <i>action</i> that should be activated on the occurrence of a certain <i>event</i> and possibly on the satisfaction of certain <i>conditions</i> . ” (Loucopoulos & Kadir, 2008)
Derivation	“...derives a new fact based on existing facts. It can be of one of two types i.e. computation, which uses a mathematical calculation or algorithm, to derive a new arithmetic value, or inference, which uses logical deduction or induction to derive a new fact.” (Loucopoulos & Kadir, 2008)

The BROOD approach provides rule templates to allow the expression of business process rules in the business process domain. The rule templates are a formal sentence pattern that act as a guideline to capture and specify business rules (Loucopoulos & Kadir, 2008). Loucopoulos and Kadir (2008) also claim that rule templates offer a way to structure business rule statements. Furthermore, language templates identify the acceptable sentence patterns for business rules statements and express the elements linking business rules and related software design elements (Loucopoulos & Kadir, 2008). In general, rule templates are applied to business process meta-model elements to constrain the target business process model instances. The BR templates that correspond to the three types of business rule typology are shown in Table 5.2.

Table 5.2: Business rule templates (Loucopoulos & Kadir, 2008)

Types	Templates
Attribute Constraint	<entity> must have may have a [unique] <attributeTerm> <attributeTerm1> must be may be <relationalOperator> <value> <attributeTerm2> <attributeTerm> must be in <list>
Relationship Constraint	[<cardinality>]<entity1> is a/an <role> of [<cardinality>]<entity2> [<cardinality>]<entity1> is associated with [<cardinality>]<entity2> <entity1> must have may have [<cardinality>]<entity2> <entity1> is a/an <entity2>
Action Assertion	When <event> [if <condition>] then <action> The templates of <event>: <attributeTerm> is updated <entity> is deleted is created <operation> <rule> is triggered The current date/time is <dateTime> <number><timeUnit>time interval from<dateTime> is reached <number><timeUnit>after<dateTime> <userEvent> The templates of <condition>: <attributeTerm1><relationalOperator><value attributeTerm2> <attributeTerm> [not] in <list> The templates of <action>: trigger <process> <operation> <rule> set <attributeTerm> to <value> create delete <entity> <userAction>
Computation	<attributeTerm> is computed as <algorithm>
Derivation	If <condition> then <fact> The templates of <fact>: <entity> <attributeTerm> is [not] a <value> <entity> may [not] <action>

Inspired by the BROOD approach we have attempted to utilize the BR templates concept in the software tool domain, specifically for our critic specification editor. This was due to the following reasons (Loucopoulos & Kadir, 2008):

- The templates use a language definition based on the context-free grammar EBNF that defines sentence patterns for rule statements;
- The templates use natural language that is easily understood to represent the rules;
- The templates provide guidance for users to help determine the rules;
- The available templates assist the inexperienced user to easily produce consistent rule statements;
- The templates provide a way to construct the rule statements;
- The templates facilitate the linking of rule statements to software design elements.

Inspired by the BR templates approach we adapted this concept to apply it to the critic specification domain, forming a set of reusable critic templates. However, it is essential to note here that not all of the defined BR templates are used for our critic specification purposes. We explain our critic templates in the following section.

5.3.2 Critic Authoring Templates

The motivation for our research in specifying critics for DSVL tools is to provide a development environment whereby tool/end-user developers are supported by Critic Authoring Templates (CATs) by facilitating/supporting simple and more effective critic authoring task.

Our approach to supporting the critic-authoring task is to adapt the concept of “business rule templates” to critic authoring. We took this approach because it has some common points with our research efforts, i.e. development of modelling environments tailored for specific domains and the properties defined in the “business rule templates” match with the description of Marama metamodel elements which is expressed using the Extended Entity Relationship (EER) descriptions. We also chose the BR templates approach to allow end users with limited programming capability to define and author critics for software tools much more easily than using OCL expressions and Java event handlers in Marama.

According to Ginige et al. (1995), authoring “involves identifying structure for the information that supports appropriate accessibility and manipulation.” (Ginige, Lowe, & Robertson, 1995) The term authoring also refers to the process of creating and save the information in a proper manner (Ginige, et al., 1995). With that, we define our critic authoring as a process of specifying and defining a critic and then saving it in a proper way that provides for accessibility and manipulation. This requires adopting suitable approaches for generating these critic structures. As we mentioned previously, we adapted the “business rule templates” approach to our critic authoring templates. We created a simplified set of Critic Authoring Templates (CATs) that allows easier input of critic rules into a DSVL tool environment.

Our CATs are applied to a target DSVL tool’s metamodel to constrain and/or reason about its target model instances. Our CATs do not utilize the complete BR templates approach; they only consist of two types: constraint templates and action assertion templates. Constraint templates specify desired or undesired states of models while action assertion templates specify what to do when an undesired state is detected (including critique generation and possible resolution actions). CATs can be chained together to specify complex patterns over a meta-tool’s model instances and complex critique/resolution strategies. Constraint templates are further divided into two types: *attribute constraint* and *relationship constraint templates*. The former are used to specify desired or undesired properties around uniqueness, optionality (null), and value check of an entity’s attributes (Loucopoulos & Kadir, 2008). The latter assert the relationship types, as well as the cardinality and roles of each entity participating in a particular relationship (Loucopoulos & Kadir, 2008). Chaining a mixture of attribute and relationship templates together allows a tool designer to specify complex detection patterns over their tool meta-model.

Action assertion templates specify an action to be activated on the occurrence of certain event or on the satisfaction of certain conditions. These include critique message generation for the tool user and/or “fix up” operations to apply to resolve detected design problem(s). These templates are shown in Table 1.

Table 5.3. Critic Authoring Templates-constraint and action assertion templates (Loucopoulos & Kadir, 2008).

Types	Templates
Attribute Constraint	<entity> must have may have a [unique] <attributeTerm> <attributeTerm1>must be may be <relationalOperator> <value> <attributeTerm2>
Relationship Constraint	[<cardinality>]<entity1> is a/an <role> of [<cardinality>]<entity2> [<cardinality>]<entity1> is associated with [<cardinality>]<entity2> <entity1> must have may have [<cardinality>]<entity2> <entity1> is a/an <entity2>
Action Assertion	When <event> [if <condition>] then <action>

In Marama, a domain-specific visual language tool meta-model is expressed using an Extended Entity Relationship (EER) diagram which specifies entities and relationships, together with their attributes. When the meta-model is equipped with sufficient information, a critic can be specified via CATs. Thus, each of the templates has a range of properties that specify the meta-model elements and associations they refer to, critique message(s) to generate for the tool user, and model update operations that need to be performed to resolve problems.

To support the critic authoring task, we have designed a form-based interface to represent the CATs. This form-based interface *allows easier input of critic templates into a DSVL tool environment*. The association of critic templates with the corresponding tool meta-model element is shown in Table 5.4, whereas the form-based interface to support the critic authoring task is shown in Figure 5.4. The usage of the CATs in a DSVL tool, specifically our Marama metatools are described and illustrated in Chapter SEVEN and Chapter EIGHT.

Table 5.4: Association of critic template properties with the tool meta-model

Critic template properties	Tool meta-model elements
<entity>	Entity
<attributeTerm>	Attribute
<cardinality>	end1Multiplicity, end2Multiplicity
<role >	associationEndName

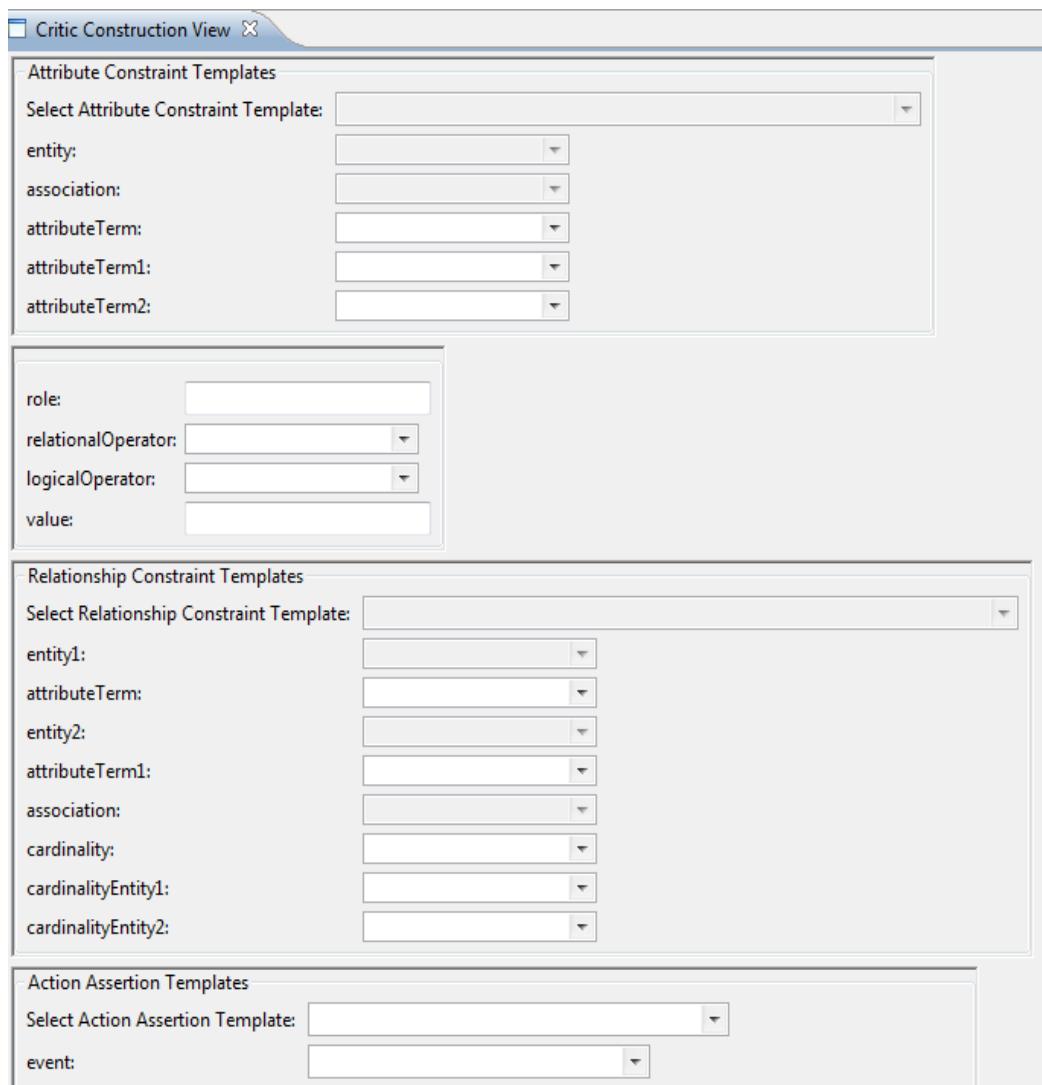


Figure 5.4: A form-based interface to represent the critic authoring templates

5.4 Visual and Template-based Critic Specification for DSVL tools

The combination of the concepts explained in the previous sections results in our combined visual (high level) and template-based (lower level) approach for specifying critics for DSVL tools. Thus, to achieve a ‘simple’ representation (Barton & Barton, 1987) and an ‘intuitive’ representation (Gurr & Tourlas, 2000), we have defined the following requirements for our tool to allow its application in DSVL tools (in our case the Marama meta tools):

1. Simple and intuitive critic specifications, with the necessary constructs/abstractions for the specification of critics;

2. Simple and intuitive critic feedback specifications, with the necessary constructs/abstractions for the specification of critic feedbacks;
3. Simple and intuitive representations in specifying complex critics;
4. Simple and intuitive visual critic specification notation and environment, embedded within a DSVL tool (Marama meta-tool);
5. Simple reuse of common critics and feedbacks, to avoid repeating specification of similar critics for different domains.

The application and examples of this visual and template-based approach for our critic specification editor/tool is described and illustrated in Chapter SEVEN and EIGHT. The following section discusses our analysis of this new critic specification editor approach from Moody's visual language design perspectives.

5.5 Analysis of Critic Specification Tool using Physics of Notations

This section presents the outcome of a brief/partial analysis of our visual notations for the critic specification editor, i.e. Marama Critic definer view. The analysis is based on Moody's new theory of visual notation design, the *Physics of Notations* (Moody, 2008). Our analysis of the visual critic specification editor with the Moody's principles is as follows:

1. Semiotic clarity.

This principle indicates "there should be a one-to-one correspondence between semantic constructs and graphical symbols used in a notation" (Moody, 2008). Assessing the semiotic clarity of a notation involves doing a mapping between the metamodel of the visual critic definer and its symbol set (visual vocabulary). When there is not a 1:1 correspondence, the following anomalies can happen: 1) **symbol deficit**- when a construct is not represented by any symbol, 2) **symbol redundancy**- when a single construct is represented by multiple symbols, 3)**symbol overload**- when a single symbol is used to represent multiple constructs, and 4) **symbol excess**- when a symbol does not represent any construct (Moody, 2008). The mapping

between the metamodel of the critic specification editor and its graphical symbols is shown in Figure 5.5. Table 5.5 shows the mappings between the metamodel constructs and the symbol set.

Table 5.5: Association of metamodel elements and graphical symbol

Metamodel element/construct	Graphical symbol
CriticShape	CriticShape,
CriticFeedbackShape	CriticFeedbackShape,
Operator	Operator,
CriticShape_CriticFeedbackShape	CriticFeedbackConn,
CriticShape_CriticShape	CriticDependencyLink,
CriticShape_Operator	OperatorConn,
Operator_CriticFeedbackShape	OperatorCriticFeedbackConn,

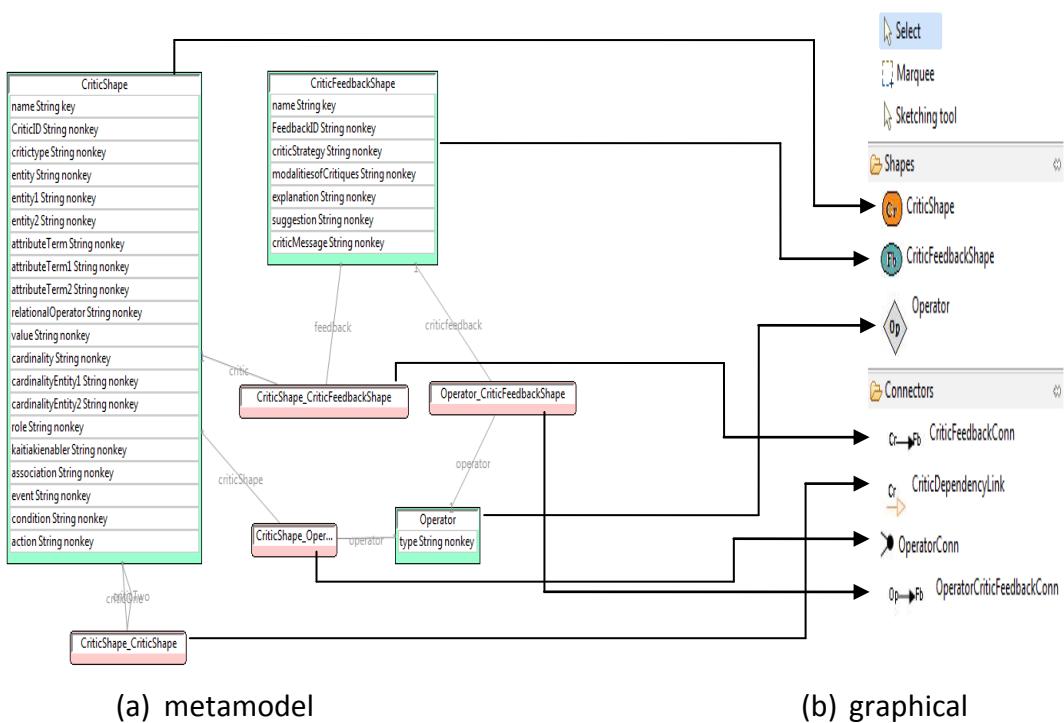


Figure 5.5: The mapping between (a) metamodel of the visual critic definer and (b) graphical symbols.

Each symbol in the visual critic specification editor provides a single meaning, defined in an advanced and independent context. The number of semantic constructs for the critic specification editor is small, so we are able to maintain a 1:1 correspondence between the constructs and the graphical symbols (i.e. a different symbol for each element). Overall, there is no symbol deficit, symbol excess, symbol redundancy, or symbol overload in the critic specification editor. This is shown in Figure 5.5.

2. Perceptual discriminability.

This principle suggests that “different symbols should be clearly distinguishable from each other” (Moody, 2008). Figure 5.5 obviously shows that the symbols for the critic specification editor can be clearly differentiated from each other. We used different shapes to represent critic, critic feedback, operator, and connectors and different icons and colours for different visual appearance. In general, the greater the visual distance between symbols, the faster and more accurately they will be recognised

(Winn, 1993). If there are slight differences, errors in interpretation can result (Moody, 2008; Moody, Heymans, & Matulevicius, 2009).

According to Moody (2008), shape plays a privileged role in perceptual discrimination, because it denotes the key basis on which objects are classified in the real world. Moody (2008) also claims, shapes that are used to represent different constructs have to be differentiated clearly. Figure 5.6 shows the different types of elements used in the critic specification editor. Three of the shapes are 2 dimensional geometric shapes, with very obvious differences between them. In particular, the shapes used to represent critic, feedback and logical operator are very different. Whereas, the other four shapes that represent semantic relationships use textual differentiation of relationships to distinguish between relationship types. Textual differentiation will be mentioned in the Principle of Dual Coding). The relationship types are shown in Figure 6 (the second row).

CriticShape 	CriticFeedbackShape 	Operator 
CriticFeedbackConn 	CriticDependencyLink 	OperatorConn 

Figure 5.6: Element types in the visual critic specification editor

3. Semantic transparency.

Semantic transparency is regarded as the “extent to which the meaning of a symbol can be inferred from its appearance” (Moody, 2008). This principle requires that symbols provide indications to their meaning. According to Moody (2008) a symbol is semantically transparent if a novice user/reader can guess the meaning only from the look of the graphical symbol (e.g. a stick figure to present a person).

The main constructs in the critic specification editor are: critic and critic feedback. We use icons to represent the two constructs. We have already explained the icons concept in an earlier section: replacing the abstract shapes with icons can improve the understanding of models by the novice users/readers (Masri, Parker, & Gemono, 2008). Furthermore, icons improve

likeability and accessibility (Bar & Neta, 2006; Petre, 1995). The icons for critic and critic feedback are distinguished based on colour and shape. Text is also used to clarify the meaning of the two constructs. This is shown in Figure 5.6. However, the symbols for the critic specification editor do not support the principle of semantic transparency. We can consider this aspect in our future work for the improvement of the critic specification tool.

4. Complexity management.

This principle refers to the “ability of a visual notation to represent information without overloading the human mind” (Moody, 2008). Moody (2008) refers to “complexity” as “diagram complexity: the number of elements (symbol instances) on a diagram.” It is very important to have an effective complexity management specially when dealing with novice users who are incapable of managing complexity (Sweller, 1994). There are claims, that excessive complexity is one of the main difficulties for end user understanding of software engineering diagrams (Moody, 2002; Shanks & Darke, 1998). Thus, to effectively represent complex situations, visual notations must provide mechanisms for modularisation and hierarchical structure (Moody, 2008).

We noted in the earlier section that our critic specification editor is an extension of our Marama meta-tools. Currently Marama-based tools are defined and developed using three metatool editors: 1) the *metamodel definer*, defining a tool’s information model; 2) the *shape designer view*, defining visual notational elements; and 3) the *viewtype definer view* to specify mappings of meta-elements to visual representations ("Marama meta-tools," 2008).

The newly created view/editor, i.e. *critic definer view* permits specification of a DSVL tool’s critics. Hence, we could say that these four different editors/views together supported the complexity management mechanism and also modularisation because problems are represented in multiple diagrams. The critic specification editor (i.e. *critic definer view*) by itself however do not support the complexity management mechanisms, which means that critic model must be represented as single monolithic diagram, no

matter how complex it becomes. The critic specification editor also has no modularisation mechanisms. Again, this aspect can be considered in our future work if there is a need to allow for multiple critic specification diagrams which of course would provide complexity management. Figure 5.7 shows an example of critics and feedbacks modelled in one diagram.

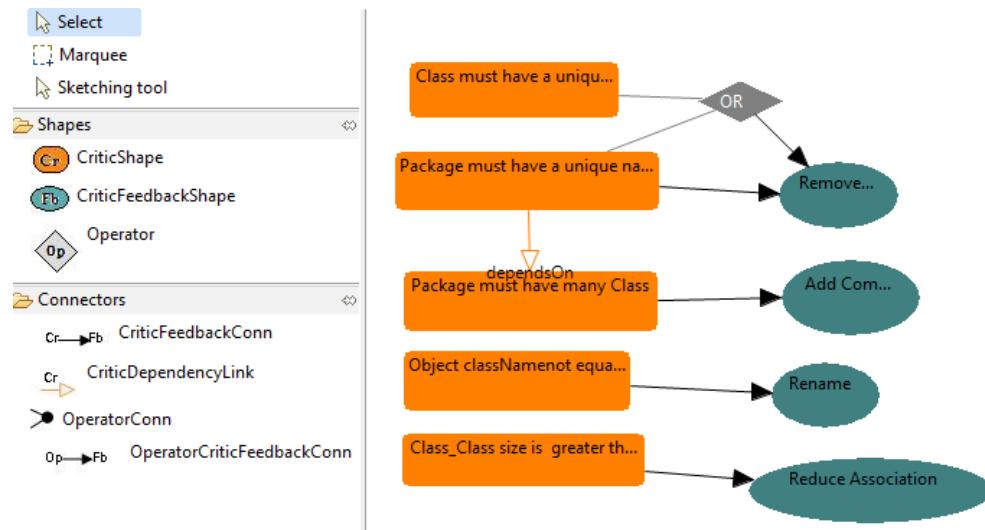


Figure 5.7: Critic specification diagram

5. Cognitive integration.

This principle suggests including “explicit mechanisms to support integration of information from different diagrams” (Moody, 2008). Moody (2008) reported that, when multiple diagrams are employed to represent a system then the cognitive integration role is necessary. This principle is closely related to the previous one-Complexity management. Siau (2004) argues if multiple diagrams are used to represent the systems, then a reader/user is required to be able to keep track of the diagrams flow and manage to integrate the information from several diagrams, and this requires additional cognitive demands (processing) on the reader/user (Siau, 2004).

In our case, the critic specification editor/view provides a diagram that models critic specification for a DSVL tool. Thus, in order to specify the tool’s critics, the information expressed in a meta-diagram (i.e. *metamodel definer view* (1)) is used as an input to the critic specification editor (i.e *critic definer view* (2)). The critic input process is performed via a form-based

critic construction editor interface, i.e. *Critic Construction view* (3). The list of available critic authoring templates is designed in a drop-down menu. A user needs to select from the drop-down menu the required critic template and the properties of that particular template are accessed from the meta-model elements. Thus, a user will only select the required property value that is shown in the drop down menu list, which can avoid the error proneness from the user when entering an input. The integration mechanism is illustrated in Figure 5.8. The defined critics can then be realized in another diagram i.e. Marama Diagram. Furthermore, the critic specification editor is integrated with the other views (i.e. the *shape designer view*, and *viewtype view*) to support the critic specification diagram.

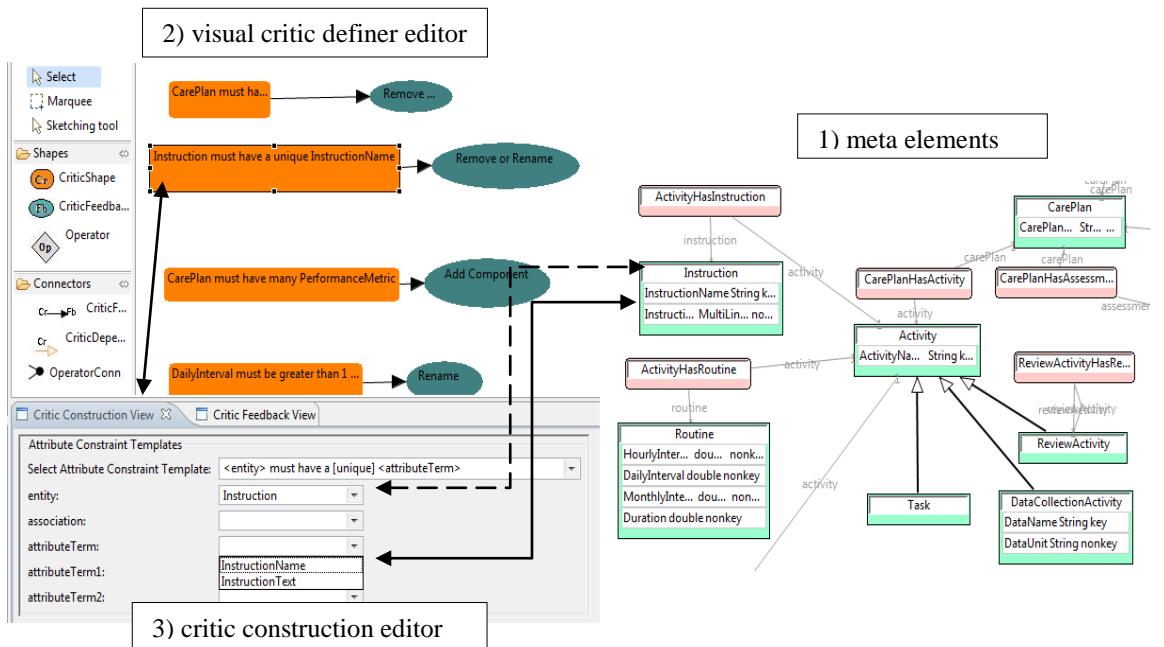


Figure 5.8: Integration between critic definer view and critic construction editor, and integration between critic construction editor and meta elements.

6. Visual expressiveness.

Visual expressiveness is related to the “number of visual variables used in a notation” (Moody, 2008). Moody (2008) points out that visual expressiveness measures the visual variation for the complete visual vocabulary. This principle measures the “utilisation of the graphic design space.” (Moody, et al., 2009).

Our critic specification editor uses only two visual variables: shape and colour. We do not use all the 8 visual variables (horizontal and vertical position, size, brightness, color, texture, shape and orientation) as the number of our graphical symbols is small (i.e. only 7 symbols/icons). Although shape is considered as one of the least powerful visual variables (G. L. Lohse, Min, & Olson, 1995) we manage to use different shapes and icons to represent the critic, critic feedback, operator, and the four connectors (*CriticFeedbackConn*, *CriticDependencyLink*, *OperatorConn* and *OperatorCriticFeedbackConn*). We also apply colour to the symbols to increase the visual expressiveness for our critic specification notation. This is shown in Figure 5.7. In fact, (Mackinlay, 1986; Winn, 1993) reported that the human visual system is very sensitive to variations in colour and easily can differentiate the colours. Thus, we believe that the shape, iconic and colour elements provide sufficient visual expressiveness.

7. Dual coding.

This principle suggests “using text to complement graphics” (Moody, 2008). Our critic specification editor uses text to define the properties of critic, critic feedback and operator, as shown in Figure 7. We use colour, icon and shape to differentiate these three elements, but we also complement them with textual annotation. This supports Moody’s (2008) assertion that text can be “usefully used as a form of redundant coding to reinforce and clarify meaning”. Also, one of the four connectors which represent the critic dependency relationship, i.e. the *CriticDependencyLink* is supplemented with a text name as shown in Figure 5.9.

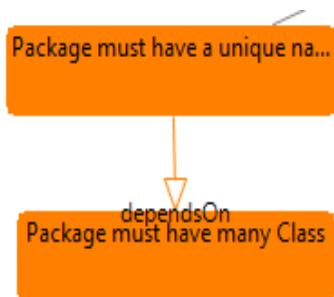


Figure 5.9: Textual encoding

8. Graphic economy.

This principle indicates that the “number of different graphical symbols should be cognitively manageable” (Moody, 2008). Only seven different symbols are used in the critic specification editor notation. These symbols are shown in Figure 5.7. Thus, our critic specification editor has a very simple and highly discriminable visual vocabulary which supports usability and end user interaction.

9. Cognitive fit.

This principle is related to the “use of different visual dialects for different tasks and audiences” (Moody, 2008). We do address this principle in terms of having multiple representations for different tasks. We stated previously that we have three other editors, namely the *metamodel definer*, *shape designer view*, and *viewtype view* within the Marama meta-tool set. We then create a new view - critic specification editor that offers simple and intuitive representations with the aim to assist especially the end user tool developers (i.e novice developers) in specifying tool’s critics. We also provide several other editors that are linked with the critic specification editor. These include the critic construction editor, critic feedback editor, and critic template editor. Ideally we provide a visual critic specification tool for authoring and generating Marama design critic implementations. This also fits well with the other visual meta-tool editors within Marama.

Analysis using Moody’s Physics of Notations principles can be used to improve the usability and effectiveness of the critic specification editor. The improvement is mainly for the purpose of interacting with end user tool developers. Thus, we will improve any minor mistakes and eliminate any potential difficulties to its usage in practice.

5.6 Conclusion

We have described our approach for specifying critics for a DSVL tool environment. The two main approaches that we employed are: visual approach and template-based approach. We introduce these concepts and then relate them to our critic specification development. The combination of these two approaches forms what we call a ‘visual and template-based approach in specifying critics for DSVL tools’.

We have explained and demonstrated the visual notations of the critic specification editor. Following that, we described our adaptation of the business rule templates to the software tool domain, specifically our critic authoring domain. We then explained the critic authoring templates that assist the end user tool developers to specify critics.

Applying the two approaches in our critic specification development has led us to carry out a brief analysis based on the Moody’s principles. We can say that we do satisfy most of the Moody’s principles (Moody, 2008) for designing effective visual notations. We demonstrate this with a target end user evaluation in Chapter NINE and we believe that with the visual and template-based approach applied to the critic specification development, end user tool developers can be supported to specify critics for a DSVL tool in a simple and effective way.

Chapter 6

Initial Prototype for Critic Specification Tool

This chapter introduces and explains the development steps of the visual and template-based approach for our critic specification tool. We explain our first attempt to employ MaramaTatau (N. Liu, et al., 2007) in specifying critics for Marama-based tools which became our motivation to develop another prototype for the critic specification tool. We then describe the second prototype, which specifies critics in the meta-model editor using a similar visual approach to MaramaTatau however tailored to the critic specification rather than the constraints domain.

6.1 Introduction

Inspired by the existing research about critic specification tools, we made an attempt to apply similar ideas to our meta-modelling tools, called Marama (Grundy, et al., 2008). Marama is a meta-tool implemented as set of Eclipse plug-ins. It includes both meta-tools and generated modelling tools (Grundy, et al., 2008). Most of the existing critic tools that we reviewed are not developed within the context of a meta-modelling tool. Our meta-tools are used to generate complex visual modelling tools, and these modelling tools could benefit from the addition of various critics. Thus, we wanted to extend our Marama meta-tools by embedding a critic design and generation component. The main purpose of our work is to assist end-user tool developers to specify and generate critics efficiently and easily. We demonstrated a proof-of-concept of our visual critic specification approach by developing a set of incremental prototypes within the Marama meta-tool.

In this chapter we present the background and motivation of our critic development approach. We describe the design and implementation of our approach for specifying DSVL tool critics via the following incremental prototypes: 1) Specifying critics using Object Constraint Language (OCL) formulas via MaramaTatau; and 2) Specifying critics at the Marama meta-model editor by creating a new functional

item, *CriticShape* associated with critic-authoring templates. These prototypes are explained and evaluated in the following sections.

6.2 Initial Prototype: Specifying Critic in a Marama Metamodel Definer views

6.2.1 Background and Motivation

The motivation for the initial prototype emerges from the work of Liu et al. on MaramaTatau (N. Liu, et al., 2007), an extension to the developed Marama metatool set (Grundy, et al., 2008). MaramaTatau offers the facility to specify behavioural extensions to Marama metamodels. The main notation used in MaramTatau is declarative Object Constraint Language (OCL) expressions. A complete description of MaramaTatau is in (N. Liu, et al., 2007).

Our initial approach (as labelled **Prototype 1** in Figure 3.1) was to experiment with applying the OCL expressions used in MaramaTatau (N. Liu, et al., 2007) to specify and implement critics for a Marama-based tool. To provide a basis for our experimentation, we developed a very simple UML class diagramming tool using Marama. The tool metamodel is defined in Marama metamodel editor, as shown in Figure 6.1.

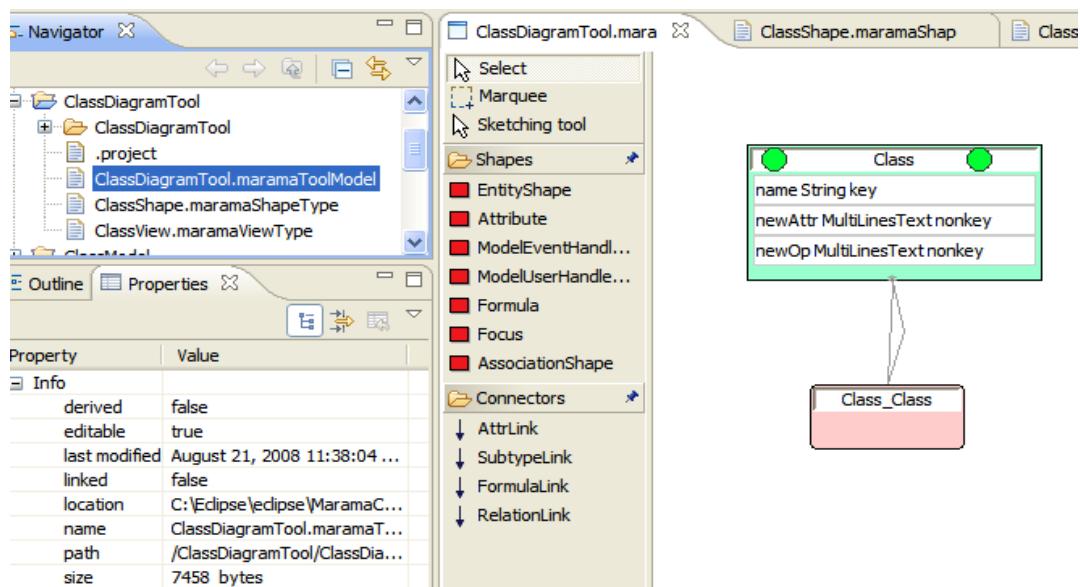


Figure 6.1: UML class diagramming tool metamodel

We identified and translated several critics for UML class design into the OCL expressions using MaramaTatau and associated them with the UML tool metamodel. A green circle annotation shown in Figure 6.1 indicates that an OCL expression has been defined to specify a critic for the UML tool. Examples of the critic statement and OCL expression in specifying critics for the UML class diagramming tool are shown in Table 6.1. These critics are then applied in the executing tool, which is at the Marama diagram level, as shown in Figure 6.2 and Figure 6.3.

