

**Visual Languages
for Event Integration Specification**

Karen Na-Liu Li

**A thesis submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy in Computer Science,
The University of Auckland, 2007.**

Abstract

The thesis demonstrates that visual approaches to modelling event-based systems are more effective than traditional textual scripts and code. We have both investigated existing approaches and developed new techniques for visual event-based system integration. We have used domain-specific visual languages with different high-level visual metaphors to specify event-handling support and provide backend processing tool support for both event integration specification and visualisation of event propagation.

We have developed three different visual metaphors for event-based system specification. The first, ViTABaL-WS, uses the Tool Abstraction (TA) metaphor to support specification of web services composition via higher level data and control flows and generation of BPEL4WS code. The second, Kaitiaki, uses an Event-Query-Filter-Action (EQFA) metaphor to allow visual primitives composition and java code generation for diagramming tool event handlers. The third, MaramaTatau, uses a spreadsheet-like metaphor to construct metamodel formulae visually to specify structural dependencies and constraints to be realised at runtime.

We have generalised from these three visual event-driven system metaphors to develop a new, generic visual event handling metaphor. From this we have built a novel multi-paradigm hybrid metamodelling environment for specifying generic event-based system behaviours.

Acknowledgements

I would like to express my endless thanks to my supervisors, Professor John Hosking and Professor John Grundy, for their enthusiasm in knowledge sharing and continuous support on both the thesis and my career development. During the many years of collaboration, I have learned from them a positive way to think, a diligent way to do, an innovative way to invent, and a generous way to give. I believe these are the factors that will drive me to a success in the near future.

My warm thanks to my parents for their love, high expectation, encouragement and support. It is now the time to reward them all my love and support.

My tearful thanks to my husband, Richard Lei Li, for his sweet love and care. His “intrusion” in my life (May 2005) made this research ever hardly progress in a year but traded surprisingly for a life ever since full of romance and joy. Many thanks to Richard for being always smart, proactive, hardworking, and most importantly, being with me in managing a wonderful life for all our family.

Accidentally once, I found these words on web, “Life itself cannot give you joy unless you really will it. Life just gives you time and space, it's up to you to fill it”, and the words have been surrounding me always to remind me what I want and how I do.

To Mum and Dad,

Thank you for giving me all the gift, encouragement and support. They make this achievement, and I am sure many future achievements, possible.

Contents

ABSTRACT	I
ACKNOWLEDGEMENTS	II
LIST OF FIGURES	VI
LIST OF TABLES.....	IX
CHAPTER 1 - INTRODUCTION	- 1 -
1.1 INTRODUCTION	- 1 -
1.1.1 <i>Events and Event-driven Systems in General</i>	- 1 -
1.1.2 <i>Textual vs. Visual Event Handling Specification</i>	- 3 -
1.1.3 <i>Goals of Research</i>	- 4 -
1.1.4 <i>Methodology</i>	- 5 -
1.2 CONTRIBUTION OF RESEARCH.....	- 8 -
1.3 THESIS ORGANISATION	- 9 -
1.4 SUMMARY	- 10 -
CHAPTER 2 - BACKGROUND AND RELATED RESEARCH	- 11 -
2.1 EVENT-BASED SYSTEMS	- 11 -
2.1.1 <i>Events in Software Architecture</i>	- 13 -
2.1.2 <i>Events in Graphics and Modelling Frameworks</i>	- 16 -
2.1.3 <i>Events in Database Systems</i>	- 18 -
2.1.4 <i>Events in Workflow Management Systems</i>	- 19 -
2.1.5 <i>Events in Distributed Computing</i>	- 20 -
2.1.6 <i>General Purpose Event Frameworks</i>	- 22 -
2.2 EVENT HANDLING SPECIFICATION AND VISUALISATION TECHNIQUES	- 24 -
2.2.1 <i>Custom Code Writing</i>	- 24 -
2.2.2 <i>Declarative Approaches</i>	- 25 -
2.2.3 <i>State-based Formalisms</i>	- 31 -
2.2.4 <i>Flow-based Approaches</i>	- 35 -
2.2.5 <i>Programming by Demonstration (Programming by Example)</i>	- 39 -
2.2.6 <i>Visual Approaches vs. Textual Approaches</i>	- 40 -
2.3 GENERAL ISSUES AND REQUIREMENTS	- 42 -
2.4 SUMMARY	- 44 -
CHAPTER 3 - MOTIVATION.....	- 45 -
3.1 OVERVIEW OF POUNAMU.....	- 45 -
3.1.1 <i>Tool Specification</i>	- 46 -
3.1.2 <i>Tool Usage</i>	- 51 -
3.2 EVALUATIONS OF POUNAMU	- 55 -
3.2.1 <i>Large Group Experiments</i>	- 55 -
3.2.2 <i>Developers of Large Applications</i>	- 57 -

3.2.3	<i>Usability of Substantial Tools Constructed Using Pounamu</i>	- 57 -
3.3	ANALYSIS OF EVALUATION RESULTS	- 58 -
3.4	SUMMARY	- 59 -
CHAPTER 4 - OVERVIEW OF OUR APPROACH		- 60 -
4.1	INTRODUCTION	- 60 -
4.2	VITABAL-WS	- 62 -
4.3	KAITIAKI	- 64 -
4.4	MARAMATATAU	- 65 -
4.5	GENERALISATION	- 66 -
4.6	SUMMARY	- 67 -
CHAPTER 5 - VISUAL WEB SERVICES COMPOSITION		- 68 -
5.1	INTRODUCTION	- 68 -
5.2	MOTIVATION	- 70 -
5.3	REQUIREMENTS	- 71 -
5.4	RELATED WORK	- 71 -
5.5	METAPHOR	- 74 -
5.6	NOTATION	- 75 -
5.7	LOAN APPROVAL EXAMPLE	- 78 -
5.8	DESIGN AND IMPLEMENTATION	- 86 -
5.9	DISCUSSION	- 87 -
5.9.1	<i>Evaluation</i>	- 87 -
5.9.2	<i>Strengths and Limitations</i>	- 89 -
5.10	SUMMARY	- 90 -
CHAPTER 6 - VISUAL GUI EDITING EVENT HANDLING		- 91 -
6.1	INTRODUCTION	- 91 -
6.2	MOTIVATION	- 92 -
6.3	REQUIREMENTS	- 95 -
6.4	METAPHOR	- 95 -
6.5	NOTATION	- 96 -
6.6	EXAMPLE OF KAITIAKI SPECIFICATIONS	- 100 -
6.7	DYNAMIC VISUALISATION OF EVENT HANDLERS	- 103 -
6.8	DESIGN AND IMPLEMENTATION	- 104 -
6.9	DISCUSSION AND EVALUATION	- 105 -
6.10	SUMMARY	- 107 -
CHAPTER 7 - VISUAL RELATIONAL FORMULA SPECIFICATION		- 109 -
7.1	INTRODUCTION	- 109 -
7.2	FROM POUNAMU TO MARAMA	- 110 -
7.3	BACKGROUND AND MOTIVATION	- 116 -
7.4	MARAMATATAU	- 118 -
7.5	CASE STUDY	- 123 -
7.6	IMPLEMENTATION	- 128 -
7.7	EVALUATION	- 129 -
7.8	SUMMARY	- 131 -
CHAPTER 8 - DESIGN OF THE GENERALISED EVENT HANDLING FRAMEWORK		- 132 -
8.1	MOTIVATION	- 132 -
8.2	BACKGROUND	- 134 -

8.3 REQUIREMENTS FOR GENERALISATION.....	- 137 -
8.4 GENERALISATION	- 138 -
8.4.1 <i>ViTABaL-WS Building Blocks.....</i>	- 138 -
8.4.2 <i>Kaitiaki Building Blocks</i>	- 141 -
8.4.3 <i>MaramaTatau Building Blocks</i>	- 143 -
8.4.4 <i>Generalised Marama Meta-tools.....</i>	- 145 -
8.4.5 <i>Program Visualisation.....</i>	- 158 -
8.4.6 <i>Framework Evolution</i>	- 161 -
8.5 SUMMARY	- 162 -

CHAPTER 9 - PROTOTYPE OF THE GENERALISED EVENT HANDLING FRAMEWORK..... - 163 -

9.1 INTRODUCTION.....	- 163 -
9.2 STRUCTURE SPECIFICATION	- 166 -
9.2.1 <i>Marama Tool Project.....</i>	- 166 -
9.2.2 <i>Marama Metamodel Definer.....</i>	- 167 -
9.2.3 <i>Marama Shape Designer</i>	- 170 -
9.2.4 <i>Marama View Type Definer.....</i>	- 174 -
9.2.5 <i>Marama Model Project and Marama Diagram</i>	- 176 -
9.3 BEHAVIOUR SPECIFICATION.....	- 178 -
9.3.1 <i>MaramaTatau Formulae.....</i>	- 178 -
9.3.2 <i>ViTABaL-WS Event Propagations</i>	- 188 -
9.3.3 <i>Kaitiaki Visual Event Handlers.....</i>	- 190 -
9.3.4 <i>Escaping to Code</i>	- 192 -
9.4 PROTOTYPE IMPLEMENTATION	- 195 -
9.5 SUMMARY	- 197 -

CHAPTER 10 - EVALUATION OF THE GENERALISED FRAMEWORK..... - 199 -

10.1 INTRODUCTION	- 199 -
10.2 EVALUATION TECHNIQUES	- 200 -
10.2.1 <i>Cognitive Dimensions</i>	- 201 -
10.2.2 <i>Evaluation against the Requirements.....</i>	- 204 -
10.2.3 <i>Large End User Survey</i>	- 205 -
10.2.4 <i>Small Experienced User Group Study</i>	- 209 -
10.3 FURTHER CONTINUOUS EVALUATION PLAN	- 211 -
10.4 SUMMARY	- 211 -

CHAPTER 11 - CONCLUSIONS AND FUTURE RESEARCH..... - 212 -

11.1 RESEARCH CONTRIBUTIONS AND CONCLUSIONS.....	- 212 -
11.2 FUTURE RESEARCH	- 214 -
11.3 SUMMARY	- 217 -

REFERENCES..... - 218 -

List of Figures

Figure 1.1. Simple event handler effects.....	2 -
Figure 1.2. Textual vs. visual event handler specifications.	4 -
Figure 2.1. Basic structure of event-based software systems. (Grundy et al, 1997)	11 -
Figure 2.2. The observer pattern. (Gamma et al, 1995)	12 -
Figure 2.3. Event flow across architecture stack (Hanson, 2005).....	13 -
Figure 2.4. Model-View-Controller abstraction (Sun, 2005).....	14 -
Figure 2.5. The GEF framework (Eclipse, GEF, 2007)	15 -
Figure 2.6. Maintaining attribute consistency between related components in CPRGs. (Grundy et al, 1996)	16 -
Figure 2.7. User interactive zoomable views (Liu et al, 2004)	17 -
Figure 2.8. Calling the notifyChanged() method. (Budinsky et al, 2003).....	18 -
Figure 2.9. WfMC Process Definition Meta-Model. (Workflow Management Coalition, 1999)	19 -
Figure 2.10. Jini™ Distributed Events Specification (Sun, 2005).....	20 -
Figure 2.11. Event Channel with Multiple Suppliers and Consumers (OMG, 2004)	21 -
Figure 2.12. (a) A Simple MDB Application (Sun, 2007), (b) MSMQ model. (MSDN, 2007)	22 -
Figure 2.13. The EBI framework: Relationships between components (left) and framework metamodel (right). (Barrett et al, 1996).....	23 -
Figure 2.14. (a) Pounamu event handler designer (Zhu et al, 2007); (b) Amulet constraint scripting (Myers, 1997)	25 -
Figure 2.15. Rule graph (Matskin and Montesi, 1998)	27 -
Figure 2.16. Reaction RuleML (Paschke et al, 2006)	27 -
Figure 2.17. A family tree (Hanna, 2002).....	29 -
Figure 2.18. The line stays attached to the box and circle even when they move. Below the graphic is the code to define the constraints on the line. (Myers, 1990)	29 -
Figure 2.19. Basic ALV architecture. (Hill, 1992)	30 -
Figure 2.20. Form chart notational elements (Weber, 2002)	30 -
Figure 2.21. Adding constraints. (Tolvanen, 2006)	31 -
Figure 2.22. Modeling a reactive system with Petri nets. (Palanque et al, 1993)	32 -
Figure 2.23. (a) State transition diagram. (b) State transition table. (Drumea and Popescu, 2004)	33 -
Figure 2.24. An event graph (Berndtsson and Mellin, 1999).....	33 -
Figure 2.25. A time graph (Berndtsson and Mellin, 1999)	34 -
Figure 2.26. BPEL4WS specification and deployment using BPWS4J,.....	36 -
Figure 2.27. Basic modelling capabilities of VEPL. (Grundy and Hosking, 1998).....	37 -
Figure 2.28. Multiple event processing, stage coordination and external process interfacing (Grundy and Hosking, 1998) - 38 -	38 -
Figure 2.29. Program method implementation. (Cox et al, 1989).....	39 -
Figure 2.30. The Alice Authoring Environment (opening scene tab). (A) The Add Object button presents a gallery of 3D objects. (B) The Object Tree, a PHIGS-like tree of hierarchical objects (C) Camera controls allow the user to drive around the scene. (D) The Undo button provides infinite animated undo. (E) The Alice Command Box for evaluating single lines of Alice script. (F) The Script tab reveals a simple text editor where the user writes scripts that control the objects in the scene. (Conway et al, 2000)	40 -
Figure 2.31. A portion of a Forms/3 form (spreadsheet) that defines a primitiveCircle. The primitiveCircle in cell newCircle is specified by the other cells, which define its characteristics. A user can view and specify formulas by clicking on the formula tabs attached to the bottom right of each cell. Radio buttons and popup menus are equivalent to cells with constant formulas. (Burnett et al, 2001)	42 -
Figure 3.1. The Pounamu approach. (Zhu et al, 2007)	46 -
Figure 3.2. Pounamu in use: (a) specification of a visual notation shape element and (b) modelling using this shape in a UML class diagram tool. (Zhu et al, 2007)	47 -
Figure 3.3. Example of the Connector designer. (Zhu et al, 2007)	48 -

Figure 3.4. Example of the metamodel designer. (Zhu et al, 2007)	- 49 -
Figure 3.5. Example of the view designer. (Zhu et al, 2007)	- 50 -
Figure 3.6. Example of the event handler designer. (Zhu et al, 2007)	- 51 -
Figure 3.7. Example modelling tool usage. (Zhu et al, 2007)	- 52 -
Figure 3.8. Examples of Pounamu DSVL tools. (Zhu et al, 2007)	- 54 -
Figure 3.9: Problems identified in 2004 and 2006 surveys.....	- 56 -
 Figure 4.1. ViTABaL-WS editing in Pounamu.	- 63 -
Figure 4.2 Kaitiaki storyboard.	- 65 -
Figure 4.3 MaramaTatau editing in Marama.	- 66 -
Figure 4.4. Event integration specification in Marama meta-tools.	- 67 -
 Figure 5.1: Conceptual model of the loan approval process.	- 70 -
Figure 5.2: Web service composition.	- 72 -
Figure 5.3: Various web service toolies and their interfaces involved in the loan approval process.	- 80 -
Figure 5.4: Composed Loan Approver web services.	- 82 -
Figure 5.5: Generating WSDL and BPEL4WS specifications from our ViTABaL-WS model.	- 84 -
Figure 5.6: Dynamic visualization of a ViTABaL-WS model.....	- 85 -
Figure 5.7: Design of our ViTABaL-WS tool.	- 86 -
 Figure 6.1. Example of a diagram-based design tool.....	- 93 -
Figure 6.2. Example of event handler textual specification.	- 94 -
Figure 6.3. The Kaitiaki EQFA metaphor.....	- 96 -
Figure 6.4. Example of addition of a new sub-page.	- 100 -
Figure 6.5. Specifying a layout constraint event handler.	- 101 -
Figure 6.6. An example of a reusable visual query.....	- 101 -
Figure 6.7. Visualising execution of a visual event handler.	- 103 -
Figure 6.8. Extensions to Pounamu (highlighted).	- 104 -
Figure 6.9. Compiling a visual event handler.	- 105 -
 Figure 7.1. The Marama approach to realising Eclipse-based visual language tools. (Grundy et al, 2006)	- 111 -
Figure 7.2. The architecture of Marama. (Grundy et al, 2006)	- 112 -
Figure 7.3. Implementation of Marama. (Grundy et al, 2006)	- 115 -
Figure 7.4. MaramaTatau visual notation.	- 119 -
Figure 7.5. (1) Model instantiation view and (2) model instance view.....	- 122 -
Figure 7.6. Formula debug view.	- 123 -
Figure 7.7. A MaramaMTE architecture view	- 124 -
Figure 7.8. Handler code implementing constraint.	- 125 -
Figure 7.9. MaramaMTE model behaviour specification.	- 126 -
Figure 7.10. Using formulae to constrain entities.	- 128 -
Figure 7.11. The initial architecture of Marama meta-tools. (adapted from Figure 7.2)	- 129 -
 Figure 8.1. A general purpose event handling framework.....	- 133 -
Figure 8.2. Relationship between patterns in the pattern language. (Roberts and Johnson, 1996)	- 135 -
Figure 8.3. Merging CPRGs organisation, ViTABaL event propagation and Serendipity event filtering/action. (Grundy et al, 1996).....	- 136 -
Figure 8.4. ViTABaL-WS event propagation definer in Marama meta-tools.....	- 140 -
Figure 8.5. Kaitiaki event handler specification (a) and its runtime execution effect (b).	- 142 -
Figure 8.6. The Marama Meta-tools approach. (Adapted from (Grundy et al, 2006)).....	- 145 -
Figure 8.7. Unified event handling in MaramaTatau and ViTABaL-WS using Kaitiaki.....	- 146 -
Figure 8.8. Event propagation definition in ViTABaL-WS and event handler definition in Kaitiaki.	- 147 -
Figure 8.9. Marama Meta-tools Event Handling Abstraction Framework.....	- 147 -
Figure 8.10. Marama EMF specification.	- 148 -
Figure 8.11. Common event handling model.....	- 153 -
Figure 8.12. MVC of Marama meta-tools.....	- 154 -
Figure 8.13. Metamodel definer in Marama meta-tools.....	- 155 -
Figure 8.14. Shape designer in Marama meta-tools.....	- 156 -
Figure 8.15. View type definer in Marama meta-tools.	- 156 -
Figure 8.16. MaramaTatau integration in View Type Definer.	- 158 -

Figure 8.17. Visual Debugger.....	- 159 -
Figure 8.18. Visual debugging MaramaTatau formulae.	- 160 -
Figure 8.19. Visual debugging a Kaitiaki event handler.....	- 161 -
Figure 9.1. Overview of tool creation with Marama meta-tools.....	- 164 -
Figure 9.2. Tool creation in Marama meta-tools.....	- 167 -
Figure 9.3. Association type specification in Marama meta-tools.....	- 168 -
Figure 9.4. Attribute specification in Marama meta-tools.	- 169 -
Figure 9.5. Sub-typing in Marama meta-tools.	- 169 -
Figure 9.6. Generation of Metamodel XML files in Marama meta-tools.	- 170 -
Figure 9.7. Shape and connector design with concrete viewers in Marama meta-tools.....	- 171 -
Figure 9.8. Exporting visual properties in Marama meta-tools.....	- 174 -
Figure 9.9. View Type Definer in Marama meta-tools.	- 175 -
Figure 9.10. A diagram (view) of a Marama model project.	- 177 -
Figure 9.11. The Model Instances view associated with a Marama model project.....	- 177 -
Figure 9.12. Runtime tool behaviour (i.e. derived value updates) enabled using Marama meta-tools.	- 179 -
Figure 9.13. MaramaTatau behaviour specification	- 181 -
Figure 9.14. Visual formula specification via clicks and highlights	- 184 -
Figure 9.15. Indication of erroneous formula compilation	- 185 -
Figure 9.16. Adding user functions.....	- 185 -
Figure 9.17. Defining view type formulas	- 187 -
Figure 9.18. A ViTABaL-WS event propagation view	- 189 -
Figure 9.19. Kaitiaki event handler specification.	- 191 -
Figure 9.20. The custom code writing approach in Marama to define event flows.	- 193 -
Figure 9.21. Marama handler hierarchy.....	- 193 -
Figure 9.22. Registering a handler to the metamodel or a view type in Marama meta-tools.....	- 195 -
Figure 9.23. The component structure of Marama meta-tools.	- 196 -
Figure 10.1. Minimal task completion.....	- 207 -
Figure 10.2. Problems identified in the survey.	- 208 -
Figure 10.3. Proposed improvements/extensions in the survey.	- 209 -
Figure 11.1. Outline of using MaramaTorua. (Huh et al, 2007)	- 216 -

List of Tables

Table 2.1. Comparison of event handling specification and visualisation techniques	- 43 -
Table 5.1: ViTABaL-WS notation overview.....	- 78 -
Table 6.1. Kaitiaki language key visual constructs.....	- 97 -
Table 6.2. Overview of Kaitiaki reusable building blocks.....	- 99 -
Table 8.1. Building blocks defined for ViTABaL-WS	- 139 -
Table 8.2. Building blocks defined for Kaitiaki.....	- 141 -
Table 8.3. Building blocks defined for MaramaTatau.	- 144 -
Table 10.1. The evaluation methods adopted by the Marama meta-tools.....	- 201 -

Chapter 1 - Introduction

This chapter discusses the core research, the exploration of three different visual event handling metaphors of event-based software development and their generalisation into a generic event handling framework. The goals of this research are described, followed by our methodology towards the research. The main contributions are then summarised. We outline the thesis organisation at the end of this chapter, with a brief description of contents for each chapter.

1.1 Introduction

The event-driven paradigm is widely used in a range of application domains due to its flexibility for constructing dynamic system interactions. This thesis initially focuses on the specification of event handling behaviours in three complex types of system: web services and business process composition systems, graphical user interface (GUI) systems, and constraint-based metamodeling systems. We subsequently propose an integrated visual approach that is generalised from the three explored exemplar approaches to specify event handling behaviours.

1.1.1 Events and Event-driven Systems in General

Events are notifications of state-changes or actions/commands. Typically the event model contains events, event generators (event source/notifier), event dispatchers, event consumers (event observer/listeners/receivers) and event handlers (reacting programs, often sharing the event listener role). An event object may contain almost every concern of an event-based application including the event name, event type, event generator, affected objects and other application-specific information. Subscription by the event consumers to the event generator is a vehicle to trigger event-based communications (MSDN, 2007).

Event-driven systems are ubiquitous in many application domains. They present loosely-coupled system behaviour (Grundy et al, 1997). Both the OMG (OMG, 2004) and Sun (Sun, 2005) strongly advocate the notion of event-driven systems. Example event-driven systems include GUI design systems, distributed systems, database systems and workflow management systems. Such systems all

incorporate events of interest, conditions (“filters”) on whether to respond to the event and action(s) to run that may modify the system state. Example approaches for specifying event-handling include scripting, Event-Condition-Action (ECA) rules, and spreadsheets (single direction constraints).

As an example of an event based system, consider a diagram-based design tool for project management, an example use of which is illustrated in Figure 1.1. This consists of a work breakdown structure view (rear) and a Pert chart view (front). We have built this tool with the Pounamu meta-tool along with many other diagram-based design tools (Zhu et al, 2007). Such applications allow end users to model complex design problems using visual notations appropriate to the domain.

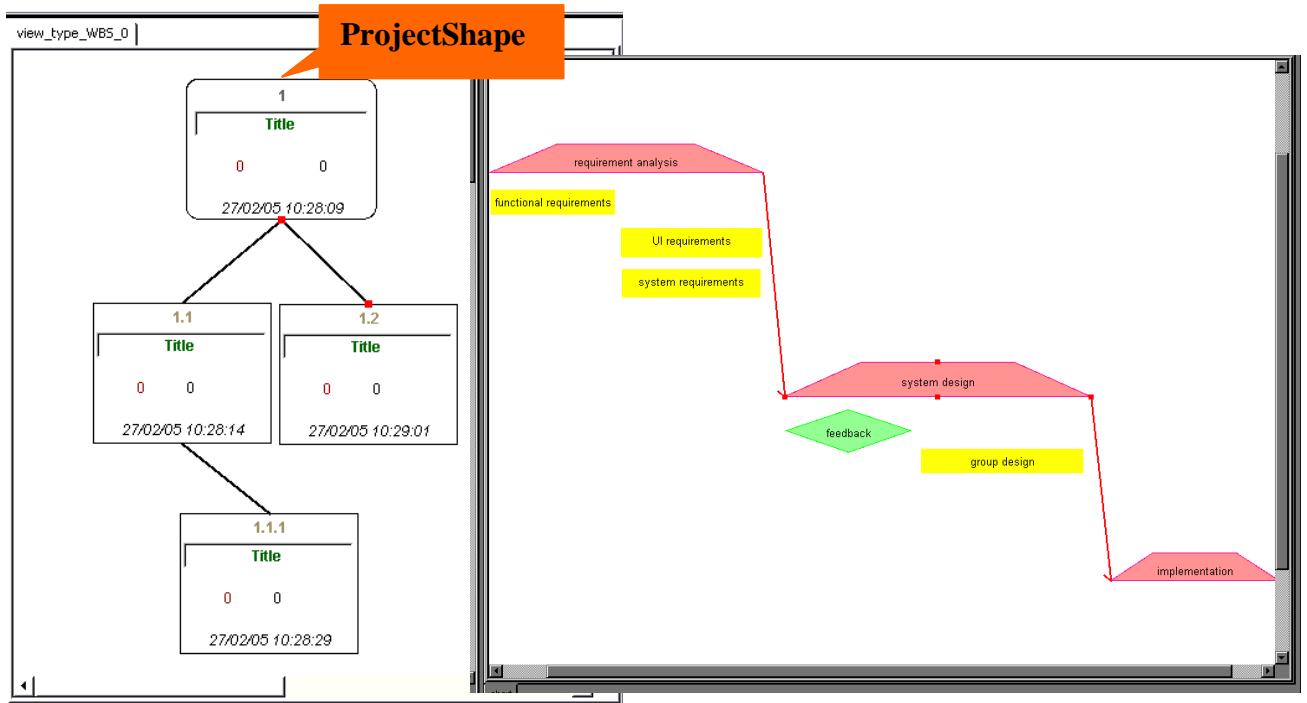


Figure 1.1. Simple event handler effects.

In Pounamu (Zhu et al, 2007) event handlers are typically used to add model/view level constraints, complex data mappings, back end data export or import e.g. code generation, and access to remote services to support tool integration and extension. Each handler specifies:

- the event type(s) that causes it to be triggered, e.g. shape/connector addition/modification, information model element change, or user action;
- any event filtering condition that needs to be fulfilled e.g. property value of shape or entity; and
- the response to make to that event, i.e. action to take, as a set of state changing operations.

In the above example, when creating a task icon for the work breakdown structure diagram, several values for its properties need to be set by the event handler. These are gathered from a range of sources. When the user adds a new ProjectShape, the event handler is fired by the generated NewShapeEvent. Information is queried and added to the new shape. This results in the project shape being given default values for attributes such as “ProjectID”, “Title”, “Duration”, “TotalCost” and “CreationDate” when it is created. Updates to the new shape are reflected in the Pounamu modelling view. Event handlers can also be used to provide automatic layout of shapes for the diagrams. The event handlers are defined to respond to a built-in Pounamu NewConnectorEvent. When a user adds a specific connector from a parent shape to a sub-shape, an event fires and the event handler locates the parent shape with all its sub-shapes, and then aligns the sub-shapes.

In general software systems, communication between components is typically achieved by procedure calls. Event-driven systems differ in that event propagations are organised between an event source and all its inter-related event sinks. When an event is generated from the event source, the event sinks are notified of the incoming events, and corresponding actions are taken based on the underlying event handling specification.

1.1.2 Textual vs. Visual Event Handling Specification

Once an event happens, handling behaviours can be triggered such that the event can be published, filtered, transformed, logged, and/or processed. The design and construction of event-driven systems can be very difficult due to complex event propagation and reaction behaviours. The currently dominant custom code writing approach requires end-users to master a programming language and API of the application domain, which is non-suitable for non-programmer end users. An appropriate high-level visual specification language and tool support should contribute to making the process less stressful (Liu et al, 2005; Zhu et al, 2007). In this thesis we explore existing event handler specification approaches and then develop new techniques for visual event-based system integration. We have used a variety of domain-specific visual languages with different high-level visual metaphors (including Tool Abstraction, Event-Query-Filter-Action and Spreadsheet) to specify event handling support and provide backend processing tool support for event integration specification and visualisation of event propagation.

Figure 1.2 illustrates the first evidence of advantages of visual event handling (b, c, and d) on top of textual scripting (a), as graphic notations are used to make event handling specifications easier to understand. Figure 1.2 (a) shows the textual scripting approach used in Pounamu to define an event

handler. Figure 1.2 (b) shows an event handler specification by composing visual primitives. Figure 1.2 (c) shows visual specification of event propagations among structural and behavioural components of a system. Figure 1.2 (d) shows the specification and visualisation of model-level dependencies via unidirectional constraints. We summarise our approaches to event handling specification in later sections.

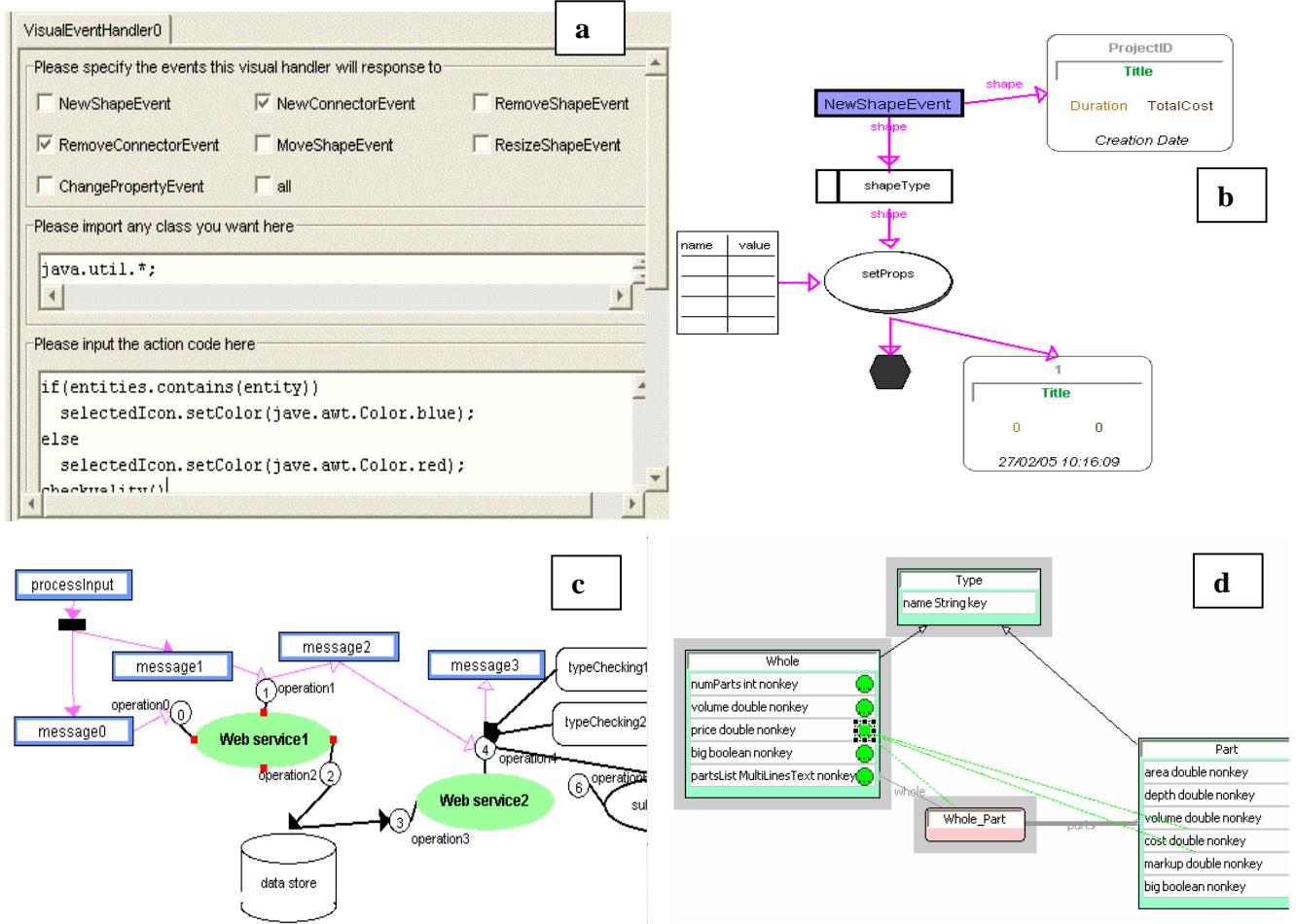


Figure 1.2. Textual vs. visual event handler specifications.

1.1.3 Goals of Research

Visual approaches, compared to custom code writing, have shown their advantages in minimising design and implementation effort and improving understandability of programs (Green and Petre, 1996; Cox et al, 1997; Grundy and Hosking, 1998; Burnett et al, 2001; Grundy et al, 2006). This suggests that a visual language that supports event integration specification is likely to be a positive approach for the design and construction of a complex event-based system. Visualisation support (tool support) for the event propagations in the running system is also necessary in order to allow users to track and control the system execution behaviour (Grundy et al, 1995; Grundy et al, 1997; Jin, 2003). Using different high-level visual metaphors for event-handling support and providing

backend processing tool support for event integration specification were our main objectives. Based on in-depth research on current event handling techniques including custom scripting, constraint-based programming models and metamodel tool event handling metaphors, our initial goal was to develop three exemplar domain-specific visual languages to examine event handler specification issues in different domains. The subsequent goal was to generalise from them to a visual metaphor and an environment for specifying general event handling integration. In achieving this our aim was that the general visual metaphor should be able to adapt the event-based communication model to a wide range of application domains, and also support complex and interactive system design and implementation.

1.1.4 Methodology

Our approach to achieving our goals was based around the following methodological steps, repeated for each metaphor:

- We began with literature review of event systems in general and compared major visual event handling techniques. The requirements for event handling specifications were identified, together with the discovery of advantages and weaknesses of the current event handling techniques.
- We then focused on problems and issues in existing event-based approaches and tool support and selected a choice of metaphor as our target of research focus.
- We designed event handling support for the selected metaphor addressing an analysed set of requirements on the problem domain.
- We proved our concept for that metaphor by developing prototype systems and examples.
- We undertook evaluation of our visual language and environment to gauge its effectiveness.

Having examined several metaphors, our next step has been to develop a high-level abstraction of these metaphors suitable for a range of application domains. We have initially developed three exemplars using such an iterative approach; they are ViTABaL-WS (Liu et al, 2005), Kaitiaki (Liu et al, 2005), and MaramaTatau (Liu et al, 2007). We have then generalised the three exemplars into a metamodeling environment to support event handling integration. Each of the three exemplars and the generalised event handling framework is briefly summarised in the following subsections.

1.1.4.1 Visual Web Services Composition

One example event-driven problem domain is web services composition. Web services have become a popular technology for building distributed systems, but there is a lack of languages and tools to

specify web service compositions at high abstraction levels, and from that generate lower-level executable process code such as BPEL4WS (IBM, 2003) and visualise, at high abstraction levels, running web services. Most approaches provide basic flow-like BPEL4WS editors or similar (Srivastava and Koehler, 2003). More abstract approaches (Fensel and Bussler, 2002; Foster et al, 2003; Tang et al, 2004; Jung and Cho, 2005; Liu et al, 2007) only support limited compositional approaches or do not support generation of BPEL4WS or similar executable forms. We have developed a new approach for complex web service composition using a high-level metaphor and visual language, called ViTABaL-WS (Liu et al, 2005), which uses a “Tool Abstraction” (TA) metaphor for describing relationships between service definitions, and multiple-views of data-flow, control-flow and event propagation in a modelled process. This supports higher level design views for service composition (as shown in Figure 1.2 (b)) that are complementary to current web services composition standards. ViTABaL-WS also supports visualisation of running processes to support architecture understanding and visual debugging of specified protocols.

1.1.4.2 Visual GUI Editing Event Handling

The second problem domain that we have focused on is visual design tools. These tools have many applications, including software design, engineering product design, e-learning, and data visualisation. Pounamu (Zhu et al, 2007) is a meta-tool we have developed for building such visual design tools. It incorporates high-level visual specifications of tool metamodels and visual language notations allowing non-programmer users to modify aspects of their tools such as appearance of icons and view compositions. However, users of visual design tools commonly wish to modify tool behaviour (Morch, 1998; Peltonen, 2000) to specify editing constraints, automated diagram and model modification, semantic constraints, computation and user interaction alerts.

Most visual design tools are “event driven”, meaning that when a user modifies a diagram in the tool, events are generated and can be acted upon to modify other content, enforce constraints, etc. (Grundy et al, 1995; Grundy et al, 1996; Zhu et al, 2007; Eclipse, 2007). We have used the event driven nature of such tools as a vehicle to provide end users with a domain specific visual language, we call Kaitiaki, with which to express both simple and complex event handling mechanisms for their diagramming tools via visual specifications (Liu et al, 2005). These include event filtering, tool state querying and action invocation as shown in Figure 1.2 (c). We have incorporated this visual language into the Pounamu meta-tool to provide end users who have little programming background, a mechanism to specify simple or complex actions to take for given event types.

1.1.4.3 Visual Relational Formula Specification

It is increasingly common to use meta-tools to specify and generate domain specific visual language tools (Kelly et al, 1996; Ferguson et al, 1999; Ledeczi et al, 2001; Eclipse GMF, 2007). A common problem for such meta-tools is specification of model level behaviours, such as constraints and dependencies. These often need to be specified using conventional code in the form of event handlers or the like. A well-known and one of the most popularly used end user programming tools nowadays is the spreadsheet (Burnett et al, 2001; Engels and Erwig, 2005), thanks to its natural tabular and one-way constraint metaphors. Formulae are designed in spreadsheets to allow declarative specification of system behaviours and automatic evaluation of them. Our third event specification metaphor attempts to adapt the spreadsheet approach to model behaviour specification.

We explored this new formula-based approach, we call MaramaTatau (Liu et al, 2007), within the context of a new visual metamodeling environment called Marama (Grundy et al, 2006), which was developed as a successor to Pounamu (Zhu et al, 2007). MaramaTatau was first developed as an external plug-in for Marama to facilitate specification of entity/association property setting and constraint checking at a model level and later integrated into a meta-toolset for Marama within Eclipse, providing a single integrated toolset for both specification and generation of Marama tools. Formula construction in MaramaTatau is similar to a spreadsheet but expressed at a type rather than instance level as shown in Figure 1.2 (d). Formulae are all interpreted as one way constraints realised at a model instance level. Error and to-do list critics provide notification to the user of constraint violations. Visualisations of formula effects are achieved via runtime visual debugging and master-details tabular model instances data views.

1.1.4.4 Generic Event Handling Specification

Based on the in-depth exploration of the three problem domains and the three corresponding visual languages, i.e. ViTABaL-WS, Kaitiaki and MaramaTatau, we have developed a generalised metaphor and a language/framework that can provide support for generic event integration specification. By abstracting from the three exemplars, a general metamodel representation that combines atomic primitives (either shared or non-shared) extended by the three visual languages is defined. This common model supports multiple metaphoric views in the style of the three exemplars and will support generation to a range of underlying implementation technologies for execution or interpretation (OCL (OMG, 2003), RuleML (RuleML Initiative, 2006), stylesheets etc.).

1.2 Contribution of Research

We have investigated thoroughly visual language metaphors suitable for specifying event handlers, addressed existing problems and applied examples to demonstrate the metaphors and prove concepts.

We have developed ViTABaL-WS, a hybrid visual programming environment for design and implementation of complex interactions and data exchanges among web service components. This exemplar tool is implemented using the Pounamu meta-tool. ViTABaL-WS uses the TA paradigm to express complex web service compositions. It provides code generation to BPEL4WS for deployment and execution from generated process models. An interactive visual debugger animates running service compositions in ViTABaL-WS by instrumenting debug service calls into the generated BPEL4WS. A conference paper titled “A Visual Language and Environment for Composing Web Services” was co-authored with Professor John Hosking and Professor John Grundy and presented in Proceedings of the 2005 ACM/IEEE International Conference on Automated Software Engineering.

We have developed Kaitiaki, a visual language and proof of concept support environment for specifying diagramming tool event handlers. This uses a metaphor of generating event, tool state queries, filters over query results and state changing actions, with dataflow between these building blocks. The support environment allows users to compose handlers from these constructs and relate them to concrete diagramming tool objects. A debugger uses the visual notation to step through a specification, animating constructs and affected diagram objects. We have added this tool to the Pounamu meta-diagramming tool and specified and generated event handlers for example tools, demonstrating the feasibility of the approach. A conference paper titled “A Visual Language and Environment for Specifying Design Tool Event Handling” was co-authored with Professor John Hosking and Professor John Grundy and presented in Proceedings of the 2005 IEEE International Conference on Visual Languages/Human-Centric Computing. Another improved conference paper titled “A Visual Language and Environment for Specifying User Interface Event Handling in Design Tools” was presented in Proceedings of the Eighth Australasian User Interface Conference in 2007.

We have developed MaramaTatau, an approach for constraint/dependency specification in a domain-specific visual language meta-tool. This borrows much from techniques used to support the spreadsheet metaphor, but in a situation with less concreteness. MaramaTatau augments the Marama meta-tools’ metamodel designer, allowing tool developers to specify formulae over metamodels,

combined with a one-way constraint system to compute values during tool usage. This allows for much simpler specification of dependency and constraint handling within Marama tools, compared to both the textual event handlers and Kaitiaki visual event handlers. A conference paper titled “MaramaTatau: Extending a Domain Specific Visual Language Meta Tool with a Declarative Constraint Mechanism” was co-authored with Professor John Hosking and Professor John Grundy and presented in Proceedings of the 2007 IEEE International Conference on Visual Languages/Human-Centric Computing.