Table 6.1. Critic statement and OCL expression

Critic Statement	OCL expression
Class must have a unique name	Class.allInstances()->forAll(c1,c2 c1 < > c2 implies c1.name < > c2.name)
Class with no name	self.name< >''
Class name should begin with a capital letter	not(let firstChar:string=self.name.substring(1,1) in firstChar < > firstChar.toUpperCase())

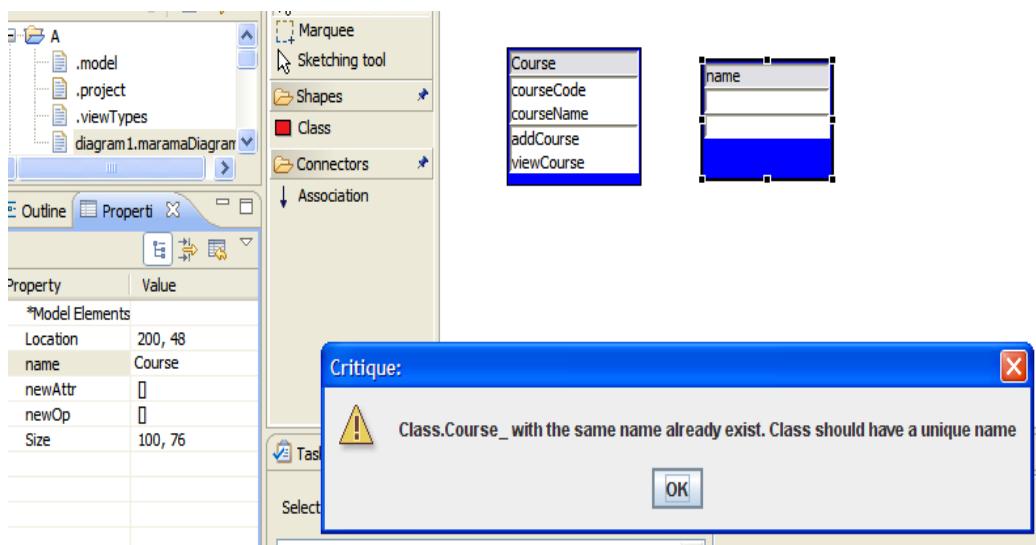


Figure 6.2: Simple critic (same named classes) violation in MaramaTatau

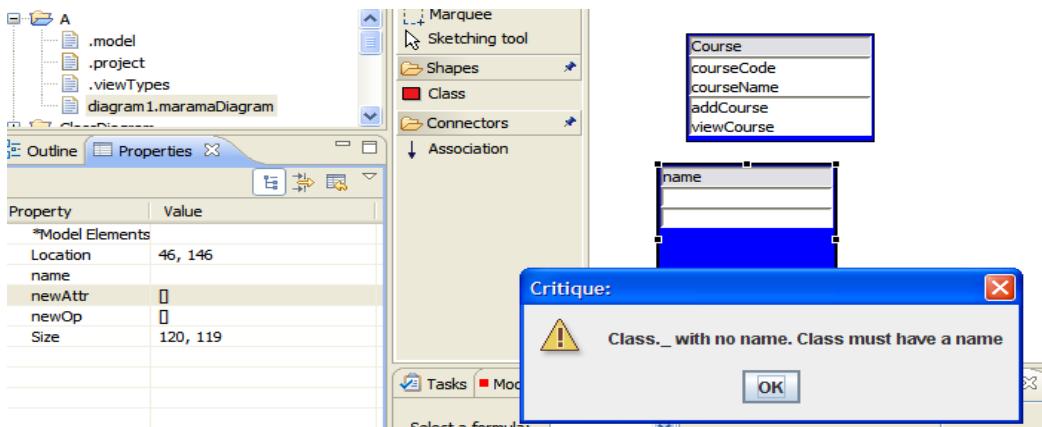


Figure 6.3: Simple critic (class with no name) violation in MaramaTatau

Our experience gained from this initial attempt demonstrated some difficulties, particularly for novice tool developers. OCL expressions are a powerful technique for expressing constraints in a meta-tool. However, some of the barriers in expressing critics using such OCL expressions include:

- OCL is not easy to understand and even harder to write (Sourrouille & Caplat, 2002) specifically for many novice users and tool developers ;
- Users who lack knowledge of OCL will have problems in specifying critics using OCL expressions. This reflects the *hard mental operations* dimension from the CDs framework (Green & Blackwell, 1998) that suggests the demand for cognitive resources. Users must remember what function is appropriate (Liu, 2007) for specifying a given critic. This argument supports previous observations made by (Sourrouille & Caplat, 2002);
- Difficulty in expressing (Sourrouille & Caplat, 2002) meaningful critics due to unfamiliarity with OCL can lead to *error proneness* as suggested by the *error proneness* dimension from the CDs framework (Green & Blackwell, 1998). The *error proneness* dimension refers to the ability of the tool to induce ‘careless mistakes’ (Green & Blackwell, 1998) . Users will make careless mistakes if they have a difficulty when specifying critics using OCL expressions. This dimension has a similar issue with the hard mental operations dimension;

- Specifying critics via OCL expressions provides a *high abstraction gradient* (Green & Blackwell, 1998) for novice users as they need to learn how to use OCL with a meta-modelling approach. As mentioned by the author of MaramaTatau (N. Liu, et al., 2007), the combination of OCL formula and spreadsheet interfaces was designed to support the target end users who are programming literate and familiar with modelling concepts for constraint/dependency specification (Liu, 2007).
- OCL is a general purpose constraint specification language, which is not designed for use in a meta-tool specification environment. It is not designed to express DSVL tool critics at all i.e. generating or enforcing design idioms for DSVL tools. Thus it lacks a “closeness of mapping” to the target domain of critic specification and implementation.
- Our assessment above was demonstrated by using Marama in two advanced software engineering courses at the University of Auckland in 2007, 2008 and 2009. In our experiments, final year Software Engineering undergraduate students and first year Computer Science post-graduate students used Marama to build simple DSVL tools with critics and constraints expressed in OCL. Most indicated in their reports that it is difficult to use OCL constraints as implemented via MaramaTatau to specify even very simple critics in their tools.

However, the attempt proved a useful stepping stone towards our understanding of the necessary building blocks for the critic specification tool. We prefer a visual specification tool for authoring and generating a Marama design critic implementation. This would then fit well with the other visual meta-tools we have developed for the Marama platform.

Due to the barriers noted above, we see an opportunity for a visual design notation to represent critics. The need to specify critics in a simple way by using an easy to use, high-level language is the motivation for our research in visual critic-authoring for domain-specific visual language tools. We also wanted a visual language with

good “closeness of mapping” to the critic authoring domain of discourse, and associated IDE support in the Marama meta-tool environment.

6.2.2 Approach

We developed a new critic-authoring support extension (as labelled **Prototype 2** in Figure 3.1) to the previously-developed Marama metatool set and applied a similar visual approach as MaramaTatau. This provides a new meta-tool facility for our Marama-based tools. The new visual critic-authoring support provides the ability to simply specify critics to Marama metamodels. Figure 6.4 illustrates the process of constructing and using critics in Marama-based tools using this approach.

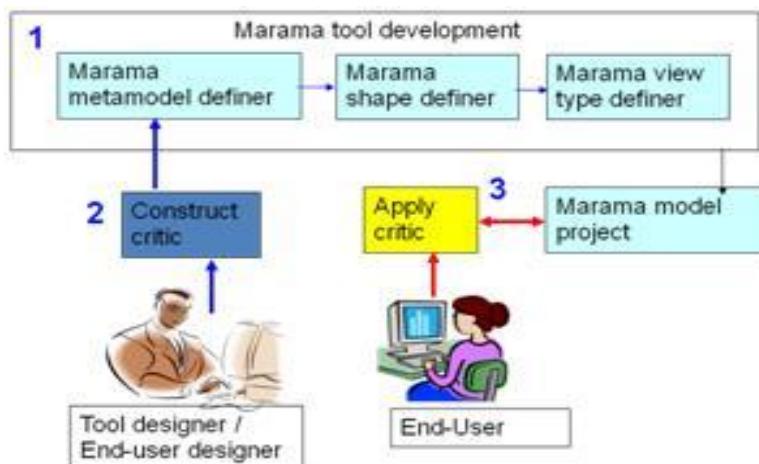


Figure 6.4: Critic development approach

Initially a target end-user developer uses the Marama meta-tools to develop a Marama-based tool (**1**). A set of core Eclipse plug-ins provides diagram and model management support for Marama modelling tools. The development of a new Marama tool starts by specifying the tool metamodel via *Marama Metamodel Definer* views. A meta-model for the tool specifies entities and relationships, together with their attributes. Once the meta-model is defined, shapes and connectors are specified via the *Marama Shape Designer* views to provide visual representations of the tool.

The next step is to specify the “view type” (i.e. specific diagram type specification) for the tool via the *Marama Viewtype Definer* views. This describes the mapping of meta-model elements to visual representations. This results in a new Marama tool

for which the tool or end-user developer can specify critics. Critics are specified in the Marama metamodel definer views (2) via a new *CriticShape* function that we have added to the metamodel editor. Once the critics are defined, a tool user can open or create new modelling projects and diagrams using the plug-ins. Critics for the tool are applied when a diagram is created. If the user creates a diagram that violates the design rules of the tool, then a critique will be displayed to notify the user about the potential errors or problems in the diagram (3).

As stated above, a new functional item, *CriticShape* (please refer to Figure 6.5) was added to the existing Marama meta-model editor to provide the visual critic-authoring support extension. This function allows end-user developers to specify and define critics based on the tool specification. It also has an appropriate underlying infrastructure allowing the critic to be generated by Marama. We associate the *CriticShape* function with a critic authoring template using a form-based style to facilitate tool and end-user developers to construct relevant critics for the new Marama tool. Critic shapes are linked to relevant tool specification elements to show users the items they are dependent on. The following section explains the creation of our initial critic authoring template.

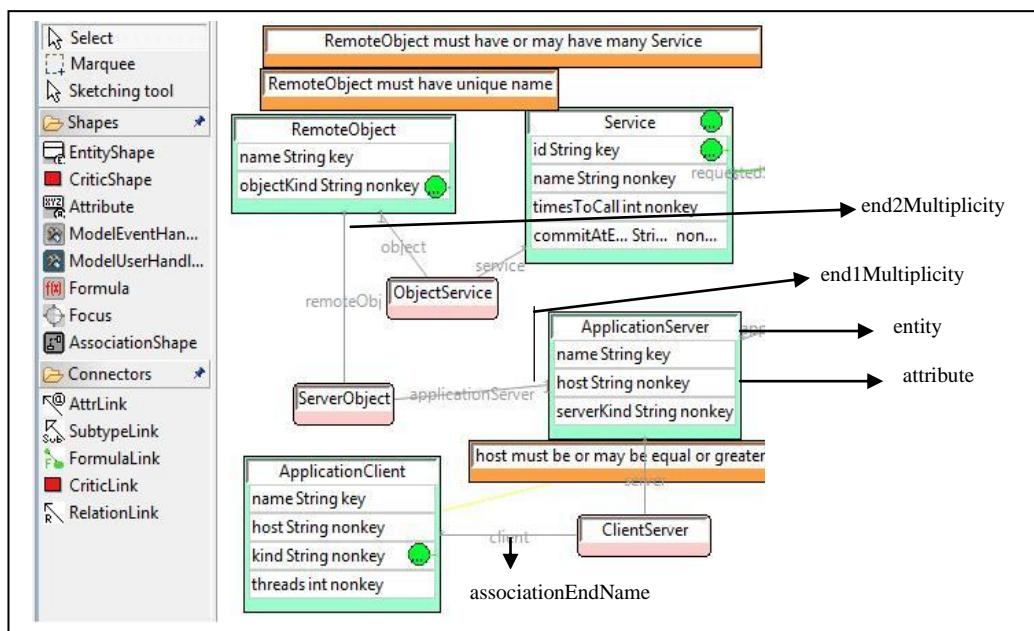


Figure 6.5: Critics specified in the meta-model definer editor

6.2.3 Initial Critic Authoring Template

Our initial attempt in specifying critic at a Marama metamodel definer view only involves the creation of a critic authoring template that focuses on two constraints from the business rule template. These two constraints are attribute constraint templates and relationship constraint templates. Table 6.2 shows the templates for each constraint. Thus, the creation of critic authoring templates adopts the attribute and relationship constraints as shown in Table 6.2. We assist the tool/end-user developers to specify critics for Marama-based tools by using these critic authoring templates via the attribute and relationship constraints.

Table 6.2: Attribute and relationship constraint templates (adopted from (Loucopoulos & Kadir, 2008)

Type	Template
Attribute Constraint	<entity> must have may have a [unique]<attributeTerm>. <attributeTerm1> must be may be <relationalOperator> <value> <attributeTerm2>.
Relationship Constraint	[<cardinality>]<entity1> is a/an <role> of [<cardinality>]<entity2>. [<cardinality>]<entity1> is associated with [<cardinality>]<entity2>. <entity1>must have may have [<cardinality>]<entity2>. <entity1> is a/an <entity2>.

Critics for Marama-based tools are specified using the Marama meta-model definer view. The tool meta-model is expressed using an Extended Entity Relationship (EER) description. This is shown in Figure 6.5. The tool meta-model elements match the properties defined in the attribute and relationship constraint templates. The association of the tool meta-model element with the critic phrase type is shown in Table 6.3.

Table 6.3: Association of tool meta-model with the critic phrase type

Tool meta-model elements	Critic phrase type
Entity	<entity>
attribute	<attributeTerm>
end1Multiplicity, end2Multiplicity	<cardinality>
associationEndName	<role>

The following section describes the implementation of the visual critic authoring support extension via the attribute and relationship constraint templates.

6.2.4 Implementation

We implemented our visual critic authoring approach by adding a new functional item to the Marama meta-model editor. This new function is called **CriticShape** with a connector, **CriticLink**. The new function provides the end-user/tool developer with a way to add several critics to a tool specification. Figure 6.6 shows the new function. Associated with the *CriticShape* is a critic authoring template. We designed a form-based interface to facilitate the critic-authoring task by end-user/tool developers. Figure 6.7 shows the association of *CriticShape* with the critic authoring template.

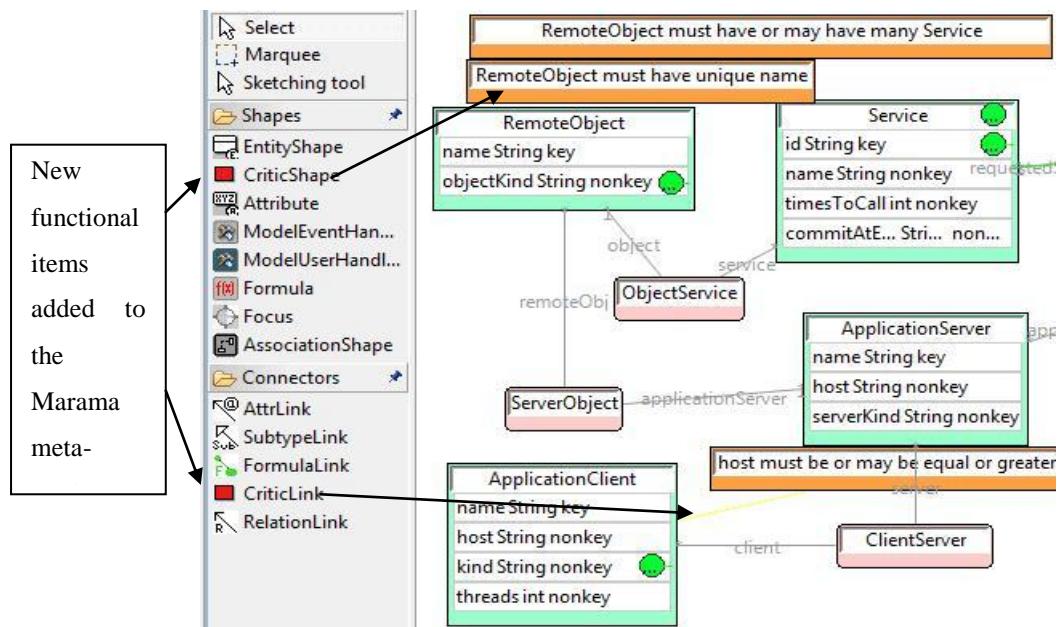


Figure 6.6: New function added in the Marama meta-model editor.

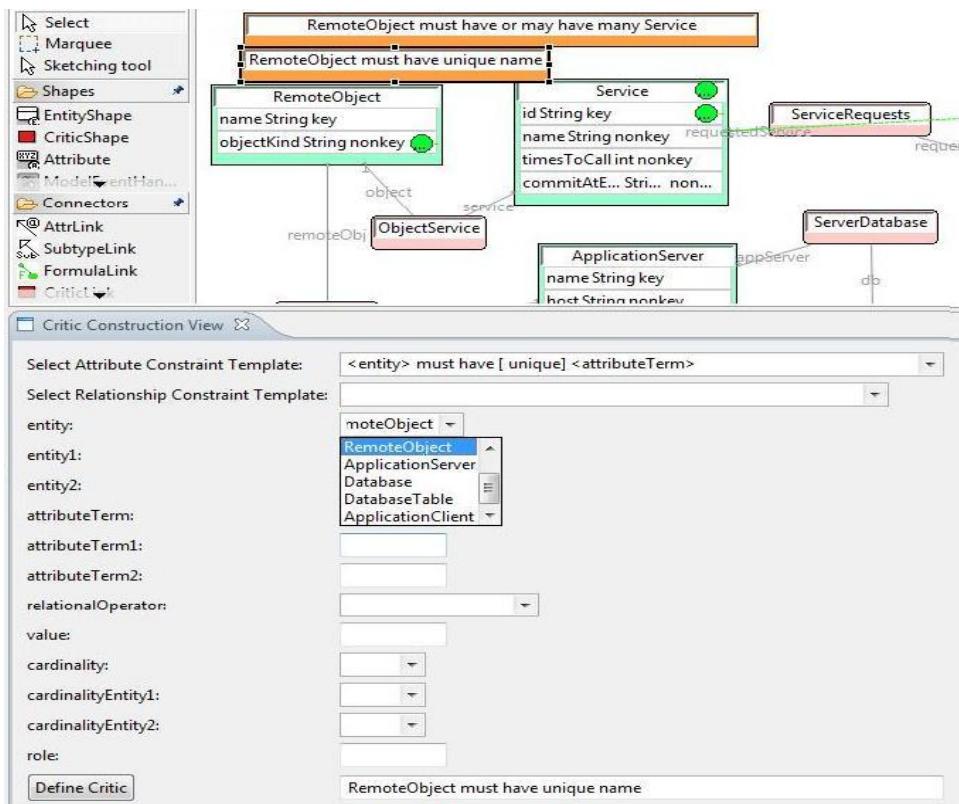


Figure 6.7: CriticShape (orange colour) linked with a critic authoring template.

The *CriticShape* and *CriticLink* functions are connected to relevant tool specification elements that represent the critic of that particular tool. The critic authoring template for this initial prototype of our visual critic authoring tool only covers the attribute and relationship constraint templates. A tool/end-user developer specifies critics by selecting the *CriticShape* function and then constructing and defining the relevant critic for the tool via the *Critic Construction View* interface. This is shown in Figure 6.7. The *CriticLink* function is used in critic authoring especially for the attribute constraint templates, where the *<entity>* is not stated. An example of this is shown in Figure 6.6, where a dotted orange line is connected to one of the entities defined in the meta-model editor.

We added a critic type folder to the Marama meta-model folder as a repository to store the list of critics that are defined for the new Marama tool. Thus, when a tool/end-user developer specifies and defines critics, these will then be shown in the *critictypes* folder as shown in Figure 6.8. Each critic is stored as an XML data file. A ‘critic engine’ loads the XML save files and instantiates and runs an ‘event listener’

in Marama for each of the critics defined for the new Marama tool. This event listener receives model update events and fires the critic implementation to implement the critic behaviour when appropriate.

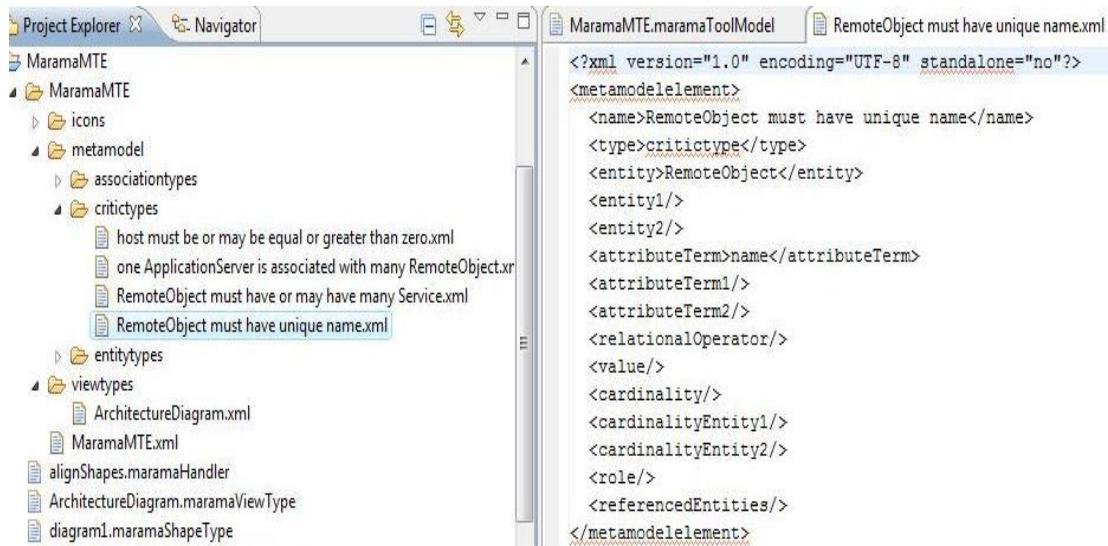


Figure 6.8: Critics store in critictypes folder.

In our initial prototype we only applied the attribute and relationship constraint templates for the critic-authoring task. There are two templates for the attribute constraints and four templates for the relationship constraints. Each of these templates represents a critic event. Thus, a ‘critic processor’ is assigned to each critic event. Whenever a model is created or changed, an event listener receives this event and decides if a particular critic is interested in the event and what action to perform.

Each critic template represents a critic type and we implement each critic as a concrete class. A critic processor class is instantiated using the stored XML information to decide which model element events it is interested in; patterns to match in terms of model state; and its action when receiving change events and matching part of the model state.

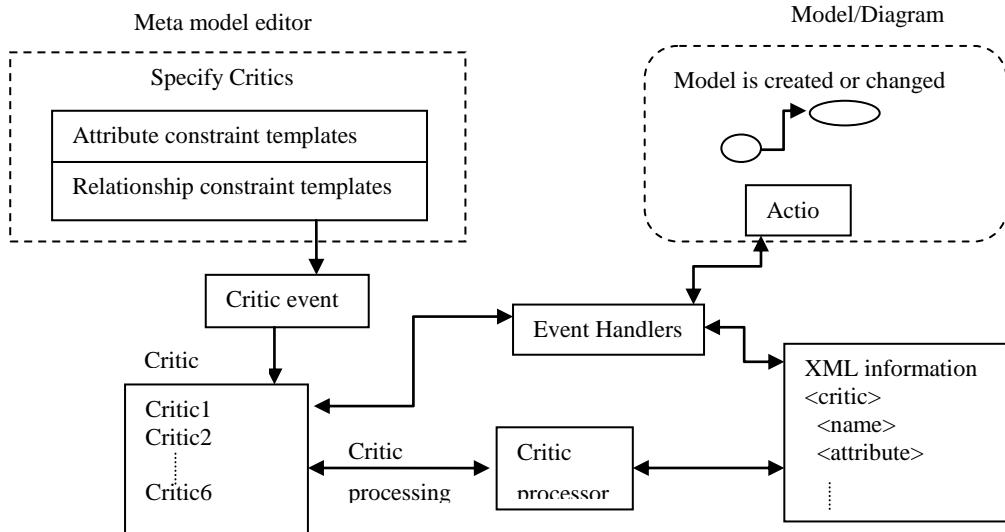


Figure 6.9: Architecture of critic processing

6.2.5 Example Usage

We demonstrate our initial prototype of visual critic authoring capabilities by applying it to an existing Marama-based tool, the MaramaMTE software architecture design tool (Grundy, Hosking, Li & Liu, 2006). Initially a tool/end-user developer specifies a design tool using a set of visual Marama meta-tools. For the MaramaMTE example, a tool developer has specified a variety of entities and associations to represent the structure of software architecture e.g. remote objects, clients, servers, services, requests, databases and various relationships. The meta-model of MaramaMTE is shown in Figure 6.10.

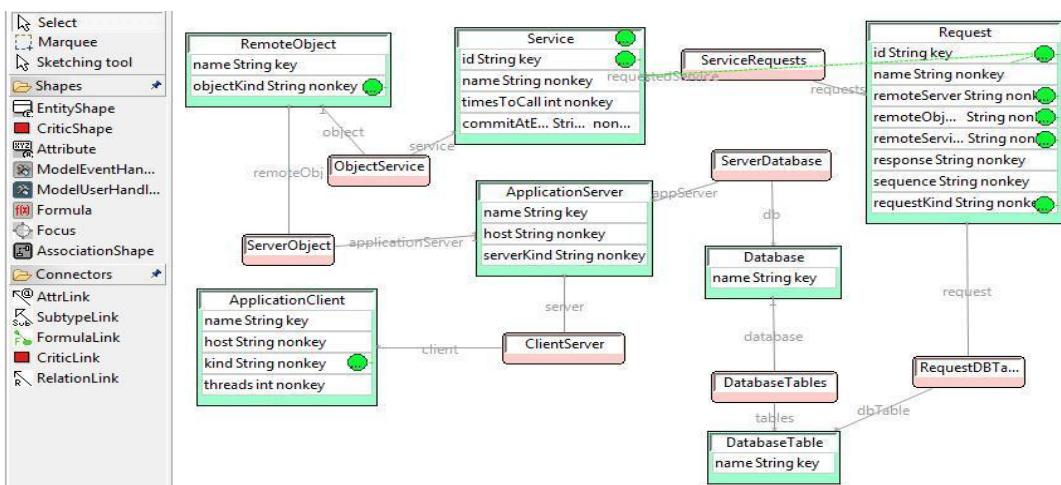


Figure 6.10: MaramaMTE metamodel definer view

The tool developer has also specified using the shape designer and view designer of MaramaMTE tool the various shapes, connectors and view (diagram) types for MaramaMTE. We then specify the relevant critics using a basic understanding and knowledge of MaramaMTE in order to generate a new version of the MaramaMTE tool with additional design critic support features. We list several critic statements that are pertinent to MaramaMTE domain and map these critic statements to critic authoring templates. Table 6.4 lists several examples of critic statements using the critic authoring templates for the MaramaMTE tool.

Table 6.4: Lists of critic statements and critic authoring templates for MaramaMTE.

Critic Statement	Critic authoring template	Template type
1. Remote object must have a unique name	<Entity> must have a [unique] <attributeTerm>	Attribute constraint
2. Threads must be greater than 3	<threads> must be <greater than> <3>	Attribute constraint
3. Remote object must have or may have many services	<entity1> must have may have <cardinality><entity2>	Relationship constraint
4. Application server is associated with many remote objects.	[<one>]<application server> is associated with [<many>]<remote object>.	Relationship constraint

The critics for MaramaMTE include completeness of the architecture design e.g. all elements linked by appropriate relationships; correctness of the architecture design e.g. no same-named services for the same remote object or same-named tables for the database; and “quality” of the architecture design i.e. checking for particular architecture styles e.g. if all services are in a single remote object; if redundancy is supported; and so on.

We specify critics for MaramaMTE via the visual *CriticShape* function from the Marama meta-model editor. Selecting the visual *CriticShape* function causes a critic authoring template in a form-based style to come into view. This view, *Critic Construction View* is displayed to guide the critic authoring task. This is shown in Figure 6.11.

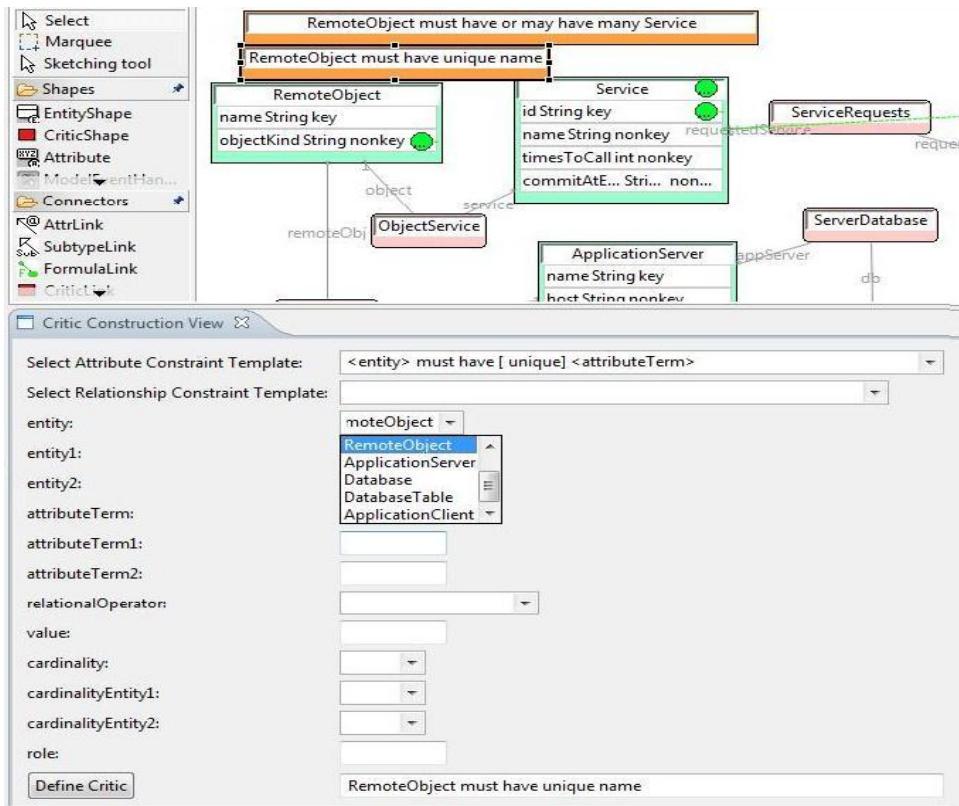


Figure 6.11: Visual CriticShape function associate with the critic authoring templates.

We specify the critics for MaramaMTE tool based on the critic authoring templates provided in the critic construction view interface. Once a critic is specified based on the selected critic template, the properties of the critic template are then selected and finally a ‘define critic’ button is clicked (refer to Figure 6.11). For example, in Figure 6.11, a critic for RemoteObject entity is being specified using an attribute uniqueness pattern to ensure RemoteObjects have a unique name. The critic shapes that represent the defined critic then appear in the meta-model editor together with other visual shapes i.e. entity shapes, association shapes, and formula shapes (MaramaTatau). The list of defined critics is then stored in a repository called *critictypes*, as shown in Figure 6.12.

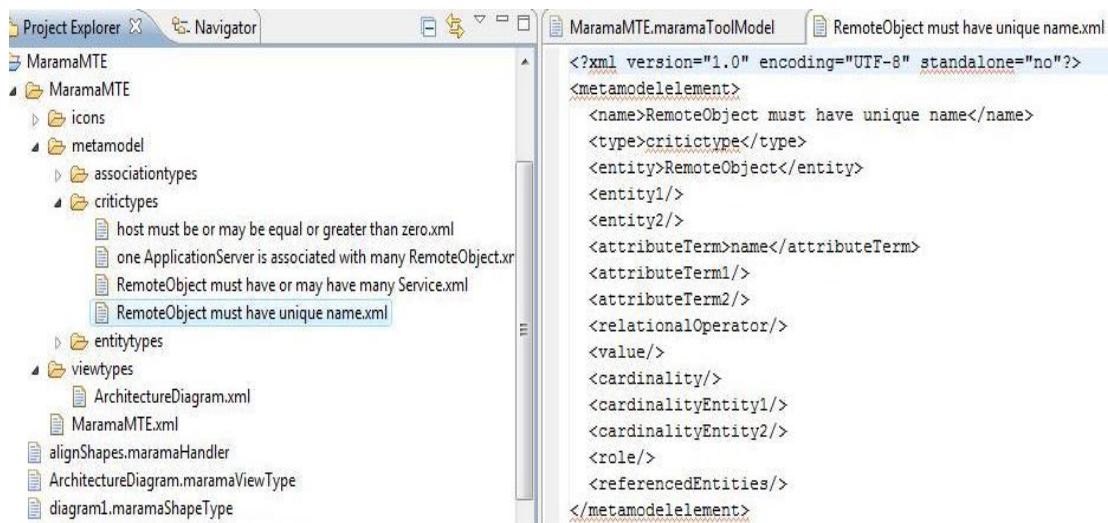


Figure 6.12: Critics for MaramaMTE are stored in critictypes folder

These critics are then applied when loading and running a Marama tool i.e. at the model or Marama diagram level. When Marama loads the definition of a tool it also loads the critic definitions. It then instantiates the generated “event listeners” on the tool meta-model elements so that when these are changed, the ‘critic engine’ is informed of the changed state. The critic engine then determines which critic(s) are associated with the change and whether the critic action criteria have been met by the current state of the design. If so, the critic action is invoked via a message to the user.

In Figure 6.13, a critic monitors and detects violation of the uniqueness constraint specified for a remote object. This is an example of a correctness critic using the attribute constraint template critic. In Figure 6.14, a critic is detecting the lack of a service for a remote object. This is an example of implementing a “completeness” critic on the design for the remote service using the relationship constraint template critic.

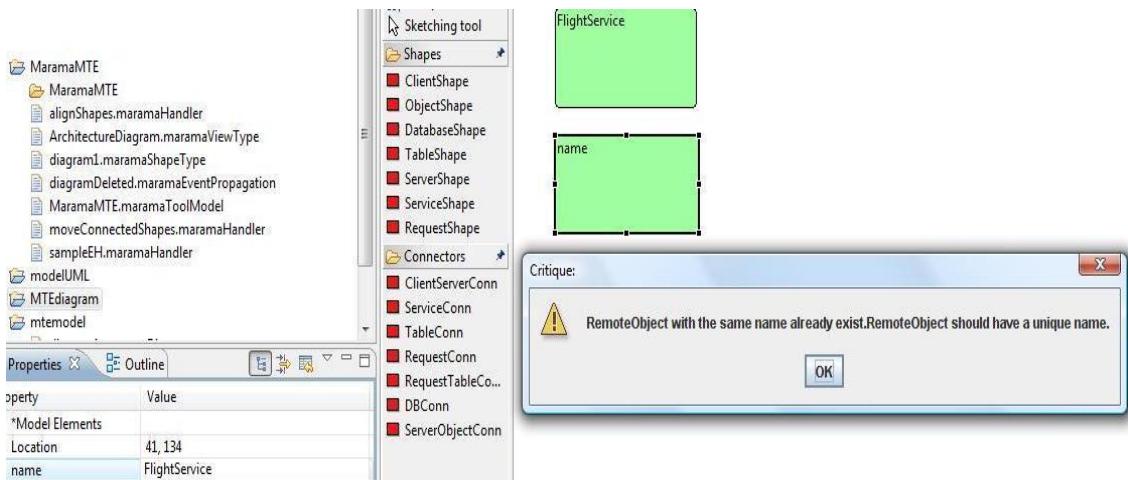


Figure 6.13: Critic statement: remote object must have a unique name.
Attribute Constraint template: <entity>must have|may have [unique]
<attributeTerm>

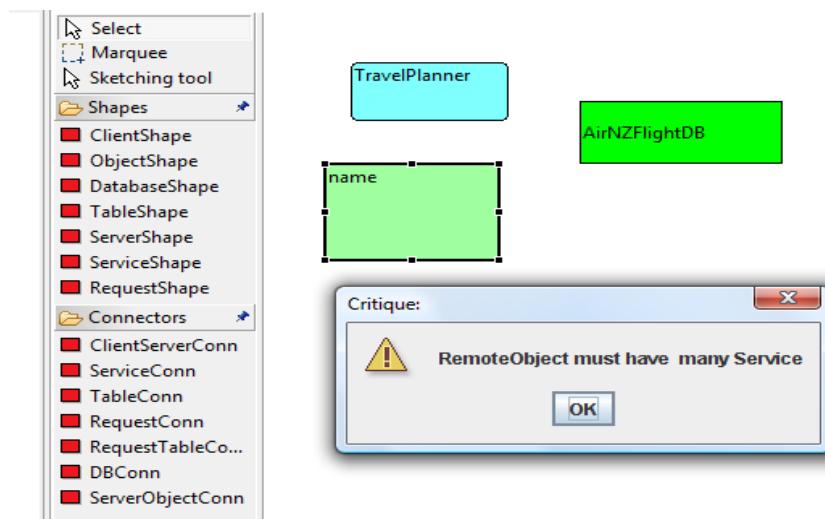


Figure 6.14: Critic statement: remote object must have many services.
Relationship constraint template: <entity1> must have | may have
[<cardinality>]<entity2>

Whenever a tool user creates or modifies one or more diagram elements that result in their design violating any design rules that were stored as critics, a critique message is displayed to warn the user about the potential problem. These messages can also be shown in an Eclipse Problem view pane making them less intrusive to the designer.

6.2.6 Preliminary Results for the Initial Prototype

We have used Cognitive Dimensions (Green & Blackwell, 1998; Green & Petre, 1996) to continuously evaluate our design. This leads to the following observations about the tradeoffs we have made in this initial prototype critic designer.

We have focused on reducing the *hard mental operations* and *error-proneness* that we experienced in the first prototype. This was achieved by simplifying critic customisation and using the more accessible business rule (BR) template approach. The BR template approach is employed for the critic authoring task. The information expressed in a meta-diagram (i.e. *metamodel definer view*) is used as an input for defining a critic. The critic input process is performed via a form-based critic construction editor interface, i.e. *Critic Construction view*. The list of available critic authoring templates is designed in a drop-down menu. A user needs to select from the drop-down menu the required critic template and the properties of that particular template are accessed from the meta-model elements. A user will select the required property value that is shown in the drop down menu list, which reduces the *error proneness* from the user when an input needs to be keyed-in.

We employed only the attribute constraint templates and the relationship constraint templates for the critic authoring task. The fact that the structure of the templates is easy to understand in representing a critic rule statement reduces the *hard mental operations* for users in specifying a critic compared to OCL expressions. This is of course, because the available critic templates are not as many as the OCL functions.

The *CriticShape* notation also exhibits a better *closeness of mapping* because the critic specification/definition imitates the critic statement that the users specified according to the given critic authoring templates.

The potential benefits of the second prototype include the manner in which it provides a simple way to specify critic rule/phrase and resultant actions. A novice end-user developer can easily construct and specify critics using the critic authoring templates. Similar to the business rule templates, the critic authoring templates also offer a structured form for expressing the critic rule/phrase. Marama instantiates

critic rule processors when opening a tool and uses Marama's built-in event handler mechanism to proactively check design changes.

However, the main limitations of this initial approach are that it currently only supports the construction of fairly simple design critics. Critics can only be defined based upon the available templates and pattern match a limited part of the model as supported in the template definition. Very complex critics are not able to be specified via the attribute and relationship constraint templates. Only limited actions are supported: notifying the user of critic feedback and undoing the previous editing operation. The critic engine implemented in Marama uses a simple approach to determine interested design critics which would need to be made more efficient if a large number of critics exist in a tool.

We also analysed our design based on *Physics of Notations* (Moody, 2008). Our attempt to create a new functional item, *CriticShape* at the meta-model editor which follow similar approach to MaramaTatau, introduced a diagram complexity within the meta-model diagram. This is against the Moody's (2008) principle on complexity management that refers to "the ability of a visual notation to represent information without overloading the human mind". By adding another visual functional item (i.e. *CriticShape*) in the meta-model diagram it increased the number of visual representations needed in the meta-model diagram to convey information to the users. This would cause difficulty especially for novice users to comprehend the diagram elements. Furthermore, according to Sweller (1994), novice users are often incapable of managing diagram complexity.

The aim of this second prototype development was to gain initial experience with implementing the business rule (BR) template concept as an alternative approach to specify and author critics. To mitigate the problems/limitations that we experienced in this prototype, we have proposed another approach which deconstructs the critic specification process into multiple design perspectives. This has meant we have ended up with several editors in place of the single combined editor we had.

6.4 Conclusion

We have described our initial prototypes to support end-user tool developers to specify critics in a simple way. Our first attempt was to experiment with the OCL expressions used in MaramaTatau to specify critics for Marama-based tools. The barriers of OCL expressions were the stepping stone to the development of an initial prototype of visual critic specification tool. Our initial prototype development for the visual critic specification tool was concerned with specifying critics at the Marama meta-model level and experimenting with the BR template approach. We recognized some problems with the initial prototype and we then developed a new approach from the initial prototype. The improvement we made in our next prototype was to specify critics in a new specification tool, called the *Marama Critic Definer* editor rather than in the meta-model editor. This new approach which is our third prototype is described in the following chapter- Chapter SEVEN.

Chapter 7

Final Prototype for Critic Specification Tool

This chapter describes our third prototype for our critic specification tool. We describe the improvements that we made based on the previous proof-of-concept prototypes that we have developed for our critic specification approach.

7.1 Background and Motivation

We outlined several problems about our initial attempts for a critic specification tool in the previous chapter (Chapter SIX). Following to the failure of our second prototype which was proved to be a non-scalable approach, we developed another prototype (we have labelled as **Prototype 3** in Figure 3.1) with several improvements that represent the requirements of the new design choice for our critic specification tool. These include:

1. Deconstructing the process of critic specification into multiple design perspectives. With this new approach, we ended up with several editors in place of the single combined editor which we had in the second prototype. While this is contrary to some design approaches, such as representational epistemology (REEP)(Barone & Cheng, 2004), it has meant we have been able to apply appropriate abstractions for each part of the process that we considered as the key requirements for our critic specification approach:
 - A *high-level visual overview* of the critics designed for a tool;
 - Highly *user accessible form-based rule template interfaces* for detailed critic specification and customisation;
 - Some *extensibility options* for more experienced tool users via the rule template textual DSL
2. We were well aware that choosing to have multiple design perspectives would introduce a *hidden dependency* issue (Green & Blackwell, 1998). The hidden dependency issue can interfere with comprehension, however an

argument by Moody in his Principal of Cognitive Integration (Moody, 2008) is that multiple views with an integrative mechanism is good and necessary. Accordingly, we would need to support juxtaposition of different perspectives in our critic specification tool.

3. Expanding the critic authoring template by considering user-specified actions via the use of an action assertion template to enable the specification of more complex critics.
4. Considering more aspects of critic feedback needed in this new approach. The new approach would enable the tool and end-user tool developers to identify and construct appropriate feedback to tool users.
5. Combining the several concepts discussed in Chapter FOUR and Chapter FIVE, it has led to the following set of requirements for our final critic specification approach:
 - o Simple and intuitive critic specifications, with the necessary construct/abstraction for the specification of critics;
 - o Simple and intuitive critic feedback specifications, with the necessary construct/abstraction for the specification of critic feedbacks;
 - o Simple and intuitive representations in specifying complex critics;
 - o Simple and intuitive visual critic specification notation and environment, embedded with a DSVL tool (Marama meta-tool);
 - o Simple reuse of common critics and feedbacks, to avoid repeating specification of similar critics for different domains.

Thus, with our third prototype, we developed a new editor called *Marama Critic Definer* specifically to support the end user tool developers to specify and author critics for their DSVL tool. The following section explains our third prototype that we used to prove the concept of our critic specification approach.

7.2 Final Prototype: the Marama Critic Definer Editor

7.2.1 Approach

Our final development approach for the visual critic authoring task is illustrated in Figure 7.1. We created a new specification tool, the *Marama Critic Definer*. Thus, the tool/end-user tool developer can specify critics for Marama-based tools via this new editor. In the existing Marama metatool set, there are three key DSVL tool specification editors: the metamodel definer view to define a tool's information model; the shape designer view to define the visual notation elements; and the viewtype definer view to specify the mappings of meta-elements to visual representations. These three editors are used to develop any new Marama-based tools (1). Once the new tool is defined and equipped with sufficient information the tool/end-user designer can then select the new *Marama critic definer* view to visually author and realize critics for their target DSVL tool specification (2). The critic authoring task is supported by two form-based interfaces, the critic construction editor and the feedback editor. These two editors assist the tool/end-user tool developers to specify critics and feedbacks in a simple and intuitive way.

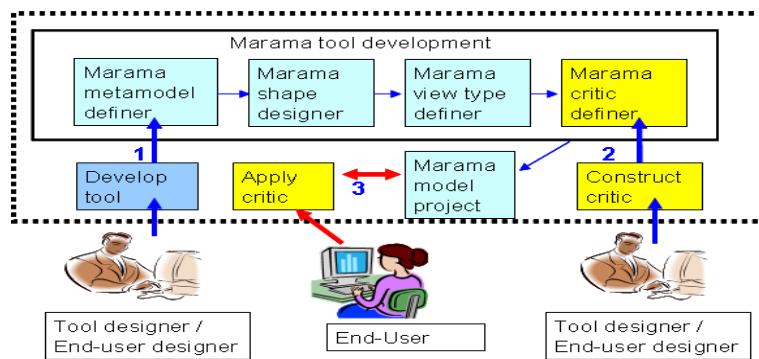


Figure 7.1: Marama Critic development approach.

Finally, a new Marama-based tool with a critic support extension is generated by Marama as a set of plug-ins. A tool end-user can then create new modelling projects and diagrams using the new tool. When a diagram is created, critics for that particular tool are instantiated. If a user creates design content that a critic identifies as problematic then a critique will be generated to notify the user about the potential

problems/errors (3). Feedback from the critic is displayed to allow the user to fix the problem/error.

The main underlying idea in our approach is to use information expressed in a meta-diagram (i.e. the Marama meta-model diagram) as input for critics to be realized in a diagram (i.e. a Marama diagram in the realized modeling tool specified by the meta-model). It is important to mention that our approach is only minimally dependent on the notation used in the meta-diagram. As we discussed earlier, the Marama meta-model diagram is expressed using an Extended Entity Relationship (EER) notation. If a richer notation is used in the future, more information can be extracted from the meta-model diagram and, thus, can be used for specifying critics and checking user diagrams. The following section explains the details of our development approach.

7.2.2 Visual Critic Definer Editor

Figure 7.2 shows a user creating a critic specification with our new specification tool, the *Marama Critic Definer*. The tool/end-user tool developer will specify critics for the Marama-based tools via this new editor. Once the editor is selected, a visual critic definer editor interface is displayed as shown by the example in Figure 7.3.

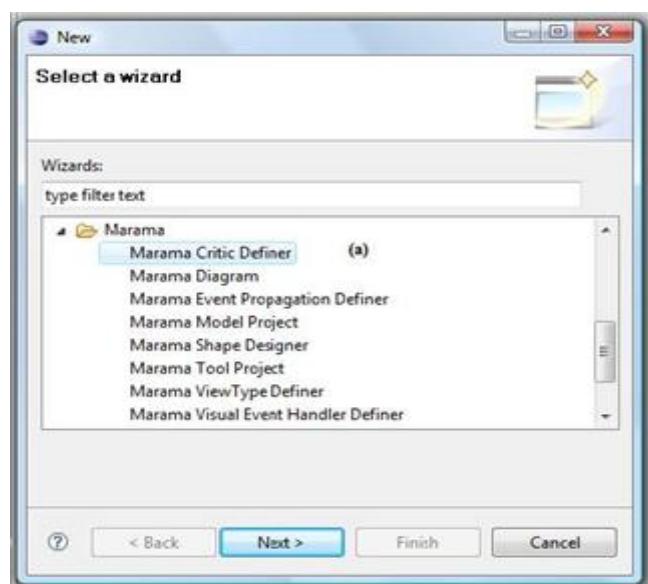


Figure 7.2: A new specification tool, Marama Critic Definer

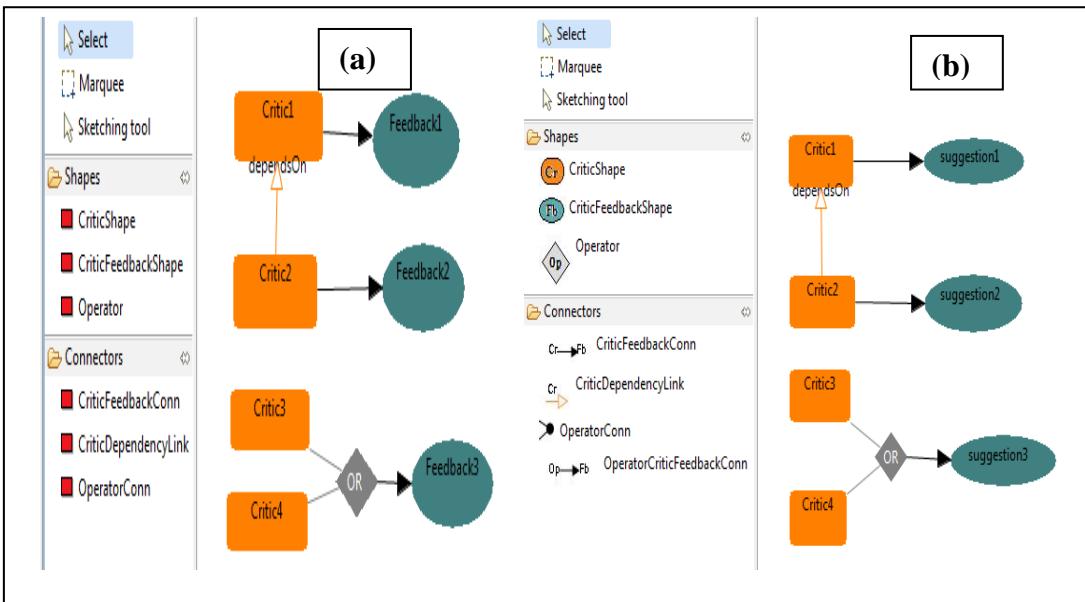


Figure 7.3: A visual critic definer editor: (a) Initial notation, (b) Improved notation

Figure 7.3 (a) shows the initial notation used in the visual critic definer editor before we improved the editor's toolbar with icon notation as shown in Figure 7.3 (b). We added the icon representation to the editor's toolbar as to be consistent with other Marama editors as well as to realise the requirements that we identified. However, please note that the evaluation survey that we conducted with target end users to assess our critic specification approach used the initial notation (Figure 7.3(a)). A few improvements to the critic designer tool were made after the evaluation. The results of our evaluation are discussed in Chapter NINE.

The visual critic definer editor has three main elements: *CriticShape*, *CriticFeedbackShape*, and *Operator*, and four connectors: *CriticFeedbackConn*, *CriticDependencyLink*, *OperatorConn*, and *OperatorCriticFeedbackConn*. The *CriticShape*, orange rounded square shape, is to allow a target end-user tool developer (or tool developer) to specify critic(s) for the developed Marama tool. The *CriticFeedbackShape*, green oval shape, is used to specify the feedback for each defined critic. After a critic is defined, the tool developer needs to specify an appropriate feedback for the critic. The grey rhombus shape is the *Operator* that holds the AND, OR, and XOR operator. The function of the *Operator* is to support the creation of composite/compound critics. The relationship between critic and

feedback is supported by the *CriticFeedbackConn* connector to indicate that each defined critic owns a defined feedback. In a case where one critic is dependent on another, a *CriticDependencyLink* connector is used to show the visual representation of the dependency. The *OperatorConn* connector is used to link a critic to a logical operator (AND, OR, and XOR), and the *OperatorCriticFeedbackConn* connector is used to link an operator to a feedback shape. A composite critic is formed in such a way. This allows complex critics to be readily built from simpler parts. We explain the function of each notation element of the visual critic definer editor in the subsequent sections.

7.2.2.1 CriticShape with Extended Critic Authoring Templates

The critic authoring template in the previous prototype only covers *constraint templates* supporting a simple critic specification. In this new development approach, we have extended the critic authoring templates by adding *action assertion templates*. Thus, the critic authoring templates support three types of template: attribute constraint templates, relationship constraint templates, and action assertion templates. Attribute constraint templates are used to specify essential properties around uniqueness, optionality (null), and value check of an entity's attributes (Loucopoulos & Kadir, 2008). The relationship constraint templates assert relationship type, cardinality and role constraints of each entity participating in a particular relationship (Loucopoulos & Kadir, 2008). Action assertion templates specify an action to be activated on the occurrence of a certain event or on the satisfaction of certain conditions (Loucopoulos & Kadir, 2008). The action assertion template allows the tool/end-user designer to specify more complex critics. Table 7.1 describes our critic authoring templates adapted from the BR template approach. The description of these templates is also given in Section 5.6 of Chapter 5. Our critic authoring templates are applied to a target DSVL tool's meta-model to review its target model instances.

Table 7.1: Critic Authoring Template (adapted from (Loucopoulos & Kadir, 2008))

Types	Templates
Attribute Constraint	$\langle entity \rangle \text{ must have / may have a [unique] } \langle attributeTerm \rangle$
	$\langle entity \rangle \langle \langle attributeTerm1 \rangle \text{ must be / may be } \langle relationalOperator \rangle \langle value \rangle \rangle / \langle attributeTerm2 \rangle \rangle$
Relationship Constraint	$[\langle cardinality \rangle] \langle entity1 \rangle \text{ is a/an } \langle role \rangle \text{ of } [\langle cardinality \rangle] \langle entity2 \rangle$
	$[\langle cardinality \rangle] \langle entity1 \rangle \text{ is associated with } [\langle cardinality \rangle] \langle entity2 \rangle$
	$\langle entity1 \rangle \text{ must have / may have } [\langle cardinality \rangle] \langle entity2 \rangle$
	$\langle entity1 \rangle \text{ is a/an } \langle entity2 \rangle$
Action Assertion	$\text{When } \langle event \rangle \text{ [if } \langle condition \rangle \text{] then } \langle action \rangle$

The critic specification is defined by selecting a *CriticShape* in the visual critic editor as shown in Figure 7.4 (top). The *CriticShape* is associated with a form-based interface designed to ease the task of specifying and authoring critics. Figure 7.4 (bottom) shows the associated *Critic Construction View* interface. The target end-user tool developers specify their critics by selecting from the available templates provided in the *Critic Construction View* interface and completing the form with required information. Critics are generated automatically after the tool developer completes the required properties for each critic.

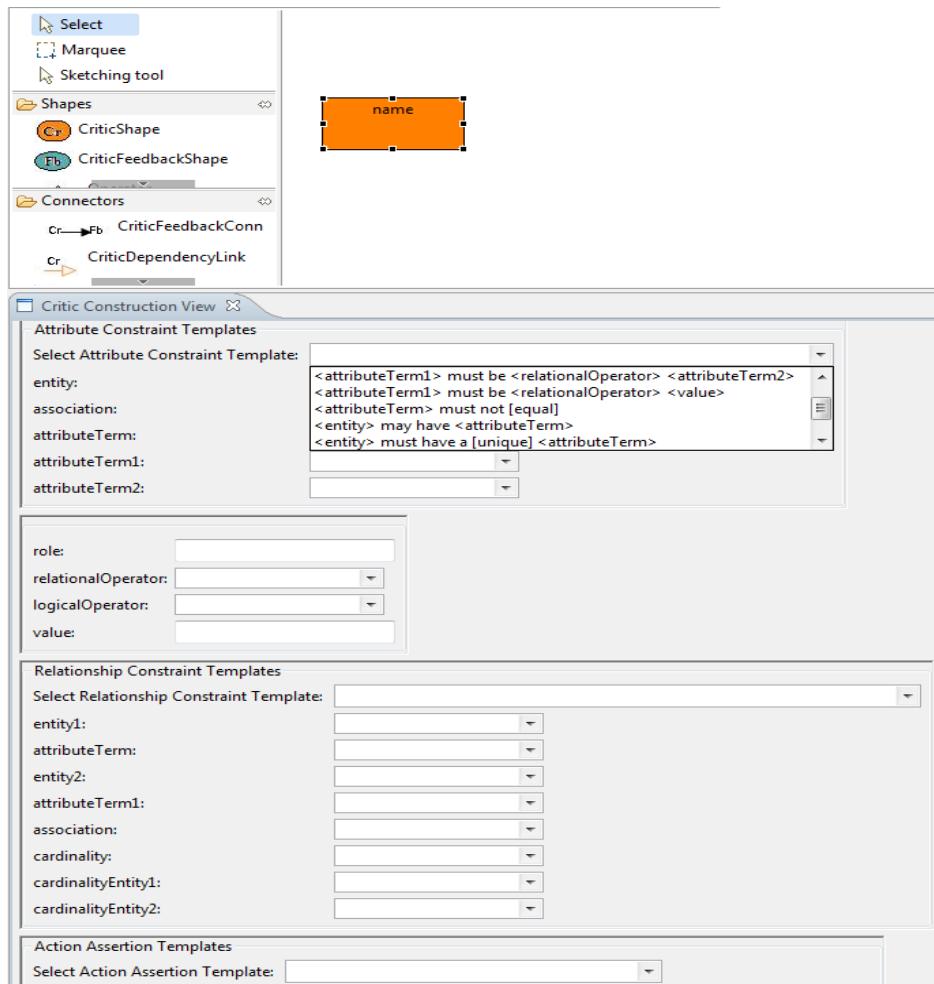


Figure 7.4: *CriticShape* (top) associated with *Critic Construction View* interface (bottom)

7.2.2.2 Critic Feedback Specification

Once the critic(s) has been defined in the visual critic definer editor, the next task is to specify feedback for the defined critic(s). This is done via the *CriticFeedbackShape* which is also associated with a form-based interface, the *Critic Feedback View*, shown in Figure 7.5.

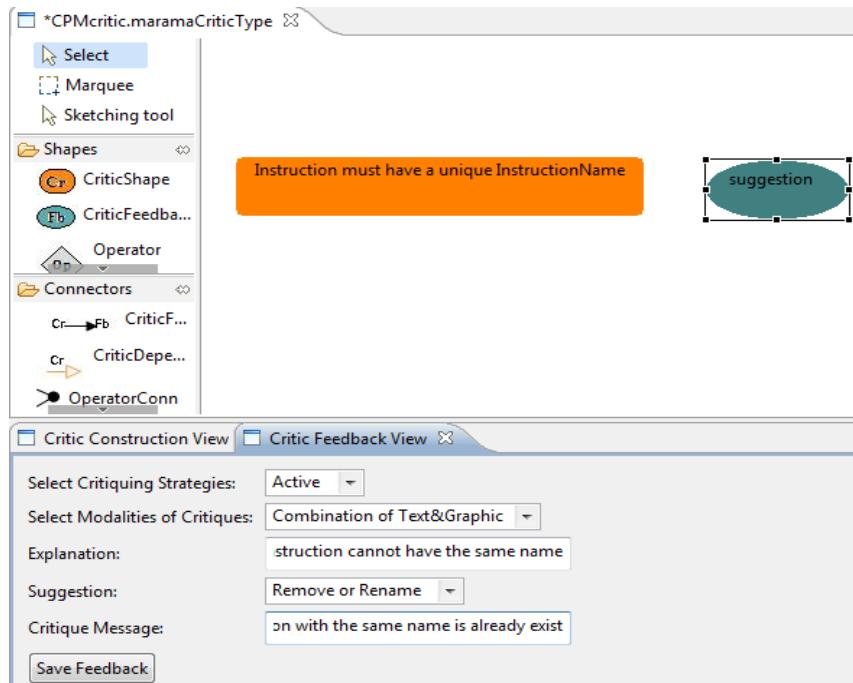


Figure 7.5: CriticFeedbackShape associated with Critic feedback view interface.

The end-user tool developer needs to specify an appropriate action to resolve the critic(s) defined for the DSVL tool. The critic feedback view has the following properties:

- (i) *Critique strategies* that determine the execution mode of the critic. This can be either active or passive. An active critic will monitor continuously a user's tasks and warns the user as soon as a critic is violated (Fischer, Lemke, & Mastaglio, 1991; Robbins, 1998) and then provides feedback (a critique). A passive critic only works when a user asks explicitly to check for a critic violation (Fischer, Lemke, & Mastaglio, 1991; Robbins, 1998). An example of a passive critic is shown in Figure 7.6. When the user selects the pop-up menu item Show Critique the critic checks the design and provides feedback to the user in the dialogue box.

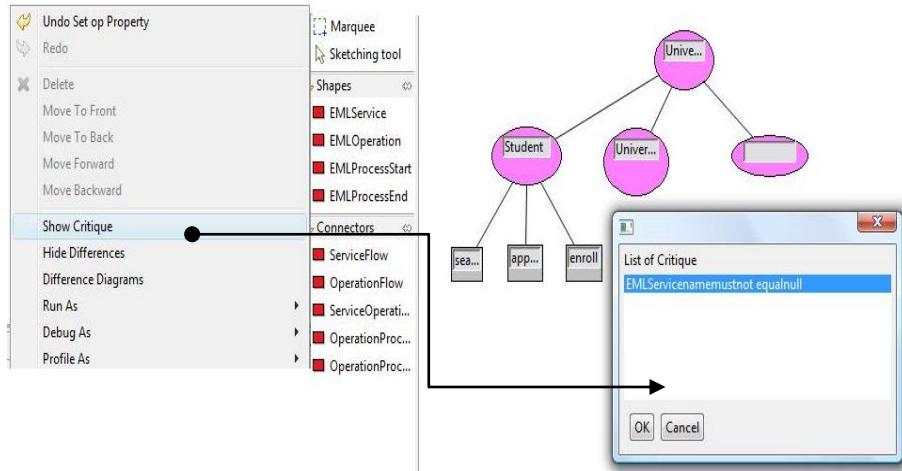


Figure 7.6: An example of passive critic

- (ii) *Modalities of critiques* (Oh, et al., 2008) involve the presentation of the critique. This can be textual, graphical or a combination of both.
- (iii) An *explanation* to represent a reason/justification of a critique. The tool developer must provide a relevant explanation to justify the critique so that the users can accept the critique given to them.
- (iv) A *suggestion* indicates an action to resolve the critic violation. Lists of actions are provided in the drop-down menu. Hence, the tool developer just needs to select an appropriate fix action for a specified critic. The suggestion only involves a simple fix action to resolve the critic.
- (v) A *critique message* specifies a textual message that is displayed for each critic that has been defined. We allow tool developers to construct their own critique message for each specified critic.

Feedbacks are generated automatically after the tool developer completes the required properties for each critic feedback. The execution of these properties is described in the following chapter- Chapter EIGHT. Once a critic and feedback are defined, these two elements are linked by the *CriticFeedbackConn* connector to indicate that a critic owns a fix action. Although (Fischer et al., 1991) state that a critic does not necessarily solve a user's problems, in our approach we expect the end-user tool developers to indicate a fix action, where possible, for each critic

defined for their DSVL tool. Figure 7.7 shows the relationship between a critic and a feedback.

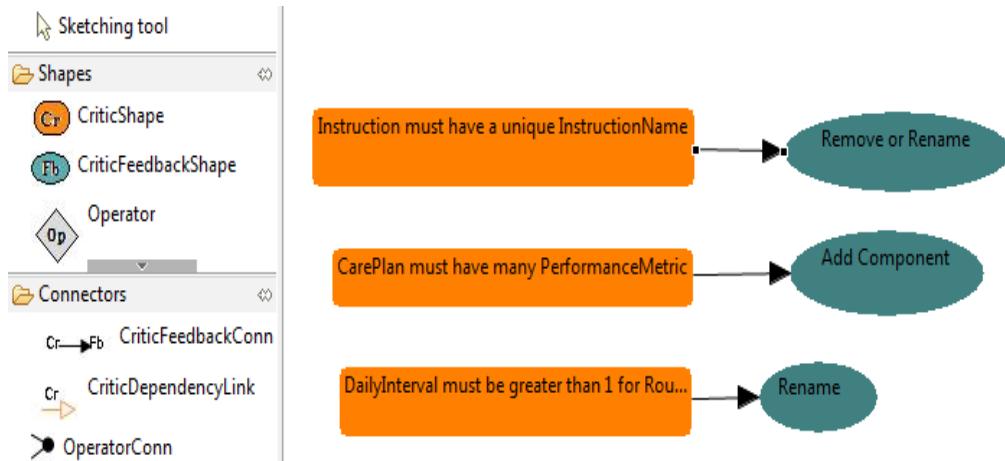


Figure 7.7: A CriticFeedbackConn connector links the critic and feedback.

7.2.2.3 Critic dependency, Operator shape, Operator and OperatorCriticFeedback connectors

Figure 7.8 shows a situation where one critic might be dependent on another critic. The dependency of critics can be represented visually by using the *CriticDependencyLink* connector (the one that takes the ‘dependsOn’ role is at the end of the arrow shape) as shown in Figure 7.8. The critic dependency link implies a sequence of critic execution between the two critics. A critic that depends on another critic will only run when the critic it depends on is not violated. For instance, in Figure 7.8 it shows the critic: “*EMLService must have a unique name*” is dependent on a critic: “*EMLService name must not be null*”. This means that the unique name critic is executed only if the service name is not null.

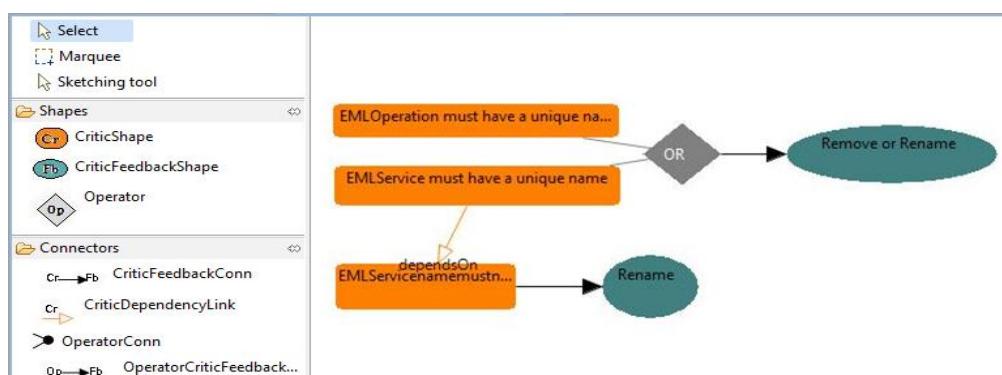


Figure 7.8: A CriticDependencyLink connects two critics

Apart from the above case, we have identified three logical operators: AND, OR, and XOR used for combining critics. A combination of critics using the logical operator AND requires all of the critic condition rules to be true for its critic feedback (i.e. fix action) to be executed for the critics. A combination of critics using the logical operator OR requires one of the critic condition rules to be true for the critic feedback to be executed. Finally the combination of critics using the logical operator XOR (“exclusive or”) requires at most one critic condition rule to be true for a critic feedback to be executed. A simple way to state the XOR is “one or the other but not both”. Figure 7.8 shows an example of the OR operator for two critic conditions. The two critics are combined with the OR operator via the *OperatorConn* connector. The feedback for the linked critics are then specified and linked with the operator using the *OperatorCriticFeedbackConn* connector. The explanation of this kind of critic is provided in the following section.

7.2.2.4 Simple and Complex Critics

We define the critics in our previous prototype development as a “unit” or “simple” critic. A unit/simple critic is a critic that was specified based on a single design model feature. Thus, the end-user tool developer constructs one critic at a time based on one BR-based model condition. For instance, a critic based on a uniqueness check for one entity can be specified using the attribute constraint template: *<entity> must have a [unique] <attributeTerm>*. This is a simple critic because it only involves a checking for a unique value for one entity. Likewise, a critic that checks for the existence of an entity can be specified using the relationship constraint template: [*<cardinality>*]<entity1> must have [*<cardinality>*] <entity2>. It is considered as a simple critic as it only checks based on one preference that is the existence of one entity. In general, critics specified using the attribute and relationship constraint templates are considered as simple critics. Figure 7.9 shows three examples of simple critics.

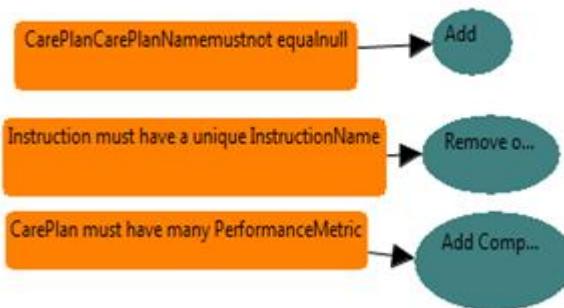


Figure 7.9: Examples of unit/ simple critics

However, in our new development approach, we wanted to allow end-user tool developers to specify both simple and “complex” critics through the visual critic definer editor. A “complex” critic is a critic that has multiple features that need to be considered. In our new approach, the end-user tool developers can construct the complex (or composite) critics by using the action assertion template and the logical operators AND, OR and XOR. Hence, end-user tool developers can specify complex critics with extended expressive power while still retaining the relative simplicity of the BR template-based approach. In addition, end-user tool developers can specify complex critics by building them from parts and also reuse simple critic parts.

An example of a complex critic is illustrated by using a simplified MaramaEML tool (a business process specification tool) as shown in Figure 7.10 (top). For instance, suppose we specify two critics with a name uniqueness constraint. A logical operator, OR can be used to link the two critics with both critics sharing a common feedback (see Figure 7.10 bottom). We consider this to be a “complex” critic because it involves more than one preference/feature. The execution semantics of these two critics is that when either one of the critics is violated the critique will be displayed and the fix action for that critic will be suggested to the user.

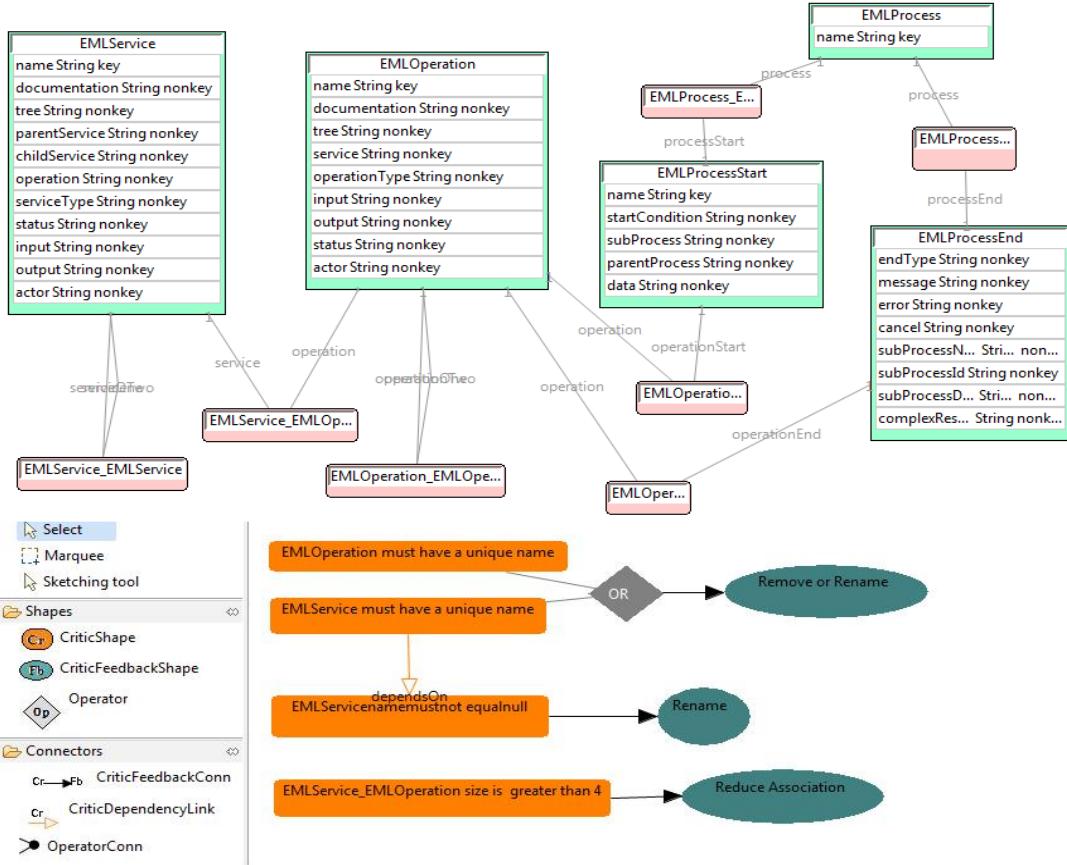


Figure 7.10: Critics specified in the critic definer editor (bottom) based on the meta-model of SimplifiedMaramaEML tool defined in the meta-model editor (top)

The action assertion template specifies an action that needs to be activated due to the occurrence of a certain event or on the satisfaction of certain conditions. The template has two options: 1) When <event> [If <condition>] then <action> and 2) When <event> then <action>. These templates can form complex critics as they involve several aspects to be assessed (that is the event, condition, and action).

For example, suppose we wish to specify a critic that constrains the service entity (i.e. *EMLService*) to have no more than four operations (i.e. *EMLOperation*). Hence, the features that need to be considered using the action assertion template are: <event>, <condition> and <action>. The event is concerned with the creation of an association link (*EMLService_EMLOperation*) between the service entity (*EMLService*) and operation entity (*EMLOperation*). The condition for the event is that the cardinality of the association link (*EMLService_EMLOperation*) is greater

than 4 and the action is to delete the new association link between the service entity and operation entity. This information is shown in Figure 7.11 that indicates there are more than single features that need to be considered. Thus, when a user runs the tool, a critique will be displayed if the event occurs to notify the user, followed by an execution of the action.