We have generalised from the three visual event-driven system metaphors and developed a new, generic visual event handling metaphor. From this we have built a novel multi-paradigm hybrid metamodelling environment for specifying generic event-based system behaviour that allows escape from code. A conference paper titled “Visual Languages for Event Integration Specification” presented our research proposal in Proceedings of the 28th International Conference on Software Engineering, 2006.

Three other papers supporting this thesis research from motivation to implementation were co-authored, these include:

- A journal paper titled “A. Pounamu: a meta-tool for exploratory domain-specific visual language tool development”, which was published in Journal of Systems and Software, Elsevier, 2007.
- A conference paper titled “Generating Domain-Specific Visual Language Editors from High-level Tool Specifications”, which was in Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, 2006.
- A workshop paper titled “Performance Engineering of Service Compositions”, which was in Proceedings of the International Workshop on Service-Oriented Software Engineering, 2006.

1.3 Thesis Organisation

The following chapters are organized as:

Chapter 2 addresses the problems and examples in the field of event handling specification, undertakes literature reviews on related work including event-based systems, together with various event handling specification and visualisation techniques.

Chapter 3 presents an in-depth analysis of problems that motivate our research and describes our approach to addressing these problems.

Chapter 4 presents overviews of our approach towards creating a general purpose event handling framework.

Chapter 5 describes the tool abstraction metaphor and the ViTABaL-WS approach in addressing event-based service-oriented architecture specification and visualisation.

Chapter 6 describes the Event-Query-Filter-Action metaphor and the Kaitiaki approach in addressing visual GUI editing event handling specification and visualisation.

Chapter 7 describes the spreadsheet metaphor and the MaramaTatau approach for dependency and constraint driven event handling.

Chapter 8 discusses the design of the generalised event handling framework and the issues discovered during the generalisation.

Chapter 9 depicts the prototype and implementation of the generalised event handling framework.

Chapter 10 evaluates the generalised event handling framework to address both its effectiveness and tradeoffs.

Chapter 11 summarises our achievements and proposes future research directions.

1.4 Summary

This chapter discussed the core research, the exploration of three different visual event handling metaphors and their generalisation into a generic event handling framework. We presented our goals and the methodology taken towards this research. We present our detailed research contributions in the following chapters.

Chapter 2 - Background and Related Research

The event-based communication model is widely used in many application domains, including software architecture, graphics and modelling frameworks, workflow management systems, database systems, and distributed computing. Event handling specifications are typically written using textual forms, which are error-prone and difficult to visualise and debug. A visual form of event handling specifications together with tool support is in contrast favourable, but is still sometimes verbose and limited in power. This chapter summarises our literature review on event-based systems, focusing on their event handling specification and visualisation support. Issues in designing event-based systems are explored and common concerns and requirements of event handling specifications are identified, which include an expressive language that describes both the static structure and dynamic behaviours of event-based systems along with environment support for modelling and tracing event propagations. Hybrid visual and textual languages and environments for event-based system specifications are of particular interest for us, because they can leverage the advantages of both visual and textual approaches.

2.1 Event-based Systems

Event-based systems are different from general software systems based on procedural calls in that an event-based system features publish/subscribe relationships and dynamic event propagations among event sources and event sinks. Components are interconnected via events and operation calls, shown in Figure 2.1, to generate the, so-called, event-based communication. This facilitates loosely coupled systems with a collection of independent components to dynamically scale and extend (Grundy et al, 1997).

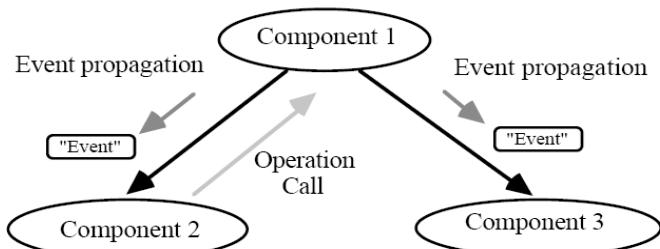


Figure 2.1. Basic structure of event-based software systems. (Grundy et al, 1997)

The publish/subscribe pattern, also known as observer pattern, specifies that one or more objects are registered to observe an event raised by the observed subject (Gamma et al, 1995). Figure 2.2 depicts the relationship between subjects and observers in the pattern, where a subject generally maintains a collection of observers. This pattern is mainly used in design and implementing event-based systems.

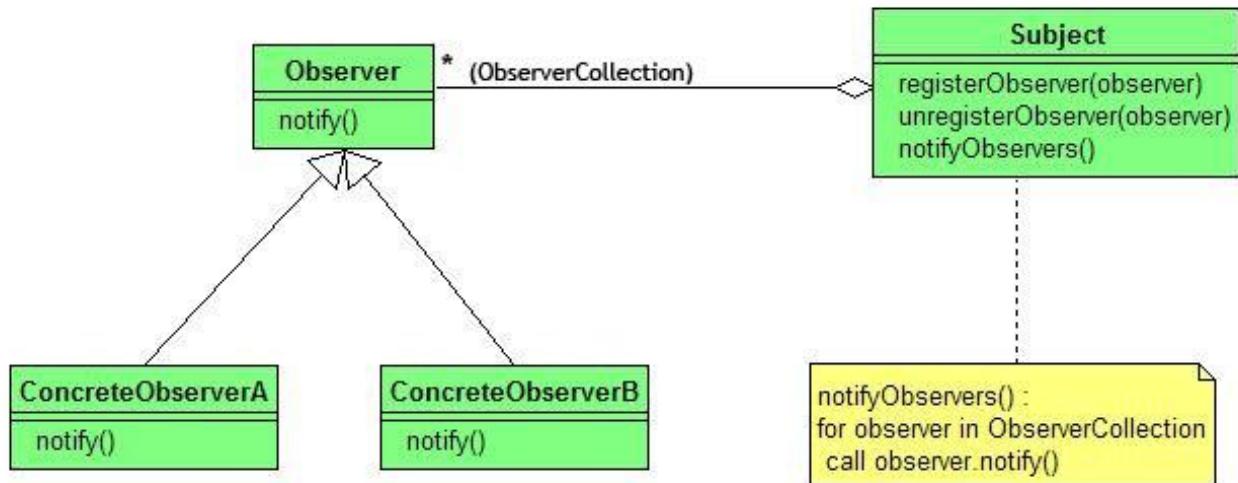


Figure 2.2. The observer pattern. (Gamma et al, 1995)

In event-based systems, an event is the occurrence of an observable activity (either generated implicitly by the system or explicitly by the user actions) or a state change in the system that may influence the execution of a user program. An event, represented as either an object or a message, is broadcast to other components that have indicated an interest in it. The act of selecting the set of recipients and posting the event to the recipients is known as event notification. Event notification can be broadly categorised as synchronous or asynchronous, with respect to the raiser of the event. If raising the event causes the signalling thread to block until it is explicitly resumed by a handler, it is termed a synchronous notification. If the thread raises the event but does not block, it is termed an asynchronous notification. Delivery of the notification eventually results in the execution of some code by its recipient (or an entity designated by the recipient), usually called the event handler (Menon et al, 1993).

Meier and Cahill (Meier and Cahill, 2002) identify a classification of major event-based communication properties on the event model and event service dimensions. In this section, we explore event characteristics and categorise event-based systems based on their major application domains, including software architecture, graphical and modelling frameworks, workflow management systems, database systems, and distributed computing. We explain the event usage and

handling in these problem domains, followed by an indication of the need for a general purpose event framework that can support the design and construction of a wide range of event-based systems.

2.1.1 Events in Software Architecture

An event-driven architecture (EDA) is a method for designing and using systems which propagate events among loosely coupled software components and services using the publish/subscribe pattern. A system of EDA typically consists of event generators, event consumers, and an intermediary event service manager. Figure 2.3 illustrates that an event can be defined as any change in a system, platform, component, business, or application process to be published, received, and responded; events can be high-level and business-oriented or low-level and technical in character. Building systems around an event-driven architecture allows these systems to be constructed in a manner that facilitates more responsiveness, since event-driven systems are more normalised to unpredictable and asynchronous environments (Hanson, 2005).

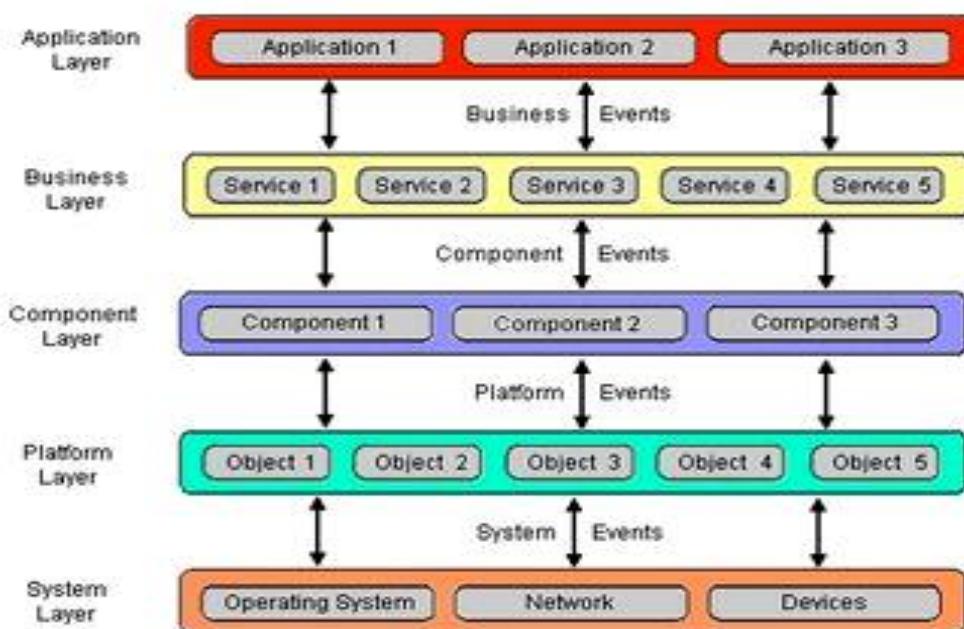


Figure 2.3. Event flow across architecture stack (Hanson, 2005).

EDA extends and complements Service-Oriented Architecture (SOA) (Sliwa, 2003) (Hanson, 2005) since services can be started by triggers such as an event. SOA focuses on business functions and EDA focuses on business events, with mutual unawareness of the loosely coupled event publisher and subscriber components (Hoof, 2006). Web service composition is a form of dynamic, component-based SOA where web services are “wired together” with messages passing from one to another.

The publish/subscribe pattern is used in the Model-View-Controller (MVC) architecture (Burbeck, 1992; Sun, 2005), as illustrated in Figure 2.4, where events cause a controller (i.e. event handler) to change a model (i.e. data), which triggers all dependent views (i.e. data display) to be automatically updated. Thus the consistency between the model and views are maintained.

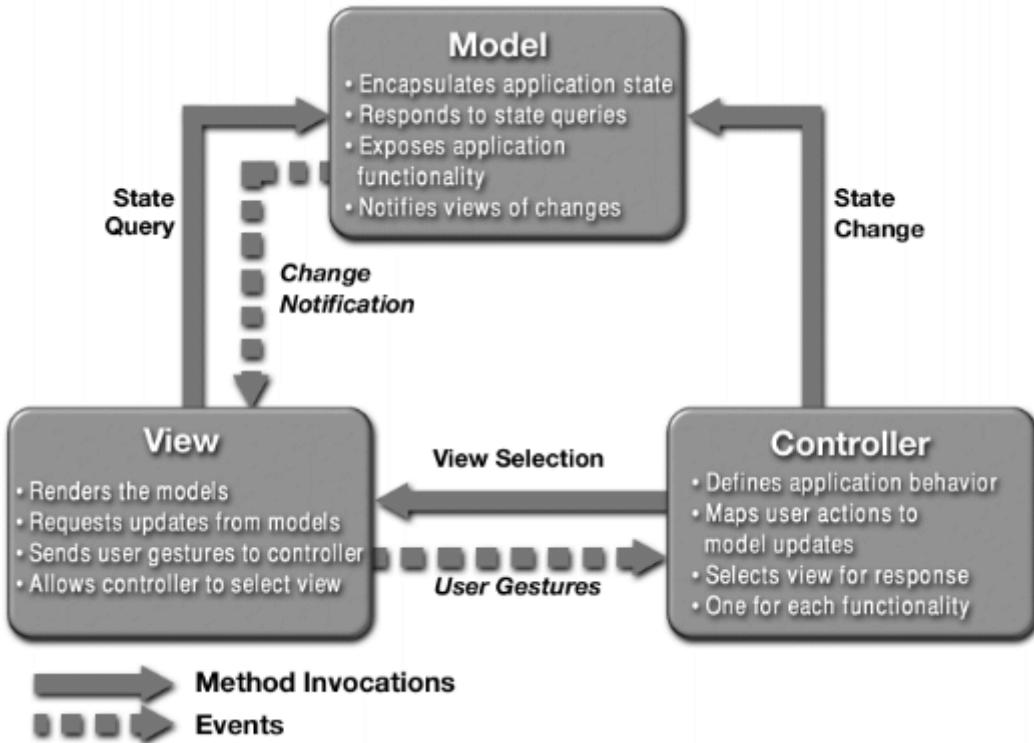


Figure 2.4. Model-View-Controller abstraction (Sun, 2005)

The Eclipse's Graphical Modelling Framework (GEF) (Eclipse, GEF, 2007) uses an MVC architecture to provide the link between an application's model and views. Requests and Commands are used in GEF in a similar way to event handlers to encapsulate interactions and their effects on the model. Figure 2.5 shows a high-level view of GEF. To enable graphical views to update according to a change in a model, an event object needs to be created in the model and fired to notify the change. The “EditParts” are the controllers which map models to view representations, listening to model events (via the `activate()` and `deactivate()` methods) and updating views accordingly (via `refreshChildren()` and `refreshVisuals()` methods) to reflect the changes in models.

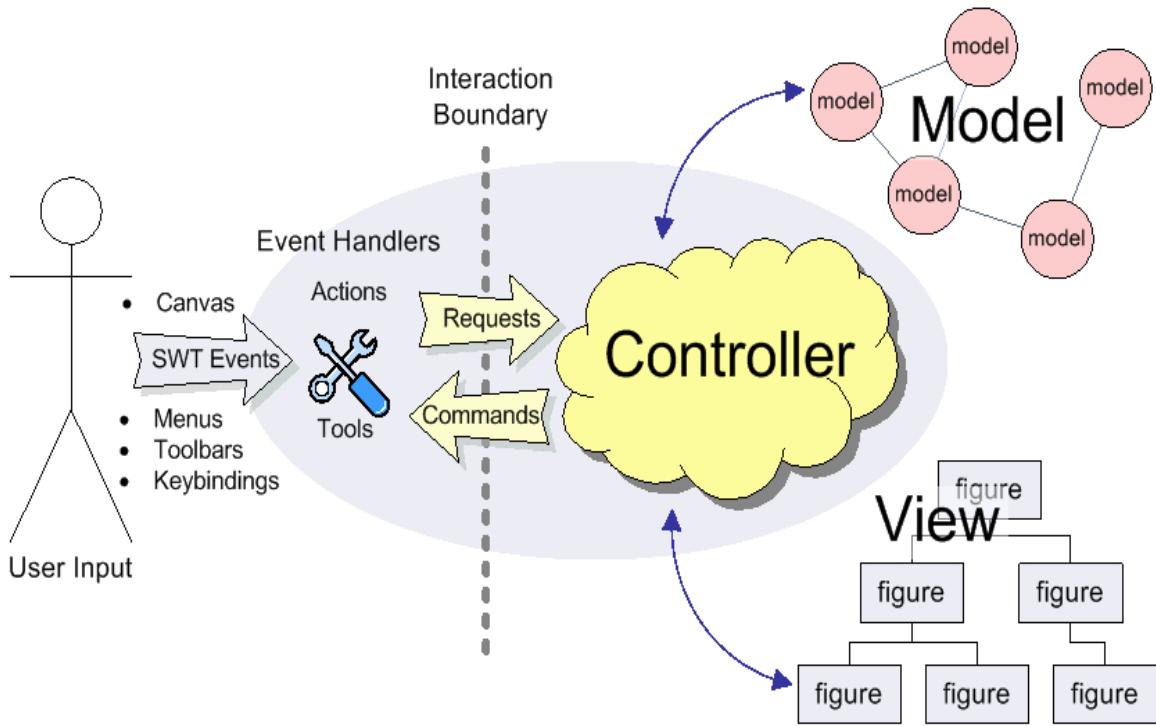


Figure 2.5. The GEF framework (Eclipse, GEF, 2007)

Multiple-view systems generally have the MVC architecture. The underlying communication and consistency management between different views of such a system is often enabled by using event-handling facilities. Examples of multiple-view systems include ViTABaL (Grundy et al, 1995), MViews (Grundy and Hosking, 1996), and C2 (Robbins et al, 1998) etc.

The change propagation and response graphs (CPRGs) (Grundy et al, 1996; Grundy et al, 1997) approach or variants is commonly used as the underlying implementation architecture of multiple view visual environments. CPRGs propagate changes, such as the notification of an icon, as change description (event) objects between component objects via relationship objects. Receiving objects interpret or store change descriptions appropriately to maintain consistency (Grundy et al, 1997). Figure 2.6 illustrates how attribute consistency is maintained between the related “dialog”, “edit field” and “caption” components in CPRGs. Change descriptions are generated and propagated to the “parts” and “caption-of” relationships, which then interpret the change descriptions and take corresponding actions to keep the referential integrity of all the inter-dependent components.

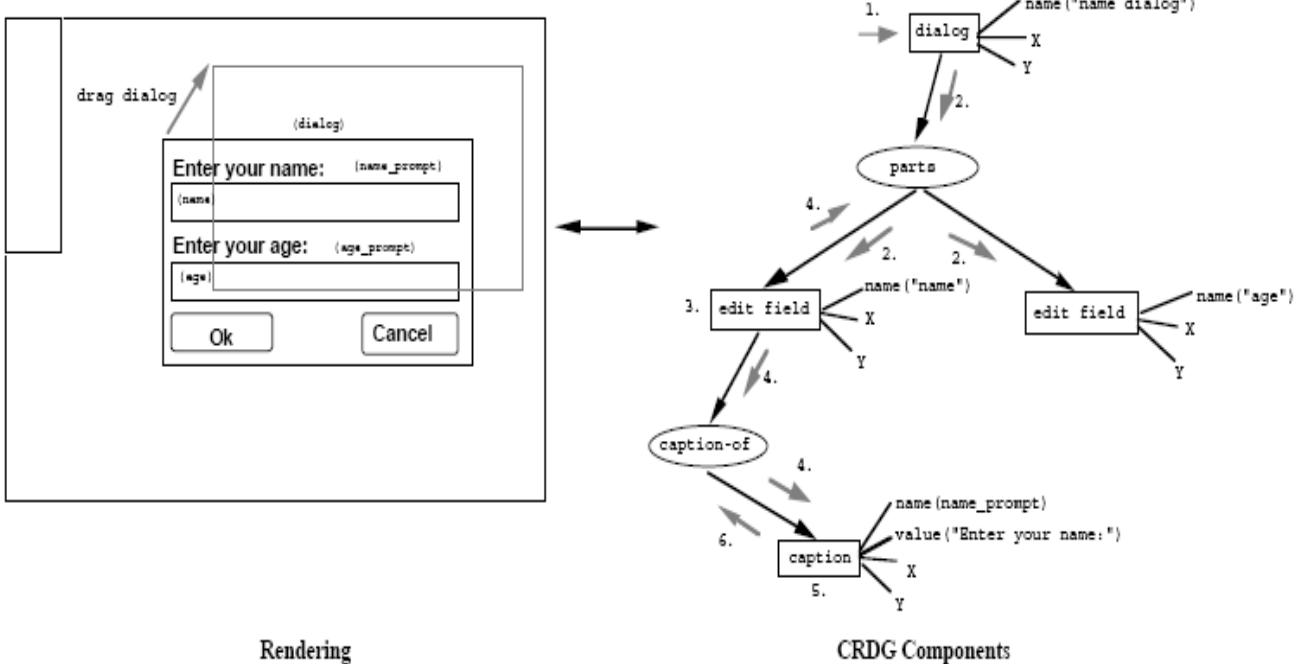


Figure 2.6. Maintaining attribute consistency between related components in CPRGs. (Grundy et al, 1996)

2.1.2 Events in Graphics and Modelling Frameworks

The delegate event model is commonly used in building graphical user interfaces (GUIs). This model is based on three entities: a GUI control, callbacks, and interfaces. The GUI control is the event source which fires events in response to user input. Callbacks are event listeners that receive events from the source and run when the events fire. Interfaces describe the protocols by which events are to be communicated, transformed or filtered (Sun, 2007; MSDN, 2007).

In the Java AWT event model, multiple event listeners can register to be notified of events of multiple types. For an example, a Button in AWT is an event source that can generate a built-in ActionEvent. The `addActionListener()` method can be called on by the Button to add an event handler which implements the `actionPerformed()` method to handle the ActionEvent. Event-handling code executes in a single thread, called the event-dispatching thread, which “ensures that each event handler finishes execution before the next one executes” (Sun, 2007).

The Jazz (Bederson et al, 2000) graphics toolkit allows Zoomable User Interfaces to be created based on Java’s event listener model. User interaction events and diagram modification events are listened and responded to by event listeners. An example application that exploits Jazz is the set of highly user interactive zoomable views (as shown in Figure 2.7) integrated into a visual language meta-tool

(Liu et al, 2004). Enhanced user-oriented view navigation and management features are provided based on a set of event handler implementations.

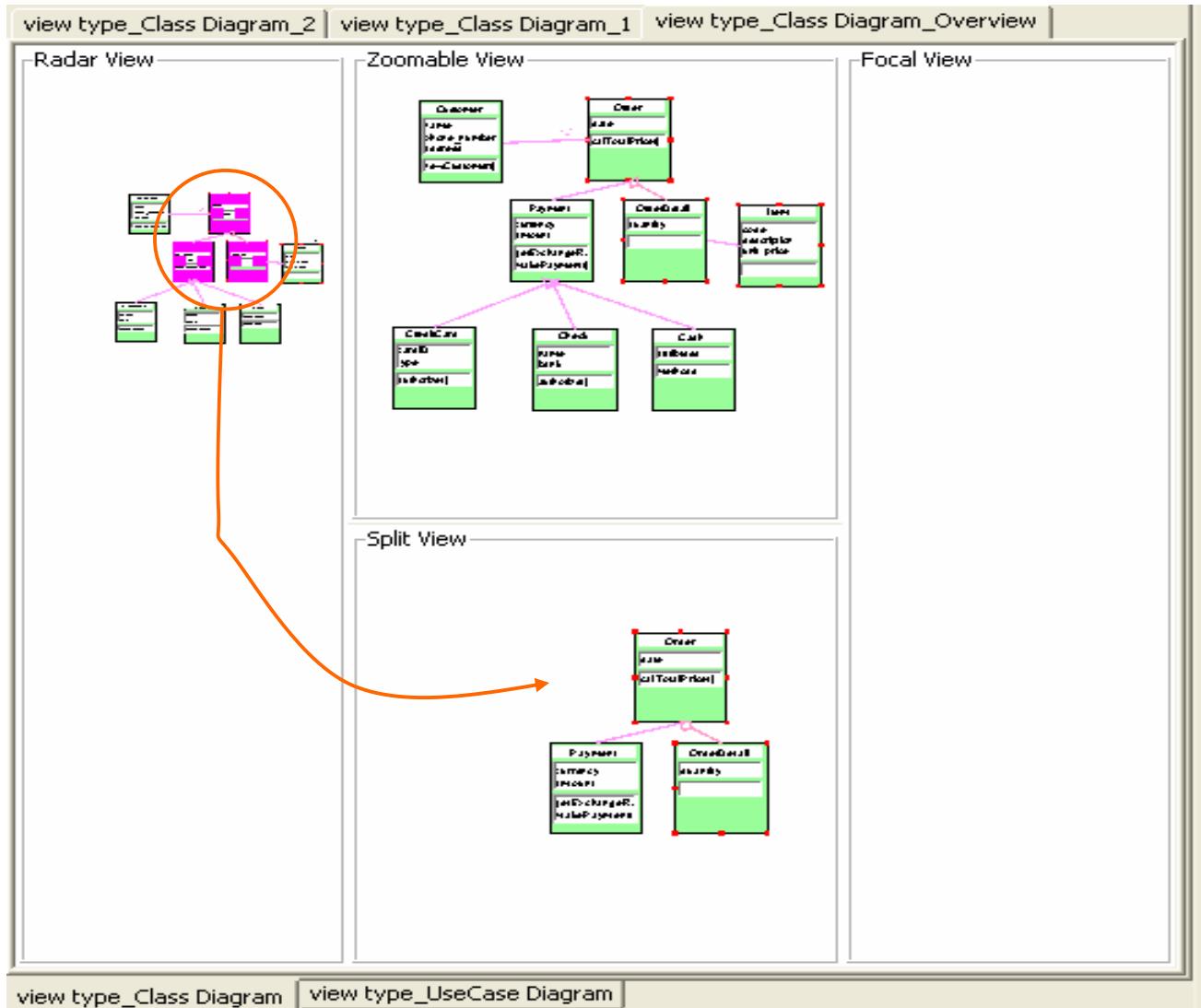


Figure 2.7. User interactive zoomable views (Liu et al, 2004)

Many modelling frameworks are “event driven”, meaning when a user modifies a model or view, events are generated and can be acted upon to update other content, enforce constraints, etc. The Eclipse Modelling Framework (EMF) (Eclipse, 2007) is a framework and code generation facility that allows definition of a model in Java, XML, or UML. An EMF model is the common high-level semantics shared by them. One specification can generate others with the corresponding implementation classes. Every generated EMF class is also a notifier, which sends notifications whenever an attribute or reference is changed. Notification observers in EMF are called adapters because they are often used to extend the behaviour of the object they are attached to. Adapters are

used extensively in EMF as observers and to extend behaviour. An adapter “can be attached to any EObject (for example, PurchaseOrder) by adding to its adapter list like this:

```
Adapter poObserver = ...
aPurchaseOrder.eAdapters().add(poObserver);” (Budinsky et al, 2003)
```

Whenever a state change occurs in the interested object, as shown in Figure 2.8, the adapter’s `notifyChanged()` method will be called to handle the change (Budinsky et al, 2003).

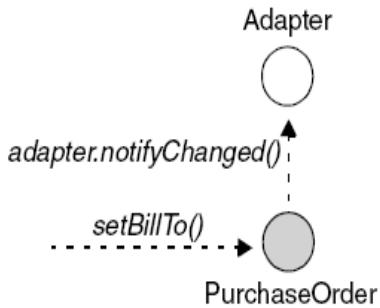


Figure 2.8. Calling the `notifyChanged()` method. (Budinsky et al, 2003)

2.1.3 Events in Database Systems

Events and reactive functionalities are used in database systems to support integrity of constraints, view maintenance, access control, and transaction management. Triggers, special kind of stored procedures, execute automatically in response to an INSERT, UPDATE or DELETE event in a database table or view (MSDN, 2007). Reactive mechanisms in active databases are usually centred on the notion of Event-Condition-Action (ECA) rules (Kiringa, 2002). An event defines the signal that triggers a rule; a condition defines the prerequisite for a rule to execute; and an action defines the way to update the system. Gatziu and Dittrich investigated the definition, detection, and management of events in the active object-oriented database system SAMOS (Gatziu and Dittrich, 93), where events are defined as a part of a rule, and be used in multiple rules.

While the ECA rules can effectively define reactive behaviours in database systems, they require the users’ capability to understand and code in database programming languages. The ECA rule specification and processing are generally separated in database systems. The rule execution is not easy to monitor, unless a formal model can be used to abstract rule bases and their relations, operations and processes, to reduce the management complexity of a database system (Li et al, 2002).

2.1.4 Events in Workflow Management Systems

Defined by the Workflow Management Coalition, workflow is “the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules.” As depicted in Figure 2.9, a workflow process defines “various process activities, procedural rules and associated control data used to manage the workflow during process enactment” (Workflow Management Coalition, 1999).

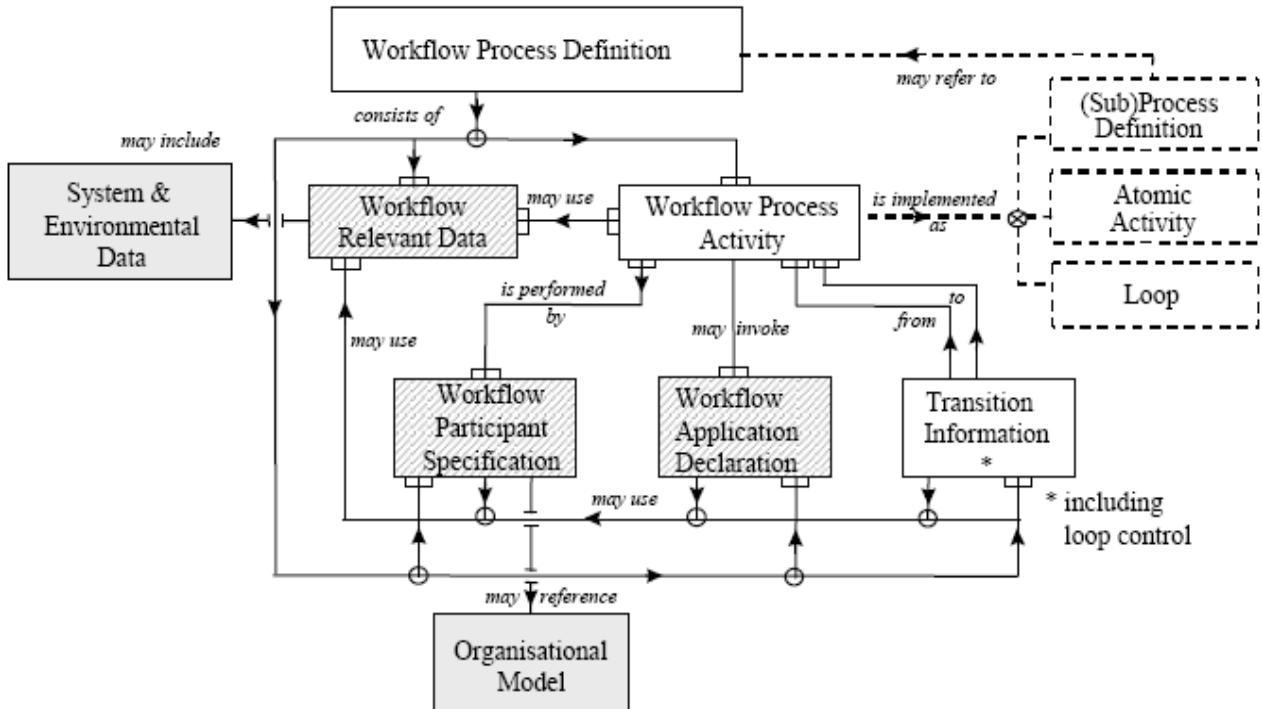


Figure 2.9. WfMC Process Definition Meta-Model. (Workflow Management Coalition, 1999)

A Workflow Management System (WFMS) is a system that “defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications” (Workflow Management Coalition, 1999). Workflow Management Systems are often event-driven. An occurrence of a particular internal or external condition to a WFMS is regarded as an event that causes rules to trigger and responding delegating actions to be taken. Examples of WFMS include Process WEAVER (Fernstrom, 1993), Oz (Ben-Shaul, 1994), Serendipity (Grundy et al, 1998), jBPM (Koenig, 2004) and BizTalk (Chappell and Associates, 2007) etc. Such environments often facilitate event-based work coordination, task automation, and system integration. These events are associated with larger system components, compared to the graphical and database events mentioned previously.

2.1.5 Events in Distributed Computing

Events and notifications are commonly seen in distributed interactive applications, where distributed user actions are events to which the applications react. A distributed event system generally involves the object that registers interest in an event, the event generator and the remote event listener (Sun, 2005). In describing the distributed event-based communication, the event model can be categorised as: peer-to-peer (e.g. Java distributed event model), mediator (e.g. CORBA event channel), and implicit model where objects subscribe to event types instead of other objects or mediators (Meier and Cahill, 2002).

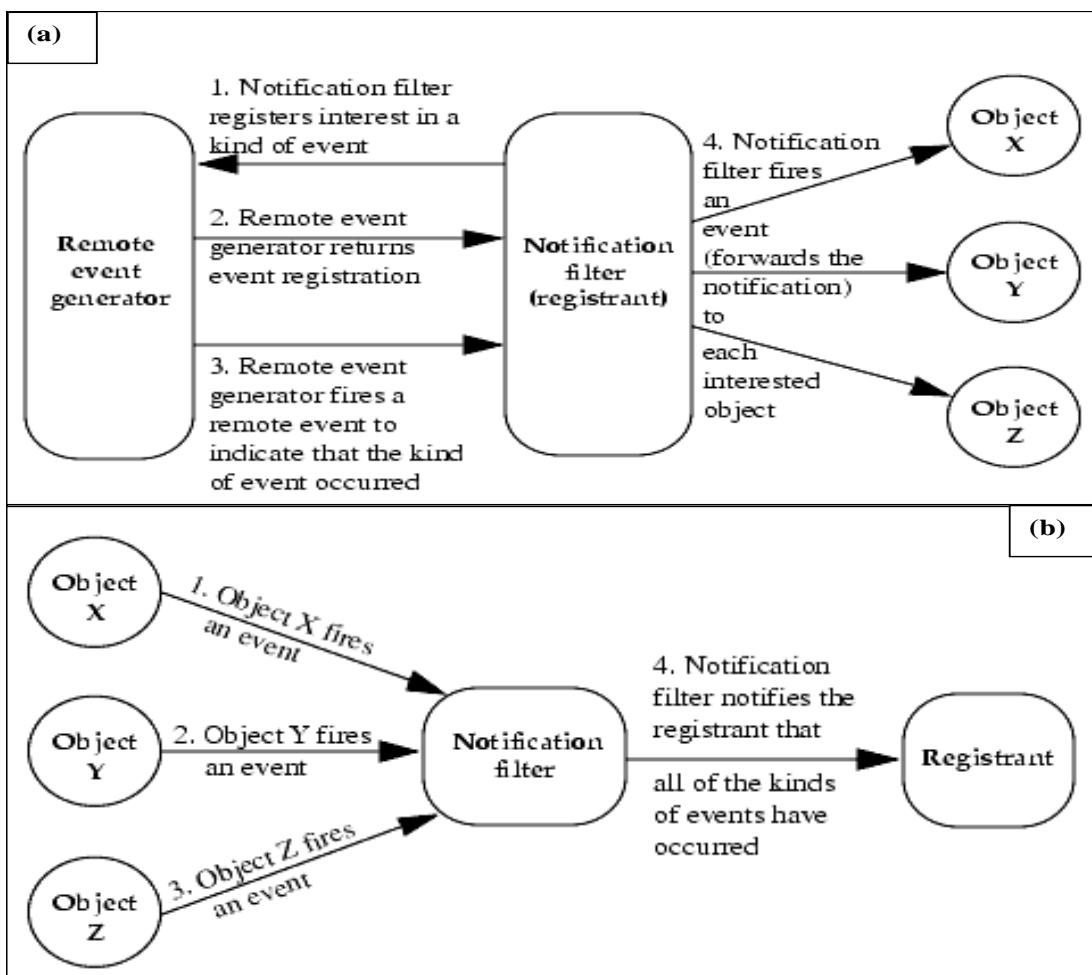


Figure 2.10. Jini™ Distributed Events Specification (Sun, 2005)

Distributed event-based programs can be implemented in Jini™ (Sun, 2005) by allowing “an object in one Java™ virtual machine (JVM) to register interest in some event occurring in an object in some other JVM, perhaps running on a different physical machine, and to receive a notification when an event of that kind occurs” (Sun, 2005). An event can be generated and passed to a remote event

listener which is physically located differently over the network. Notification filters and notification multiplexing/de-multiplexing mechanisms are also introduced, as illustrated in Figure 2.10, to be used to help minimise network traffic. Notification multiplexing (as shown in Figure 2.10 (a)) enables the filter to receive a notification and forward it to each of the objects that had registered its interest in the event notification. Notification de-multiplexing (as shown in Figure 2.10 (b)) enables the filter to receive a set of events that the object is interested in and deliver them to the object.

OMG defines a set of event service interfaces that enable publish/subscribe communications between objects. There are two roles defined for objects: the supplier (i.e. event generator) role and the consumer role. The Event Service facilitates anonymous suppliers/consumers, decoupled asynchronous communication, group communication among suppliers and consumers, abstraction for distribution, and abstraction for concurrent event handling. Suppliers and consumers establish communications via CORBA requests. Event channels are intermediaries that allow asynchronous communications between multiple suppliers and consumers (OMG, 2004). Figure 2.11 illustrates channel component and passing of events among multiple suppliers and consumers.

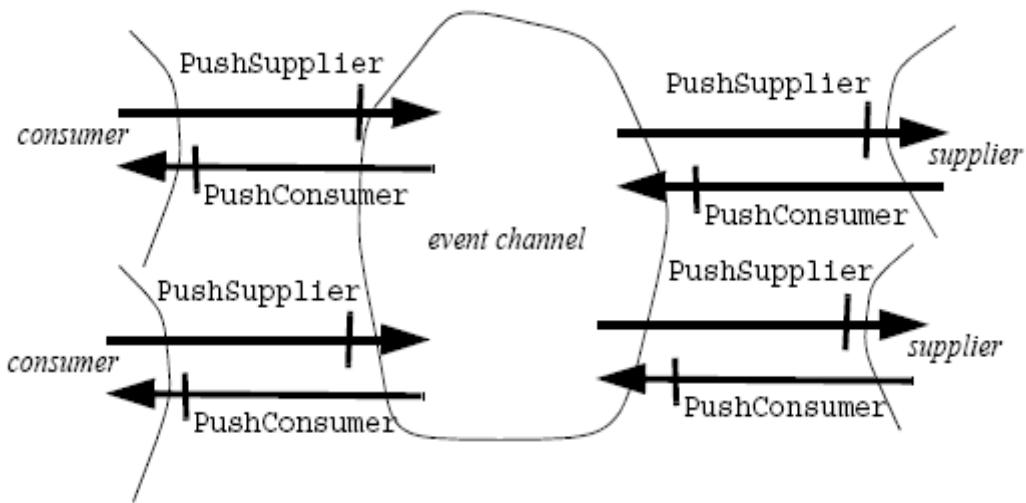


Figure 2.11. Event Channel with Multiple Suppliers and Consumers (OMG, 2004)

Message-Oriented Middleware (MOM) systems offer message-based anonymous and asynchronous communication mechanisms. Sun's Java Beans (Sun, 2007) are based on the reactive paradigm. Figure 2.12 (a) shows a simple Message-Driven Bean (MDB) example, where the application client sends messages to the message queue, and the Java EE server delivers the messages to the instances of the message-driven bean, which then processes the messages (Sun, 2007). Microsoft Message Queuing (MSMQ) is also based on the MOM model. Figure 2.12 (b) shows the MSMQ queue's

communication model, where queues store messages from a client and forward them to the service. Using such a MOM model ensures high availability of communications in distributed systems (MSND, 2007).

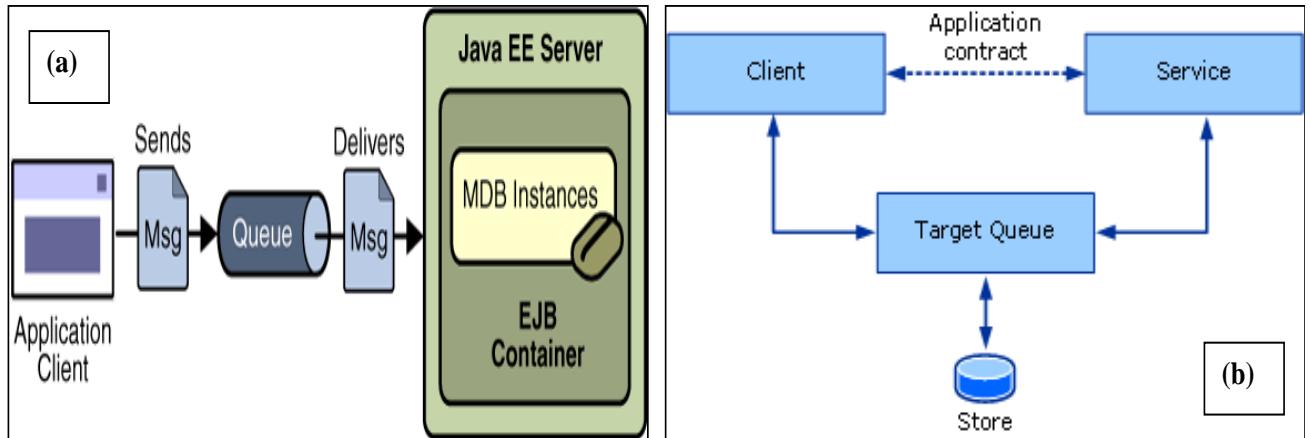


Figure 2.12. (a) A Simple MDB Application (Sun, 2007), (b) MSMQ model. (MSDN, 2007)

Other applications of events in distributed systems include JEDI (Gugola et al, 2001) which is an object-oriented infrastructure that supports the development and operation of event-based systems especially distributed workflow management systems, and the web services eventing (Box at el, 2006) specification which describes a protocol allowing event-based communication between web services.

2.1.6 General Purpose Event Frameworks

Since events are commonly used in many application domains, it seems desirable to have a general purpose event framework that can support the design and construction of a wide range of event-based systems. Such a framework should allow reuse of design and implementation thus saves a huge amount of the development effort and time. Though there exist many frameworks for high-level event modelling, they generally lack identified support for event handling specification.