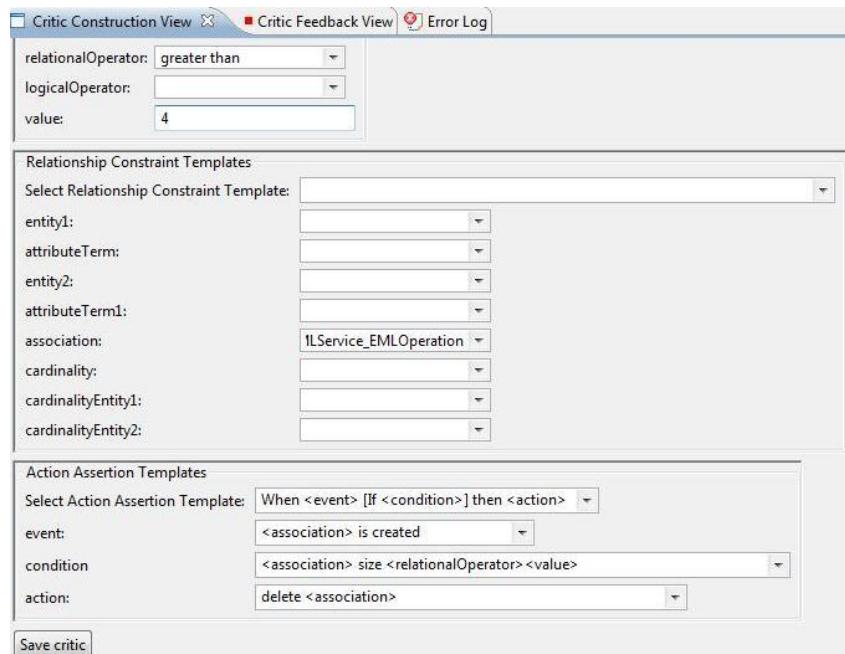


Figure 7.11: A critic specified using an action assertion template.

7.2.2.5 Critic Template Editor

In our previous prototype development, we managed to specify tool critics based on the BR templates. This was because the structure of the templates is straightforward and easy to understand especially the attribute and relationship constraint templates. However, the structure of the constraint templates of the BR approach does not provide the mixture and combination of the attribute and relationship constraint templates. This limitation is resolved with our new critic authoring templates through the development of a *Critic Template* editor.

We mentioned earlier that in specifying critics, end-user tool developers need to select the appropriate template provided in the *Critic Construction View* interface (see Figure 7.4) to define their tool critics. However, we do not limit our critic authoring templates to the ones proposed in the BR templates. We wanted end-user tool developers to be able to specify their own critic templates for reuse. Hence, we

have developed the *Critic Template* editor to support the development of new critic templates. In a case where an available critic template does not provide the desired critic specification, we allow the end-user tool developer to construct a new critic template via the *Critic Template* editor (see Figure 7.12). We also allow the critic template to have a mixture of attribute and relationship constraint templates.

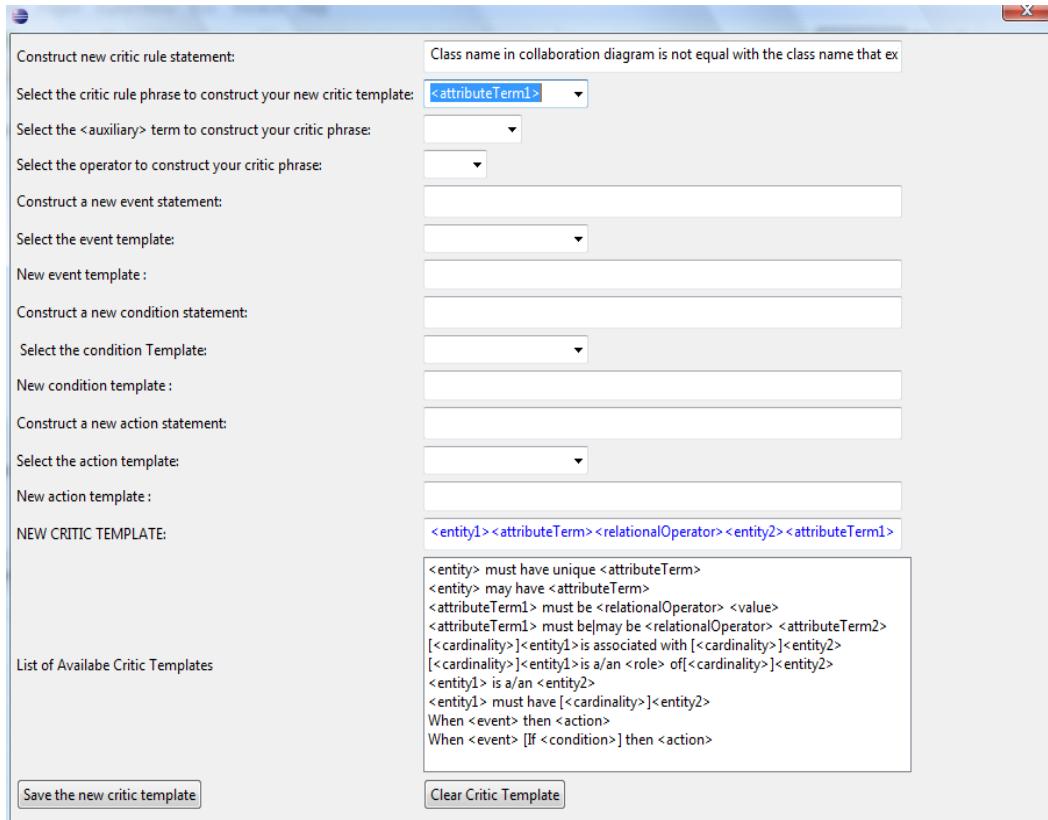


Figure 7.12: A new critic template created in the Critic Template editor.

The end-user tool developer initially needs to construct the new critic statement/phrase that describes the critic situation. The critic statement should reflect the information expressed in the Marama meta-model diagram for that particular DSVL tool. Based on the critic statement, the developer selects the necessary properties to form a new critic template that represents the new critic statement that has been defined. After specification, the new critic template is listed in the available templates and can be used to specify critics. Thus, the available template list can be expanded according to the new critic templates created in the critic template editor. Our critic authoring templates are not as highly expressive as natural language rule statements, but provide sufficient expressiveness to allow end-

user tool developers to understand, modify and possibly author critic rule expressions with little support from expert tool developers.

We provide a critic authoring guideline to assist end-user tool developers to author their own critic template if the required template is not available in the critic template list. The critic authoring guideline shows what phrases are allowed to use to author/express an appropriate critic rule template that represents a critic statement. The critic authoring guideline is explained in the following section.

7.2.2.6 Critic Authoring Guideline

Our critic authoring templates are applied to a target DSVL tool's meta-model to review its target model instances. We have developed a general critic authoring guideline to assist end-user tool developers in specifying their DSVL tool critics. The description of the critic authoring template guideline is added to the critic construction editor interface so that the new end-user tool developers can understand the critic authoring template style and they can use it to specify appropriate critics for their tool. This is shown in Figure 7.13.

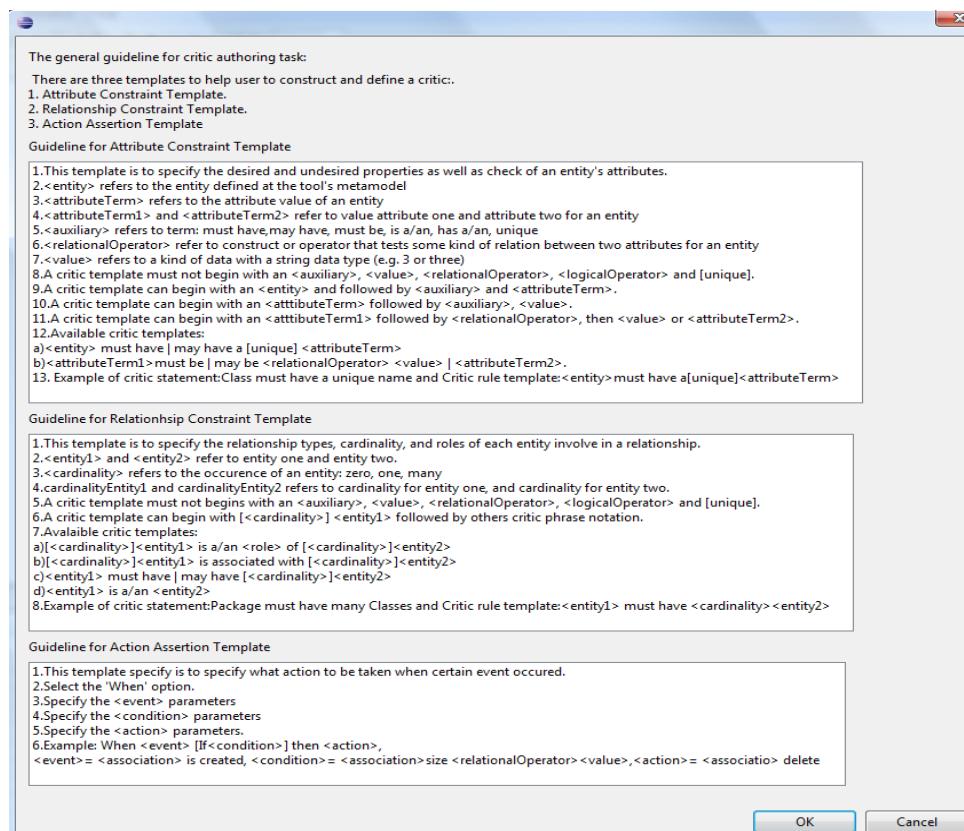


Figure 7.13: A guideline for the critic authoring template style.

Critic Authoring Guideline.

I. Purpose

The purpose of this guideline is to provide guidance to the end-user tool developers in specifying critics via the critic authoring templates.

II. Scope

This guideline applies to the end-user tool developers who want to add critic support to their DSVL tool. The development of the DSVL tool is within the Marama meta-tools environment.

III. Definitions

For the purpose of this guideline, the following definitions shall apply.

Critic phrase notation	Meaning
<entity>	<entity> is a type of entities defined in the Marama meta-model diagram
<attributeTerm>	<attributeTerm> is a type of an attribute for an entity
<association>	<association> is a type of associations defined in the Marama meta-model diagram
<role>	refers to the <i>associationEndName</i> for an association type defined in the Marama meta-model diagram
<cardinality>	refers to the <i>end1Multiplicity</i> and <i>end2Multiplicity</i> defined for an association type
<relationalOperator>	Operators that check relation between two entities or two attributes. Consists of: equal, not equal, greater than, less than, equal or greater than, equal or less than.
<logicalOperator>	Logical operators that connect two or more parameters/statements. Consists of: AND, OR, and XOR.
<auxiliary>	<auxiliary> is a term functioning to provide semantic information to a critic statement. Consists of: 'has a/an', 'is a/an', 'may be', 'must be', 'may have', 'must have' and 'unique'
<value>	is a kind of data with a string data type
<event>	Event is a part to specify a signal that triggers an invocation of a critic rule template.
<condition>	Condition is a part that provides a logical test causes an action to be carried out.
<action>	Action is a part that consists of updates or invocations on the entity attributes.
<A> 	Choice of A and B. Is either A or B.
[A]	A is optional.

IV. Guideline

Attribute Constraint Template

1. This template specifies desired and undesired properties as well as checks constraints on an entity's attributes.
2. A critic template **must not** begin with an <auxiliary>, <value>, <relationalOperator>, <logicalOperator>, <attributeTerm> , <attributeTerm1> , <attributeTerm2> or [unique].
3. A critic template can begin with an <entity> followed by <auxiliary> and <attributeTerm>.
4. A critic template can begin with an <entity> followed by <attributeTerm> and <auxiliary>, <value>.
5. A critic template can begin with an <entity> followed by <attributeTerm1> and <relationalOperator>, then <value> or <attributeTerm2>.
6. Available critic templates:
 - <entity> *must have / may have a [unique] <attributeTerm>*.
 - <entity><<attributeTerm1>>*must be / may be <relationalOperator> <value> / <attributeTerm2>>*.
7. Example:
 - <Class> *must have a [unique] <name>*.
 - <Class><operation> *may be <equal> <2>*.

Relationship Constraint Template

8. This template specifies the relationship types, cardinalities, and roles of each entity involved in a relationship.
9. A critic template **must not** begin with an <auxiliary>, <value>, <relationalOperator>, <logicalOperator> or [unique].
10. <entity1> and <entity2> refer to entity one and entity two respectively.
11. cardinalityEntity1 and cardinalityEntity2 refer to the cardinality for entity one, and that for entity two.
12. A critic template can begin with [<cardinality>] <entity1> followed by other critic phrase notation.
13. Available critic templates:
 - [<cardinality>]<entity1> *is a/an <role> of [<cardinality>]<entity2>*
 - [<cardinality>]<entity1> *is associated with [<cardinality>]<entity2>*
 - <entity1> *must have / may have [<cardinality>]<entity2>*
 - <entity1> *is a/an <entity2>*
14. Example:
 - <Package> *must have [<many>]<Class>*
 - [<many>]<Request> *is associated with [<one>]<Service>*

Action Assertion Template

15. This template is to specify what action to take when certain event occurs.
16. Select the ‘When’ option.
17. Specify the <event> parameters.
18. Specify the <condition> parameters.
19. Specify the <action> parameters.
20. Example:
 - When <event> [If <condition>] then <action>
 - <event> = <association> is created
 - <condition> = <association> size <relationalOperator><value>
 - <action> = delete <association>

The critic authoring guideline helps to prevent the end-user tool developers from authoring an invalid critic rule template. The following are some examples of invalid and valid structures of critic rule templates.

1. Examples of invalid critic rule templates:
 - <auxiliary><attributeTerm><entity>
 - <relationalOperator><entity1><entity2>
 - <value><relationalOperator><attributeTerm>
 - <entity1><relationalOperator>
 - <logicalOperator><attributeTerm1><attributeTerm2>
2. Examples of valid critic rule templates:
 - <entity> <auxiliary><attributeTerm>
 - <entity1><attributeTerm><relationalOperator><entity2><attributeTerm1>
 - [<cardinality>]<entity1> is associated with [<cardinality>]<entity2>
 - <entity1><auxiliary><cardinality><entity2>
 - <entity1><logicalOperator><entity2><auxiliary><attributeTerm>

7.2.2.6 Critic and Feedback Repository

Critics and feedbacks defined for a DSVL tool are stored in an XML format in the Marama tool repository. Critics are stored in a `critictypes` folder whereas the feedbacks are stored in a `feedbacktypes` folder, as shown in Figure 7.14.

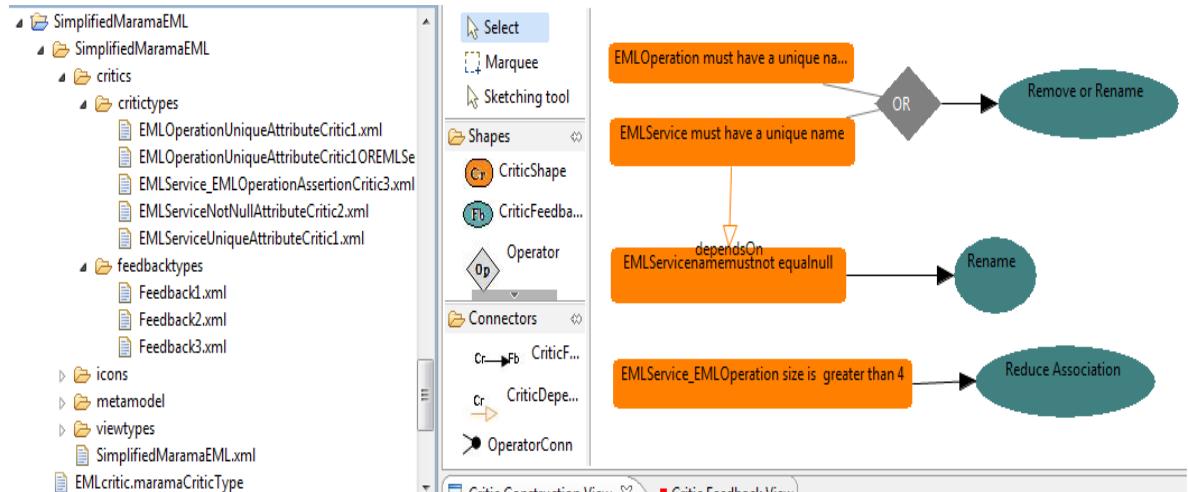


Figure 7.14: Critic (critictypes folder) and feedback (feedbacktypes) repository browser.

Once critic and feedback mechanisms have been specified, parameters are passed on to template classes to construct critic and feedback handling objects. They are then instantiated into the tool when executed.

7.3 Summary of the Implementation

The new approach with the *Marama Critic Definer* (see Figure 7.2) comprises four major components to support end-user tool developers to perform a critic specification task. These four components are the four new specification editors that we designed and prototyped to support our new critic development approach (refer to Figure 7.15):

1. Visual critic definer editor
2. Critic construction editor
3. Critic feedback editor
4. Critic template editor

We have described the functions of the four editors in the previous section and examples of their utility are described in the Case Studies chapter (i.e. Chapter

EIGHT). Figure 7.15 shows a high level architecture view of the Marama meta-tools and the extension of the *Marama Critic Definer* view.

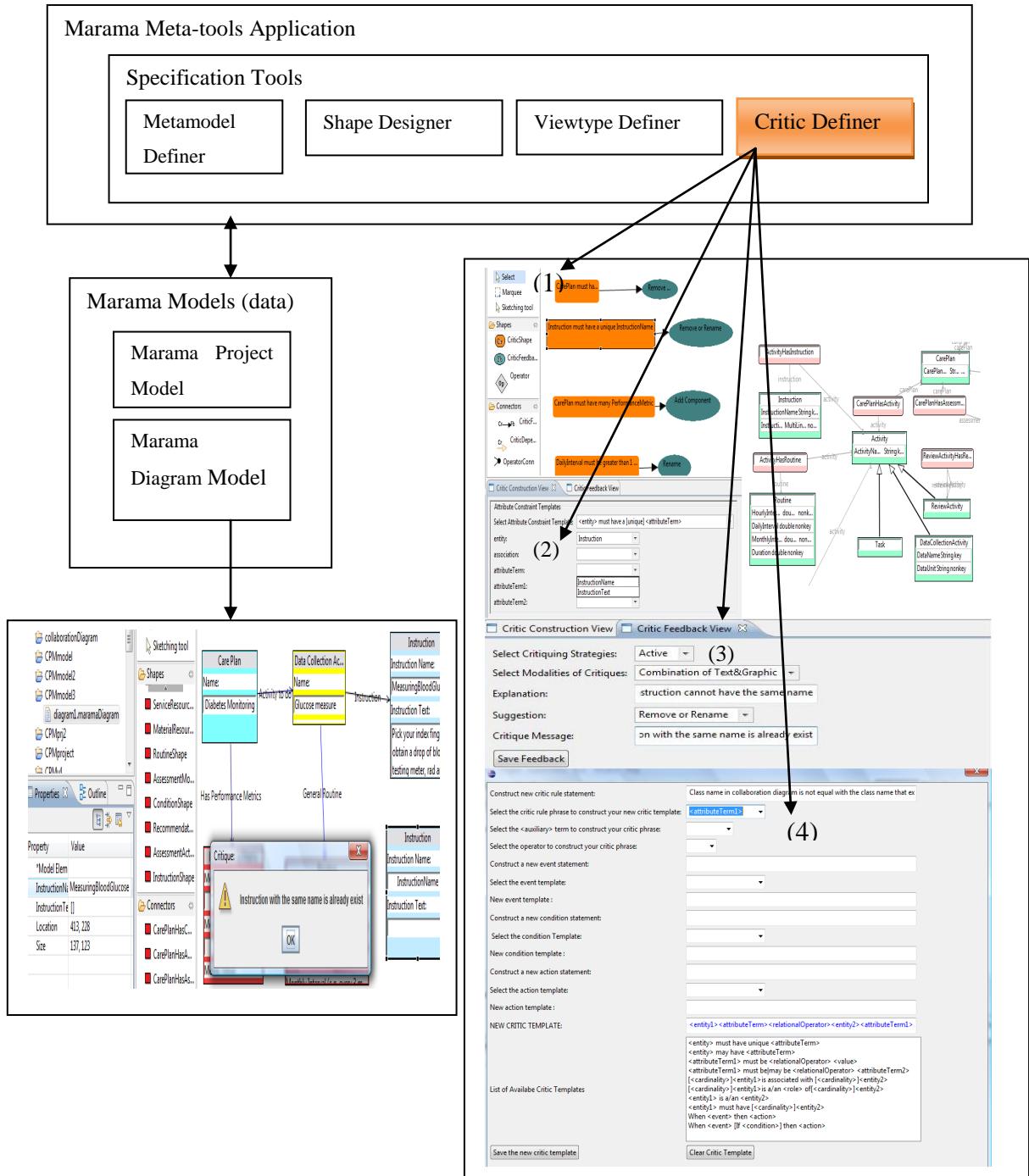


Figure 7.15: Architecture view of the Marama meta-tools and the extension of Marama Critic Definer view

7.4 Conclusion

We have described our final prototype for our critic specification approach to support end-user tool developers in specifying critics and feedback in a simple way for DSVL tools. We illustrated our visual and template-based approach to support the task of end user specification on critics and feedback using examples for Marama-based DSVL tools. A notational representation is offered to end-user tool developers to specify critics for their DSVL tools without the need to have an in-depth technical knowledge of critic construction. We also provide a critic authoring template-based approach as an alternate style for the critic specification task. Our tool supports end-user tool developers in customising critics and introduces a new critic template via a critic authoring guideline and critic template editor. We have demonstrated a proof-of-concept of our critic specification approach by implementing a prototype of it within Marama meta-tool. We have evaluated our resulting prototype with target tool developer end-users. In the following chapter we describe a more comprehensive set of case studies illustrating the usage of the approach in Chapter EIGHT. Results of the final prototype evaluation are provided in Chapter NINE.

Chapter 8

Case Studies

This chapter describes three case studies that we used to demonstrate and evaluate the utility of the critic specification editor for Marama DSVL tools. We begin by introducing and describing the first case study - Marama VCPM that explains the use of constraint templates provided by our critic specification editor. We then describe the second case study - MaramaEML that demonstrates the action assertion templates of our critic specification editor. We then describe our third case study - MaramaUML that illustrates the customizing of a critic authoring template via our critic template editor. The chapter ends with some conclusions based on the results from these case studies.

8.1 Introduction

Gable (1994) suggests that using a case study approach can help us to understand the problem being explored (Gable, 1994). Furthermore, according to Perry et al (2006), case studies are now well-accepted in software engineering and are often used in research projects “to understand, to explain or to demonstrate the capabilities of a new technique, method, tool, process, technology or organizational structure” (Perry, Sim, & Easterbrook, 2006).

Three case studies are described in this chapter. The purpose of using three case studies is to understand, explain and demonstrate the utilities of our critic specification editor in three different domains of DSVL tools (specifically Marama-based tools). These three case studies are: a visual care plan modelling language tool (medical domain), a simplified MaramaEML tool (business process domain), and a MaramaUML tool (UML diagramming domain). We chose a diverse set of domains in order to effectively prove the concept. The tools for these three case studies were developed using the Marama environment via its three main editors: Marama *meta-model definer*, to specify a tool’s meta-model; the Marama *shape designer*, to design the tool’s visual notational elements; and the Marama *viewtype definer*, to specify

mappings of meta-model elements to visual representations. We then used the newly developed editor from this thesis research, i.e. the *critic specification editor*, to specify a range of critics for these exemplar DSVL tools. The three case studies are explained in the following sections.

8.2 Case Study I: A Visual Care Plan Modelling Language (VCPML) Tool

We have chosen the Visual Care Plan Modelling Language (VCPML) tool which was designed by (Khambati, 2008) as our first case study of adding critics to a DSVL tool. We chose this tool for the reasons that it was from a medical domain, specifically the health care planning domain, and it was developed using the Marama platform. The purpose of this case study is to understand and demonstrate the utility of our critic specification editor in a medical domain of DSVL tools. Hence, we applied our critic specification editor to the VCPML tool to see how critics can be specified. For this case study we explored the Constraint Templates of the critic specification editor to specify the VCPML tool's critics.

8.2.1 Case Study Description

A visual care plan model language (VCPML) was designed to support health care providers to capture health treatment and management information commonly contained in guidelines for chronic illness treatment into a more formal, structured and digital manner (Khambati, Grundy, Warren, & Hosking, 2008). The health care professionals can model complex health care plans which comprised of different types of health care activities, performance metrics (goals), assessment modules, and other sub-care plans using the VCPML (Khambati, 2008). Figure 8.1 shows the meta-model defined for the VCPML tool with the necessary entities, attributes and associations. The four main types of components that form a care plan are: performance metrics, health care activities, assessment modules and other health care plans. Hence, the care plan entity has association with the performance metric entity, activity entity, assessment module entity and other care plan entity. Similarly, the activity entity is composed of other entities: instruction entity, routine entity, and resource entity. The activity entity can be a simple task, a data collection activity or

a review activity. The assessment module entity is a decisional task flow which is composed of assessment component entity that can be a conditional component, assessment action, and treatment recommendation. These entities are shown in Figure 8.1. A detailed explanation on this meta-model can be found in (Khambati, 2008).

The tool is then realized by modelling a care plan for diabetes management and this is shown in Figure 8.2 (Khambati, 2008). In Figure 8.2, a glucose measurement activity is modelled for one patient. From that model, it shows that the activity has a routine to conduct in every 2 days, and also has instructions on how it should be conducted. In addition, the patient requires certain material resources (i.e. testing meter, testing strips and testing pen) to perform the activity. The patient also needs to record his/her blood glucose sugar which is measured in mmol/L data unit.

The original VCPML Marama tool developed by Khambati had very little constraint support to validate models and no design critic support to provide feedback to users. Hence, it was an excellent exemplar to explore the utility of our new critic design meta-tool extension to Marama.

To illustrate our critic specification editor in action, we show several examples of critics and feedbacks defined using it. As we mentioned in previous chapters, a critic specification is dependent on the information expressed in the tool's meta-model. We applied the meta-model of the visual care plan modelling language (VCPML) tool shown in Figure 8.1 to specify the simple critics for the VCPML tool. The following section demonstrates several examples of critics for the VCPML.

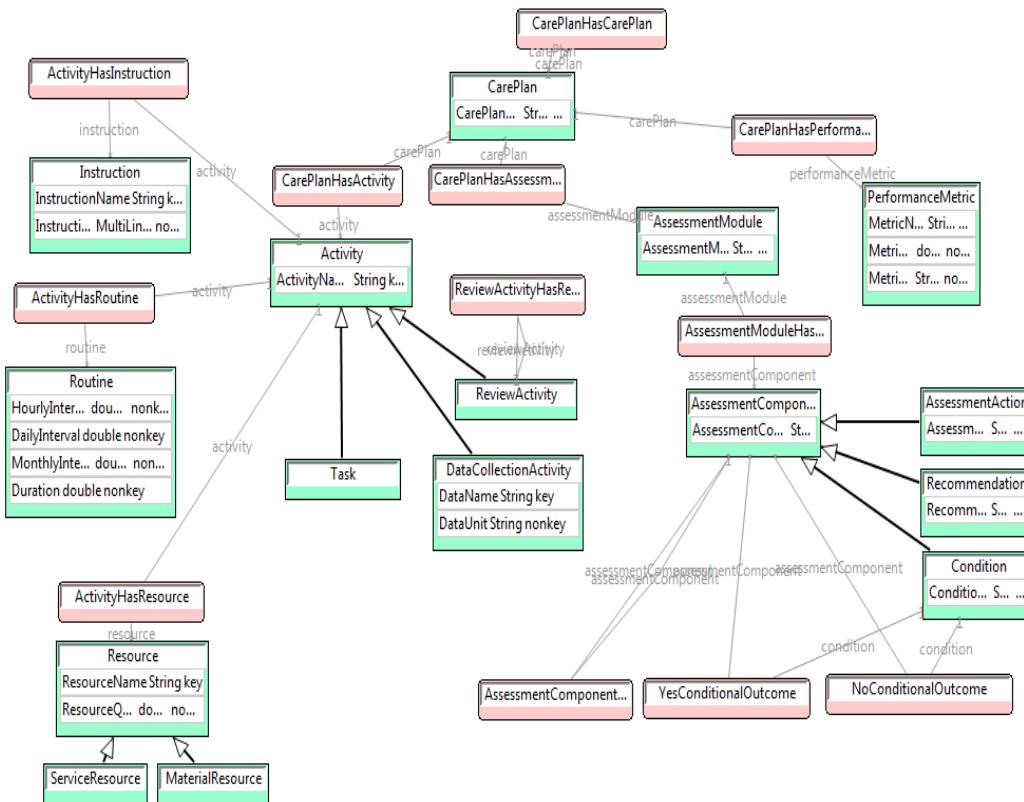


Figure 8.1: The VCPML meta model (Khambati, 2008)

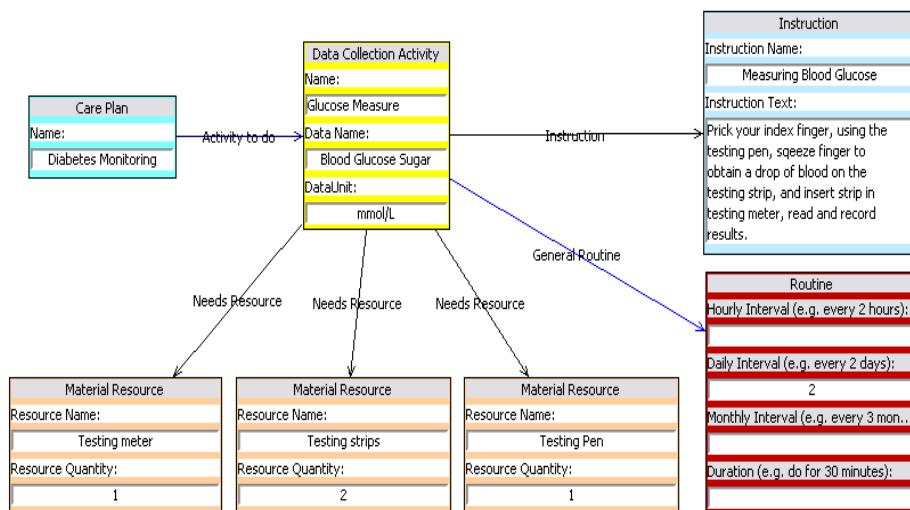


Figure 8.2: An example of the VCPML model: A care plan for diabetes management (Khambati, 2008))

8.2.2 Example Usage

We defined three simple critics for the visual care plan model language (VCPML) tool based on the constraint templates shown in Table 8.1. The constraint templates

used can be divided into attribute constraint templates and relationship constraint templates. The three examples of critics for the VCPML tool are shown in Table 8.2. Table 8.2 shows the concerned elements from the tool's meta-model, as shown in Figure 8.1, a critic statement/phrase, critic template syntax, a template type and a feedback (or fix action) to resolve the specified critic.

Table 8.1: Attribute and relationship constraint templates (Loucopoulos & Kadir, 2008).

Type	Template
Attribute Constraint	<entity> must have may have a [unique]<attributeTerm>. <entity><attributeTerm1> must be may be <relationalOperator> <value> <attributeTerm2>.
Relationship Constraint	[<cardinality>]<entity1> is a/an <role> of [<cardinality>]<entity2>. [<cardinality>]<entity1> is associated with [<cardinality>]<entity2>. <entity1>must have may have [<cardinality>]<entity2>. <entity1> is a/an <entity2>.

Table 8.2: Examples of critics and feedbacks for VCPML tool

Tool's meta-model element	Critic statement	Critic template	Type	Feedback
Instruction	Instruction must have a unique InstructionName	<entity> must have may have a [unique]<attributeTerm>.	Attribute constraint	Rename or Remove one of the component
Routine	Daily Interval must be greater than 1	<<attributeTerm1> must be may be <relationalOperator> <value>	Attribute constraint	Rename the item
CarePlan	Care Plan must have many performance metrics	<entity1>must have may have [<cardinality>]<entity2>.	Relationship constraint	Add the component

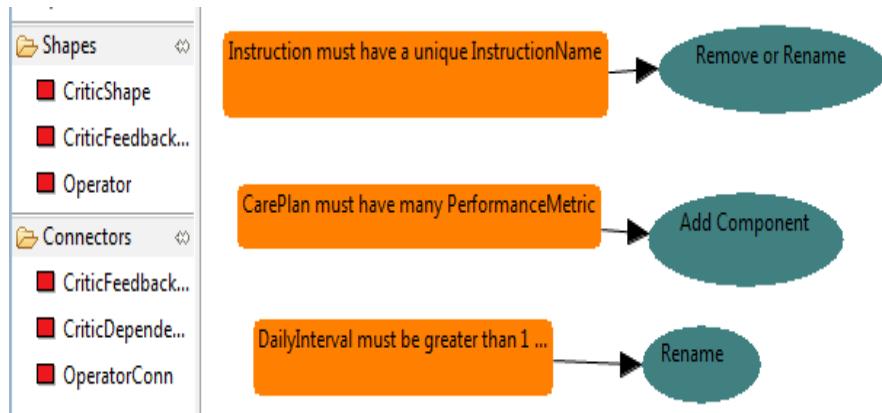


Figure 8.3: A CriticFeedbackConn connector links the critic and feedback.

The first critic in Table 8.2 shows the critic statement derived from the attribute constraint template as “*Instruction must have a unique InstructionName.*” The statement ‘*Instruction*’ and ‘*InstructionName*’ are correspondingly associated to *Instruction* entity and *InstructionName* attribute as shown in Figure 8.1. A name uniqueness constraint has been specified for an *Instruction* entity using the attribute constraint template in the *CriticConstructionView* editor. The *Instruction* entity and *InstructionName* attribute have been selected as the entity and attribute term respectively. This is shown in Figure 8.4.

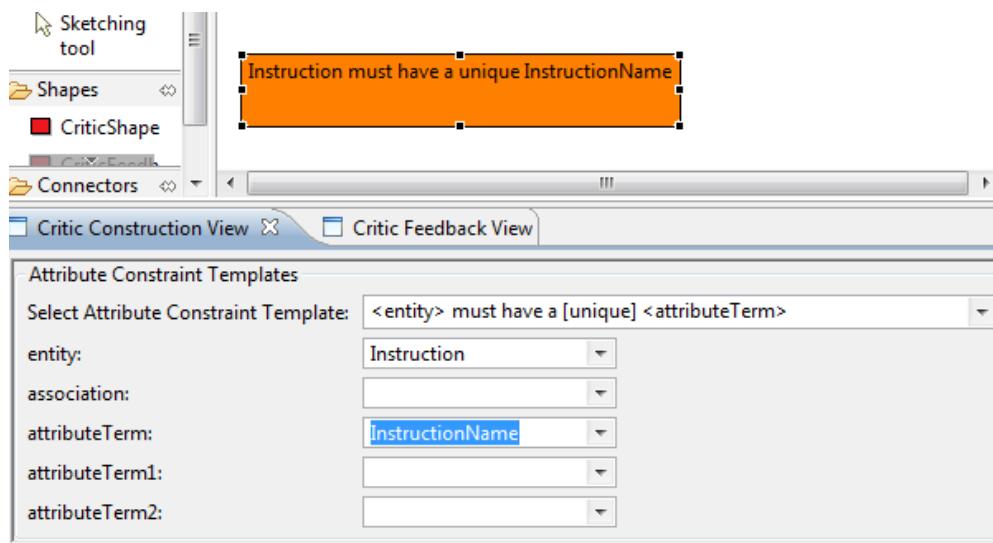


Figure 8.4: A uniqueness name critic via the attribute constraint template

Another critic is related to a cardinality constraint on the relationship between the CarePlan and PerformanceMetric entities, specified in a relationship constraint template. The critic statement “*CarePlan must have many performance metrics*” indicates that the *CarePlan* and *PerformanceMetric* are respectively associated to CarePlan entity and PerformanceMetric entity in the tool’s meta-model. The statement ‘many’ represents the cardinality of the second entity, i.e. Performance Metric. This critic specification is shown in Figure 8.5.

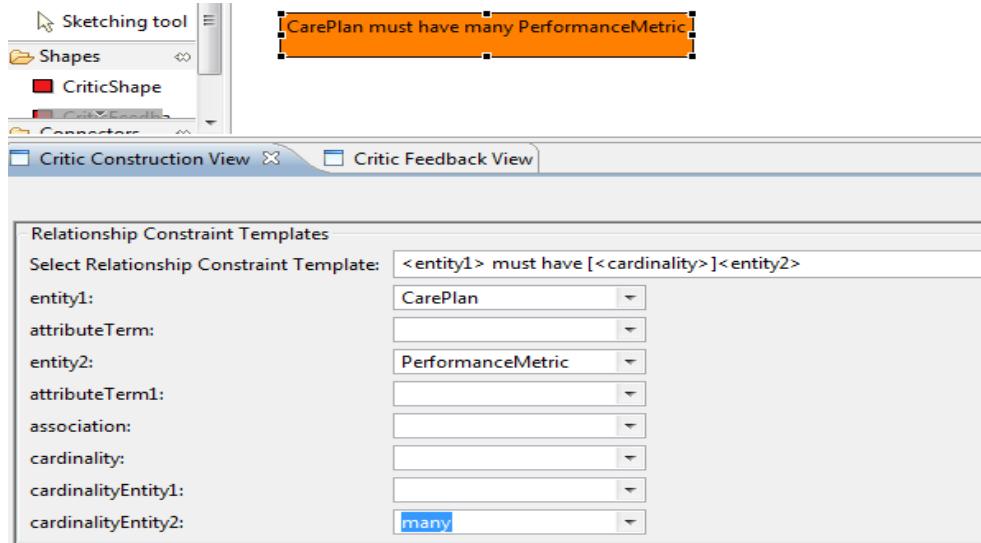


Figure 8.5: A critic on cardinality constraint using the relationship constraint template.

All properties in the tool’s meta-model are available in the *critic construction editor*, selectable via drop down menus.

Feedback actions for each critic have to be specified and defined. We show one example of the feedback specified for the uniqueness name critic shown in Figure 8.4. The feedback for the defined critic is done via the *CriticFeedbackShape* (a green oval shape) which is associated with a form-based interface, the *Critic Feedback View*, as shown in Figure 8.6. The critic feedback editor has five properties as shown in Figure 8.6: critiquing strategies (active/passive); modalities of critiques (text/graphic/combination of text and graphic); explanation; suggestion (list of possible actions); and critique message. All of the required properties have to be filled in. We have described these properties in Chapter SEVEN (section 7.3.3.2).

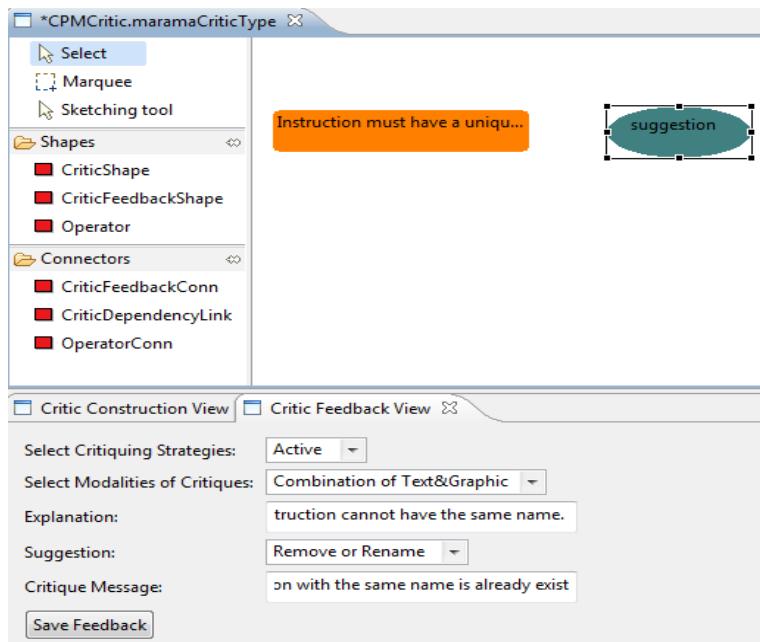


Figure 8.6: Critic feedback for the uniqueness name critic

Once the properties have been specified then a button, *Save Feedback* (refer to Figure 8.6) is selected. The feedback specification for other critics goes through the same process. All critics have been specified as *active critics* with appropriate explanation and fix messages to resolve them. Critics are generated automatically after the end-user developer completes the required properties for each critic.

The execution of a critic specified in Figure 8.4 is shown in Figure 8.7, Figure 8.8 and Figure 8.9. Presentation of the critique message and the fix action are based on properties that were specified in the *critic feedback editor*. In Figure 8.7, a uniqueness name critic is violated for the Instruction entity due to the same name that existed in the two entities. A critique message is displayed to warn the user about the error. Furthermore, an explanation and suggestion are offered to the user to resolve the problem as shown in Figure 8.8 and Figure 8.9. In Figure 8.9, an action to rename the property value is selected and a new name is given to the Instruction entity.

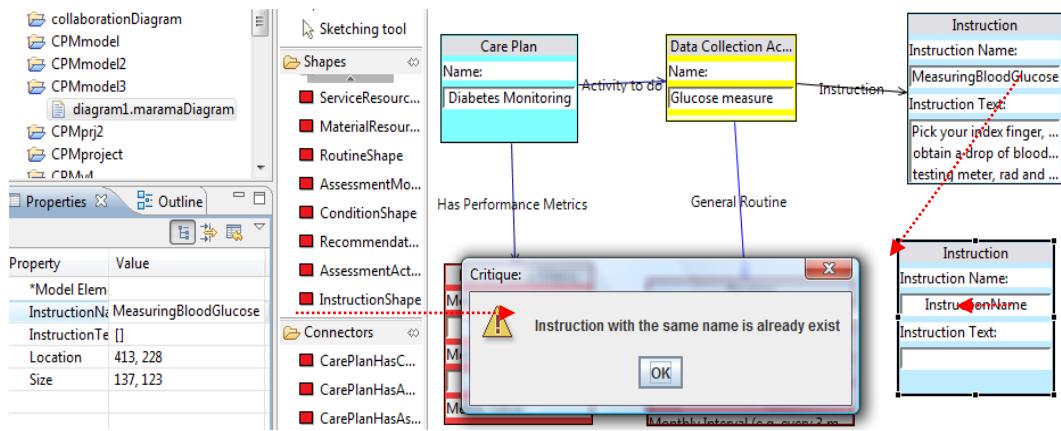


Figure 8.7: A critique message is displayed when a uniqueness name critic is violated

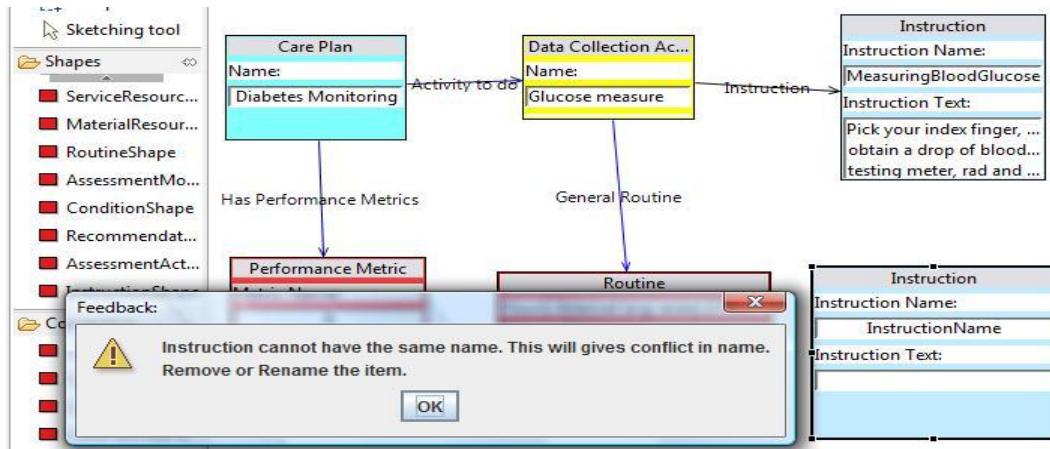


Figure 8.8: A critic feedback with a brief explanation and suggestion

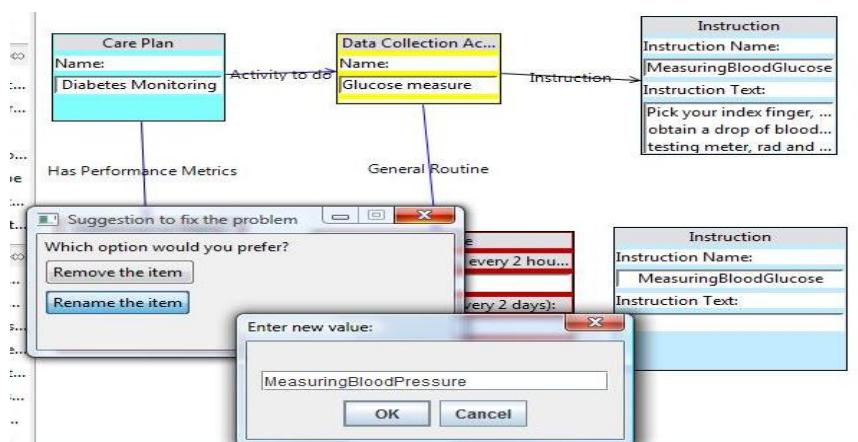


Figure 8.9: The fix action for the uniqueness name critic.

Specifying these three example critics was very straightforward using our critic specification tool. Their specifications could not be compared to hard-coded critics specified using the existing Marama meta-tools as none of them existed for MaramaVCPML prior to this case study. However, similar critics could be implemented using the existing Marama meta-tools using a combination of OCL constraints and Java coded event handlers. These would be much more time consuming to specify and debug than using our new critic specification tool. In particular, giving feedback via a dialogue box or problem marker would require writing Java code. Specifying a unique name property requires careful use of an OCL constraint in the meta-model editor which we have found to be non-intuitive for meta-tool users in previous evaluations. Unlike the OCL expressions and event handlers, our simple critics can be easily packaged and reused or combined into composite critics in our critic design tool.

8.3 Case Study II: A Simplified Marama EML Tool

The purpose of this second case study is to illustrate the use of our critic specification editor in a business process modelling domain. We choose to do this with MaramaEML (Li, Hosking, & Grundy, 2007b) which is a complex visual tool for business process modelling. The original MaramaEML was previously designed and developed using the Marama meta-tool (Grundy, et al., 2008; Grundy, et al., 2006) for creating Enterprise Modelling Language (EML) specifications (Li, Hosking, & Grundy, 2007a; Li, et al., 2007b). EML uses a tree layout to represent the basic structure of a service. However, for clarity reasons, we have designed and developed a simplified version of MaramaEML by highlighting several main components of the MaramaEML tool. We then applied our critic specification editor to the simplified MaramaEML tool to see how critics can be specified. For this case study we explored the complex critic features of the critic specification editor. As the original MaramaEML tool had a number of constraints and critics specified using Java event handlers we were able to directly compare the task of designing critics using this approach to using our new critic design meta-tool. We were able to interview the original developer of MaramaEML and obtain feedback on the

effectiveness and efficiency of specifying critics using our new critic design tool compared to using the existing Marama meta-tools.

8.3.1 Case Study Description

One of the facilities provided by the simplified MaramaEML tool is to model business processes. Figure 8.10 shows the simplified meta-model for the MaramaEML with some of the relevant entities, attributes and associations. As shown in Figure 8.10, MaramaEML's main features include service entity, operation entity and process entity. A service entity is to imply a task within a business process of an organization. An operation entity is to represent an atomic activity that is included in a service. A process entity has two types of entities: process start entity and process end entity. The process start entity is to represent the start of a process. The process end entity is to indicate the end of a process. Associations between the required entities are created so as to support the modelling of the business process structure. All services, operations and processes are organized in a tree structure to model a business process system.

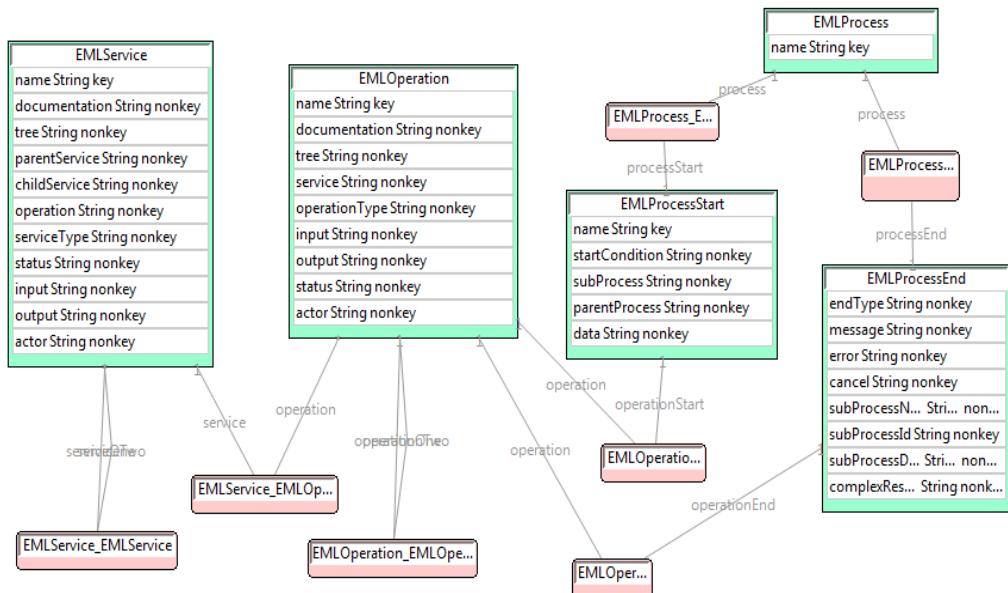


Figure 8.10: Meta model for the simplified MaramaEML

We adopted the following basic rules for the EML structure from (Li, 2010).

Table 8.3: Basic rules of EML tree structure (adopted from (Li, 2010))

Basic rules:
<ul style="list-style-type: none"> • An enterprise system must have at least one Service tree. • Every single service tree must have one and only one service node. It may (or may not) include an arbitrary number of sub-service nodes (zero or more). • A service node is always at the top of the single service tree structure. It must include at least one Operation node (directly or indirectly). It may (or may not) include an arbitrary number of sub-services. • A sub-service is inside a service or sub-service node. It must include at least one Operation (directly or indirectly) and may (or may not) has arbitrary number of sub-services. • An Operation is the leaf node of the service tree. It cannot include any service, sub-service or other operations.

Figure 8.11 shows a simple example of a MaramaEML structure model for a basic university enrolment service (modified from (Li, et al., 2007b)). We used the example from (Li, et al., 2007b) however, presenting only a part of the university enrolment service model. Figure 8.11 shows that the student service, university service, and StudyLink are sub-services of the university enrolment service. These are represented in the oval shape. Each service may (or may not) include a sub-service. The university service includes four embedded services (i.e. enrolment office, finance office, credit check and department). Each service must include at least one operation. The operation entity is represented in a rectangle shape. For instance, the Student Service manages four operations: search courses, apply enrolment, apply loan and make payment.

To illustrate our critic specification editor in action, we applied the meta-model of the MaramaEML shown in Figure 8.10 to specify the possible critics for the simplified MaramaEML tool.

The following section demonstrates several examples of critics for the simplified MaramaEML tool.

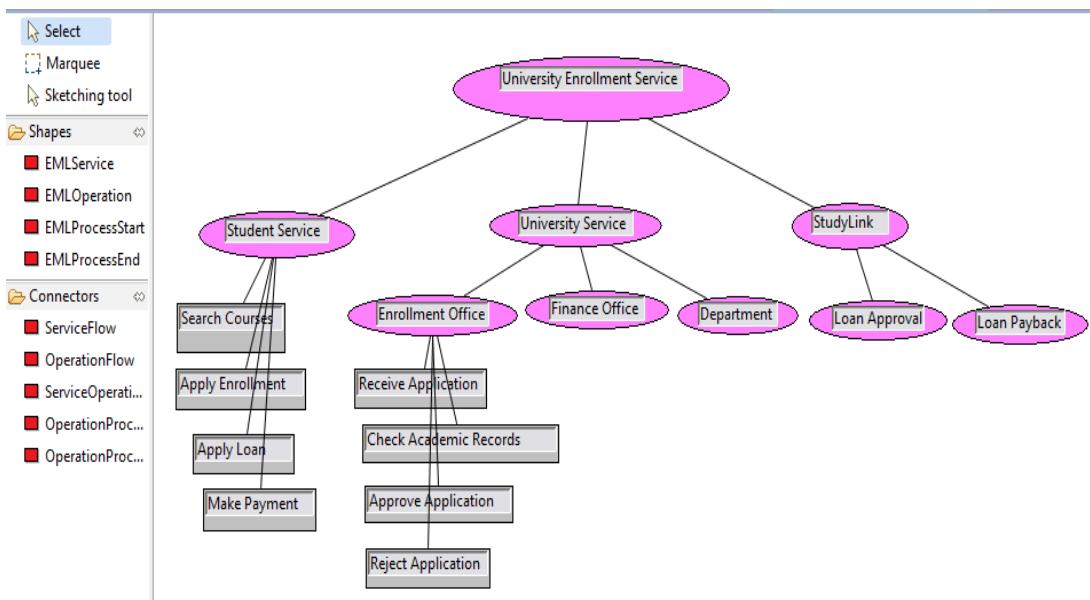


Figure 8.11: University enrollment service model using a simplified MaramaEML (modified from (Li, et al., 2007b))

8.3.2 Example Usage

Figure 8.12 shows several possible critics for the MaramaEML tool. These include examples of complex critics using action assertion templates and the logical operators (OR, AND and XOR). The situation that involves the dependency of critics (i.e. between the second critic and the third critic as shown in Figure 8.12) is not illustrated here as it is already explained in previous chapter in the section 7.2.2.3.

The action assertion templates specify an action to be activated on the occurrence of a certain event or on the satisfaction of certain conditions. These include critique message generation for the tool user and “fix up” actions that can be applied to resolve detected design problem(s). The action assertion template is as below:

When <event> [if <condition>] then <action>

The bottom-most critic in Figure 8.12 is an example of complex critic using the action assertion template. Suppose we wish to specify a critic that constrains the service entity (i.e. *EMLService*) to have no more than four operations (i.e. *EMLOperation*). This might be sensible in order to encourage designers to split large hierarchies of services into smaller, more manageable and understandable groups as

our evaluation of MaramaEML found that service entities with large numbers of operations look cumbersome to the end users.

The features that need to be considered using the action assertion template are: <event>, <condition> and <action>. The event is concerned with the creation of an association link (*EMLService_EMLOperation*) between the service entity (*EMLService*) and operation entity (*EMLOperation*). The condition for the event is that the cardinality of the association link (*EMLService_EMLOperation*) is greater than 4 and the action is to delete the new association link between service entity and operation entity. This information is shown in Table 8.4. A critic for this case can be specified by defining the relevant properties for event, condition and action in an action assertion template as shown in Figure 8.13 that indicates there are more than single features that need to be considered. Thus, when a user runs the tool, a critique is displayed if the event occurs to notify the user, followed by an execution of the action. The execution of this critic is shown in Figure 8.14.

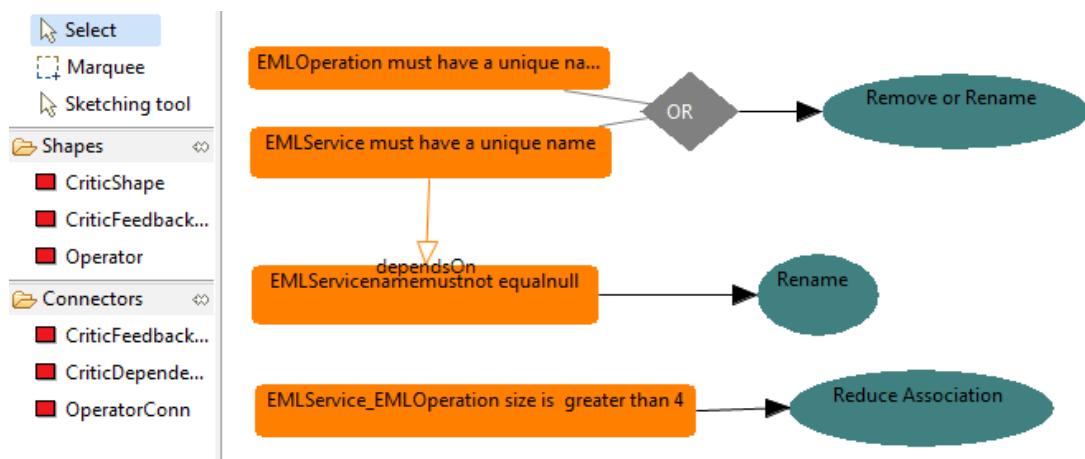


Figure 8.12: Critics specified in the critic definer editor based on the meta-model of SimplifiedMaramaEML tool.

Table 8.4: Action assertion template:
when<event>[If<condition>] then <action>.

Template	Template instance
<event> = <association> is created	<event> = <EMLService_EMLOperation> is created
<condition> = <association> size <relationalOperator> <value>	<condition> = < EMLService_EMLOperation > size <greater than> <4>
<action> = delete <association>	<action> = delete < EMLService_EMLOperation >

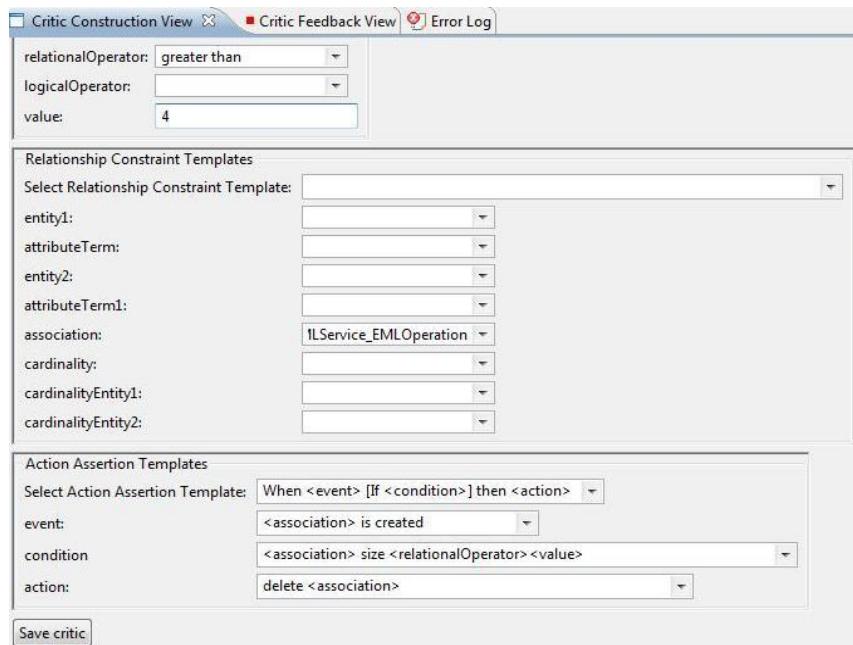


Figure 8.13: A critic specified using an action assertion template.

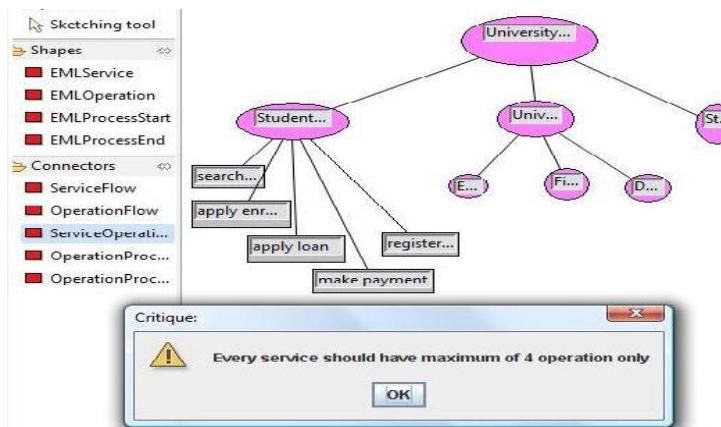


Figure 8.14: Action assertion critic execution after the trigger event occurs: a critique is displayed to warn the user

Another example of such a complex (or composite) critic is when using the logical operators AND, OR and XOR. This approach allows users to specify complex critics by building them from parts. Importantly it also facilitates hierarchical reuse of simple (or other complex) critic parts. The topmost critic in Figure 8.12 is a complex critic, where two simple critic conditions, in this case two name uniqueness constraints, have been connected to OR to share a common feedback element. Table 8.5 shows the specification of the two critics. It is considered as a complex critic because it involves more than one preference. The execution semantics of these two critics are that when either one of the critic conditions is violated the critique will be

displayed and the fix action for that critic is suggested to the user. The execution of this critic is shown in Figure 8.15.

Table 8.5. Attribute constraint template:
<entity> must have a [unique] <attributeTerm>.

Template	Template instance
<entity> must have a [unique] <attributeTerm>	<EMLService> must have a unique <name>
<entity> must have a [unique] <attributeTerm>	<EMLOperation> must have a unique <name>

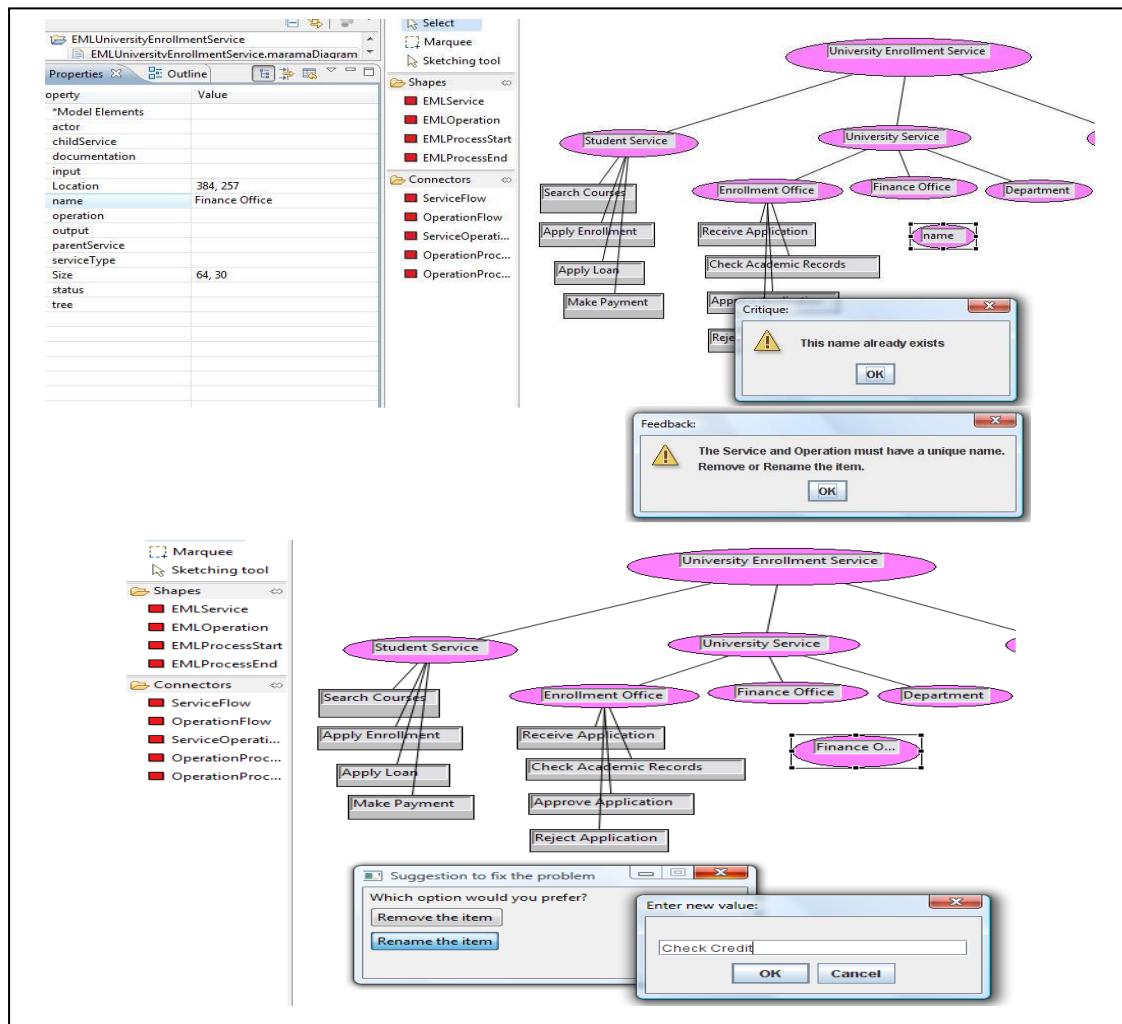


Figure 8.15: Feedback of a complex critic using the logical operator OR (top) and fix action for this critic (bottom).

We are able to compare the specification of these critics using our critic specification tool with specification with the original critics in MaramaEML. The original MaramaEML used Java event handlers to implement similar constraint

testing and feedback to the user. However, it did not generally implement fix-up options for the end user to invoke. Specifying constraints and feedback was time-consuming and was difficult to maintain in MaramaEML as the meta-model evolved over time. Similarly, as MaramaEML has several integrated modelling notations and a canonical meta-model, it was a complex task to implement inter-notation constraints. Using our critic designer on the canonical meta-model is straightforward and implementations of critics that took several hours to specify, test and evolve can be done in a matter of minutes. Understanding the critics is far easier than browsing and understanding the previous individual Java event handlers, which comprised hundreds of lines of Java code with Marama API calls. In contrast, as seen in these examples, visual and form-based critic specifications are very clear, concise, understandable, reusable and maintainable.

8.4 Case Study III: MaramaUML Tool

In this section, we present our final case study, a MaramaUML tool to demonstrate the utility of our critic specification editor in customizing/tailoring critic authoring templates. The MaramaUML tool provides a simplified Unified Modelling Language (UML) class diagram view and collaboration diagram view. We designed and developed a simplified UML tool for the purpose of clarity in explaining the task of customizing the critic authoring templates.

8.4.1 Case Study Description

The Unified Modeling Language (UML) offers several types of diagrams that can be employed to model the static structure and dynamic behaviour of a software system. In this case study, we have chosen a class diagram to represent the static model structure of a software system and a collaboration diagram to represent the dynamic behavioural model of a software system. We have designed and developed a simple MaramaUML tool using the Marama meta-tool editors. In this case study, we concentrated on class diagrams and collaboration diagrams for the conceptual perspective. This could be extended to cover other UML diagram types in the future, but the coverage is sufficient to illustrate the application of our critic approach.

We specified several entities and associations to represent the structure of a class diagram and a collaboration diagram. The basic items of a class diagram are: package; class with the properties name, attribute and operation; and associations between these items. Similarly, the collaboration diagram depicts objects and links between objects. The class of each object included in the collaboration diagram must be defined and the object may optionally be named. The basic items for a collaboration diagram are: object with the properties class name and object name; message; and relationships between these items. The meta-model for the MaramaUML tool that defines the structure of a class diagram and a collaboration diagram is shown in Figure 8.15.

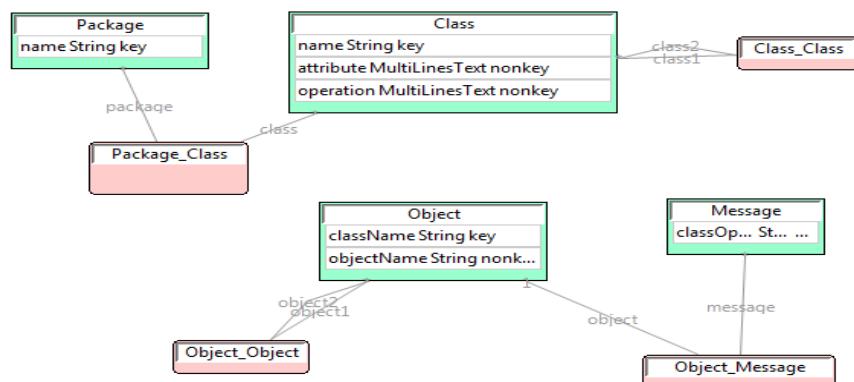


Figure 8.15: Metamodel for MaramaUML tool.

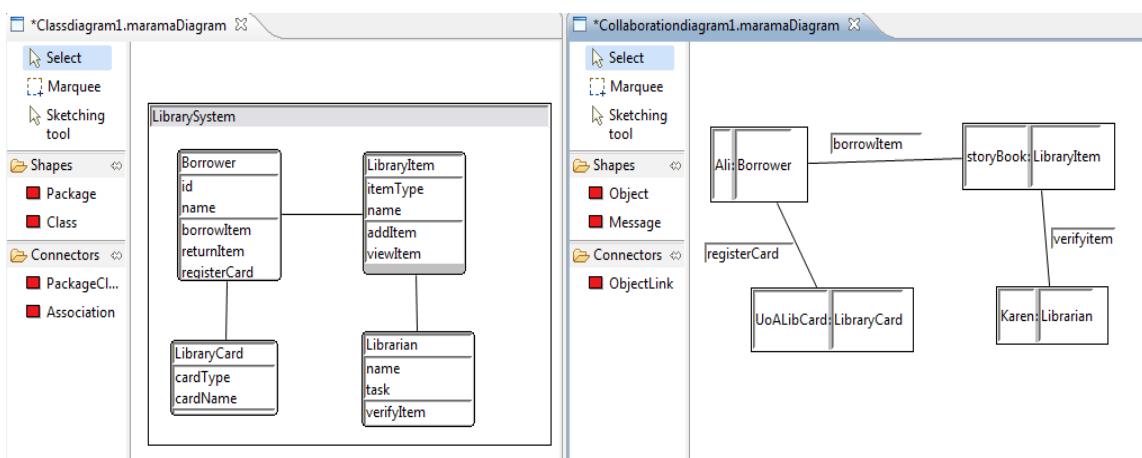


Figure 8.16: Class diagram example (left) and Collaboration diagram example (right)

The MaramaUML tool provides two diagrams: Class diagram view and Collaboration diagram view. Figure 8.16 (left) shows an example of UML class diagram modelling the concepts and relationships in a library system. Every class is

distinguished by its name, by a collection of properties, and by a collection of operations provided by the class. The class diagram (left) represents the structure of an early design-time snapshot of a simplified library system: a *Borrower* for borrowing and returning library items, a *LibraryCard* for certifying the borrower's right to use the library, a *LibraryItem* for recording the library items, and a *Librarian* for processing and verifying the library data. A *Borrower* class is related to *LibraryCard* and *LibraryItem* classes. A *Librarian* class is related to *LibraryItem* class.

Figure 8.16 (right) shows an example of UML collaboration diagram modeling the objects interaction in a library system. An early design of a collaboration diagram (right) is used on the instance level to describe the interaction among the objects (instances of classes) and messages passes between them (Paige, Ostroff, & Brooke, 2002). The *UoALibCard* object is an instance of the *LibraryCard* class, the *storybook* object is an instance of the *LibraryItem* class, the *Ali* object is an instance of the *Borrower* class and the *Karen* object is an instance of the *Librarian* class. The diagram also shows messages being passed between the objects. For instance, the *Ali* object passes a message, *borrowItem* to the *storybook* object.

The two diagrams: class diagram and collaboration diagram are two fundamental models that can be used to represent a system as shown in Figure 8.16. Once the MaramaUML tool was defined, critics and feedbacks were specified via our critic specification editor. The following section illustrates an example of tailoring the critic authoring templates via the critic template editor, followed by an example of using the newly created critic template to specify a critic for the simplified MaramaUML tool. This shows how users of MaramaUML could tailor critics to their own preferences and needs using our high-level critic design tool facilities. In all other UML tools that we are aware of, including the earlier versions of Marama-implemented UML tools, such tailoring would require expert knowledge of the tool infrastructure, detailed use of the tool's scripting and/or programming language, or would not be supported at all (the case for most UML tools that we are aware of).