Barrett et al defined a generic framework for event-based system integration, named the EBI framework (Barrett et al, 1996). It defines a flexible object-oriented reference model for both the static and dynamic specification of event-based systems. Participants in the EBI framework communicate via four types of components: registrars, routers, message transforming functions (MTFs), and delivery constraints (DCs). Participants are interacting software modules as either informers or listeners. Registrars establish communication relationships among participants. Routers

deliver messages among participants. MTFs transform messages in transit. DCs control the delivery of messages against some rules. Figure 2.13 depicts the EBI framework components; the left figure shows the component interaction relationships, and the right figure shows the abstract data types of the framework components.

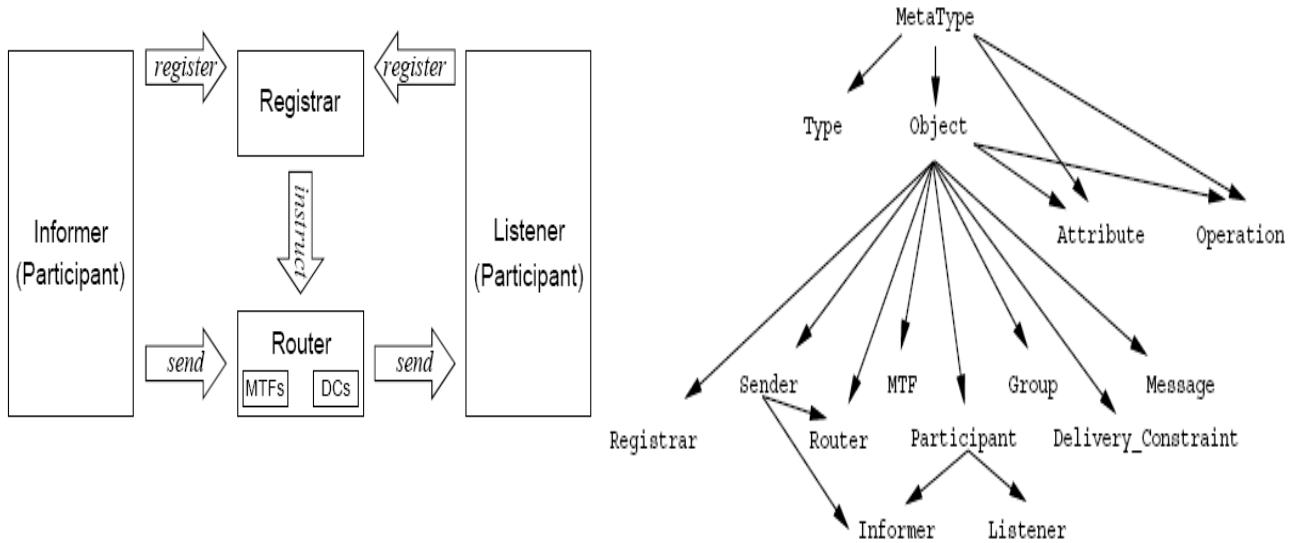


Figure 2.13. The EBI framework: Relationships between components (left) and framework metamodel (right). (Barrett et al, 1996)

Yeast (Krishnamurthy and Rosenblum, 1995) is a general purpose platform for specifying distributed event-based applications. Systems are constructed in Yeast via a high-level language with predefined object classes and attributes to support specification and matching of event patterns. Yeast supports a rich collection of client commands that allow users to interactively query and manage the status of their specifications. Furthermore, security of all client interactions is ensured. Rapide (Rapide Design Team, 1997) is an architecture description language that supports systems to be constructed via architecture definitions. It uses event-based simulations to find event sequences, causalities and constraint violations. Complex Event Processing (CEP), an application of Rapide concepts, assists in understanding of a distributed enterprise system by organising the activities of the system in an event abstraction hierarchy. ZOOM (Jia et al, 2005) provides an event model processed by an event-driven framework to bind the structural, behavioural and UI models of a software specification together.

Though there exists many high-level event specification frameworks, there is still a lack of a state-of-the-art framework for generic event handling specification. Most of the above event specification frameworks support custom event handlers; they neither exploit compositional primitives (such as

queries, filters, actions, and constraints) as reusable event handler building blocks, nor support tracing and visualising event propagations between them.

2.2 Event Handling Specification and Visualisation Techniques

Event handling plays an important role in event-based systems. All of the above systems need a way to handle the receipt of an event, i.e. need to take some action in response to it. The major kinds of event handling responses from the above systems include broadcasting events, regenerating events, and consuming events according to ECA rules. There are various techniques for specifying event handlers, including custom code writing, flow-based approaches, declarative programming, visual programming and hybrid approaches. Among these approaches, some are easy to use, some require developers to have expertise of programming languages, and some provide both static structure and dynamic behaviour views of an event handler. In this section we briefly examine the most common approaches used for specifying event handling in different applications.

2.2.1 Custom Code Writing

Custom code writing is typically the most commonly used way to design and construct an event-based system (Grundy et al, 1995; Jin, 2003; Zhu et al, 2007). Users have full and flexible control over the behaviour that needs to be specified in an event handler. However, it requires the user to be a competent programmer, and a great deal of time needs to be invested on concerns such as language syntactic details rather than focusing on the conceptual problem of the system. In some circumstances, due to the lack of visualisation support, it is often very hard to debug code when the application becomes sophisticated. Therefore, the quality of the resulting application can not be guaranteed. Problems of maintenance, reusability and extensibility of the application arise as well (Jin, 2003).

Examples of reactive systems configured by writing custom code include the following: frameworks, such as Suite (Dewan and Choudhary, 1991), Meta-Moose (Ferguson et al, 1999), and Unidraw (Vlissides and Linton, 1989) which require modifications to the tool's code, with an edit-compile-run cycle; some Tcl/Tk-based tools may be modified while in use (Welch and Jones, 2003), but this requires use of the Tcl programming language; and Pounamu (Zhu et al, 2007) supports direct modification via an API at runtime (as shown in Figure 2.14 (a)). However, usually only programmers familiar with the tool architecture can make such modifications. Many end users of such tools are not programmers and have difficulty in using textual, programmatic scripting languages to tailor their design tools (Zhu et al, 2007).

A common alternative approach supporting run-time modification is scripting. This is supported, for example, by Amulet (Myers, 1997) (as shown in Figure 2.14 (b)) and Peltonen's UML tool (Peltonen, 2000). MetaEdit+ (Kelly et al, 1996) also provides a custom scripting language for report generation while GME (Ledeczi et al, 2001) uses OCL (OMG, 2003) as a scripting language for constraint specification. These again require the knowledge of scripting languages thus are difficult for non-programmer users to understand and use (Green and Petre, 1996; Zhu et al, 2007).

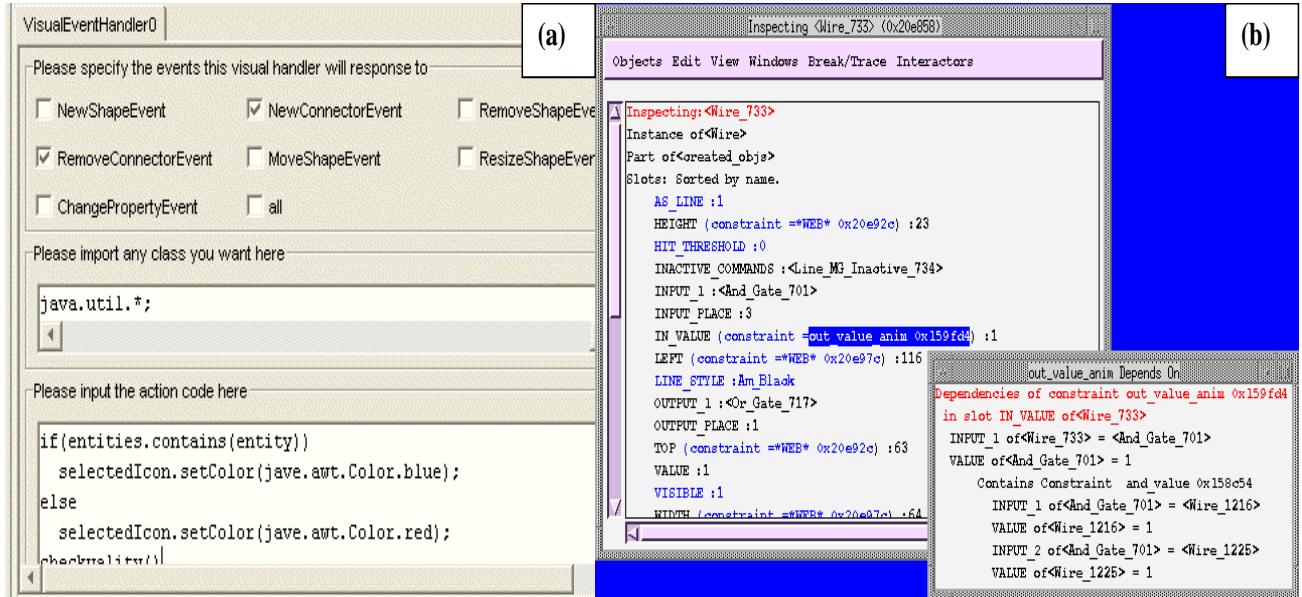


Figure 2.14. (a) Pounamu event handler designer (Zhu et al, 2007); (b) Amulet constraint scripting (Myers, 1997)

2.2.2 Declarative Approaches

Declarative programming models allow users to define “what” rather than “how”. They provide another way of specifying event handling using relationships between data as opposed to control flow (Burnett and Ambler, 1992). Examples include rule-based, functional and constraint-based programming models. Declarative approaches allow users to ignore the implementation details, but instead concentrate on the problem semantics of the event handler by specifying relationships between objects, or conditions guarding the state transition of an object (Jin, 2003). Declarative systems usually have an embedded runtime engine to automate the evaluation of these relationships or conditions. Details are provided in the following subsections.

2.2.2.1 Rule-based Specification

Rules as a declarative modelling technique can be integrated into an existing object-oriented modelling technique (e.g. UML) to specify behaviour on data models such as formulating

constraints, handling events, deriving new attribute values and modelling strategies in business and engineering. Rules provide a useful level of abstraction, allowing the designer to focus on important behaviour (Taentzer, 1999).

Rule-based specifications are widely used in artificial intelligence systems for modelling intelligent behaviours, such as learning, searching, and problem-solving in computer-aided design and configuration (Jin, 2003). In rule-based systems, users can compose a rule with actions, preconditions and post-conditions; a rule interpreter evaluates the conditions against a knowledge base and executes the matched actions. An example of rule-based systems is AP5 (Cohen, 2006), which allows users to "program" at a more "specificational" level, in other words, focus more on what the user wants the machine to do and less on the details of how. With regard to event-based systems, rule-based modelling technique can be used to specify the preconditions and postconditions of a particular event or multiple event occurrences (Jin, 2003). Rule-based systems are easy to understand, and suitable for small-scale, static problem domains. However, the rule-based paradigm is not suitable for modelling large-scale, dynamic software systems. Rule-based systems are limited in their expressiveness, optimization is only based on heuristics, and they are limited with regard to reasoning solutions (Felfernig et al, 2003). Rule interpretations are generally computation intensive, which limits the performance and real-time response (Jin, 2003).

Most rule-based approaches exemplify "Event-Condition-Action" based languages where the user specifies an event of interest; conditions ("filters") when the action(s) should be run in response to the event; and action(s) to run to modify the tool's state. Other Event-Condition-Action rule-based languages have been developed for a variety of domains, including building and tailoring design tools (Costagliola et al, 2002; Ledeczi et al, 2001; Lewicki and Fisher, 1996), user interface event handling (Berndtsson et al, 1999; Jacob, 1996), process modelling (Grundy et al, 1998), database rule handling (Matskin and Montesi, 1998) and middleware for event detection and composition (Buchmann et al, 2004). Figure 2.15 shows a rule graph (Matskin and Montesi, 1998) that represents a particular active rule with three types of nodes: event nodes, condition nodes and action nodes. Event nodes correspond to events which switch on active rules. Condition nodes correspond to conditions to be checked in order to trigger rules. Action nodes represent actions performed by rules. Directed edges on rule graph connect event node with conditional node and condition node with action node.

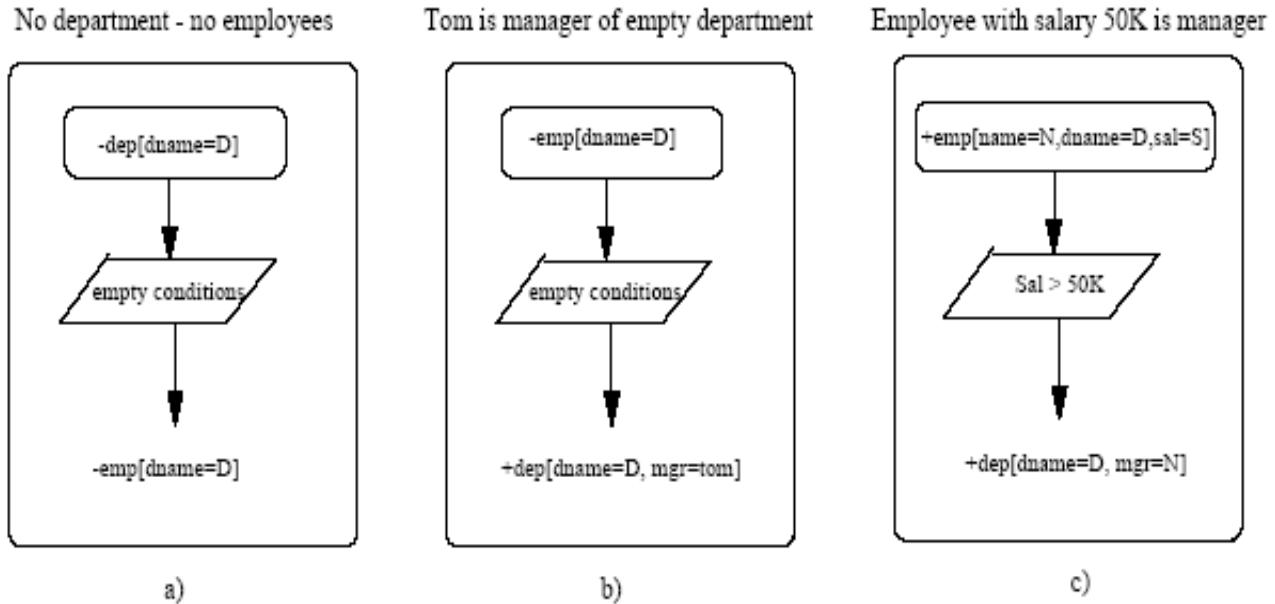


Figure 2.15. Rule graph (Matskin and Montesi, 1998)

Reaction RuleML (Paschke et al, 2006) is an XML-based language for reaction rules. It “incorporates different kinds of reaction rules from various domains such as active-database ECA rules and triggers, forward-directed production rules, backward-reasoning temporal-KR event/action/process logics, event notification, messaging, active update, transition and transaction logics” (Paschke et al, 2006) (as illustrated in Figure 2.16). These rules can be specified globally, with other derivation rules or integrity constraints, or locally, nested within other derivation or reaction rules. Different rule processing styles are incorporated such as actively pulling or detecting events by monitoring, passively listening to events, and reasoning events and actions’ effects.

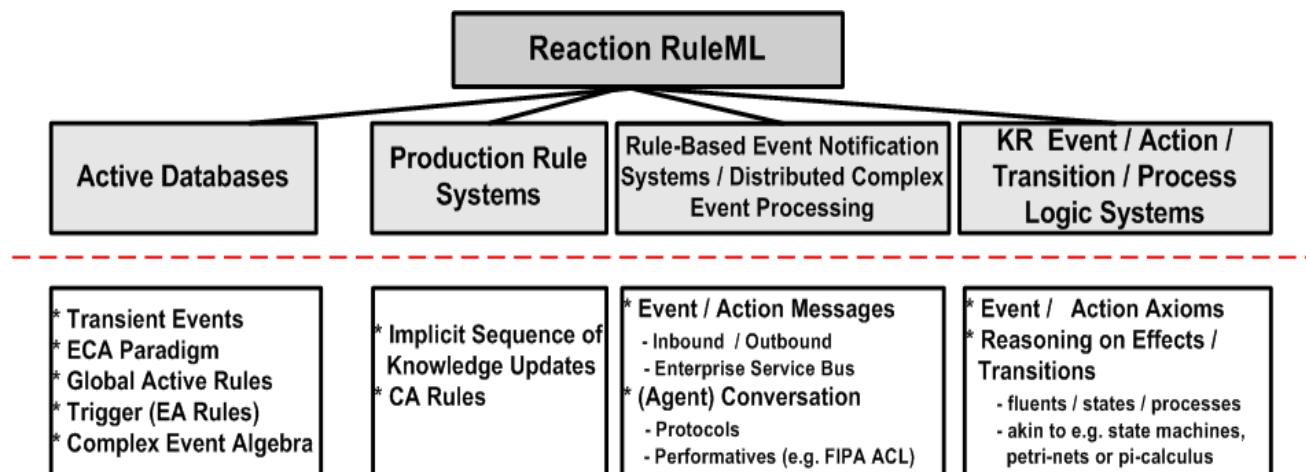


Figure 2.16. Reaction RuleML (Paschke et al, 2006)

The above mentioned rule-based approaches often suffer from use of inappropriate, textual rule-based languages which are not suitable for end users. They rely on many abstract concepts like control structures and variables. They have limitations on the expressive power of the languages; difficulties in visualising and debugging learned rules; and limitations of reconfiguration power, including compile-time rather than run-time changes (Ledeczi et al, 2001; Costagliola et al, 2002; Liu et al, 2007).

2.2.2.2 Functional programming

Functional programming is a declarative programming paradigm that places emphasis on the application of functions rather than state changes. It features higher-order functions, non-strict semantics (lazy evaluation), data abstraction, equations, pattern matching, and strong typing (Hudak, 1989). Haskell (Haskell, 2007) is a popular general purpose functional programming language used to date.

Higher-order functions treat functions as values and give them the same first class status. This makes the function the primary abstraction mechanism over values (Hudak, 1989). Higher-order functions are very efficient for calculating values associated with lists, where, for example, they can be applied to every member of a list and return an updated list. For example, the function `map (*2)` `[1, 2, 3, 4]` returns `[2, 4, 6, 8]` (Haskell, 2007).

The language Z (Wordsworth, 1992) is a well-known approach to formal specification. It specifies systems as functions. Z uses a state machine model to describe a “system as an automaton with a state from a potentially infinite state space and a state transition function” (Weber, 2003). It is declarative rather than procedural, because the system state is determined by values taken by variables, and operations are expressed by relationship between preconditions and post-conditions. Variable declarations and related predicates are encapsulated into schemas. Complex specifications are facilitated by schema calculus via composition.

Vital (Hanna, 2002) is an interactive graphical environment that uses a functional programming language in a spreadsheet with direct data display. It allows graphical display of data structures in a format defined by a datatype indexed stylesheet. Figure 2.17 shows an exemplar family tree stylesheet. It supports demand-driven evaluation of values by the action of the user scrolling around an unbounded workspace.

```
data Person = Unknown | Person {name:: String} | Parents (father:: Person, name:: String, mother:: Person)
```

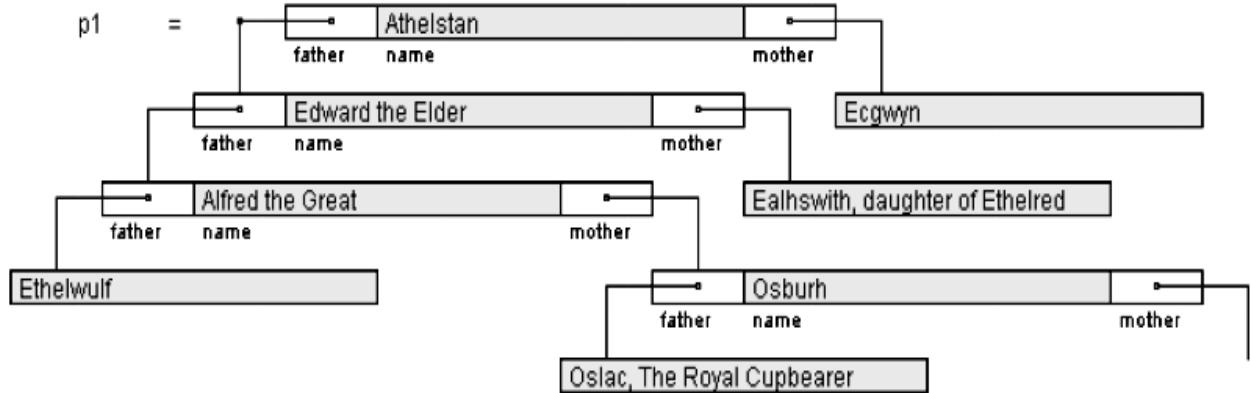


Figure 2.17. A family tree (Hanna, 2002)

Functional programming languages are terse, clear, expressive and powerful; however they may still be error-prone in specification due to their high abstraction gradient. Supplementing them with visual representations could potentially make them easier to visualise, debug and understand (Kelso, 2002).