8.4.2 Example Usage

One example of the possible critics that can be specified for the MaramaUML tool is to define a critic rule that checks for elements of consistency between a collaboration diagram and a class diagram. For instance, the objects in a collaboration diagram must include/define a class name of classes that are already defined in a class diagram. In other words, for each object defined in the collaboration diagram, there should be a class name that belongs to a class diagram, so that, objects defined in the collaboration diagram have a corresponding class that has been defined in the class diagram. This description is shown graphically in Figure 8.17. Thus, we can say that the object's class name in a collaboration diagram must be equal to a class name that has been defined in a class diagram. If a design violated this critic rule, then a feedback is displayed to warn the tool user about the consistency error in the diagram design. During exploratory design this critique may be ignored i.e. the inconsistency tolerated by designers. However, it must be resolved at some point or otherwise the resultant UML model is, by definition, incorrect.

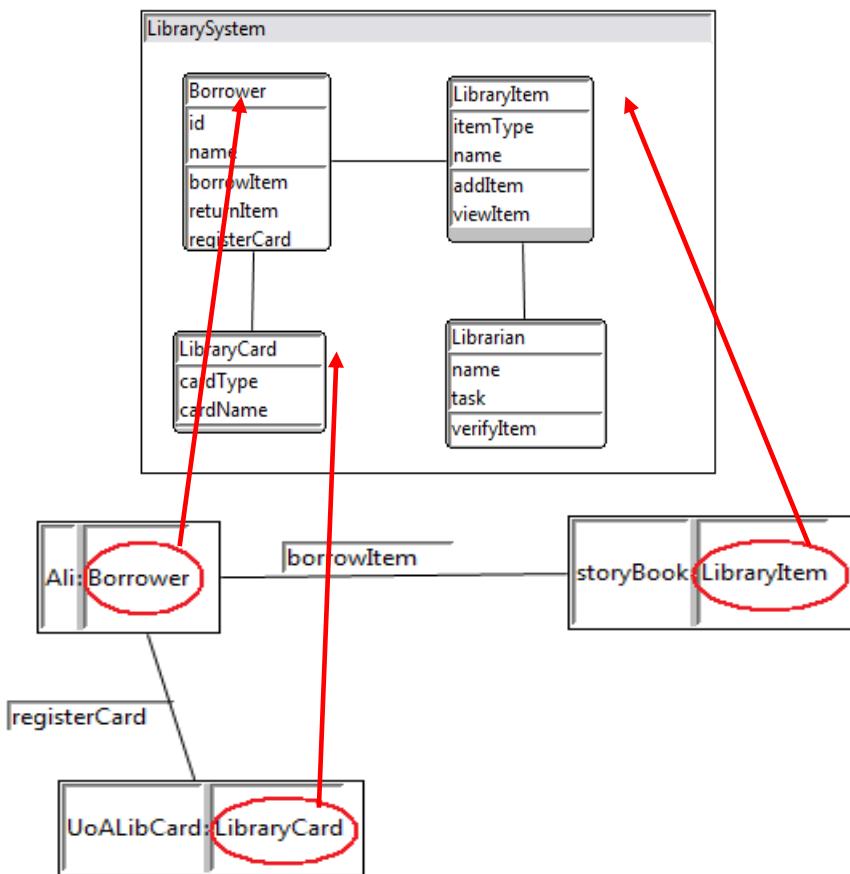


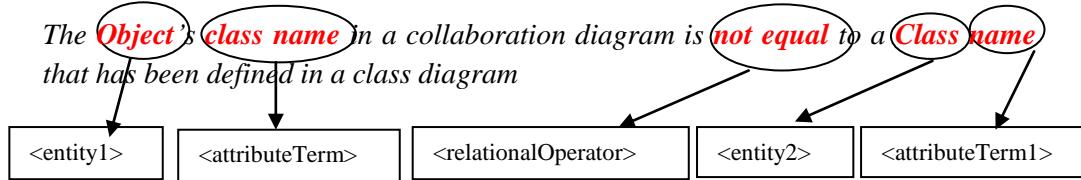
Figure 8.17: Graphical representation of a consistency rule between collaboration diagram (bottom) and class diagram (top)

The available critic authoring templates that employ the business rule templates do not, however, support the above situation. In a case where the available template does not support the desired critic specification, we allow the end-user tool developer to construct/customize a new critic template via a *Critic Template* editor, shown in Figure 8.18. The following are the steps in constructing a new critic authoring template:

1. Construct a critic statement/expression in natural language that describes the critic condition. The critic statement should reflect the information expressed in the MaramaUML tool's meta-model;

New critic rule statement: The Object's class name in the collaboration diagram is not equal to a Class name that has been defined in a class diagram

2. Identify the appropriate critic rule phrase that can represent the critic statement. This is shown below:



3. Selected critic rule phrase then form a new critic authoring template to be used in specifying a critic:

New critic authoring template syntax:

<entity1><attributeTerm><relationalOperator><entity2><attributeTerm1>

4. After specification, the new critic template is listed in the available templates and can be used to specify critics. Thus, the available templates list can be expanded according to the new critic templates created in the critic template editor. Such basic critics can also be used in complex, composite critics as illustrated in the previous MaramaEML case study.

The steps are shown in Figure 8.18. With the new critic template, the consistency critic rule to check the existence of classes in a collaboration diagram and a class diagram can be specified as:

Critic name: **<Object><className><not equal><Class><name>**

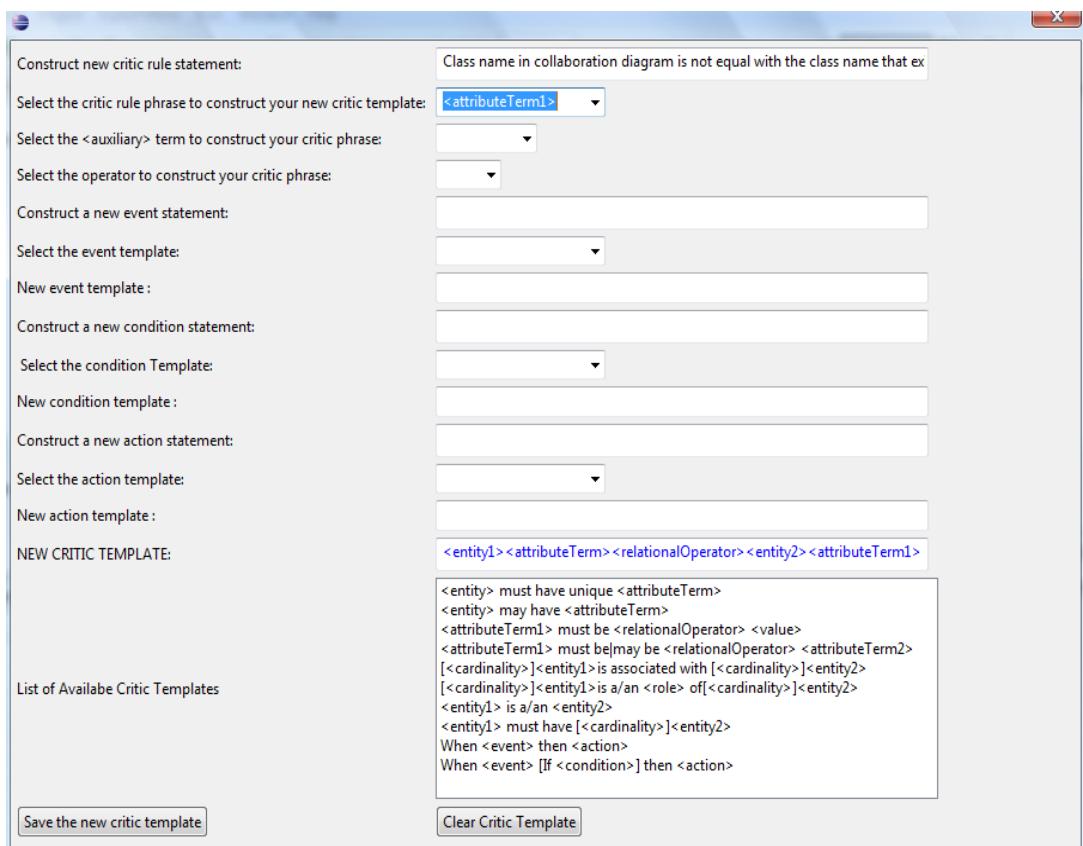


Figure 8.18: A new critic template created in the Critic Template editor.

A critic then can be specified using the new critic template as shown in Figure 8.19 (the bottom critic). The other critics in Figure 8.19 have similar characteristics to the one that we described in the earlier section.

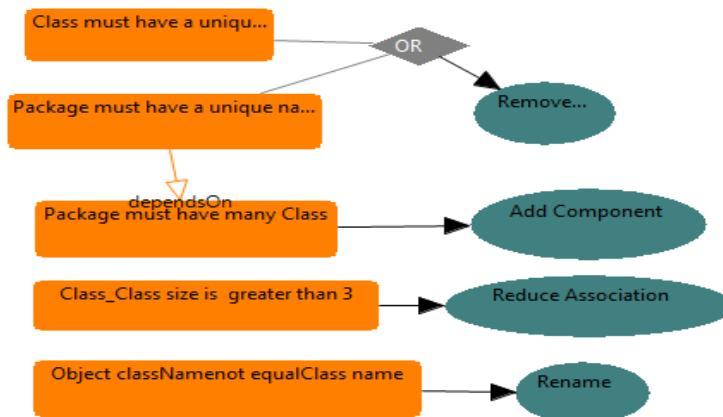


Figure 8.19: New critic authoring template:
 $<\text{entity1}><\text{attributeTerm}><\text{relationalOperator}><\text{entity2}><\text{attributeTerm1}>$
 (bottom critic)

The execution of this new critic authoring template is shown in the following figures: Figure 8.20, Figure 8.21 and Figure 8.22. In Figure 8.20, a critique message is displayed in the collaboration diagram due to the fact that the class name defined in the collaboration diagram does not correspond to a class that has been defined in the class diagram. The critique message warns the user about the error and provides an explanation together with a suggestion to resolve the error. This is shown in Figure 8.21. The critique message, explanation, and suggestion to fix the error are actually based on the properties that were specified in the critic feedback editor.

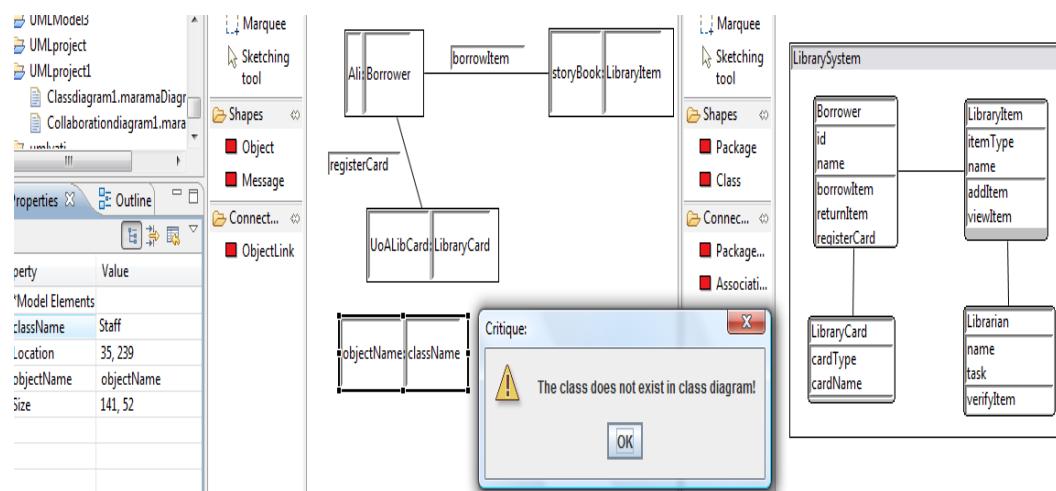


Figure 8.20 A critique is displayed when a consistency critic rule is violated.

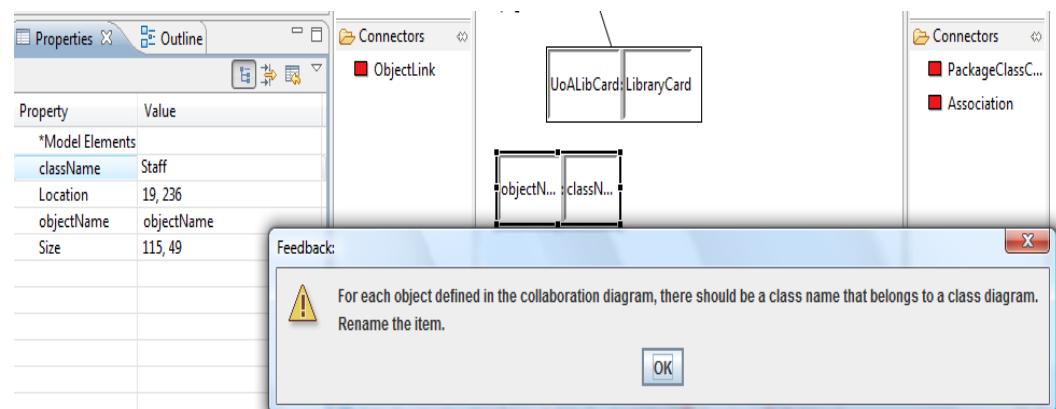


Figure 8.21: A critic feedback displays a brief explanation and suggestion.

In Figure 8.22, it shows an action to resolve the error by renaming the class name in the collaboration diagram. A new class name that corresponds to an existing class in a class diagram is then defined in the dialogue box (i.e. *Enter new value*).

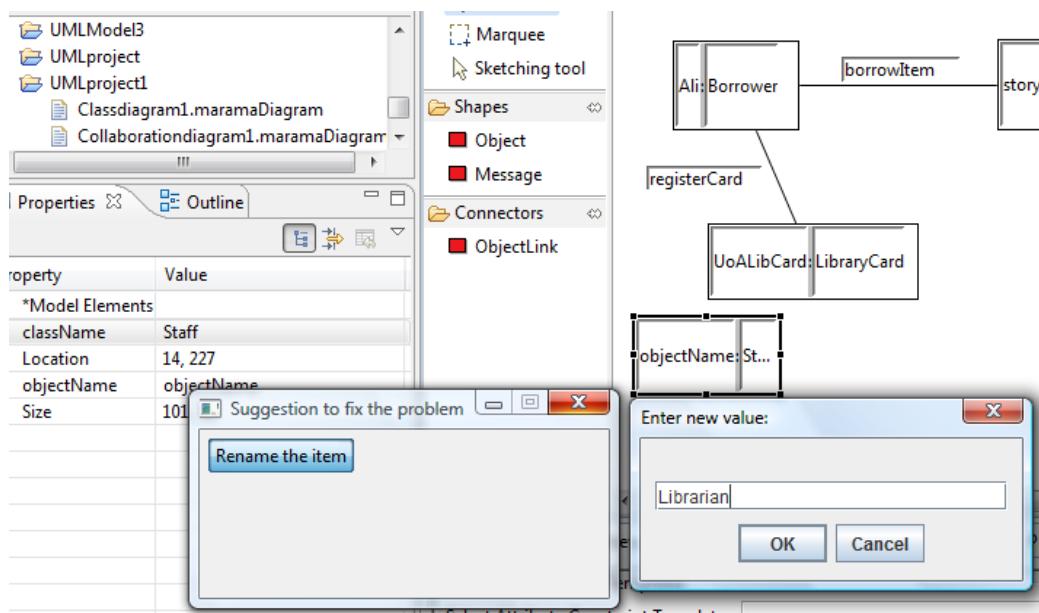


Figure 8.22: A fix action to resolve the consistency critic rule

In our earlier versions of MaramaUML tools such a constraint had to be implemented with a Java event handler. This needed considerable coding and Marama API calls to fully implement the example shown, around 100 lines of Java code including supporting feedback to the user and fix-up action support. Writing such code requires expert knowledge of the Marama APIs, as well as advanced Java programming skills. It is error-prone, difficult to maintain and very difficult to abstract and reuse.

In contrast, the critic for the MaramaUML example shown above demonstrates that the critic specification tool can be extended by adding new design critic templates to be instantiated for a DSVL domain. The critic template can be reused and any complex critic built from the template can be composed of multiple reused basic critics. The visual language and form-based property editors used to specify this complex critic are easy to understand for tool developers and even for end users in this example (UML users would find the critic and DSVL tool meta-modelling and visual language specification straightforward as they are familiar with the class and collaboration diagram concepts themselves). Finally, the critic template and complex critic are understandable, maintainable, reusable and ultimately reconfigurable – the MaramaUML user could switch off the critic or extend it with further constraints, feedback or alternative fix-up rules using the critic designer tool.

8.5 Discussions and Conclusions

We have applied our new critic specification editor to three very different domains of Marama-based tools, i.e. Marama visual care plan modelling language (VCPML) - a health care plan modelling tool, a simplified MaramaEML (Enterprise Modelling Language) – business process modelling tool, and a MaramaUML tool - a class and collaboration diagramming tool.

We described our approach in specifying critics for Marama-based tools through three case studies and each of these case studies included critic specifications and applications as we mentioned previously. We have developed a prototype of a critic specification editor that consists of two editors, including a Critic Construction editor that comprises of the critic authoring templates and a critic template editor, and a Critic Feedback editor to specify a critic feedback (critique). The main aim of our prototype is to demonstrate the utility of the critic specification editor when integrated into the Marama meta-tools. We have illustrated the utility of the critic specification editor with three different case studies. Our purpose of using three different case studies is to show that the utility of the critic specification editor can extend to a range of different domains of DSVL tools.

The first case study involves specifying simple critics for a health care plan modelling tool (i.e. VCPML) using the constraint templates (attribute and relationship constraint templates). The available templates provided in the Critic Construction editor are very straightforward to specify a critic. The structured form of the critic authoring templates makes it easier and quicker to specify critics for the VCPML tool compared to using OCL and/or Marama event handlers coded in Java. The syntax of the critic authoring templates matched the information expressed in the tool's meta-model and this makes the critic authoring task much easier for tool developers. However, the limitation in the first case study is that we did not address complex critics for the VCPML tool, as the constraint templates only support fairly simple design critic construction. We speculate that with some training end users of MaramaVCPML may even be able to understand, reconfigure and specify new critics i.e. health professionals could extend their MaramaVCPML designer critics.

The second case study deals with a business process modelling tool, a simplified version of an earlier developed MaramaEML tool. The main aim of the second case study is to demonstrate the utility of the critic specification editor in specifying complex critics. The specification of complex critics is performed via the action assertion template. Several properties/features have to be considered and assessed when specifying a complex critic for the tool and these properties however have to be matched with the tool's meta-model elements. Through the action assertion template, complex critics can be specified, provided that all the necessary properties of event, condition and action have been defined. We also show how a complex critic can be specified through the use of logical operators-OR, AND and XOR. The logical operators can be used to link several similar critics and provide a common critic feedback, like the one that we explained in the case study. With the second case study we managed to show that complex critics are possible to be specified via the critic specification editor. We were able to compare specification with our critic designer to earlier implementation of the same critics using existing Marama meta-tool OCL constraints and Java event handlers. Our critic specification tool approach was proved far easier, quicker, and maintainable than our existing meta-tool support. However, the main limitation of our second study is that we did not illustrate many examples of complex critics from the action assertion template and also the logical operators that are likely to arise.

Finally, we described a third case study that was concerned with a simplified MaramaUML tool. The difference of this case study with the previous two is that the MaramaUML tool provides two diagrams: a class diagram and a collaboration diagram. The properties of these two diagrams come from one meta-model. The previous case studies only support one model/diagram of their tools. The main objective of the third case study is to illustrate the task of customizing the critic authoring templates when a desired critic specification is not supported by the existing critic authoring templates. An aim of our research was to enable end user tool developers to be able to create their own critic template in a situation where the list of available templates cannot support their desired critic statement. Thus, we demonstrate in the case study how a critic authoring template can be customized using a critic template editor. With the critic template editor, new critic templates

can be constructed to expand the list of available templates that appear in the critic construction editor. Once the new critic authoring template is defined, the required critic statement can then be specified for that tool. In this case study, we show an example of creating a new critic authoring template which is a consistency critic template that is concerned with one aspect of the consistencies between a collaboration diagram and a class diagram. The limitation from the third case study is that we did not address the issue of expressive power that the critic template editor provides to the end use tool developers.

In one of the three case studies we did illustrate the critic feedback editor that is used to provide the feedback information once a critic is defined. The critic feedback process is applied for the other case studies. The function and properties of the critic feedback editor are described in the previous chapters. In general, we believe that we have managed to explain, demonstrate and provide understanding of the utility of our critic specification editor which we integrated with the Marama meta-tools. It is important to note that although each case study explains and illustrates a different utility of the critic specification editor, the utilities of the critic specification editor actually can be applied across all three different Marama tools. The exactly same critic specification tool was used for each case study with no tailoring to the target DSVL tool domain.

Chapter 9

Evaluation

This chapter presents the evaluation of our final critic specification prototype for domain-specific visual language tools. We begin by introducing the concepts of evaluations and usability evaluations. Then we introduce the Cognitive Dimensions of Notations framework (CDs) and describe the criteria to evaluate a tool's usability. We then explain the design/method of our survey carried out to assess whether the visual and template-based critic authoring tool effectively supports end-user developers in specifying critics for DSVL tools. We analyse the survey results and present the findings before we conclude the chapter.

9.1 Introduction

Evaluation is an essential activity in software engineering. According to Gena and Weibelzahl (2007), evaluations are applied in software development to verify the quality and feasibility of initial products such as mock-ups and prototypes as well as of the final system/tool (Gena & Weibelzahl, 2007). Conducting an evaluation can supply direct information about how people use the system/tool and the problems with a specific interface (Holzinger, 2005). In addition, useful feedback from the evaluation can help tool developers with redesign of the system/tool (Gena & Weibelzahl, 2007). There are various types of evaluation and the one that we focus in this chapter is usability evaluation. We briefly introduce several concepts and definitions regarding usability evaluation before we describe the methods that we applied for evaluating our prototype tool, i.e. the critic specification editor (Marama Critic definer).

A considerable number of studies have discussed and published information on usability testing/usability evaluation. These include (Blecken & Marx, 2010; Hartson, Andre, & Williges, 2003; Holzinger, 2005; Jacko & Sears, 2003; Khan, Israr, & Hassan, 2010; Leventhal & Barnes, 2008; Lund, 1998; Nielson, 1993;

Rubin, 1994). Most studies suggest that usability plays an essential task in the improvement and development of effective and efficient systems/tools.

Rubin (1994) describes the term usability testing as “a process that employs participants who are representative of the target population to evaluate the degree to which a product meets specific usability criteria.” Another study which is recently published in (Hwang & Salvendy, 2010) suggests that usability evaluation is “essential to make sure that software products newly released are easy to use, efficient, and effective to reach goals, and satisfactory to users.” Though the different terms “usability testing” and “usability evaluation” are used here it is very clear that both focus on the aspect of usability. Leventhal and Barnes (2008) report the definition of usability from the international standard ISO 9241-11 as: “Usability: the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.” In another published article by Holzinger (2005), usability is defined as “the ease of use and acceptability of a system for a particular class of users carrying out specific tasks in a specific environment.” Hence, we can say that usability testing and usability evaluation serve the same role which is to evaluate a system or product that can be used by the potential users in order to achieve specific usability aims such as ease of use, efficiency, effectiveness and satisfaction for users. Here we use the term usability evaluation to describe our evaluation method.

When conducting an evaluation, one has to consider whether it is a formative evaluation or summative evaluation. Formative evaluations are evaluations that take place throughout a system’s or tool’s development to improve a design. Summative evaluations are conducted after a system/tool is completed to assess a design (Blecken & Marx, 2010; Hartson, et al., 2003). In our case, we conducted a summative evaluation and this is explained in the following sections. There are several methods for evaluating usability, such as Think Aloud (TA), Heuristic Evaluation (HE), Cognitive Walkthrough (CW), field observation and questionnaires. Information on these methods can be found in (Gena & Weibelzahl, 2007; Holzinger, 2005; Hwang & Salvendy, 2010; Nielson, 1993).

This chapter describes the methods used to evaluate our prototype tool, which are: questionnaires, observation and think aloud. We also applied the Cognitive Dimensions of Notations framework (CDs) for discussing the usability of our tool. CDs have been used to design a set of generalised questionnaires intended for system/tool users evaluating the usability of a tool (Blackwell & Green, 2000). We used some of these to design a questionnaire that integrates with the CDs elements. We also made use of the USE (Usefulness, Satisfaction, and Ease of Use) Questionnaire from (Lund, 1998) to design a questionnaire that deals with usability issues. In addition, we used the observation and think aloud approach while our participants worked on our prototype tool. All of these are explained in the following sections.

9.2 Cognitive Dimensions of Notations framework (CDs)

The Cognitive Dimensions of Notations framework (CDs) as reported in (Blackwell et al., 2001) “is a framework for describing the usability of notational systems ... and information artifacts ... ”. The CDs are used to support the non-specialists in evaluating usability of information-based artefacts (Green & Blackwell, 1998). Many researchers have employed CDs for the purpose of usability evaluation for their tools, such as (Pereira, Mernik, Cruz, & Henriques, 2008), (Li, et al., 2007b), (Tukiainen, 2001), (Cox, 2000), (Green & Petre, 1996), and many others. According to Green and Blackwell (1998), the CDs approach aims to provide surface analysis rather than extensive analysis. The CDs were proposed to discuss the usability tradeoffs that occur when designing diverse tools and systems (Green & Blackwell, 1998; Green & Petre, 1996). Table 9.1 presents a brief description of the Cognitive Dimensions adopted from (Blackwell, et al., 2001). The details of the CDs approach that comprises these fourteen dimensions can be found in many published articles, e.g. (Green & Petre, 1996), (Green & Blackwell, 1998), (Blackwell & Green, 2000), (Blackwell, et al., 2001), (Green, Blandford, Church, Roast, & Clarke, 2006), and others.

We apply the CDs approach in the form of a questionnaire about our critic specification editor to review the usability of the editor for specifying critics for the

DSVL tools, specifically for our Marama-based tools. However, we do not include all the dimensions as stated in the CDs. The results from the survey questionnaire are discussed in section 9.5.

Table 9.1: The meaning of each dimensions (Blackwell, et al., 2001)

Dimension	Meaning
Viscosity	Resistance to change
Visibility	Ability to view components easily
Premature Commitment	Constraints on the order of doing things
Hidden Dependencies	Important links between entities are not visible
Role-Expressiveness	The purpose of an entity is readily inferred
Error-Proneness	The notation invites mistakes and the system gives little protection
Abstraction	Types and availability of abstraction mechanisms
Secondary Notation	Extra information in means other than formal syntax
Closeness of Mapping	Closeness of representation to domain
Consistency	Similar semantics are expressed in similar syntactic forms
Diffuseness	Verbosity of language
Hard Mental Operations	High demand on cognitive resources
Provisionality	Degree of commitment to actions or marks
Progressive Evaluation	Work-to-date can be checked at any time

9.3 The Four Criteria to Evaluate Usability

Another issue that needs to be considered when conducting a usability evaluation is the criteria or elements that need to be assessed by the potential users of the tool under evaluation. There are several models that can be employed to perform a usability evaluation, such as Shackel's model of usability, Nielson's model of usability and Eason's model of usability (Leventhal & Barnes, 2008). The Shackel model identifies the four items of usability as effectiveness, learnability, flexibility,

and attitude. In the Nielson model, there are five dimensions that are contributed to usability: easy to learn, efficient to use, easy to remember, few errors, and subjectively pleasing. Similarly, the Eason model recognises three aspects for usability: 1) system (user interface) characteristics- ease of use, ease of learning, task match; 2) task characteristics – frequency and openness, and 3) user characteristic- knowledge, motivation and discretion. Researchers can adopt those usability models as their guidelines and create their own criteria to evaluate their tool. For instance, in (Khan, et al., 2010) studies, they selected five different criteria to evaluate the usability of their tool, (i.e. *ThinkFree doc.*). Their criteria are: effectiveness, efficiency, satisfaction, learnability and utility. We are not adopting any specific usability models, however, we have used items from the USE Questionnaire (Lund, 1998) to design our survey questionnaire for evaluating the usability of our prototype tool.

We defined four elements to evaluate the usability of our prototype tool. These are:

1. Usefulness – refers to how useful the tool is in helping the users to be more effective and able to accomplish a task in an easier way.
2. Ease of use – refers to how easy the users can work with the tool's interface after they have understood the tool.
3. Ease of Learning – refers to how easy the users can learn and understand the new/untried tool.
4. Satisfaction – refers to the user's satisfaction in using/working with the new tool.

We designed a short questionnaire that can be used to measure the four elements of usability for users. We selected several questions from (Lund, 1998) and developed our survey questionnaire with 12 questions, 3 for each of the usefulness, ease of use, ease of learning and satisfaction categories. These questions are in Section two of the questionnaire part of the survey. The results are discussed in the section 9.5.

9.4 Design of the Survey

In this section, we present the design of our survey carried out to evaluate the prototype of our visual language-based tool, critic specification editor (i.e. *Marama Critic definer*). The objectives of the survey are:

1. To evaluate the visual design critic authoring tool to test the tool's usability and effectiveness in constructing critics for domain-specific visual language (DSVL) tools.
2. To obtain qualitative information on user perceptions of the 'template-based critic authoring' - whether it is easy and useful for generating critics for their DSVL tools.

Our survey is structured into two parts. Part one involves a task list and an observation. The task list contains the tasks needed to be completed by a participant. The observation was conducted and data was collected while the participant performed the tasks. Part two provided a questionnaire to be answered by the participant once he/she had completed the tasks. We asked participants to participate in this survey on a voluntary basis and their participation was treated anonymously. In the following, we describe the observation and questionnaire design, the method used to evaluate the usability of the critic specification editor as well as the end users' subjective comments.

9.4.1 The Observation Design

The observation method was used to achieve the second objective of the survey stated above. We applied a combination of two methods for the observation: 1) unobtrusive observation and 2) obtrusive observation. With unobtrusive observation, the participant was observed in how they used the tool. Thus, we learned whether the participant could use the tool in an easy and efficient way. The aspects that we wished to observe from the participants were: 1) how participants defined critics for the developed tool; 2) did participants managed to complete the critic-authoring task and 3) how participants navigated different parts of the tool. With obtrusive observation, participant was asked to speak out what he/she thought while using the

tool. Hence, we learned from the participants more about the usefulness and the acceptance of the tool. These two methods are performed at the same time as the participants doing their task on the given tool. We wanted the participants to feel relax while doing the task, so we allowed them to express what they think about the tool via a think aloud approach. We collected the observation data while the participants were performing the tasks. We also collected the views/comments expressed by the participants.

The observation was carried out when a participant performed a set of tasks that he/she was required to do while interacting with the prototype tool. In both methods, no personal information about the participant was collected as participation in this survey was treated anonymously.

9.4.2 The Questionnaire Design

In this section, we describe our questionnaire. According to Blecken and Marx (2010), questionnaires can be applied for both summative and formative evaluations and also can assist to acquire quantitative information on user judgement of a system or tool (Blecken & Marx, 2010). Furthermore, questionnaires can serve to assess an entire tool or only partial aspects of a tool (Blecken & Marx, 2010). We have used a questionnaire in our survey to assist us in performing the usability evaluation for our prototype tool.

We designed our questionnaire based on the CDs approach and the original Blackwell and Green questionnaire (Blackwell & Green, 2000). The questionnaire from (Blackwell & Green, 2000) acted as a guideline for us to identify the relevant items to put in our survey questionnaire.

Our questionnaire has two sections: 1) background information; and 2) prototype tool information. Section one contains four questions to reflect the background of the participant. The questions we designed for this were based on the questions from (Blackwell & Green, 2000). Section two consists of six categories that we classified as: 1) usefulness; 2) ease of use; 3) ease of learning; 4) satisfaction; 5) cognitive dimensions of critic authoring task; and 6) open end question to which participants

can freely respond. We employed a Likert scale to obtain participants' feedback about the usability of our prototype tool (i.e. critic specification editor). For each question statement in this section, we classified the responses as 5-point Likert rating scales: 1=strongly disagree, 2=disagree, 3= undecided, 4= Agree and 5=strongly agree.

For categories (1) to (4), we designed questions based on the USE questionnaires (Lund, 1998), whereas category (5) was based on the questionnaire from (Blackwell & Green, 2000).

Overall, the questionnaire is comprised of twenty seven different questions which the selected participants filled in to evaluate the usability of the critic specification editor. Before the end user evaluation took place we gained an ethics approval from the University of Auckland Human Participants Ethics Committee. Please refer to Appendix A: Evaluation Survey for the questions.

9.4.3 Survey Method

Invitations to the survey were made to potential participants who had basic background knowledge of the Marama meta-tools. We managed to gather a group of 12 volunteer researchers and students who met the background requirement and who were interested in both modelling and the development of modelling tools to support their work. Four of the participants were computer science researchers, who have used the Marama meta-tool to develop tools for their research work. Another 8 participants were postgraduate Computer Science students who had taken a course in which Marama had been introduced and involved in a coursework assessment.

The usability evaluation survey was conducted individually with the volunteer participants. The participants were given a description of how to use the prototype tool, i.e. critic specification editor (Marama critic definer) and the functions involved with it. We then asked the participants to perform five different tasks. The tasks had to be carried out respecting any constraints. These tasks were:

1. Task 1: Explore the Marama tool that was given;
2. Task 2: Identify critics for the tool;
3. Task 3: Add the critics to the tool using the critic authoring templates;

4. Task 4: Run the critics;
5. Task 5: Construct a critic via a formula function.

We also observed how the participants went about using the critic specification editor. Participants were asked to think aloud and give suggestions about the tool. After performing all of the five tasks we distributed the survey questionnaire to participants to collect their responses. Participants filled out the questionnaire at their own pace without supervision. We then collected the response data for our analysis. In general, each participant took less than 1 hour to perform the evaluation survey. The result of the survey and analysis are discussed in the following section.

9.5 Survey Result and Analysis

In this section, we present the survey results and analysis.

9.5.1 Analysis of Task List and Observation

Part one of the evaluation survey was to observe how the participants use/work with Marama meta-tools and carry out the five tasks that we structured in the survey. As the participant performed the five tasks, the aspects that we wanted to observe from the participants were: 1) how participant defined critics for the developed tool; 2) did participant manage to complete the critic-authoring task and 3) how participant navigated different parts of the tool. Participants were also encouraged to say aloud while interacting with the tool.

We mentioned in our observation design that we used a combination of unobtrusive and obtrusive observation methods. These two methods were performed at the same time as the participants performing their task on the given tool. While doing their work, the participants were asked to express what they think about the tool via a think aloud approach. We collected the required data that we observed while the participants performed the tasks and also any views/comments expressed by the participants.

The possible observations for each task are reviewed below:

1. Task 1. Explore the Marama tool that was given to you.

The first task was to allow the participants to explore the three main editors of the Marama meta-tools, i.e. *Metamodel definer view*, *Marama Shape Designer view*, and *Marama Viewtype Definer view*.

Observation results: All participants appeared to be familiar with the three editors based on their previous basic knowledge on Marama meta-tools. They understood the function of each editor and were able to navigate between the three editors.

2. Task 2. Identify critics for the tool.

The second task was to let the participants think up and list several critic statements that were relevant to the given Marama-tool. The participants also needed to identify an appropriate feedback (fix action) for each of the identified critics. A space is provided in the survey form for them to write their critic statements in English.

Observation results: With the think aloud approach, the participants communicate with the researcher/observer to gain an understanding of a critic statement. All the participants managed to understand a simple critic statement for the given Marama tool after one or two examples of critic statements were shown by the observer. However, most of the participants did not write down the critic statements in the space given in the form, but instead they preferred to proceed with task 3 to define their critic statements. Below are the examples of critic statements for a MaramaUML tool written by one of the participants on their survey form.

Critic	Feedback
“- Class should have a unique name property” “ –Class should not have more than (some limit) of other classes associated to it”	“-Rename or Remove the class” “-Remove the association”

3. Task 3. Add critics to the tool using the critic authoring templates.

The third task was to allow the participants to implement their chosen Marama critic by specifying the tool critics via the *Marama Critic Definer* views. The participants defined their tool's critic by selecting a *CriticShape* icon which automatically associated with a form-based interface, the *Critic Construction View*. The participants then selected the appropriate templates from this interface (Attribute constraint template/ Relationship constraint template/ Action Assertion template) that could represent their critic statement. Next the participants identified the feedback (fix action) for the critics that had been defined. The participants then selected the *CriticFeedbackShape* icon which is also automatically associated with a form-based interface, *Critic Feedback View*. The participants subsequently selected the necessary fix action listed in the interface. Once the participants were satisfied with their critics and feedbacks, the participants saved their work.

Observation results: All participants managed to perform the third task; however they needed some guidance from the tool developer (i.e. the researcher who acted as the observer). The critic authoring templates were not easily understandable by the participants for first time use due to their unfamiliarity with the critic templates concept. We did provide a critic authoring template guideline in the tool but it is unreasonable to expect first time users to pick them up quickly and have a good understanding of the templates. However, most participants found it interesting to specify critics just by selecting the appropriate template and then select the necessary fix action that had been suggested. Overall, the participants managed to complete the critic-authoring task by specifying simple critics using the templates and then specifying the fix action for the critics. We got some useful feedback through the think aloud method and below are some of the comments.

Participants' comments:

- “It would be easier to specify critics after the critic authoring templates are well understood”;
- “It is hard for a first time user to specify critics using the templates. However, after regular use of the tool it would be easy”;
- “It takes time to understand the templates and also to select the appropriate templates to represent a critic”.

4. Task 4. Run critics.

The fourth task was to allow participants to see how the critics are implemented in the Marama Model Project and Marama Diagram for the given Marama tool. The participants created a simple diagram and tried to violate the critic rules that they defined in task 3. The participants could see the critic message and feedback (fix action) which was displayed at the Marama diagram they created.

Observation results: All of the participants were impressed with the displayed critic message and feedback that was generated at their Marama diagram. They found it interesting to use the critic authoring templates to generate tool’s critics.

5. Task 5. Critic via formula function.

The fifth task asked participants to construct simple critics using the Object Constraint Language (OCL) via the *Formula* icon that already exists in the Marama meta-tool. The participants then saved the Formula and ran the critic as per task 4. The participants needed to open the Eclipse-Problem View to see the response to the critics’ violation.

Observation results: This task was unfortunately not performed by most of the participants because they appeared to forget the required OCL expressions. They had previously learned OCL expressions but they were not familiar with using OCL to express a critic. Thus, for this last task, the researcher/observer ended up showing a simple example of an OCL

expression to represent a critic. The critics' responses were then displayed in the Eclipse-Problem View when a critic rule was violated in the Marama diagram. The feedback that we received from most participants through the think aloud method is as below.

Participants comments:

- “Prefer to use the templates instead of using OCL expression or through coding in specifying critics”;
- “Using the templates to specify critics are much easier compared to OCL expression or coding”.

9.5.2 Analysis of Questionnaire Responses

The second part of the survey was to answer the survey questionnaire. The questionnaire was in two sections. Section one was to obtain the background information of the participants. The aim of this section was to find out whether the participant is a skilled, intermediate or novice user of the tool under evaluation, and whether the participant has experience of other similar tools. The following table shows the four questions in section one that was answered by the twelve participants.

Table 9.2: Section 1- Background information

Participants	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Participant type	PS	Res	Res	Res	Res							
Q1. Level of proficiency	S	N	I	I	I	I	I	I	S	S	S	I
Q2. Used similar tools?	No	Yes	Yes	Yes	Yes	Yes						
Q3. Developed design critics	Yes	No	No	No	Yes	No	No	No	Yes	No	Yes	No
Q4. Name of the Marama tool	Marama UML	Marama W	Marama X	Marama maY	Marama maZ							

Note: PS = Postgraduate student, Res = Researcher.

Q1. S = Skilled, I = Intermediate, N = Novice

From the Table 9.2, it shows that seven participants are intermediate users; four participants are skilled users and one participant is a novice user in using the Marama meta-tools. From the twelve participants, more than half of them never used other tools similar to the Marama meta-tools. We then asked whether participants had experience designing critics for any of their software tools. Only four participants had done so. The eight postgraduate students were given the same Marama tool to perform a usability evaluation. However, the researchers were using their own simplified Marama tool to perform the usability evaluation on the critic specification editor.

In an earlier section, we mentioned that the participants needed to perform several tasks before they could answer the questionnaire. The section after the background information aimed to obtain prototype tool information. There are six categories in this section, comprising twenty three questions of which three questions were asked about **usefulness**, three questions were asked about **ease of use**, three questions were asked about **ease of learning**, three questions were asked about **satisfaction**, ten questions were asked about **cognitive dimensions of the critic authoring task**, and **one open ended question** was asked on how the tool could be improved. The participants' responses for the six categories are discussed below.

A. Usefulness, Ease of Use, Ease of Learning and Satisfaction

The questions in this section focused on standard usability questions, such as usefulness, ease of use, ease of learning and satisfaction. The usability responses shown in Table 9.3 and Figure 9.1 were highly positive. In all questions by far the majority of participants answered that they agreed or strongly agreed, indicating the tool had strong appeal, and was perceived to be highly usable, useful, easy learning and highly satisfied by our target end users. Please refer to Table 9.3 and Figure 9.1, which show the participants' responses based on a 5 point Likert scale (1=strongly disagree, 5=strongly agree).

Table 9.3: Usability responses

Section 2- Prototype Tool Information	SD(1)	D(2)	U(3)	A(4)	SA(5)
A. Usefullness					
It is usefull	0	0	0	5	7
It helps me be more effective	0	0	0	7	5
It makes the things I want to accomplish easier to get done	0	0	1	6	5
B. Ease of Use					
It is easy to use	0	0	1	8	3
It is user friendly	0	0	2	6	4
I don't notice any inconsistencies as I use it	0	1	2	5	4
C. Ease of Learning					
I learned to use it quickly	0	1	0	7	4
I easily remember how to use it	0	0	1	6	5
It is easy to learn to use it	0	1	1	6	4
D. Satisfaction					
I am satisfied with it	0	0	0	7	5
I would recommend it to a friend	0	1	1	6	4
It is fun to use	0	0	3	6	3

Legend: SD=Strongly Disagree, D=Disagree, U=Undecided, A=Agree, SA= Strongly Agree

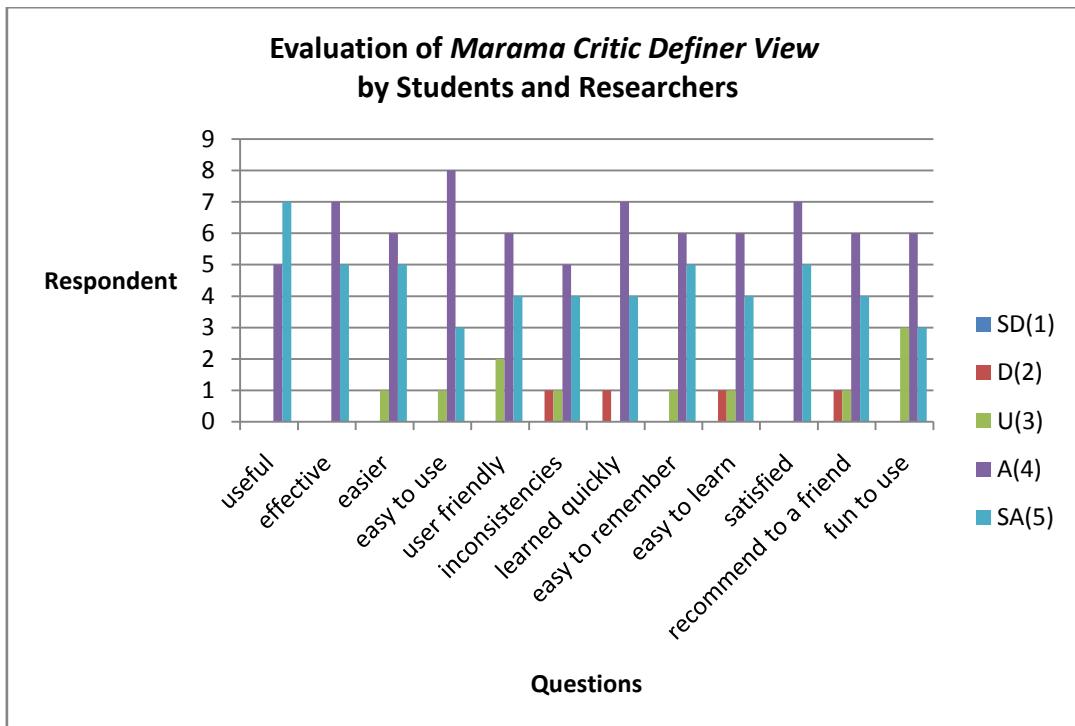


Figure 9.2: Usability responses.

B. Cognitive dimensions analysis of the critic authoring task

We adapted the questionnaire designed by (Blackwell & Green, 2000) based on the Cognitive Dimensions of Notations (CDs) framework. This provided questions targeted at each of the cognitive dimensions as we were interested in the tradeoffs amongst those dimensions that participants observed. For this section too, the participants answered on a 5-point Likert scale (1=strongly disagree, 5=strongly agree). Please refer to Table 9.4 and Figure 9.2 to see the CDs responses.

Table 9.4: Cognitive dimension responses

E. Cognitive Dimensions of Critic-Authoring Task	SD(1)	D(2)	U(3)	A(4)	SA (5)
It is easy to see various parts of the tool	0	1	0	9	2
It is easy to make changes	0	0	1	7	4
The notation is succinct and not long-winded	0	0	2	7	3
Some things do require hard mental effort	0	4	4	3	1
It is easy to make errors or mistakes	0	5	4	3	0
The notation is closely related to the result	0	0	0	8	4
It is easy to tell what each part is for when reading the notation	1	0	2	5	4
The dependencies are visible	0	0	2	7	3
It is easy to stop and check my work so far	0	0	1	6	5
I can work in any order I like when working with the notation	0	0	0	8	4

Legend: SD=Strongly Disagree, D=Disagree, U=Undecided, A=Agree, SA= Strongly Agree

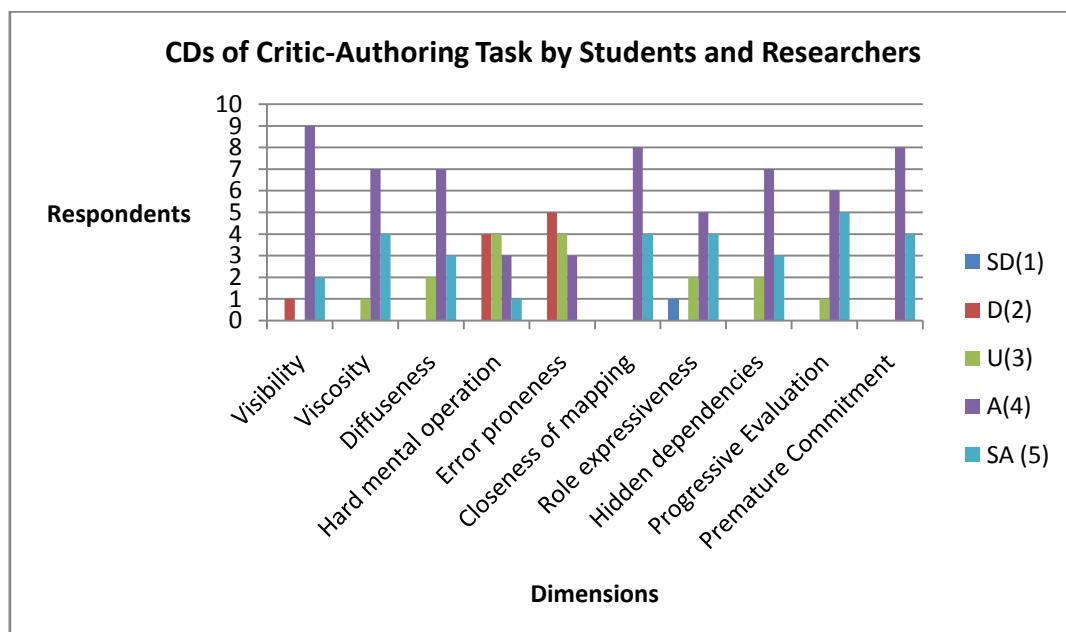


Figure 9.3: CD questionnaire responses.

Figure 9.2 shows the responses to the questions concerning each of the cognitive dimensions. In the following section we discuss each of these in more detail.

Visibility

This CD indicates the ability to view various aspects of the tool easily. Nine out of the total 12 participants answered that it is easy to see various parts of the tool. The Marama Critic Definer view shows two simple visual notations to represent critic (i.e. *CriticShape*) and critic feedback (i.e. *CriticFeedbackShape*), and two connectors to show the link between critic and critic feedback (i.e. *CriticFeedbackLink*), and dependency between critics (*CriticDependencyLink*). The *CriticShape* and *CriticFeedback Shape* are associated with form-based interfaces to assist the user in specifying a critic and a critic feedback. The only respondent who doubted the easiness to see various parts of the tool commented that was due to the lack of understanding of the meta-tool concept and as a novice user it is hard to see the function of various parts of the tool.

Viscosity

Viscosity reflects a design's resistance to change. Eleven participants said that it was easy to make changes. The user can easily change critics and critic feedback that have been defined in the Marama Critic Definer view. Only one respondent answered undecided. This is probably due to the small size of a critic specification instance.

Diffuseness

Diffuseness refers to the verbosity of language, i.e. the number of symbols required to express a meaning using the language. Ten participants answered that the notation is succinct and not long-winded. The participants commented that the notation is a straightforward representation of a critic and its feedback, as well as the connectors that link them. A critic is defined via three templates provided in the critic construction interface. Thus, a user only defines a critic based on the selected template. Whereas to specify a critic feedback it only requires the user to define five properties: critiquing strategies, modes of critiques, explanation, suggestion, and critique message. Two participants replied that they were undecided to this element.

Hard Mental Operations

This dimension reflects the degree of demand on cognitive resources. Four participants disagreed, four participants agreed (3 agree and one strongly agree) and four were undecided as to whether using the tool required hard mental effort. The four participants who agreed claimed that they needed to concentrate and think carefully before using the critic templates to specify a critic. This may be because the users were unfamiliar with the critic authoring templates. Our aim was to provide a way of making the critic specification much easier, but at the heart of it, critic specification task itself is something difficult to do. However, a regular use of the templates can overcome the cognitive load.

Error Proneness

Error proneness refers to the ability of the tool to induce ‘careless mistakes’. Five participants disagreed, three participants agreed and four were undecided as to whether the tool was likely to induce mistakes. This dimension has a similar issue as the hard mental operations dimension. The participants who answered it is easy to make mistakes raised the issue that unfamiliarity with the templates can cause users to make mistakes in specifying critics. This is an initial barrier which can be overcome by more frequent use of the tool. However, five participants found it has low error proneness as the notation is very straightforward and supported by a form-based interface which is familiar to most users.

Closeness of Mapping

This dimension reflects the closeness of the representation to the domain it describes. All of the participants agreed that the Marama Critic definer view provides a notation that is closely related to the domain. The critic definition closely relates to the critic statement/phrase that the user specified based on the available templates. The specification of a critic feedback is straightforward by just clicking on the required options and adding the explanation and a critique message.

Role Expressiveness

Role expressiveness indicates that the relationships among components should be obvious. Nine participants answered it is easy to tell what each part is for when

reading the notation. Only one respondent disagreed and two participants were undecided. In the Marama Critic definer view, it is obvious how to specify a critic and a critic feedback because it only involves two simple notational elements with each associated with a form-based interface.

Hidden Dependencies

This dimension assesses the existence of hidden links among parts of the tool. Ten participants said that the dependencies are visible and two participants are undecided. Hidden dependencies are primarily between the visual critic definer view and the form based template views. Moody (2008) argues that this type of hierarchical dependency is of positive benefit in his Principal of Complexity Management (Moody, 2008).

Progressive Evaluation

Progressive evaluation indicates the ability to test code as it is being developed. Eleven participants answered it is easy to stop and check work progress. The Marama Critic definer view allows the critic and critic feedback specifications to be evaluated at any stage. Partially completed critics and feedbacks for a developed Marama tool can be executed as well. Critics and Feedbacks properties can be edited easily and any new changes will take effect during the model execution of the tool.

Premature Commitment

This dimension reflects the order of steps that a user must follow to achieve a specific outcome. All of the participants agreed that there are no premature commitments in the Marama Critic Definer view. The user can freely specify a critic using any templates (attribute, relationship or action assertion). However, the user does need to define a critic first before a critic feedback can be specified and linked with the defined critic. This dependency is obviously seen as a natural one by end users as they do not appear to regard it as forcing them to prematurely commit to something at a point where they are not ready. The user does not need to have a complete set of critics and critic feedbacks to be specified in the Marama Critic definer view. The user can add a critic as well as the critic feedback for the Marama tool incrementally as he/she encounter new critics.

C. Open ended question to improve the tool design

We also provided the participants with an open ended question and space for them to write comments about how to improve the critic authoring tool. In general, the comments/feedbacks suggested that specifying critics visually and via a template-based style is simple and effective. Issues that raised by some participants to improve the tool are shown in Table 9.5. These issues are discussed in Chapter TEN.

Table 9.5: Participants' Comment

Participant	Comment
1	“Overall, it is pretty cool, maybe HCI is one aspect to improve by using AI feedbacks.”
2	“I find the tool may be hard to understand initially for a novice user with little experience with meta-tools. I think it is not easy to learn at first because the critic definer is a tool for the meta-tool and the levels of abstraction is high. However, I feel that a regular user of this tool would find the functions easy to remember after learning it for the first.”
3	“Include the templates as visual entities. Possibly also connect the shapes of critics with the shapes of the entities they affect (and the other way around)”
4	none
5	“Everything is good”
6	“Templates should be explained better. Some bugs need to be freed”
7	“The view of the feedback critic/critic construction shall be automatically focused when creating/extending a critic”
8	“UI cleaned up e.g. Criticfeedback toolbar opens when a feedback is selected. Icons associated with each template appear on the feedback shapes? Form should be dependent on which template is selected, so user doesn't accidentally fill in incorrect or unnecessary fields.”
9	“Visual representation relation between critic and tool model, and graphical notation in critic, e.g., highlight”
10	“It will be nice to involve colours in the screen to show different critics.”
11	“Noticed 1-1 mapping of critic & feedback, suggested adding feedback into critic shape, with connection & layout automatically created.”
12	none

9.7 Conclusion

In this chapter, we have presented and described a usability evaluation survey of our prototype tool, i.e. *Marama Critic Definer* to specify critics for a DSVL tool. Like most tools or systems, we believe that our critic specification tool can benefit considerably from end user involvement in evaluation. The evaluation survey is based on the combined use of two approaches: observation and questionnaires. The

Cognitive Dimensions framework and four usability criteria are applied and specified in the questionnaire form. We conducted our survey with twelve participants and each participant conducted the evaluation individually. Though the sample size is small, we complied with the general rule suggested by Hwang and Salvendy (2010) for usability evaluation: the 10±2 Rule (Hwang & Salvendy, 2010).

The survey results have shown a good degree of satisfaction of our participants with our critic design tool integrated with the Marama meta-tools. The survey results demonstrated that for most participants our approach appears to be useful in assisting these participants in the critic specification task. Our approach also appears to nicely complement the other components of the Marama meta-tools and is integrated with these.

However, limitations of the tool are also revealed through the survey results. Thus, some minor improvements are needed to improve the usability of the critic specification editor integrated with Marama meta-tools. The evaluation survey has also provided a number of suggestions to improve the critic specification editor (Marama Critic Definer). These suggestions are listed in the previous section and are later considered for our future work.

Chapter 10

Conclusions and Future Work

This chapter concludes this thesis by presenting a research summary of the work carried out in responding to the research question. It discusses the overall research results as well as the limitations and strengths of the research. This chapter also suggests some future work to extend the research followed by a brief summary at the end of this chapter.

10.1 Research Summary

We have described our research work extending the use of “critics” into meta-tool environments that implement domain-specific visual language tools with an aim to support end-user tool developers to simply specify critics for domain-specific visual language (DSVL) tools.

In **Chapter 1** of the thesis, we identified that critic authoring continues to be a challenge despite critics having been recognised as an efficient feedback-providing mechanism in diverse domains. The process of authoring or customising critics is not an easy task to be performed especially by novice and end-user tool developers. Furthermore, we realised that critics have not been adopted within meta-modelling tools that implement DSVL tools. As a result, we proposed to provide a critic specification approach within a meta-tool environment that is accessible to end-user tool developers for specifying critics for DSVL tools. We formulated research questions that enabled us to identify possible solutions for our proposition.

Through our review of the available literature on critics and constraint specifications presented in **Chapter 2**, we showed that the use of critics is often applied in application domains and constraint specification is common for meta-tool environments. The process of defining constraints for meta-tool environments is hard as it requires good knowledge of programming skills, it uses a formal approach and it involves heavy cognitive load. Thus, we wanted to provide a critic

specification that was tailored to critic authoring and user accessible to replace the complex constraint specification approach.

A methodology to organise this research work was described in **Chapter 3**. We identified several important steps that were required to attain the research aim. Each step in the methodology produced artefacts: critic taxonomy, prototypes, evaluation results and so on which reflects the following chapters of this thesis.

Review of the related literature concerning critics resulted in a new critic taxonomy described in **Chapter 4**. We proposed our critic taxonomy based on several aspects that characterised critics (or critiquing systems). These aspects are gathered widely from the critic literature. We identified eight groups for our critic taxonomy: critic domain, critiquing approach, modes of critic feedback, critic rule authoring, critic realisation approach, critic dimension, types of critic feedback, and types of critic. We applied our taxonomy to ten tools that have critic support. The mapping of the tools to our critic taxonomy shows that the practice of critics is supported by the critic taxonomy. Furthermore, this critic taxonomy development has assisted us in identifying the needs of our own critic specification tool.

In **Chapter 5** we described our approach for specifying critics for a DSVL tool environment. A visual and template-based approach was introduced in this chapter. We described our adaptation of business rule templates to the software tool domain, specifically our critic authoring domain. We described the visual notation for our critic specification approach. We also analysed the visual notation design of our critic specification tool based on the Physics of Notations (Moody, 2008). Our visual notation design approach satisfied some of Moody's principles. The combination of the two approaches resulted in what we call a "visual and template-based approach of critic specification for DSVL tools".

Our initial attempt at critic specification development was described in **Chapter 6**. We developed our first prototype for critic specification using MaramaTatau (N. Liu, et al., 2007) and this was a useful stepping stone for us to understand the necessary building blocks for an improved critic specification approach. Experience gained from prototype 1 led us to develop prototype 2 for our critic specification

approach. From prototype 2, we gained experience in applying the business rule template concept as an alternative approach to specify critics. However, we recognised some problems with prototype 2 and resolved these with a new approach by developing prototype 3.

The final development of our critic specification approach, i.e. prototype 3, was described in **Chapter 7**. We created a new critic specification tool, *Marama Critic Definer* that is accessible to end-user tool developers for critic-specification task. This new approach comprises four main components: visual critic definer editor, critic construction editor, critic feedback editor, and critic template editor. The function of these editors is described in this chapter. We offered a notational representation and critic authoring templates to end-user tool developers to specify critics for their DSVL tools without the need to have a deep technical knowledge of critic construction. We defined a critic authoring guideline to assist end-user tool developers in specifying critics and authoring a new critic template which can be done via critic template editor.

The utility of prototype 3, which represents our critic specification approach, is presented in **Chapter 8**. We proved our concept for critic specification within three different domains of DSVL exemplar tools using three case studies. We described a health care plan modelling tool as our first case study to demonstrate the critic specification task. The second case study concerned a business process modelling tool. We illustrated the specification of complex critics for this tool. The final case study concerned UML design. In this case study, we described the task of customising the critic authoring template when a desired critic specification is not defined in critic authoring templates. We also illustrated the function of the critic feedback editor in one of the case studies. We claimed that our critic specification approach can be applied across different domains of DSVL tools.

In **Chapter 9** we presented an evaluation of our critic specification approach via an end-user evaluation survey. We defined four usability criteria and ten elements from the Cognitive Dimensions framework in our questionnaire to evaluate our critic specification approach and tool. The usability responses that we obtained from the

evaluation were highly positive indicating that the critic specification had strong appeal. The Cognitive Dimensions responses that we received from the evaluation were also encouraging and each of the dimensions were discussed in this chapter. Through these evaluations, we were able to establish that critic specification and implementation for domain specific visual languages can be made accessible to end-user tool developers. We were also able to show that the combination of a notational representation and a critic authoring template-based approach was useful, highly usable, easy to learn and of high satisfaction to our target end-user tool developers.

Limitations of the Research

Not surprisingly, the evaluations exposed some limitations of our research. These limitations can be ameliorated in future work. These include:

- Critic and feedback specification can only be specified based on the predefined templates that were implemented for the prototype critic specification tool. We designed and developed our critic authoring template based on BR templates (i.e. attribute constraint templates, relationship constraint templates and action assertion templates) to support/prove the critic specification process. Similarly, our critic feedback specification only support limited actions to resolve defined critics;
- Currently the modes of critiques for the critic specification tool only support a textual style without the use of graphical style. For instance, one respondent suggested to consider highlighting (e.g. with colour) the design item that triggered a critic. Similarly, another respondent recommended to consider colouring to differentiate different types of critic;
- During the tool's evaluation, the guideline that provides explanation of the critic templates developed in the critic construction editor was a minimal guideline. However, we can easily resolve this issue by providing more detailed guidelines and examples in the tool to assist user to specify critics;

- The critic and critic feedback icons are not automatically associated with the critic construction editor and critic feedback editor. The user had to select the required editor to perform the required task;
- In general, a potential weakness of the research is that the presented approach and tool may be of little interest or benefit to expert tool developers. However this research would likely provide benefit to the majority of novice, intermediate and end-user tool developers, which was our target audience.

These minor limitations observed in our tool can be improved in future work.

Strengths of the Research

The implementation of our critic specification approach and tool contributed several benefits. These include:

- A simple way to express and define critic condition specifications based on the structured critic rule templates given, making it easier for end-user tool developers to author and realise critics;
- A simple way to express and define critic feedback specifications based on structured templates also making it easier for end-user tool developers to specify and realise critic feedback;
- The process of authoring critics and their feedback is made easier through the combination of the visual specification editor (i.e. critic definer editor) and the two form-based template editors (i.e. critic construction editor and critic feedback editor);
- The critic specification tool provides guidelines (i.e. critic authoring guidelines) for the user to customise critic rule templates through the critic template editor;

- The critic authoring templates facilitate the linking of critic statements to meta-model elements.

10.2 Research Contributions

We have described our critic development approach to support end-user tool developers to specify critics in an effective and easy way. The research discussed in this thesis contributes to the field of software engineering particularly in the area of critic tools and critiquing systems development. The main contributions from this research are as follows:

5. This research provides a taxonomy of critics that can assist other users/designers or developers in obtaining relevant information about critics. Our critic taxonomy identified eight groups: critic domain, critiquing approach, modes of critic feedback, critic rule authoring, critic realisation approach, critic dimension, types of critic feedback, and types of critic. We believe that our critic taxonomy will be useful to critic developers in providing a meaningful way of describing and reasoning about critics. We also believe that our critic taxonomy is useful in guiding the critic developer towards realising robust critic capabilities by comparing and contrasting different critic dimensions.
6. This research provides a visual way of expressing or constructing critics for domain-specific visual language (DSVL) tools. Notational representation of critic authoring facilities is offered to end-user designers to express critics for their DSVL tools. Furthermore, this research provides support for end-user tool developers who want to express critics for their specific tool without the need to have a comprehensive technical knowledge on expressing and constructing critics.
7. This research provides a critic authoring template-based approach which is much easier and quicker to author critics compared to other approaches for designing and realising the critics. An end-user tool developer uses the critic authoring template to generate critic rule templates. The critic rule

templates (CR) adapt the business rule (BR) templates which are currently applied in the business process domain. We attempted to apply the critic rule templates in the software tool domain. By using the critic authoring templates, it is fairly easy for end-user tool developers to introduce new critic template or modify existing critics in the tool.

8. This research included prototype development of a visual critic authoring tool which was embedded in the existing Marama meta-tool and which acts as a proof-of-concept of our approach. We evaluated the prototype using an end user study conforming to the Cognitive Dimensions (CD) approach (Green & Blackwell, 1998) and usability aspects. We also analysed our design notation using the Physics of Notations (PON) principles (Moody, 2008).

10.3 Future Work

Several areas for further research are as follows:

- To extend the critic capabilities by allowing the critics to check the tool's meta-model elements i.e. meta-critic. At present our critic specification approach only manages critic specification for a DSVL tool which resulted from a defined meta-model element. Critics that were specified are used to check any potential problems of a model/diagram for that modelling tool. Thus, we can expand the critic capabilities by offering critics when defining meta-model elements. The idea is to construct critics that are able to check potential problems at the meta-model level. This will allow critics to be specified for two stages, i.e. critics for the meta-model level and critics for the model/diagram level.
- To consider including other elements from the taxonomy group. One limitation in our research is that we have not incorporated as many of the elements from the taxonomy as is desirable. For instance, in future, we can expand our critic specification approach to add a graphical style where

appropriate to deliver critiques instead of just textual messages. Similarly, we can consider adding positive and negative critics in the critic specification tool as another way to provide critics to tool's users. However, all these have to be examined carefully in terms of their relevance to incorporate in the critic specification tool.

- To improve and provide a better critic authoring template by considering a visual representation for each of the item/properties in the template. At present our critic specification approach applies a textual and visual approach in specifying critics and feedback. In future, we can potentially replace more elements of the textual approach for the template with visual notational representations. This will allow new templates to be specified in a more visual manner, with actions realised using Marama's other visual specification tools.

10.4 Summary

This research arose from the need to have a critic specification approach for domain-specific visual languages and to provide accessibility for end-user tool developers to specify critics in an effective and easy way. A combination of a visual notational representation and a template-based approach were developed for the critic specification approach and demonstrated via three case studies of different domains for DSVL exemplar tools. A formal end-user evaluation was employed to evaluate and proof the concept of a critic specification approach. Thus we can say that critic specification and implementation for domain specific visual languages can be made accessible to end-user tool developers. In addition, the combination of a notational representation and a critic authoring template-based approach is another useful approach to support end-user tool developers in the critic specification task.

Appendix A

Participation Information Sheet

(Head of Department)



Department of Computer Science
Level 3, Science Centre
Building 303
38 Princes St
The University of Auckland
Private Bag 92019
Auckland

Tel: 09 373 7599

PARTICIPANT INFORMATION SHEET (HEAD OF DEPARTMENT)

Title: Evaluation of Template-based Critic Authoring for Domain-Specific Visual Language Tools

My name is Norhayati Mohd Ali and I am a PhD student at the Department of Computer Science, The University of Auckland. I am conducting research on visual design critic authoring template-based approach that supports end-users or tool designers in the construction of critics for domain-specific visual language (DSVL) tools. This research is under the supervision of Professor John Hosking and Professor John Grundy. Our research investigates the 'Visual design critic authoring template-based approach' as an alternative approach for constructing critics in an efficient and simple way. A prototype of visual design critic authoring tool, called *Marama Critic Definer* has been developed. Part of our research involves an evaluation of this prototype regarding its usability and effectiveness for specifying and constructing critics for DSVL tools.

As a Computer Science Head of Department, we would like to ask your permission to allow us to have access to students who enrolled in COMPSCI 732 course and SOFTENG 450 course and permit the students to participate voluntarily in our survey. Participation in this survey is on a voluntary basis and there will be no financial compensation. The survey is performed in an anonymous way. No personal information will be collected during the survey. We would like you to provide us the assurance that neither the students' grades nor academic relationships with the department staff members will be affected by either refusal or agreement in students' participation. Your support would be greatly appreciated.

This research is funded by the Ministry of Higher Education, Malaysia. If you have any queries regarding this survey, please do not hesitate to contact me. You can email me at: nmoh044@aucklanduni.ac.nz. Alternatively, you may phone me at 0210 -2421890. You may also contact my supervisor, Professor John Hosking at john@cs.auckland.ac.nz or 09 373 7599 ext 88297.

For any queries regarding ethical concerns you may contact the Chair, The University of Auckland Human Participants Ethics Committee, The University of Auckland, Office of the Vice Chancellor, Private Bag 92019, Auckland 1142. Telephone 09 373-7599 extn. 83711.

APPROVED BY THE UNIVERSITY OF AUCKLAND HUMAN PARTICIPANTS ETHICS COMMITTEE ON 02 December 2009 for (3) years, Reference Number 2009/492

Appendix B

Participation Information Sheet

(Student)



Department of Computer Science
Level 3, Science Centre
Building 303
38 Princes St
The University of Auckland
Private Bag 92019
Auckland

Tel: 09 373 7599

PARTICIPANT INFORMATION SHEET (STUDENT)

Title: Evaluation of Template-based Critic Authoring for Domain-Specific Visual Language Tools

My name is Norhayati Mohd Ali and I am a PhD student at the Department of Computer Science, The University of Auckland. I am conducting research on visual design critic authoring template-based approach that supports end-users or tool designers in the construction of critics for domain-specific visual language (DSVL) tools. This research is under the supervision of Professor John Hosking and Professor John Grundy. Our research investigates the 'Visual design critic authoring template-based approach' as an alternative approach for constructing critics in an efficient and simple way. A prototype of visual design critic authoring tool, called *Marama Critic Definer* has been developed. Part of our research involves an evaluation of this prototype regarding its usability and effectiveness for specifying and constructing critics for DSVL tools.

You are invited to participate in this survey as you are either postgraduate student who enrolled COMPSCI 732 course or 4th year undergraduate student who enrolled SOFTENG 450 course. Your comments and assistance would be greatly appreciated.

Participation in this survey is on a voluntary basis and there will be no financial compensation. The survey is performed in an anonymous way. No personal information will be collected during the survey. You can be assured that neither your grades nor academic relationships with the department staff members will be affected by either refusal or agreement to participate. This assurance is given by the Computer Science Head of Department. You can withdraw yourself from the survey at any time. Completing the required tasks in the survey and submitting the evaluation is an indication of consent but as the evaluation is anonymous, no answers can be withdrawn once the evaluation is submitted.

If you consent to participate in this survey, the participation involves one visit to the Computer Science Undergraduate Laboratory, approximately 1 hour. You will be given an explanation together with a demonstration of what need to be done. A task list and questionnaire sheet will be given to you before you start using the prototype tool. You will be asked to perform a number of tasks on the prototype tool and once you completed the task, you will be asked to answer the questionnaire sheet given to you. You also will be observed to allow the researcher to learn whether the tool is easy and efficient to use and also to know more about the usefulness and acceptance of the tool. You will be observed based on the following aspects: a) how you manage to complete the task given to you; b) how you define critics for a tool developed in Marama; c) how you navigate different parts of the tool; and d) your verbal responses while using the tool. The observations will take place only while you perform the tasks on the prototype tool. There will be note-taking while you perform the tasks and also while you are responding or commenting when using the prototype tool. However, no personal information will be collected in this observation process. Audio-tape, video-tape and any other electronic means such as Digital Voice Recorders are not used in this survey.

After completing the tasks you will be asked to answer the questionnaire sheet. Once you completed the questionnaire, you need to put in the box that will be placed in the lab. There will be no coding to your questionnaire as it is treated anonymously. The observation and questionnaires data will be compiled and analysed, and the results will be used for a PhD thesis and for other academic publications. Results also will be available to participants on request. The observation and questionnaires data will be stored for SIX (6) years for the purpose of peer review and further research. When the observation and questionnaires data is no longer needed, it will be destroyed using the paper shredder.