2.2.2.3 Constraint-based Specification

Constraints are presented in many user interface applications to specify object relationships and runtime enforcement. Garnet (Myers, 1990) is a constraint-based user interface toolkit facilitating visual layout constraints and consistency maintenance. Figure 2.18 shows an example of Garnet's code-based constraint specification. Model-integrated computing (Ledeczi et al, 2001) uses metamodelling to define domain-specific modelling languages and generate design environments enforcing model-level constraints.

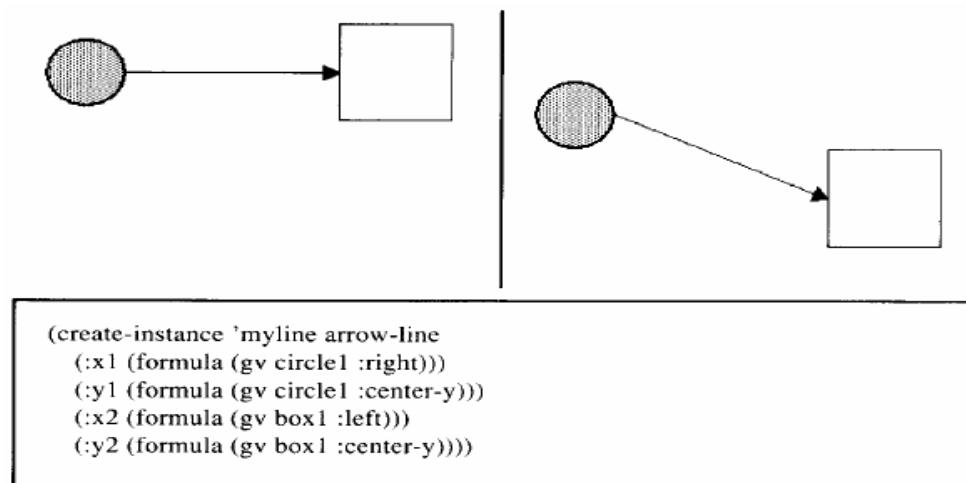


Figure 2.18. The line stays attached to the box and circle even when they move. Below the graphic is the code to define the constraints on the line. (Myers, 1990)

The Abstraction-Link-View paradigm (ALV) (Hill, 1992) is an approach similar to the MVC (Burbeck, 1992; Sun, 2005) architecture in the sense that it manages consistency between model and views. In addition, AVL introduced the use of link objects (as shown in Figure 2.19) to connect and constrain multiple user interface views and a single shared abstraction. Constraints are specified in a structured manner in link objects, so the views can ignore the shared abstraction and vice versa.

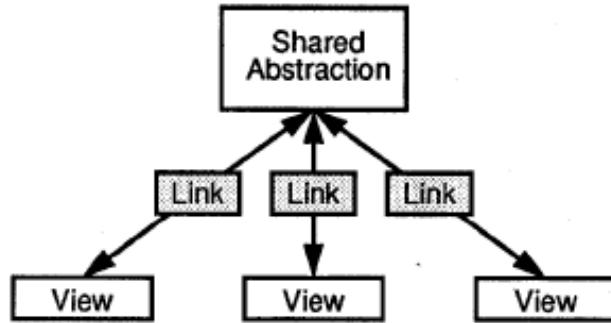


Figure 2.19. Basic ALV architecture. (Hill, 1992)

Form Chart (Weber, 2002) uses bipartite transition diagrams to define states of client and server, and transitions from client pages to server actions. Dialogue Constraint Language (DCL) is used in Form Charts. DCL is an extension of OCL (OMG, 2003) and defines special-purpose constraint types which are used to annotate state transition constraints. Figure 2.20 illustrates the use of an enabling condition constraint to specify under which circumstances this transition is enabled, and that of a client output constraint to specify the data submitted from a client page.

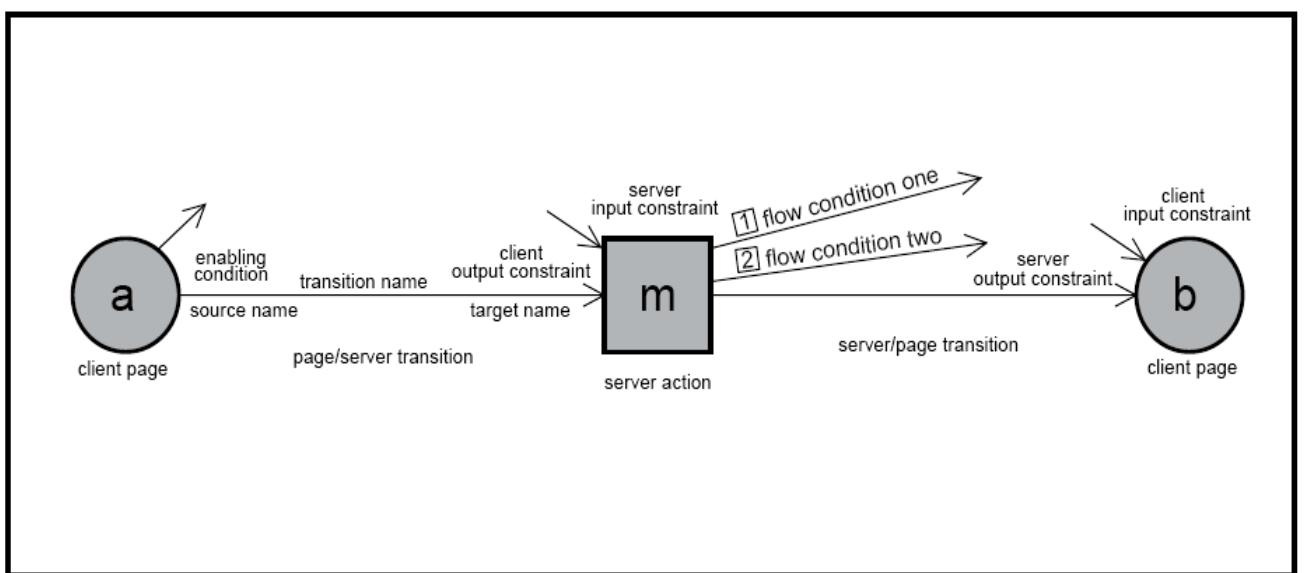


Figure 2.20. Form chart notational elements (Weber, 2002)

Constraints do not necessarily need to be specified as code. MetaEdit+ (Tolvanen, 2006) allows constraints to be specified and visualised in the graphical metamodel using a wizard-like approach. Figure 2.21 shows its visual editor for adding constraints to graphs.



Figure 2.21. Adding constraints. (Tolvanen, 2006)

Constraints are usually enforced via a constraint solver, which acts as a back-end engine to automatically interpret and check the constraints against the model. Alloy (MIT, 2006) is a declarative language for structural modelling. Complex structural constraints and behaviour can be specified based on first-order logic using Alloy and solved by the Alloy Analyser, which is a constraint solver.

Though constraint-based languages are powerful in specifying object relationships, usually they need an efficient constraint solver to interpret and enforce the constraints. Limitations of the constraint solver mean that some constraints are not possible to express (Jin, 2003). Complex constraint may be hard to specify and visualise for the end users. Debugging of constraints is also hard.

2.2.3 State-based Formalisms

State information is necessary to be represented in system models, as states are the concerns of both the structural and behavioural aspects of a system. Various state-based formalisms are used in specifying system workflows. We describe Petri Nets, Finite State Machine, and Event and Time graphs in this section.

2.2.3.1 Petri Nets

The Petri net is a powerful modelling formalism widely used in analysing and simulating workflows and distributed systems described in Section 2.1. It uses a terse set of graphical symbols to specify

system structure and behaviour. The language notation consists of places, transitions and directed arcs. Places (graphically represented as ellipses) represent data. Transitions (graphically represented as rectangles) represent activities. Arcs (graphically presented as arrows) connect places with transitions to represent activities' inputs/outputs, prerequisites/consequences, or state changes. An arc can have an associated expression to describe how the state of the Petri net changes.

Petri nets have been used in modelling reactive systems. PUIST (Li et al, 1997) uses a Petri net notation for specifying the static form and dynamic behaviour of graphical user interfaces. Both hierarchical and recursive compositions of GUI components can be specified using this visual formalism. Palanque et al considered defining interface places as a subset of places to represent the interface between the system being modelled and its environment (Palanque et al, 1993). Each event is directly modelled in the system by the deposit of a token in an interface place. An incoming event may trigger different actions in the system, as depicted in Figure 2.22.

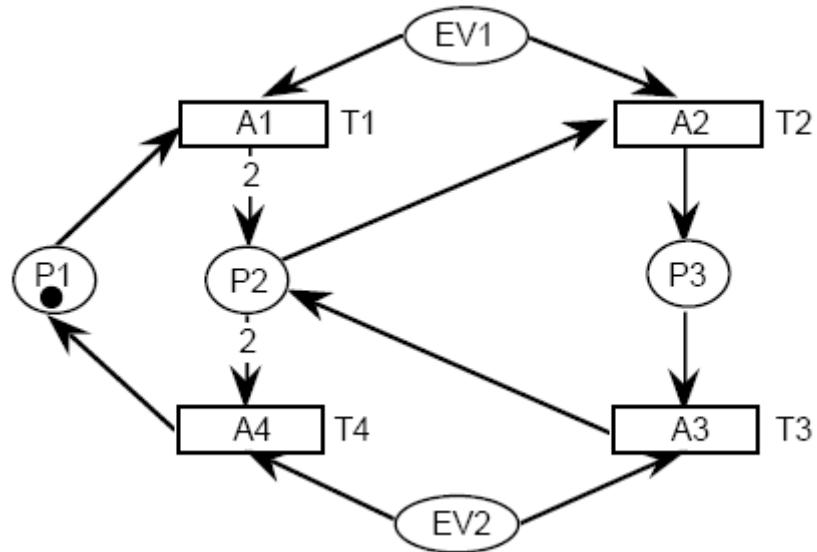


Figure 2.22. Modeling a reactive system with Petri nets. (Palanque et al, 1993)

2.2.3.2 Finite State Machines

Behaviour can be represented as a finite state machine (FSM) with a set of states, inputs, transitions and actions. A state stores information; an input supplies data; a transition describes a state change from one to another; and an action triggers a transition (Wagner et al, 2006). There are two representations of FSMs: state transition diagram and state transition table (Drumea and Popescu, 2004; Wagner et al, 2006). Figure 2.23 shows an example of graphical representations of them.

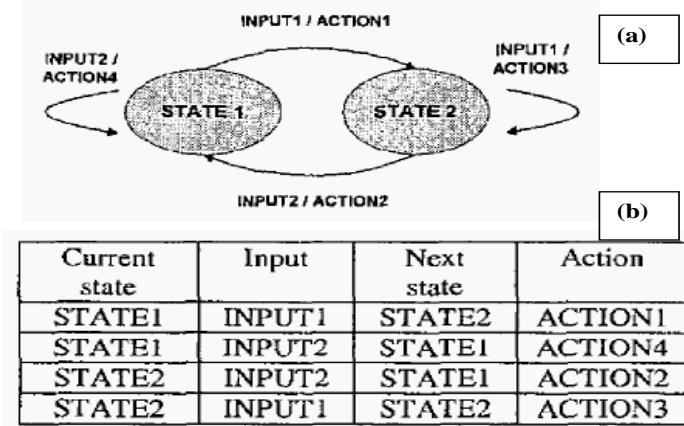


Figure 2.23. (a) State transition diagram. (b) State transition table. (Drumea and Popescu, 2004)

FSMs are good at representing states of an event detection process in a reactive system, but it is not possible to identify participating event occurrences and event consumption modes or to extract the structure of a composite event (Berndtsson and Mellin, 1999), thus they are not ideal for representing event handling.

2.2.3.3 Event and Time Graphs

Event graphs can be viewed as directed acyclic graphs with nodes and leaves. An event graph uses a time line to represent the event history, nodes to represent event operators, leaves to represent primitive events, and arcs to represent a connection between a node and its two children. Event graphs make it easy to determine the structure of composite events. Hence, they support understanding of the state perspective (Berndtsson and Mellin, 1999). Figure 2.24 shows the detection of the composite event E_6 for recent event consumption mode using an event graph.

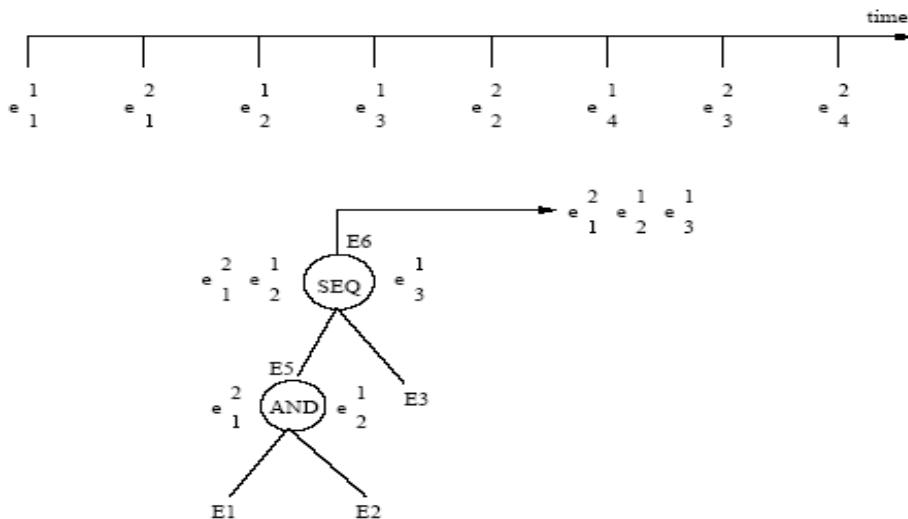


Figure 2.24. An event graph (Berndtsson and Mellin, 1999)

Although event occurrences are visually presented, their role (indicator or terminator) in the event detection process is not explicitly visualised. These must be inferred manually by the user, by looking at the event history, participating event occurrences and event operators shown in the event graph. One potential solution to this is to use event graphs in combination with time graphs, since time graphs can visualize initiators and terminators (Berndtsson and Mellin, 1999). The use of time graphs to visualise composite events and event consumption models was introduced by Chakravarthy et al (Chakravarthy et al, 1994). A time graph uses a time line to represent the event history, which presents the semantics of composite events. Each event occurrence in the event history is marked in order of occurrence on the time line. Each time interval depicts the detection of a composite event for a given event consumption mode and includes one initiator, zero or more participating primitive events, and one termination event. Figure 2.25 depicts the detection of a composite event using a time graph for four event consumption modes: recent, chronicle, continuous, and cumulative.

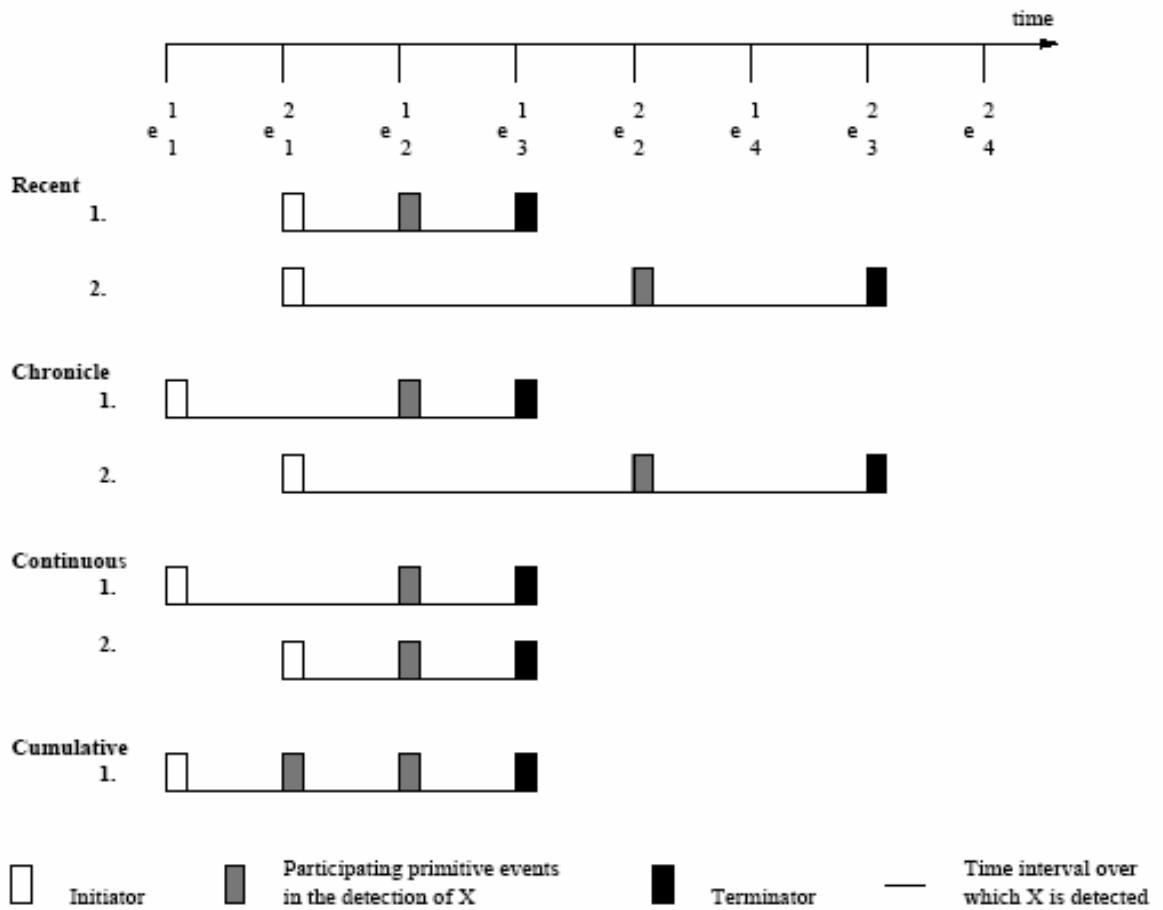


Figure 2.25. A time graph (Berndtsson and Mellin, 1999)

The advantage of using time graphs is that it is possible to see which event instances participate and their role in the detection of a composite event. However, time graphs do not provide any

information concerning the structure of the composite event, e.g. event operators are missing. Thus, time graphs support understanding of the outcome perspective and need to be used together with additional information which describes the structure (Berndtsson and Mellin, 1999).

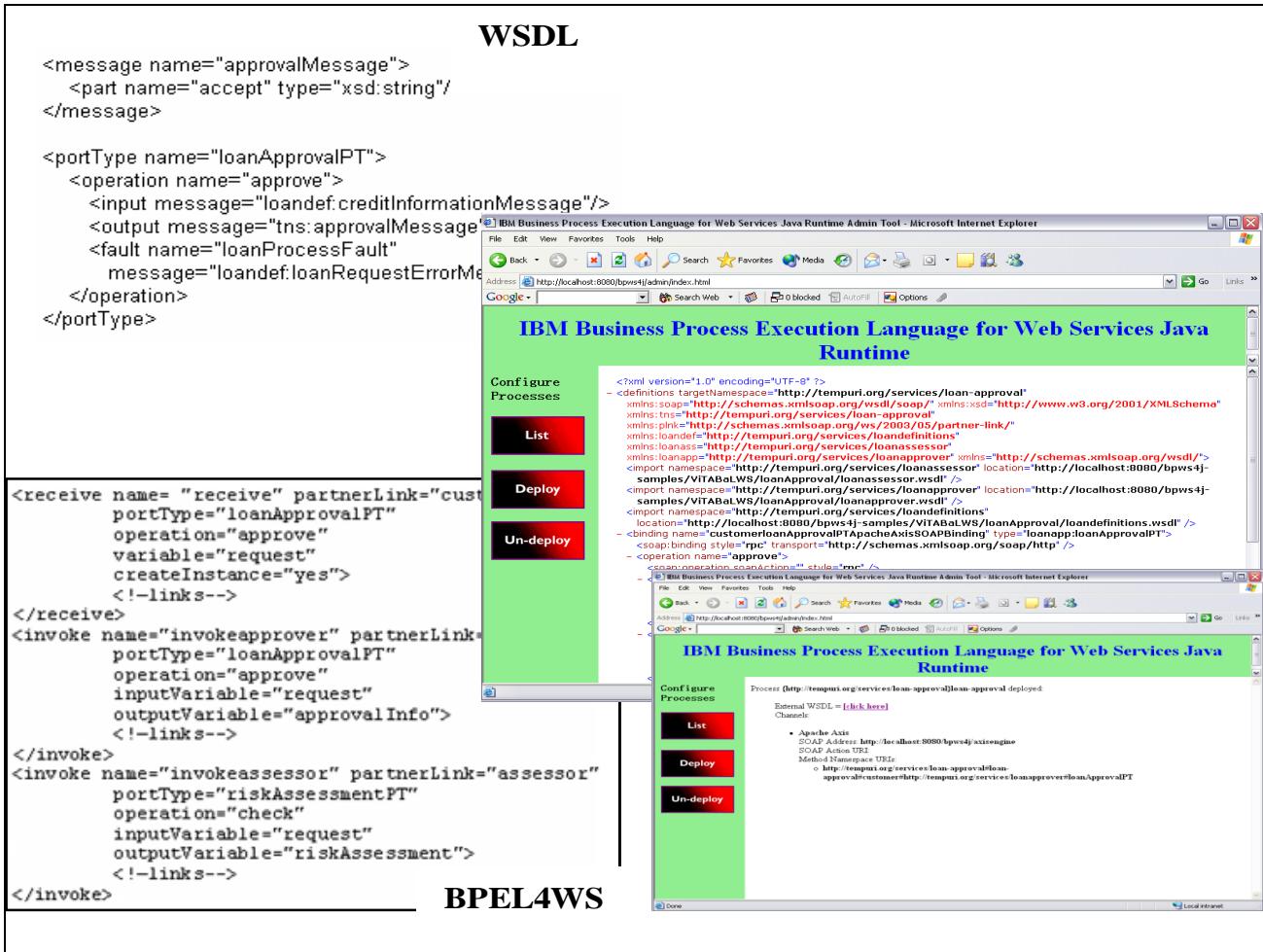
State-based approaches in general usually ignore the structural elements of a system and focus on representing runtime changes. This results in isolating dynamic behaviour specifications from the static structure which is the backbone of the system. Another disadvantage is that state-based approaches convey many low level details, especially when rendering a highly concurrent system, a large number of states need to be presented, which raises scalability issues (Kraemer and Herrmann, 2007).

2.2.4 Flow-based Approaches

Flows can be used to effectively represent dependencies between states and activities based on execution sequence or conditions. We describe two flow-based approaches: workflow and dataflow.

2.2.4.1 Workflow

As described in the Section 2.1.4, workflow supports procedural designs via the execution order and dependencies of activities. It includes definitions of sequence, parallelism and synchronization, decision making, split and merge, loop, start, terminate and cancel an activity. Usually this leads to a significant amount of additional control to be included in a model, thus making large systems difficult to understand and communicate. Workflow metaphors are typically used in much recent research on system composition. Simple workflows are, however, insufficient to describe the integration and co-ordination of complex system. For example, conditional execution is needed; some links are sequential data-flow from one to another; some asynchronous; services may subscribe to events; conversion of messages may be needed; and so on. Examples include BPEL4WS (IBM, 2003), BPML (ebPML, 2002), and jBPM (Koenig, 2004). Many are described as “business process modelling languages”. Different composition languages support different levels of abstraction, fault-recovery, transaction modelling, and component inter-relationships. Most are textual scripts that are interpreted at run-time by workflow or business process flow engines, for instance, BPEL4WS processes are interpreted by the BPWS4J (IBM, 2002) engine as shown in Figure 2.26. Such textual scripts are often challenging to read, error-prone to write, and reusability can be limited.



**Figure 2.26. BPEL4WS specification and deployment using BPWS4J,
adapted figure (Liu et al, 2005)**

Serendipity (Grundy and Hosking, 1998) exploits a set of visual components to model work processes, work plans and history for a particular project in the Serendipity environment. These include process stage, artefact, tool and role representations. It has in addition the Visual Event Processing Language (VEPL) to permit visual specification of arbitrary event handling and event-triggered rules in process modelling. VEPL includes two basic constructs, filters and actions, which receive events from stages, artefacts, tools or roles, or other filters and actions. Filters match received events against user-specified criteria, passing them onto connected filters and actions if the match succeeds. When actions receive an event, they carry out one or more operations in response to the event (Grundy and Hosking, 1998). Figure 2.27 describes the basic modelling capabilities of VEPL. Figure 2.28 shows a simple event-handling view that illustrates the use of VEPL for inter-stage work coordination.

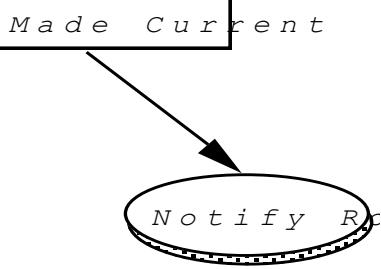
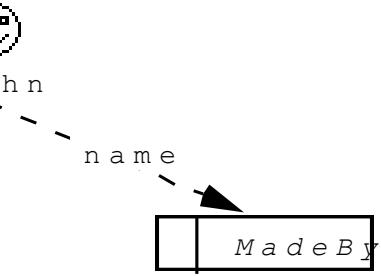
Notational symbol Example	Description
	A filter definition. Filters receive events (from process stages, artefacts, tools, roles, other filters or actions) and if the event matches the defined selection criteria for the filter, the event is passed onto the connected filters and/or actions. A filter or action reused from a template filter/action definition has its name in italics.
	An action definition. Actions receive events (from process stages, artefacts, tools, roles, filters or other actions) and respond to the event by performing some action (which often generates other events). Actions can pass on events to other filters and/or actions.
	An event flow into/from a filter or action. Events may be process stage enactment events, artefact update events, tool events, some event caused by a role (i.e. user), or an event generated by an action. For example, if Made Current decides an enactment event flowing into it means a process stage has been made the current enacted stage, then the Notify Role action is invoked to notify another user about this event.
	Usage flow into a filter or action. These specify parameters of the filter/action. For example, the MadeBy filter is parameterised by a role name which it uses to decide whether some event was caused by a particular role. In the example, that role name is instantiated to "john" by the usage connection to the role process model component.

Figure 2.27. Basic modelling capabilities of VEPL. (Grundy and Hosking, 1998)

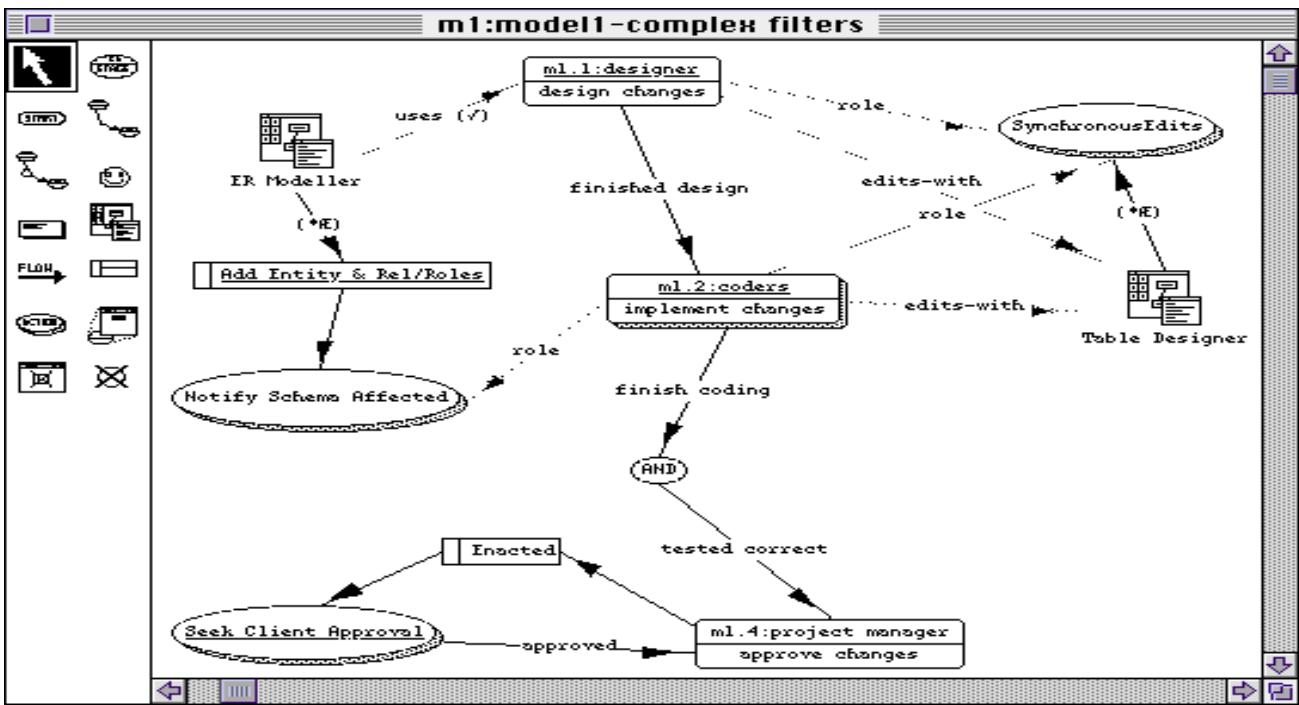


Figure 2.28. Multiple event processing, stage coordination and external process interfacing (Grundy and Hosking, 1998)

2.2.4.2 Data flow diagram

A data flow diagram is a directed graph that describes the flow of data between system components. It supports hierarchical decomposition of processes into sub-diagrams, which however, cannot be reused in different specifications (Weber, 2003). InterCONS (Smith, 90) and ConMan (Haeberli, 1988) are domain-specific visual dataflow languages which has certain primitives associated with user interactions (e.g. buttons, sliders). These primitives accept interactive input events and generate integer outputs, which can be routed into other dataflow nodes for further calculation (Burnett and Ambler, 1992). Besides UI primitives, Prograph (Cox et al, 1989) provides generality by facilitating dataflow connectibility to the low-level Macintosh Toolbox, allowing direct access and manipulation of the various Macintosh data structures, but this leaves the realm of high-level programming to do so (Burnett and Ambler, 1992); an example is illustrated in Figure 2.29, where Prograph uses iconic symbols to represent data and actions. The Biopera Flow Language (Pautasso, 2003) is a generic visual flow language for coordinating software components. It focuses on data flow, execution sequence and fault handling and all can be specified with a simple visual syntax. However it lacks modelling capability for event subscription and various other service relationships like call-backs. The visual syntax is verbose as both data and data bindings must be specified.

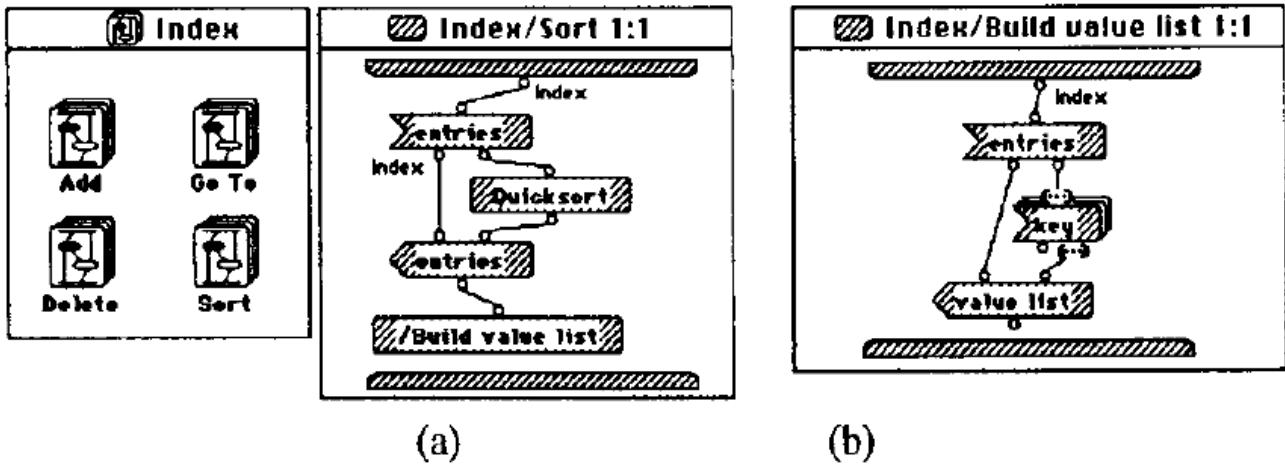


Figure 2.29. Program method implementation. (Cox et al, 1989)

The main disadvantage of flow-based approaches in general is that “Cobweb and Labyrinth problems appear quickly” when modelling a complex system. End users have to “deal with either very complex diagrams or many cross-diagram implicit relationships” (Li et al, 2007).

2.2.5 Programming by Demonstration (Programming by Example)

Programming by Demonstration (PBD) or Programming by Example (PBE) is a technique that allows unskilled users to perform actions interactively to demonstrate the desired behaviour of a system, and programs can then be generalised from the recorded actions (Myers, 1997). PBD approaches have been used to specify behavioural constraints in some systems, often together and most notably in children’s programming environments such as KidSim (Smith et al, 1995) and Agentsheets (Repennig and Sumnet, 1995). Alice (Conway et al, 2000) is an authoring tool for scripting and prototyping 3D object behaviours. Object state and behaviour are specified via user scripts, which are executed to demonstrate how objects respond to user interactions. Figure 2.30 shows the Alice environment that supports programming by demonstration via user interactions.

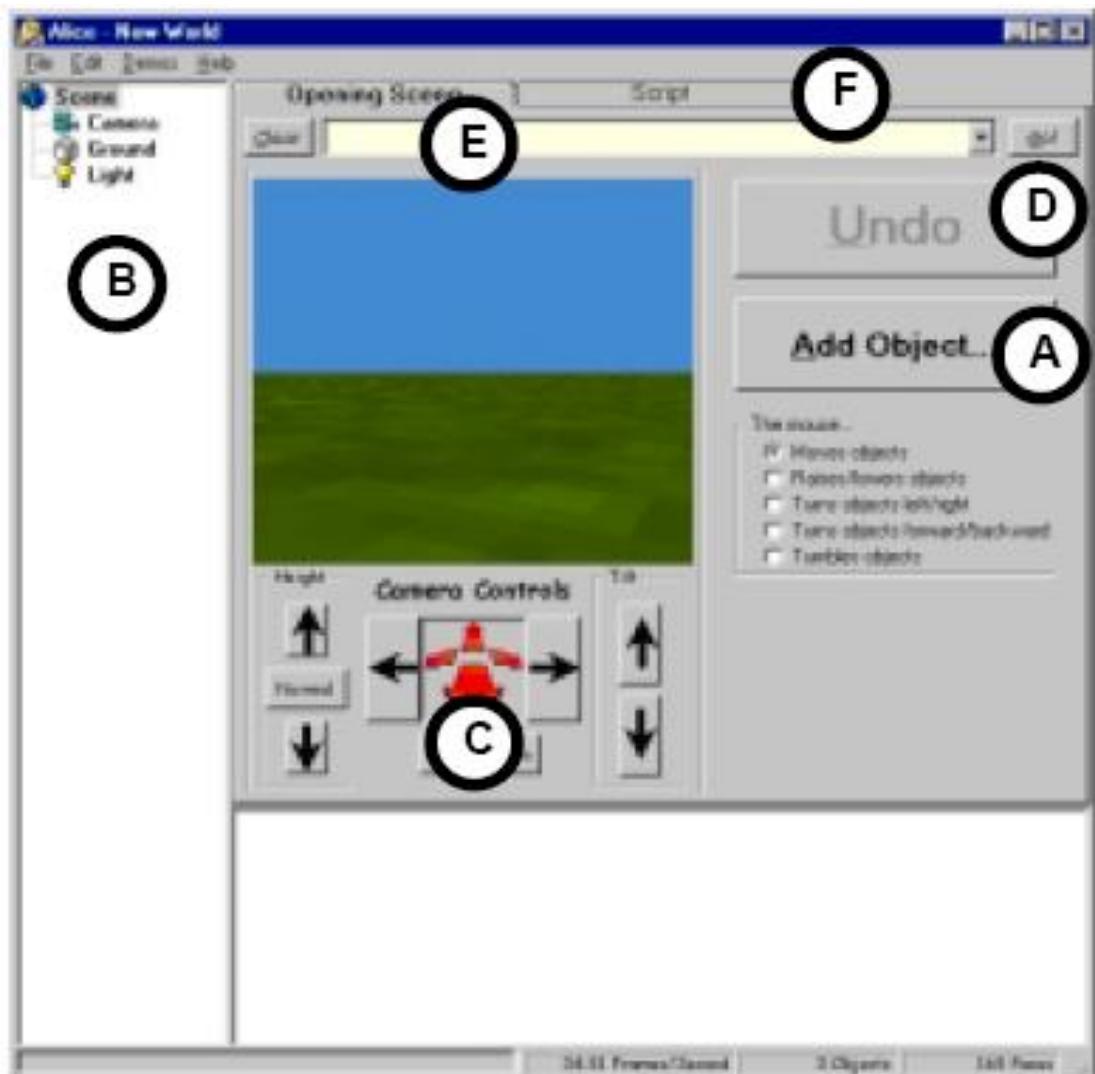


Figure 2.30. The Alice Authoring Environment (opening scene tab). (A) The Add Object button presents a gallery of 3D objects. (B) The Object Tree, a PHIGS-like tree of hierarchical objects (C) Camera controls allow the user to drive around the scene. (D) The Undo button provides infinite animated undo. (E) The Alice Command Box for evaluating single lines of Alice script. (F) The Script tab reveals a simple text editor where the user writes scripts that control the objects in the scene. (Conway et al, 2000)

PBD approaches are generally limited in specification power. They put the burden of programming tasks on the user, which is generally not desirable as the changes made by the user are error-prone and hard to control (Sheth, 1994).

2.2.6 Visual Approaches vs. Textual Approaches

Scripting and declarative approaches normally adopt a textual representation, which generally presents difficulties for system visualisation and debugging. Visual programming languages and

environments (Graphical notations and tools) can be used to simplify the development process. Visual approaches, as opposed to custom code writing/scripting, have advantages such as allowing concrete representations of concepts, minimising design and implementation effort, improving understandability, expressiveness in both static and dynamic specification and visual debugging running event-based systems (Hirakawa and Ichikawa, 1992; Cox et al, 1997; Conway et al, 2000; Burnett et al, 2001; Zhu et al, 2007). Visual languages and environments such as the previously illustrated Serendipity (Grundy and Hosking, 1998) tend to be more readily understandable by end users than many textual, rule-based languages.

It has been a long running debate as to whether pure visual approaches have limitations in expressiveness, while some tasks can simply be expressed in a terse and concise way using text (Schiffer and Frohlich, 1994; Gottfried and Burnett, 1997; Neag and Tyler, 2001; Costagliola et al, 2004). For large complex systems, one single paradigm may not be effective enough to address all aspects or concerns. Combining multiple complementary paradigms in an environment could significantly lift the specification power.

Vista (Schiffer and Frohlich, 1994) is a visual multi-paradigm programming environment that combines object-oriented programming with signal flow and dataflow for the construction of reactive and transformational systems. It avoids visual overload by permitting text input whenever useful. HANDS (Myers et al, 2004) also uses similar approaches.

Forms/3 (Burnett et al, 2001) is a general purpose, declarative form-based language and environment that support procedural abstraction, data abstraction and graphics output in the spreadsheet paradigm. The spreadsheet paradigm provides a declarative approach to programming, characterised by a dependence-driven and direct-manipulation working model. A form in Forms/3 is a collection of cells or groups of cells called matrices and abstraction boxes. The programmer can directly manipulate and define formulae for the cells. A formula is a side-effect-less functional expression that calculates values based on inherent dependencies. Forms/3 combines the use of visual representation of text for formulae and graphics for direct manipulation with concrete, immediate feedback. Programming with events in Forms/3 is no different than any other kind of programming, since events are treated as simple values and vice versa. This allows a visual, high-level approach to event-based programming (Burnett et al, 2001). Figure 2.31 illustrates the language in use.

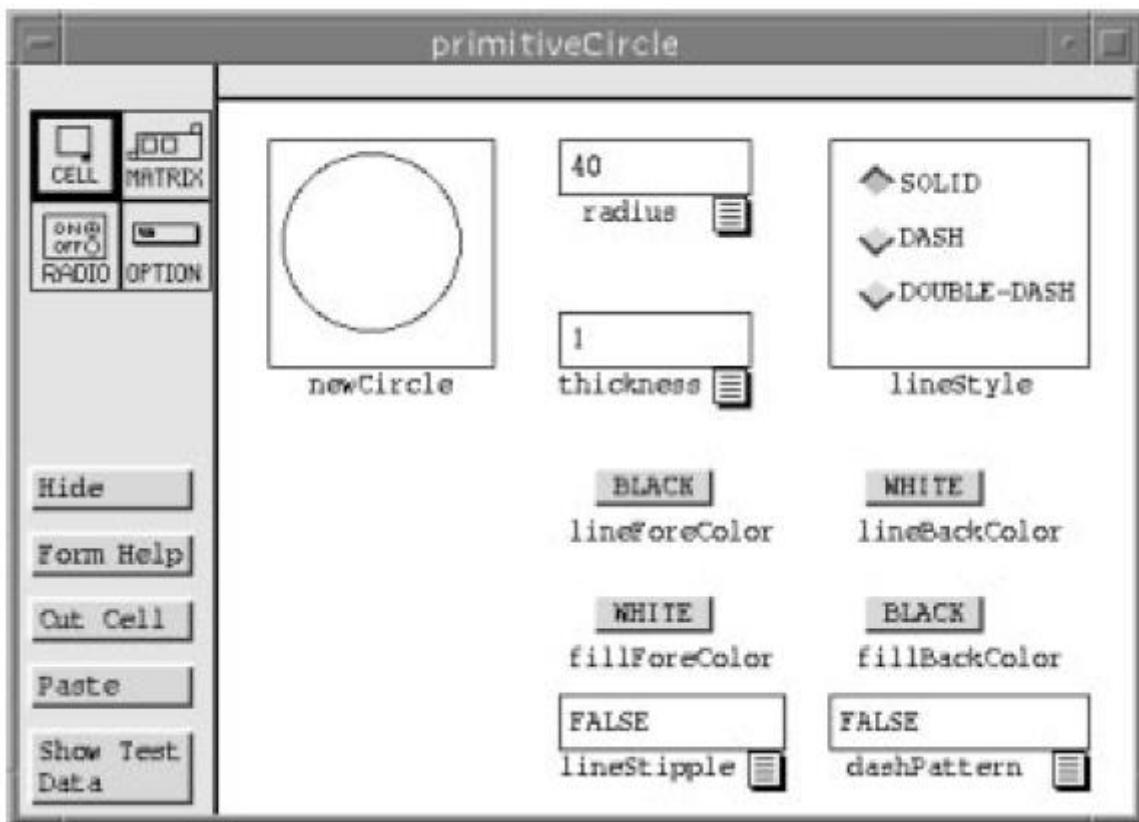


Figure 2.31. A portion of a Forms/3 form (spreadsheet) that defines a primitiveCircle. The primitiveCircle in cell newCircle is specified by the other cells, which define its characteristics. A user can view and specify formulas by clicking on the formula tabs attached to the bottom right of each cell. Radio buttons and popup menus are equivalent to cells with constant formulas. (Burnett et al, 2001)

Combining multiple paradigms such as declarative programming and visual programming in a system specification can expedite the system construction process, given that good usage of each paradigm allows separation of modelling concerns and merging them to achieve a system solution (Burnett and Ambler, 1992). The development process can be less intensive and less error-prone compared to using a single programming model (Jong, 1997).