This research is funded by the Ministry of Higher Education, Malaysia. If you have any queries regarding this survey, please do not hesitate to contact me. You can email me at: nmoh044@aucklanduni.ac.nz. Alternatively, you may phone me at 0210 -2421890. You may also contact my supervisor, Professor John Hosking at john@cs.auckland.ac.nz or 09 373 7599 ext 88297, or the Head of Department, Associate Professor Robert Amor at trebor@cs.auckland.ac.nz or 09 373 7599 ext 83068, or you can write to us at:

Department of Computer Science,
The University of Auckland
Private Bag 92019
Auckland.

For any queries regarding ethical concerns you may contact the Chair, The University of Auckland Human Participants Ethics Committee, The University of Auckland, Office of the Vice Chancellor, Private Bag 92019, Auckland 1142. Telephone 09 373-7599 extn. 83711.

APPROVED BY THE UNIVERSITY OF AUCKLAND HUMAN PARTICIPANTS ETHICS COMMITTEE ON 02 December 2009 for (3) years, Reference Number 2009/492.

Appendix C

Consent Form (Head of Department)



Department of Computer Science
Level 3, Science Centre
Building 303
38 Princes St
The University of Auckland
Private Bag 92019
Auckland

Tel: 09 373 7599

CONSENT FORM (HEAD OF DEPARTMENT)

This Consent Form will be held for a period of six (6) years.

Title: *Evaluation of Template-based Critic Authoring for Domain-Specific Visual Language Tools.*

Researcher: Norhayati Mohd.Ali

I have read and understood the Participant Information Sheet. I understand the nature of the research and why I have been asked for permission and assurance of this research. I have had the opportunity to ask questions and have them answered. I agree to support the survey.

- I agree to allow the researcher to have access to the students who enrolled in COMPSCI 732 course and SOFTENG 450 course.
- I agree to permit the students to participate voluntarily in the survey.
- I understand there will be no payment to the student who participates in the survey.
- I understand that all of the data collected from the survey will be non-identifying.
- I agree to provide the assurance that neither grades nor academic relationship with any departmental staff members will be affected by either refusal or agreement to students' participation in the survey.

Name: _____

Signature & Date: _____

APPROVED BY THE UNIVERSITY OF AUCKLAND HUMAN PARTICIPANTS ETHICS COMMITTEE ON 02 December 2009 for 3 years, Reference Number 2009/492.

Appendix D

Consent Form (Student)



Department of Computer Science
Level 3, Science Centre
Building 303
38 Princes St
The University of Auckland
Private Bag 92019
Auckland

Tel: 09 373 7599

CONSENT FORM (STUDENT)

This Consent Form will be held for a period of six (6) years.

Title: *Evaluation of Template-based Critic Authoring for Domain-Specific Visual Language Tools.*

Researcher: Norhayati Mohd.Ali

I have read and understood the Participant Information Sheet. I understand the nature of the research and why I have been selected to participate in this research. I have had the opportunity to ask questions and have them answered. I understand that I can withdraw at any time but that data already recorded cannot be withdrawn. I agree to take part in the survey.

- I understand that I will not be paid for the time taken to participate in this survey.
- I understand that all of the data collected from the survey will be non-identifying.
- I understand that I will be observed while doing a task on the prototype tool if I agree to participate in this survey. No audio-tape, video-tape or any other electronic means such as Digital Voice Recorders is used in this survey.
- I understand that I will need to fill up a questionnaire at the end of the task if I agree to participate in this survey.
- I understand that only the researcher and her main supervisor will have access to the questionnaire and observation data.
- I understand that the observation and questionnaire data may be used to review the research outcomes both to improve the notation and software tool and in publications about the survey.
- I understand that data will be archived or stored for six years and then destroyed.

- I understand that the Computer Science Head of Department have provides assurance that neither my grades nor academic relationship with any department staff members will be affected by either refusal or agreement to participate.
- I understand that at the conclusion of the survey, a summary of the results will be available from the researcher upon request.

Name: _____

Signature & Date: _____

APPROVED BY THE UNIVERSITY OF AUCKLAND HUMAN PARTICIPANTS ETHICS
COMMITTEE ON 02 December 2009 for 3 years, Reference Number 2009/492.

Appendix E

Survey:Evaluation of Template-based Critic Authoring for Domain-Specific Visual Language Tools



Department of Computer Science
Level 3, Science Centre
Building 303
38 Princes St
The University of Auckland
Private Bag 92019
Auckland
Tel: 09 373 7599

Survey: Evaluation of Template-based Critic Authoring for Domain-Specific Visual Language Tools

Note: *The survey is structured into TWO parts. Part one, provides the task list that need to be done by you. Observation data will be collected while you are performing the tasks. Part two, provides a questionnaire that should be answered by you once you have completed the tasks.*

Statement

<input type="checkbox"/>	I have read the Participant Information Sheet and have understood the nature of the survey and I agree to take part in this survey. (please tick ✓)
--------------------------	---

PART ONE: Task List and Observation

Purpose: To allow the participant to develop a Marama-based tool using the Marama metatools. After the tool development, the participant needs to add several critics to the tool. Please take note, that participant will be observed on how he/she use the tool. Participant can ask question while doing the task. Observation data will be collected during participant doing his/her task.

Instruction: Please read and perform the following task steps.

Task 1. Explore the Marama tool that was given to you.

1. Metamodel for that tool is on the *Marama Metamodel Definer* views.
2. Shapes and connectors for that tool are on the *Marama Shape Designer* views.
3. The mapping of meta-elements to visual representations is on the *Marama Viewtype Definer* views.

Task 2. Identify critics for the tool.

1. Think and list several critic statements that are relevant to the given Marama-tool.
2. Identify and list an appropriate feedback (fix action) for each of the critic.
3. Use the following table to list your critic and feedback.

Critic	Feedback

Task 3. Add Critics to the tool using the critic authoring templates.

1. Design Marama critic type, by specifying the tool critics via the *Marama Critic Definer* views. Refer to Figure 1.

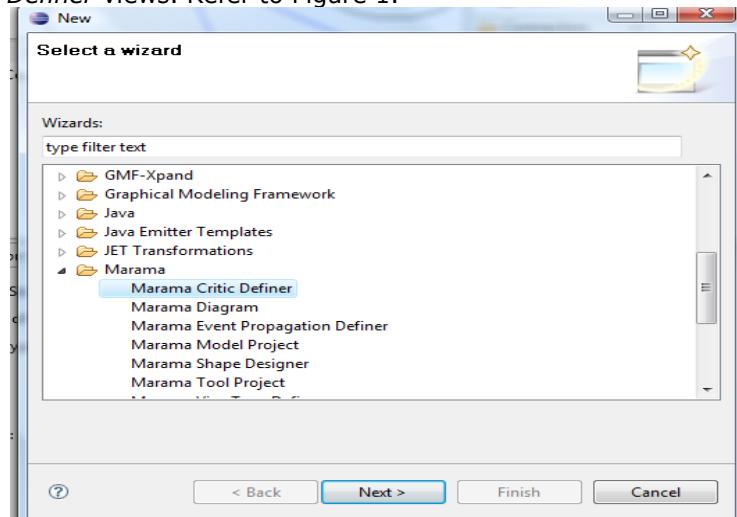


Figure 1: Marama Critic Definer

2. Define a critic for the tool by selecting the *CriticShape* icon. Associate with this *CriticShape* is a form-based interface, called *Critic Construction View*. Refer to Figure 2. To open this view, select **Window->Show View-> Other->Marama Editor->Critic Construction View**.

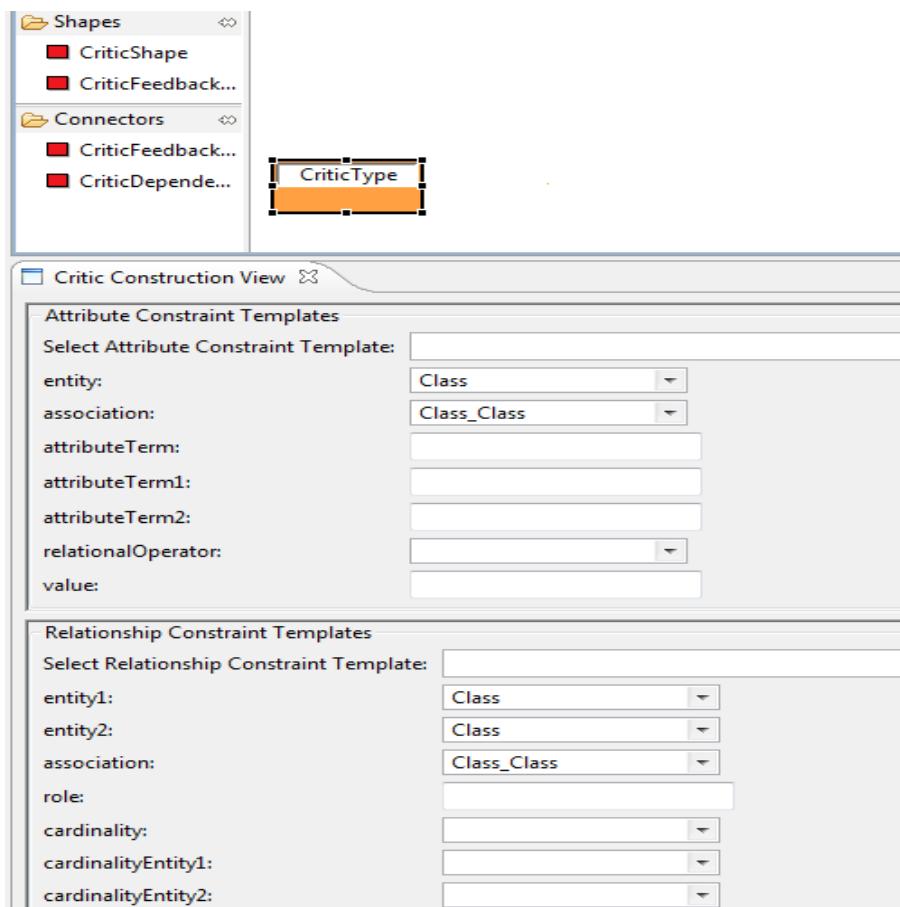


Figure 2: *CriticShape* with Critic Construction View interface.

3. To define critic, you can select from the list of available critic authoring templates- a) Attribute Constraint Template, b) Relationship Constraint Template, c) Action Assertion Template. After define the critic select 'Save Critic' button.
4. If the critic that you want to construct is not supported by the available critic templates, you can select the "Critic Template Editor" button to allow you to construct new critic template. Then click 'OK' and get back to *Critic Construction View* to define the critic.
5. Define the feedback (fix action) for the critics defined by selecting the *CriticFeedbackShape* icon. Associate with this *CriticFeedbackShape*, is a form-based interface, called *CriticFeedback View*. Refer to Figure 3. To open this view, **Window->Show View-> Other->Marama Editor->Critic Feedback View**.

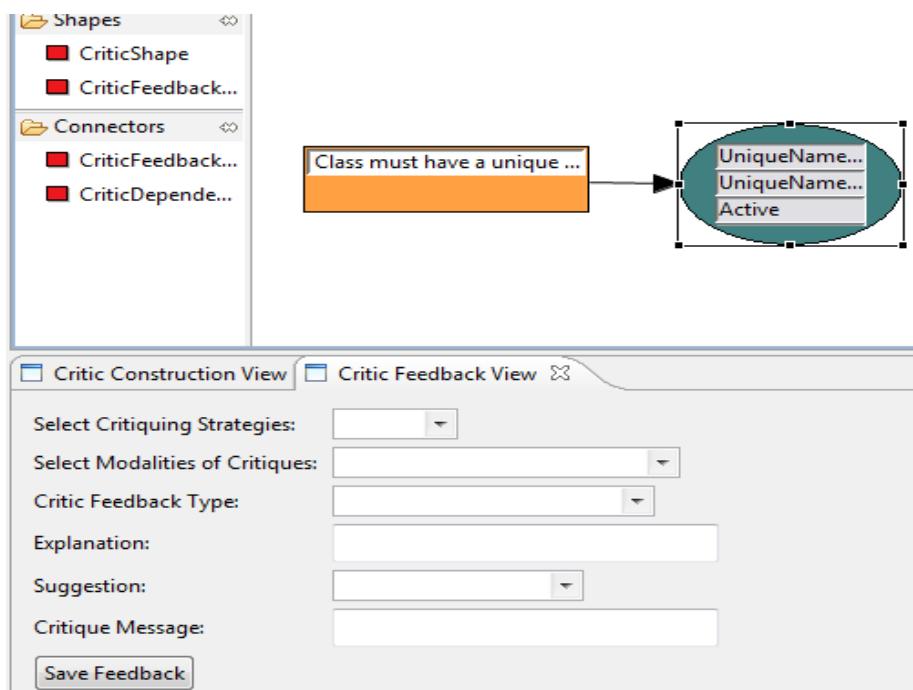


Figure 3: *CriticFeedbackShape* with Critic Feedback View interface.

6. Once you satisfied with the critics and feedbacks that you defined, then you can save it.

Task 4. Run Critics

1. Create a Marama Model Project for your tool
2. Create a Marama diagram.
3. Try to violate the critic rules to see whether critic and feedback is displayed at the Marama diagram.
4. End of task in specifying critics and feedbacks via critic authoring templates.

Task 5. Critic via formula function

1. Try to construct the same critic using the Object Constraint Language (OCL) via the formula icon.
2. Save the formula.
3. Run the critic the same way you did in Task 3(1-2-3). However, you need to open the Problem view to see the critics' violation.

End of Task.

After you complete the above task, please answer the questionnaires in PART TWO.

PART TWO: Questionnaire

Instruction:

Please answer the following questions.

Section (1)- Background Information.

1. How do you rate yourself in using Marama metatools? (tick one box)

	Proficient/skilled
	Intermediate
	Novice

2. Have you used similar tools like Marama metatools? If so, please name them.
-

3. Have you developed a software tool where you add design critics for that tool? If so, please name the tool and critic types.
-

4. Name the tool that was given to you using the Marama metatools.
-

Section (2)- Prototype Tool Information.

Please rate your agreement with the following statements about how you feel in general when using **Marama Critic Definer view** (a new specification tool that represents the visual design critic authoring template approach). Just circle or tick out the level of agreement that applies using the following scale:

1:Strongly Disagree (SD) 2:Disagree (D) 3:Undecided (U) 4: Agree (A) 5:Strongly Agree (SA)

A. Usefulness:

It is useful.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

It helps me be more effective.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

It makes the things I want to accomplish easier to get done.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

B. Ease of Use:

It is easy to use.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

It is user friendly.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

I don't notice any inconsistencies as I use it.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

C. Ease of Learning:

I learned to use it quickly.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

I easily remember how to use it.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

It is easy to learn to use it.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

D. Satisfaction:

I am satisfied with it.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

I would recommend it to a friend.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

It is fun to use.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

E. Cognitive Dimensions of Critic-Authoring Task:

It is easy to see various parts of the tool.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

It is easy to make changes.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

The notation is succinct and not long-winded.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

Some things do require hard mental effort.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

It is easy to make errors or mistakes.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

The notation is closely related to the result.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

It is easy to tell what each part is for when reading the notation.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

The dependencies are visible.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

It is easy to stop and check my work so far.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

I can work in any order I like when working with the notation.

Strongly Disagree 1-----2-----3-----4-----5 Strongly Agree

- F.** After completing this questionnaire, can you think of obvious ways that the design of the template-based critic authoring tool could be improved? What are they?
-
-

Thank you for your time!

Please let us know if you have any queries about this questionnaire or the survey we are conducting. Questions or concerns can either be directed to the researcher, Norhayati (nmoh044@aucklanduni.ac.nz) or to the course lecturer, Professor John Hosking (john@cs.auckland.ac.nz), Dept. of Computer Science.

References

- Bar, M., & Neta, M. (2006). Humans Prefer Curved Visual Objects. *Psychological Science*, 17(8), 645-648.
- Bardohl, R. (2002). A visual environment for visual languages. *Science of Computer Programming*, 44, 181-203.
- Barton, B. F., & Barton, M. S. (1987). Simplicity in Visual Representation: A Semiotic Approach. *Journal of Business ad Technical Communication*, 1(9), 9-26.
- Bergenti, F., & Poggi, A. (2000). Improving UML Designs Using Automatic Design Pattern Detection. *International Conference on Software Engineering and Knowledge Engineering (SEKE 2000)*, 336--343.
- Bezivin, J., & Jouault, F. (2006). Using ATL for Checking Models. *Electronic Notes in Theoretical Computer Science*(152), 69-81.
- Blackwell, A. F., Britton, C., Cox, A., Green, T. R. G., Gurr, C., Kadoda, G., et al. (2001). Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. *4th International Conference on Cognitive Technology*, 325-541.
- Blackwell, A. F., & Green, T. R. G. (2000). A Cognitive Dimensions Questionnaire Optimised for Users. *12th Workshop of the Psychology of Programming Interest Group*, 137-154.
- Blecken, A., & Marx, W. (2010). Usability Evaluation of a Learning Management System. *The 43rd International Conference on System Sciences*, 1-9.
- Brown, A. L. (1988). Motivation to learn and understand: On taking charge of one's own learning. *Cognition and Instruction*, 5(4), 311-321.
- Catarci, T., Massari, A., & Santucci, G. (1991). Iconic and Diagrammatic Interfaces: An Integrated Approach. *IEEE Workshop on Visual Languages*, 199-204.
- Coelho, W., & Murphy, G. (2007). ClassCompass: A Software Design Mentoring System. *ACM Journal on Educational Resource in Computing*, 7(1), 1-18.
- Costagliola, G., Lucia, A. D., Ferrucci, F., Gravino, C., & Scanniello, G. (2008). Assessing the usability of a visual tool for the definition of e-learning processes. [Journal]. *Journal of Visual Languages and Computing* 19, 721-737.

- Cox, K. (2000). Cognitive Dimensions of Use Cases- feedback from a student questionnare. *Proceedings of Twelfth Annual Meeting of the Psychology of Programming Interest Group (PPIG-12)*, 99-121.
- Czarnecki, K., & Helson, S. (2003). Classification of Model Transformation Approaches. *OOPSLA '03 Workshop on Generative Techniques in the Context of Model_Driven Architecture*, 1-17.
- Dashofy, E. M., Hoek, A. v. d., & Taylor, R. N. (2002). Towards Architecture-Based Self-Healing Systems. *Proceedings of the First Workshop on Self-healing Systems*, 21-26.
- de Souza, C. R. B., Jr., J. S. F., & Goncalves, K. M. (2000). A Group Critic System for Object-Oriented Analysis and Design. *Fifteenth IEEE International Conference on Automated Software Engineering*, 313 - 316
- de Souza, C. R. B., Oliveira, H. L. R., da Rocha, C. R. P., Goncalves, K. M., & Redmiles, D. F. (2003). Using Critiquing Systems for Inconsistency Detection in Software Engineering Models. *International Conference on Software Engineering and Knowledge Engineering (SEKE 2003)*, 196-203.
- Ebert, J., Suttenbach, R., & Uhe, I. (1997). Meta-CASE in Practice: A Case for KOGGE. *Lecture Notes in Computer Science*, 1250/1997, 203-216.
- Fischer, G. (1987). A Critic For LISP. *10th International Joint Conference on Artificial Intelligence*, 177-184.
- Fischer, G. (1989). Human-Computer Interaction Software: Lessons Learned, Challenges Ahead. *IEEE Software*, 6, 44-52.
- Fischer, G., Lemke, A. C., & Mastaglio, T. (1991). Critics: An Emerging Approach to Knowledge-Based Human Computer Interaction. *International Journal of Man-Machine Studies*, 35, 695-721.
- Fischer, G., Lemke, A. C., Mastaglio, T., & Morch, A. I. (1991). The Role of Critiquing in Cooperative Problem Solving. *ACM Transactions on Information Systems*, 9(3), 123-151.
- Fischer, G., & Mastaglio, T. (1990). A conceptual framework for knowledge-based critic systems. *Decision Support Systems*, 7, 355-378.
- Fischer, G., Nakakoji, K., Ostwald, J., Stahl, G., & Sumner, T. (1993). Embedding critics in design environments. *The Knowledge Engineering Review*, 8(4), 285-307.
- Florijn, G. (2002). *RevJava-Design critiques and architectural conformance checking for Java Software*: Software Engineering Research Centre.

- Gable, G. G. (1994). Integrating Case Study and Survey Research Methods: An Example in Information Systems. *European Journal of Information Systems*, 3(2), 112-126.
- Gena, C., & Weibelzahl, S. (2007). Usability Engineering for the Adaptive Web. In P. Brusilovsky, A. kobsa & W. Nejdl (Eds.), *The Adaptive Web, LNCS* (Vol. 4321, pp. 720-762): Springer-Verlag Berlin Heidelberg.
- Gertner, A. S., & Webber, B. L. (1998). TraumaTIQ: Online Decision Support for Trauma Management. *IEEE Intelligent Systems*, 32-39.
- Ginige, A., Lowe, D. B., & Robertson, J. (1995). Hypermedia Authoring. *IEEE Multimedia*, 2(4), 24-35.
- Gray, J., Bapty, T., & Neema, S. (2000). Aspectifying Constraints in Model-Integrated Computing. *Proceedings of OOPSLA*.
- Green, T. R. G., & Blackwell, A. F. (1998). Cognitive Dimensions of Information Artefacts: a tutorial. from <http://www.ndirect.co.uk/~thomas.gree/workStuff/Papers>
- Green, T. R. G., Blandford, A. E., Church, L., Roast, C. R., & Clarke, S. (2006). Cognitive dimensions: Achievements, new directions, and open questions. *Journal of Visual Languages and Computing*, 17, 328-365.
- Green, T. R. G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2), 131-174.
- Grundy, J., & Hosking, J. (2003). SoftArch: Tool Support for Integrated Software Architecture Development. *International Journal of Software Engineering and Knowledge Engineering*, 13(2), 125-151.
- Grundy, J., Hosking, J., Huh, J., & Li, K. N.-L. (2008). Marama: an Eclipse Meta-toolset for Generating Multi-view Environments. *International Conference on Software Engineering*, 819-822.
- Grundy, J., Hosking, J., Zhu, N., & Liu, N. (2006). Generating Domain-Specific Visual Language Editors from High-Level Tool Specifications. *21st IEEE International Conference on Automated Software Engineering*, 25-36.
- Guimaraes, R. L., Neto, C. d. S. S., & Soares, L. F. G. (2008). A Visual Approach for Modeling Spatiotemporal Relations. *DocEng 2008*, 285-288.
- Gurr, C., & Tourlas, K. (2000). Towards the Principled Design of Software Engineering Diagrams. *International Conference on Software Engineering* 509-518.

- Hagglund, S. (1993). Introducing expert critiquing systems. *The Knowledge Engineering Review*, 8(4), 281-284.
- Hartson, H. R., Andre, T. S., & Williges, R. C. (2003). Criteria for Evaluating Usability Evaluation Methods. *International Journal of Human-Computer Interaction*, 15(1), 145-181.
- Hill, J. H., Gokhale, A., & Schmidt, D. C. (2010). Template Patterns for Improving Configurability and Scalability of Enterprise Distributed Real-time and Embedded System Testing and Experimentation. 1-19. Retrieved from <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.150.7040&rep=rep1&type=pdf>.
- Holzinger, A. (2005). Usability Engineering Methods for Software Developers. *Communications Of The ACM*, 48(1), 71-74.
- Hwang, W., & Salvendy, G. (2010). Number of People Required for Usability. *Communications Of The ACM*, 53(5), 130-133.
- Irandoost, H. (2006). *Critiquing systems for decision support* (Technical Report No. DRDC Valcartier TR 2003-321): Defence Research and Development Canada.
- Jacko, J. A., & Sears, A. (2003). *The Human-Computer Interaction Handbook*. New Jersey: Lawrence Erlbaum Associates, Inc.
- Jaramillo, J. d. L., Vangheluwe, H., & Moreno, M. A. (2003). Using Meta-Modelling and Graph Grammars to Create Modelling Environments. In P. Bottani & M. Minas (Eds.), *Electronic Notes in Theoretical Computer Science* (Vol. 72, pp. 36-50): Elsevier Science B.V.
- Karsai, G., Nordstrom, G., Ledeczi, A., & Sztipanovits, J. (2000). Specifying Graphical Modeling Systems Using Constraint-based Metamodels. *IEEE International Symposium on Computer-Aided Control System Design*, 89-94.
- Kelly, S., Lyytinen, K., & Rossi, M. (1996). MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment *Advanced Information Systems Engineering* (Vol. Volume 1080/1996, pp. 1-21): Springer Berlin/Heidelberg.
- Khambati, A. (2008). *A model driven care plan modelling system*. Unpublished Master, University of Auckland.
- Khambati, A., Grundy, J., Warren, J., & Hosking, J. (2008). Model-driven Development of Mobile Personal Health Care Applications. *The 23rd IEEE/ACM International Conference on Automated Software Engineering*, 467-470.

- Khan, M. A., Israr, N., & Hassan, S. (2010). Usability Evaluation of Web Office Applications. *First International Conference on Intelligent Systems, Modelling and Simulation*, 146-151.
- Kleppe, A., & Warmer, J. (2002). The Semantics of the OCL Action Clause *Object Modeling with the OCL, Lecture Notes in Computer Science* (Vol. 2263, pp. 213-227): Springer-Verlag Berlin Heidelberg.
- Knauss, E., Luebke, D., & Meyer, S. (2009). Feedback-Driven Requirements Engineering: The Heuristic Requirements Assistant. *IEEE 31st International Conference on Software Engineering*, 587 - 590.
- Lemke, A. C., & Fischer, G. (1990). A Cooperative Problem Solving System for User Interface Design. *Eight National Conference on Artificial Intelligence*, 479-484.
- Leventhal, L. M., & Barnes, J. A. (2008). *Usability Engineering: Process, Products, and Examples*. Upper Saddle River, New Jersey: Pearson Prentice Hall.
- Li, L. (2010). *An Integrated Visual Approach for Business Process Modelling*. Unpublished PhD, University of Auckland, Auckland.
- Li, L., Hosking, J., & Grundy, J. (2007a, 12-16 June). *EML: A Tree Overlay-Based Visual Language For Business Process Modelling*. Paper presented at the Proceeding of ICEIS 2007, Funchal, Madeira, Portugal.
- Li, L., Hosking, J., & Grundy, J. (2007b). Visual Modelling of Complex Business Process with Trees, Overlays and Distortion-based Displays. *IEEE Symposium on Visual Languages and Human-Centric Computing*, 137-144.
- Liu, H., Rowles, C. D., & Wen, W. X. (1995). Critics for Knowledge-Based Design Systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(5), 740-750.
- Liu, N., Hosking, J., & Grundy, J. (2007). MaramaTatau: Extending a Domain Specific Visual Language Meta Tool with a Declarative Constraint Mechanism. *IEEE Symposium on Visual Languages and Human-Centric Computing* 95-103.
- Lohse, G. L., Biolsi, K., Walker, N., & Reuter, H. H. (1994). A Classification of Visual Representations. *Communications Of The ACM*, 37(12), 36-49.
- Lohse, G. L., Min, D., & Olson, J. R. (1995). Cognitive Evaluation of System Representation Diagrams. *Information & Management*, 29, 79-94.
- Lohse, J., Reuter, H., Biolsi, K., & Walker, N. (1990). Classifying Visual Knowledge Representations: A Foundation for Visualization Research. *Visualization '90*, 131-138.

Loucopoulos, P., & Kadir, W. M. N. W. (2008). BROOD: Business Rules-driven Object Oriented Design. *Journal of Database Management*, 19(1), 41-73.

Lund, A. (1998). USE Questionnaire Resource Page. from <http://usesurvey.com/IntroductionToUse.html>

Mackinlay, J. (1986). Automating the Design of Graphical Presentations of Relational Information. *ACM Transactions on Graphics*, 5(2), 110-141.

Maiden, N. A. M., & Sutcliffe, A. G. (1994). Requirements Critiquing Using Domain Abstractions. *First International Conference on Requirements Engineering*, 184-193.

Marama meta-tools. (2008). from <https://wiki.auckland.ac.nz/display/csidst/Marama+Meta-tools>

Masri, K., Parker, D., & Gemino, A. (2008). Using Iconic Graphics in Entity Relationship Diagrams: The Impact on Understanding. *Journal of Database Management*, 19(3), 22-41.

McCarthy, K., et al. . (2005). Experiments in Dynamic Critiquing. *International Conference on Intelligent User Interfaces (IUI '05)*, 175-182.

McCarthy, K., Salamo, M., Coyle, L., McGinty, L., Smyth, B., & Nixon, P. (2006). Group Recommender Systems: A Critiquing Based Approach. *International Conference on Intelligent User Interfaces*, 267-269.

McCormick, B. H., DeFanti, T. A., & Brown, M. D. (1987). Visualization in scientific computing-a synopsis. *IEEE Computer Graphics And Applications*, 7(7), 61-70.

McGinty, L., & Smyth, B. (2003). Tweaking Critiquing. *Proceedings of the Workshop on Personalization and Web*.

Mehzer, T., Abdul-Malak, M. A., & Maarouf, B. (1998). Embedding critics in decision-making environments to reduce human errors. *Knowledge-Based Systems*, 11, 229-237.

Miller, P. (1986). *Expert Critiquing Systems: Practice-based Medical Consultation by Computer*. New York: Springer-Verlag.

Moody, D. L. (2002). Complexity Effects On End User Understanding Of Data Models: An Experimental Comparison Of Large Data Model Representation Methods. *Tenth European Conference on Information Systems*.

Moody, D. L. (2006). What Makes a Good Diagram? Improving the Cognitive Effectiveness of Diagrams in IS Development. *15th International Conference in Information Systems Development (ISD 2006)*, 481-492.

- Moody, D. L. (2008). The "Physics" of Notations: Towards a Scientific basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6), 756-779.
- Moody, D. L., Heymans, P., & Matulevicius, R. (2009). Improving the Effectiveness of Visual Representations in Requirements Engineering: An Evaluation of *i** Visual Syntax. *2009 17th IEEE International Requirements Engineering Conference*, 171-180.
- Nielson, J. (1993). *Usability Engineering*. London: Academic Press Limited.
- Oh, Y., Do, E. Y.-L., & Gross, M. D. (2004). Intelligent Critiquing of Design Sketches.
- Oh, Y., Gross, M. D., & Do, E. Y.-L. (2008). Computer-Aided Critiquing System. *Computer Aided Architectural Design and Research in Asia (AADRIA)*, 161-167.
- Oh, Y., Gross, M. D., Ishizaki, S., & Do, E. Y.-L. (2009). Constraint-based Design Critic for Flat-pack Furniture Design. *17th International Conference on Computers in Education*, 19-26.
- Paige, R. F., Ostroff, J. S., & Brooke, P. J. (2002). Checking the Consistency of Collaboration and Class Diagrams using PVS. *Proceedings of Fourth Workshop on Rigorous Object-Oriented Methods (ROOM4)*.
- Pereira, M. J. V., Mernik, M., Cruz, D. d., & Henriques, P. R. (2008). Program Comprehension for Domain-Specific Languages. *ComSIS*, 5(2).
- Perry, D. E., Sim, S. E., & Easterbrook, S. (2006). Case Studies for Software Engineers. *International Conference on Software Engineering 2006*, 1045-1046.
- Petre, M. (1995). Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications Of The ACM*, 38(6), 33-44.
- Pisan, Y., Richards, D., Sloane, A., Koncek, H., & Mitchell, S. (2003). Submit! A Web-Based System for Automatic Program Critiquing. *Fifth Australasian Computing Education Conference (ACE2003)*, 20, 59-68.
- Pohjonen, R. (2005). Metamodeling Made Easy-MetaEdit+ (Tool Demonstration). In R. Gluck & M. Lowry (Eds.), *Lecture Note in Computer Science* (Vol. 3676, pp. 442-446): Springer-Verlag Berlin Heidelberg.
- Qattous, H. (2009). Constraint Specification by Example in a Meta-CASE Tool. *ESEC/FSE Doctoral symposium 2009*, 13-16.

- Qiu, L., & Riesbeck, C. (2008). An Incremental Model for Developing Educational Critiquing Systems: Experiences with the Java Critiquer. *Journal of Interactive Learning Research*, 19(1), 119-145.
- Qiu, L., & Riesbeck, C. K. (2003). Facilitating Critiquing in Education: The Design and Implementation of the Java Critiquer. *International Conference on Computers in Education (ICCE)*.
- Qiu, L., & Riesbeck, C. K. (2004). An Incremental Model for Developing Educational Critiquing Systems: Experiences with the Java Critiquer. *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications*, 908-916.
- Redmiles, D. F. (1998). *Applying design critics to software requirements engineering*.
- Reilly, J., McCarthy, K., McGinty, L., & Smyth, B. (2005). Incremental Critiquing. *Knowledge-Based Systems*, 18, 143-151.
- Robbins, J. E. (1998). *Design Critiquing Systems* (Technical Report). Irvine: Department of Information and Computer Science, University of California.
- Robbins, J. E., & Redmiles, D. F. (1998). Software architecture critics in the Argo design environment. *Knowledge-Based Systems*, 11(1), 47-60.
- Robbins, J. E., & Redmiles, D. F. (2000). Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. *Journal of Information and Software Technology*, 42(2), 79-89.
- Robey, D., Welke, R., & Turk, D. (2001). Traditional, Iterative, and Component-Based Development: A Social Analysis of Software Development Paradigms. *Information Technology and Management*, 2, 53-70.
- Rubin, J. (1994). *Handbook Of Usability Testing: How to Plan, Design and Conduct Effective Tests*. New York: John Wiley & Sons, Inc.
- Shanks, G. G., & Darke, P. (1998). Understanding Corporate Data Models. *Information & Management*, 35, 19-30.
- Siau, K. (2004). Informational and Computational Equivalence in Comparing Information Modelling Methods. *Journal of Database Management*, 15(1), 73-86.
- Silverman, B. G. (1992). Survey of Expert Critiquing Systems: Practical and Theoretical Frontiers. *Communications Of The ACM*, 35(4).
- Silverman, B. G., & Mehzer, T. M. (1992). Expert Critics in Engineering Design: Lessons Learned and Research Needs. *AI Magazine*, 13, 45-62.

- Sourrouille, J. L., & Caplat, G. (2002). Constraint Checking in UML Modeling. *The 14th International Conference on Software Engineering and Knowledge Engineering*, 217-224.
- Tianfield, H., & Wang, R. (2004). Critic System- Towards Human-Computer Collaborative Problem Solving. *Artificial Intelligence Review*, 22, 271-295.
- Tolvanen, J.-P. (2004). MetaEdit+: Domain-Specific Modeling for Full Code Generation Demonstrated [GPCE]. *OOPSLA '04*, 442-446.
- Tolvanen, J.-P., Pohjonen, R., & Kelly, S. (2007). Advanced Tooling for Domain-Specific Modeling:MetaEdit+. *The 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07)*, 243-250.
- Trochim, W. M. K. (1989). Outcome Pattern Matching and Program Theory. *Journal of Evaluation and Program Planning*, 12(4), 355-366.
- Tukiainen, M. (2001). Evaluation of the Cognitive Dimensions Questionnaire and Some Thoughts about the Cognitive Dimensions of Spreadsheet Calculation. *13th Workshop of the Psychology of Programming Interest Group*.
- Tyugu, E. (2007). *Algorithms and architectures of artificial intelligence, Frontiers in AI and Applications* (Vol. 159): IOS Press.
- Vahidov, R., & Elrod, R. (1999). Incorporating critique and argumentation in DSS. *Decision Support Systems*, 26, 249-258.
- Wang, Y. (2009). A Formal Syntax of Natural Languages and Deductive Grammar. *Fundamenta Informaticae*, 90(4), 353-368.
- Winn, W. D. (1993). An Account of How Readers Search for Information in Diagrams. *Contemporary Educational Psychology*, 18, 162-185.
- Xiyong, Z., & Xingwang, Z. (2006). Implementation of a Template-based Approach for Mass Customization of Service-oriented E-business Applications. *2006 International Conference on Systems, Man, and Cybernetics*, 4666-4670.
- Zhu, N., Grundy, J., Hosking, J., Liu, N., Cao, S., & Mehra, A. (2007). Pounamu; A meta-tool for exploratory domain-specific visual language tool development. *The Journal of Systems and Software*, 80(8), 1390-1407.