2.3 General Issues and Requirements

Our research aims to provide a general purpose framework for event handling integration. From an analysis of the above related work, which includes various event handling specification and visualisation techniques in event-based systems, we can derive the strengths and weaknesses of each at a high-level, as illustrated in Table 2.1. Textual/Scripting approaches provide the user with flexible control in extending system behaviours, but are not suitable for end users due to the difficulty of

visualising and understanding their specifications. Declarative semantics of rule-based and constraint-based modelling techniques allow the user to specify the relations between components and declarative approaches usually provide system-level support to automate their execution and maintenance, but runtime visualisation of the behaviour execution is usually suppressed (Jin, 2003). State-based approaches allow easy analysis of system runtime changes and simulations, but system structural details are usually under represented. Flow-based techniques enable high-level specification of inter-communications among system components, but usually fail to present the structural and behavioural details at the same time for runtime visualisation. Program by demonstration approaches focus on dynamic behavioural changes and visualisations, however they are generally limited in specification power. A hybrid visual and textual approach that takes all the advantages of the above individual approaches can potentially possess more capabilities for effectively specifying event-based systems.

Technique	Static structure specification	Dynamic behaviour specification	Runtime visualisation
Textual/Scripting	✗	✗	✗
Declarative	✓	✓	✗
State-based	✗	✓	✓
Flow-based	✓	✓	✗
Program by demonstration	✗	✓	✓
Hybrid visual and textual	✓	✓	✓

Table 2.1. Comparison of event handling specification and visualisation techniques

The issues in the above explored specification and visualisation approaches to event-based system integration need to be addressed properly in our work, in order to provide a feasible event handling framework for a wide range of event integration support. This led to the following set of requirements for our event handling integration framework:

- An expressive language that describes both the static structure and dynamic behaviours of event-based systems is needed. The language should have the ability to compose building blocks and models for reuse.
- Collaborative use of textual and visual notations in multiple views of different levels of abstraction should be allowed to accommodate the specification needs. The framework

should have the advantage of ease of use for novice users but also expressive power for experienced tool developers (Grundy et al, 1998).

- A canonical event handling model that is easy to extend is required as the fundamental framework infrastructure.
- The framework should allow users to add event propagation and event handling behaviour to running tools in a fast and easy way.
- Environment or software tool support for specifying event handling, tracing event propagations and visualising results using graphical structures is needed. The visualisations need to incorporate both the static system dependency structure and the dynamic event handling behaviour of running event-based systems.
- Event-based system executions are highly time related, and many phenomena may occur in a very short time (and conversely, there can be large gaps between clusters of events). A real-time visualisation would be too fast as it may prevent users from actually seeing all the phenomena he/she is interested in (Coupaye et al, 1999). Step-by-step visualisation that is interactively controlled by the user is thus required.

2.4 Summary

The aim of this research is to explore existing event-based systems for their event handling and visualisation support and then design a general purpose event handling framework. In this chapter, we conducted a literature review of related work in this field, which includes event handling specification techniques and visualisation support for design and construction of event-based systems. We have compared the existing approaches and identified the strengths and weaknesses of each together with general issues relevant to the event-based programming paradigm. These form the basis for our further research development to achieve our research goal.

Chapter 3 - Motivation

This chapter describes the motivation of this thesis. We start with an introduction to the Pounamu (Zhu et al, 2007) project, and then describe various evaluations performed on Pounamu. A key feature of Pounamu is the use of “event handlers” to allow tool developers to specify handling of various events for user interfaces and for models. The current event handler definer in Pounamu requires the user to write Java code fragments to compose the handler. The user has to be familiar with the Pounamu API to complete even a simple event handler specification task. One of our primary motivations for this research was to replace such textual Java code scripts with more appropriate and powerful visual tool support for event handler specification.

3.1 Overview of Pounamu

Domain-specific visual language tools have become important in many domains of software engineering and end user development. However building such tools is very challenging with a need for multiple views of information and multi-user support, the ability for users to change tool diagram and metamodel specifications while in use, and a need for an open architecture for tool integration. Pounamu is a meta-tool for realising such visual design environments (Zhu et al, 2007).

Figure 3.1 shows the main components of Pounamu. Users initially specify a meta-description of the desired tool via a set of visual specification tools. These define:

- The appearance of visual language notation components, via the “Shape Designer”, which has shape and connector variants;
- Views for graphical display and editing of information, via the “View Designer”;
- The tool’s underlying information model as metamodel types, via the “Metamodel Designer”; and
- Event handlers to define behaviour semantics, via the “Event Handler Designer”, which has textual variants.

Tool projects are used to group individual tool specifications.

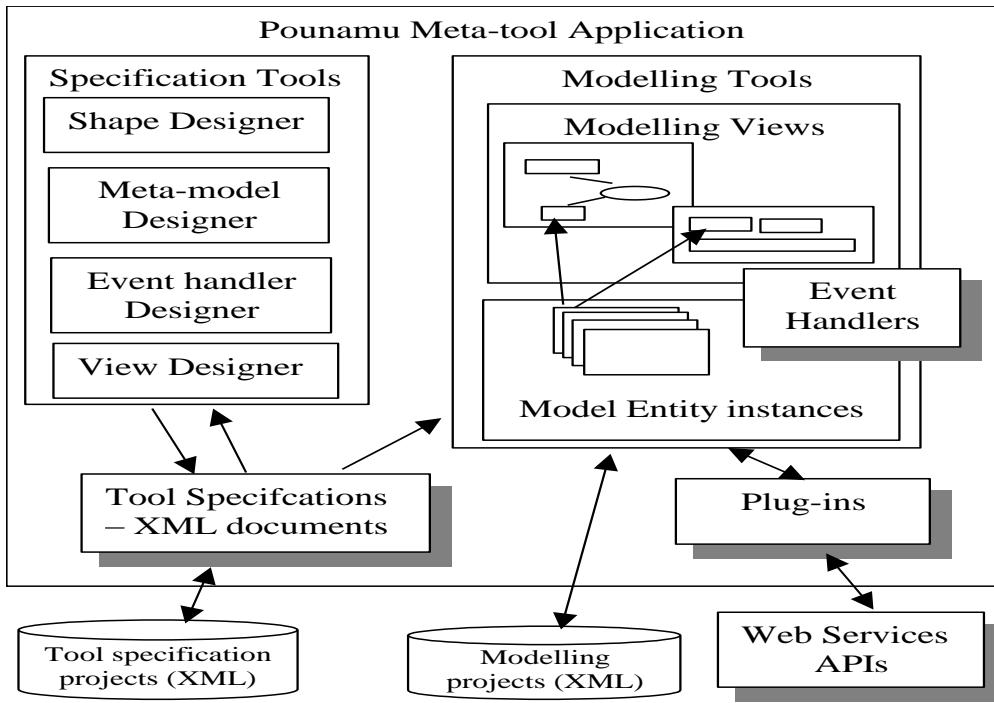


Figure 3.1. The Pounamu approach. (Zhu et al, 2007)

Having specified a tool or obtained someone else's tool project specification, users can create multiple project models associated with that tool. Modelling tools allow users to create modelling projects, modelling views and edit view shapes, updating model entities. Pounamu uses an XML representation of all tool specification and model data, which can be stored in files, a database or a remote version control tool. Pounamu provides a full web services-based API which can be used to integrate the tool with other tools, or to remotely drive the tool.

3.1.1 Tool Specification

Figure 3.2 (a) shows an example of the Pounamu shape designer in use. On the left a hierarchical view provides access to tool specification components and models instantiated for that tool. In the centre are visual editing windows for displaying tool specification components and model instances. Here, a shape is defined representing a generic UML class icon. To the right is a property editing panel supplementing the visual editing window. General information is provided in a panel at the bottom.

Figure 3.2 (b) shows a UML class diagramming tool, which uses the shape icon defined in Figure 3.2 (a) to model a “person” class, and two subclasses “student” and “staff”. The same shape specification could be reused for other modelling tools associated with the same (e.g. a class element in a package diagram) or different metamodel elements.

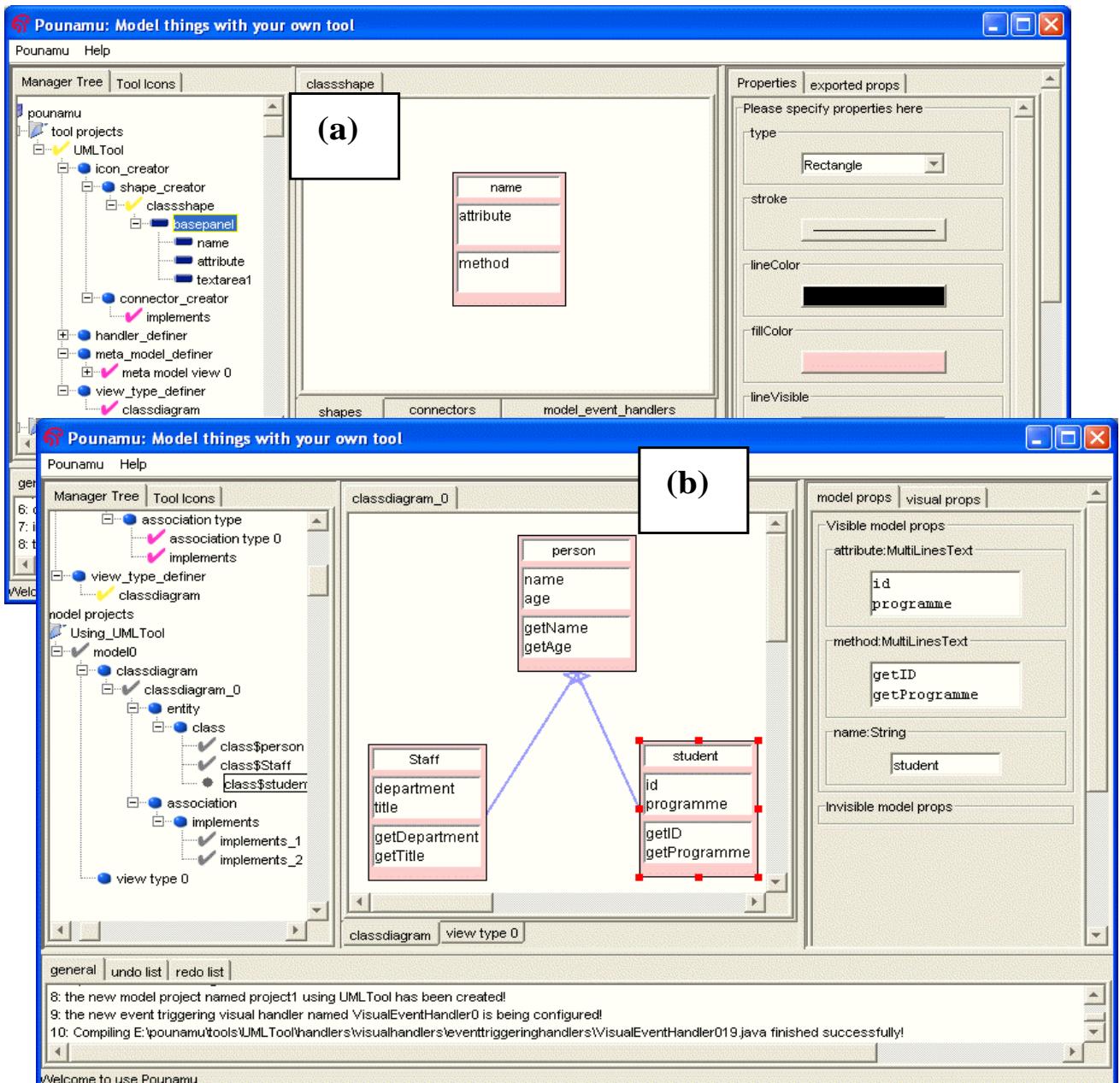


Figure 3.2. Pounamu in use: (a) specification of a visual notation shape element and (b) modelling using this shape in a UML class diagram tool. (Zhu et al, 2007)

The Shape Designer allows visual elements (generalised icons) to be defined. These consist of Java Swing panels, with embedded sub-shapes, such as labels, single or multi-line editable text fields (with formatting), layout managers, geometric shapes, images, borders, etc. For example, the icon in Figure 3.2 (a) consists of a bordered, filled rectangular panel, with three sub-shapes, a single line textfield for the name, and two multi-line textfields for the attribute and operation parts of the class icon. The property sheet pane (right) allows names and formatting information to be specified for each shape component. Fields that are to be exposed to the underlying information model are also

specified using a property sheet tab. Form-based interface can also be defined by using a single shape specification defining the whole form.

The Connector Designer allows specification of inter-shape connectors, such as the UML generalisation connector shown in Figure 3.3. The tool permits specification of line format, end shapes, and labels or edit fields associated with the connector's ends or centre.

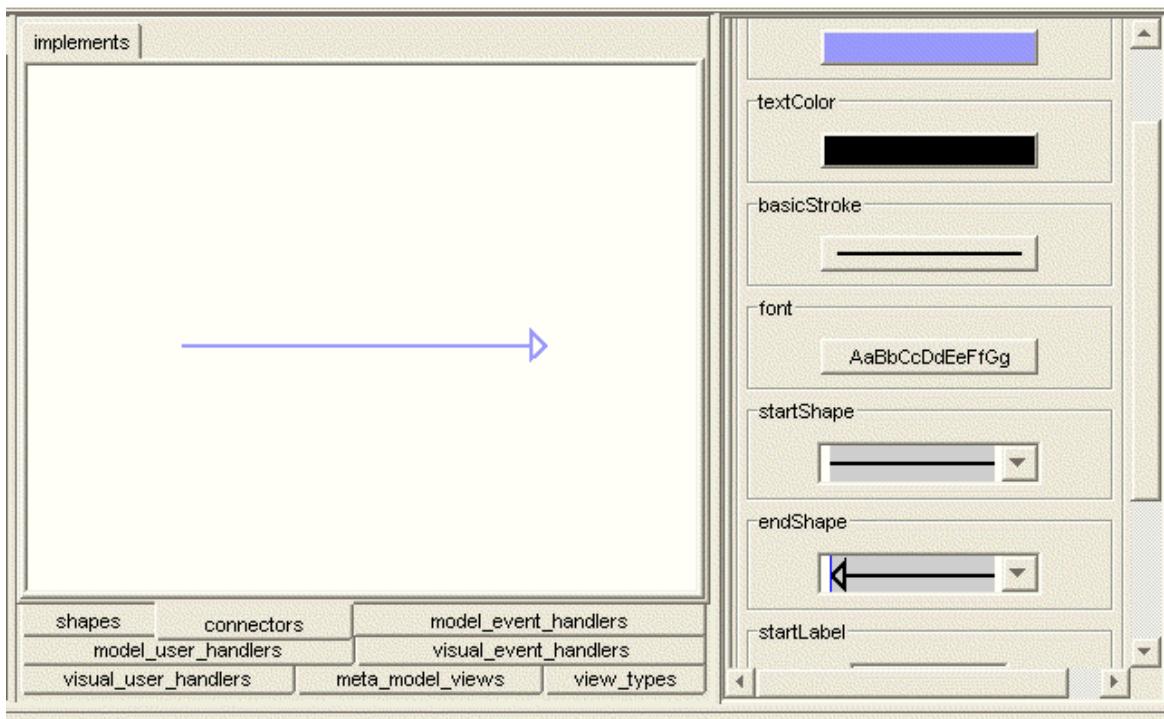


Figure 3.3. Example of the Connector designer. (Zhu et al, 2007)

The underlying tool information model is specified using the *metamodel designer*, shown in Figure 3.4. This uses an Extended Entity Relationship (EER) model as its representational metaphor. This was chosen because the representation is simple and hence accessible to a wide range of users. For example, the metamodel in Figure 3.4 contains two entities representing a UML class and UML object, each with properties for their names attributes and methods, class type etc. An “instanceOf” association links class and object entities and an “implements” association links classes. The metamodel tool supports multiple views of the metamodel, allowing complex metamodels to be presented in manageable segments.

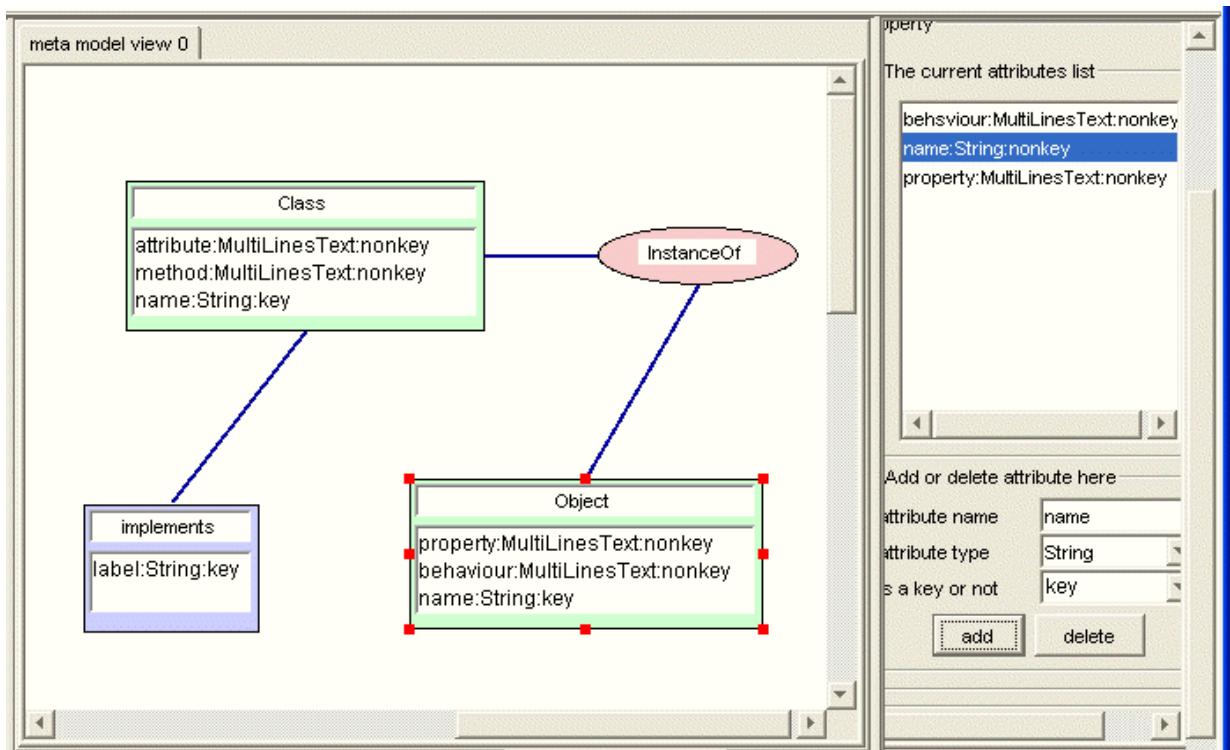


Figure 3.4. Example of the metamodel designer. (Zhu et al, 2007)

The View Designer, shown in Figure 3.5, is used to define a visual editor and its mapping to the underlying information model. Each view type consists of the shape and connector types that are allowed in that view type, together with a mapping from each such element to corresponding metamodel element types. Menus and property sheets for the view editor and view shapes can also be customised using this tool. For example, Figure 3.5 shows the specification of a simple UML class diagramming tool, consisting of UML class icon shapes, and generalisation connectors. Figure 3.5 shows that the “classshape” icon maps to the *class* metamodel entity type, and their selected properties map as shown. Mappings supported in this tool are limited to simple 1-1 mappings of elements (single or multi-valued) between view instance and information model instance. More complex mappings can be specified using event handlers as described below. Multiple view types can be defined, each mapping to a common information model. For example, other view types for sequence diagrams or package diagrams can be defined for the simple UML tool.

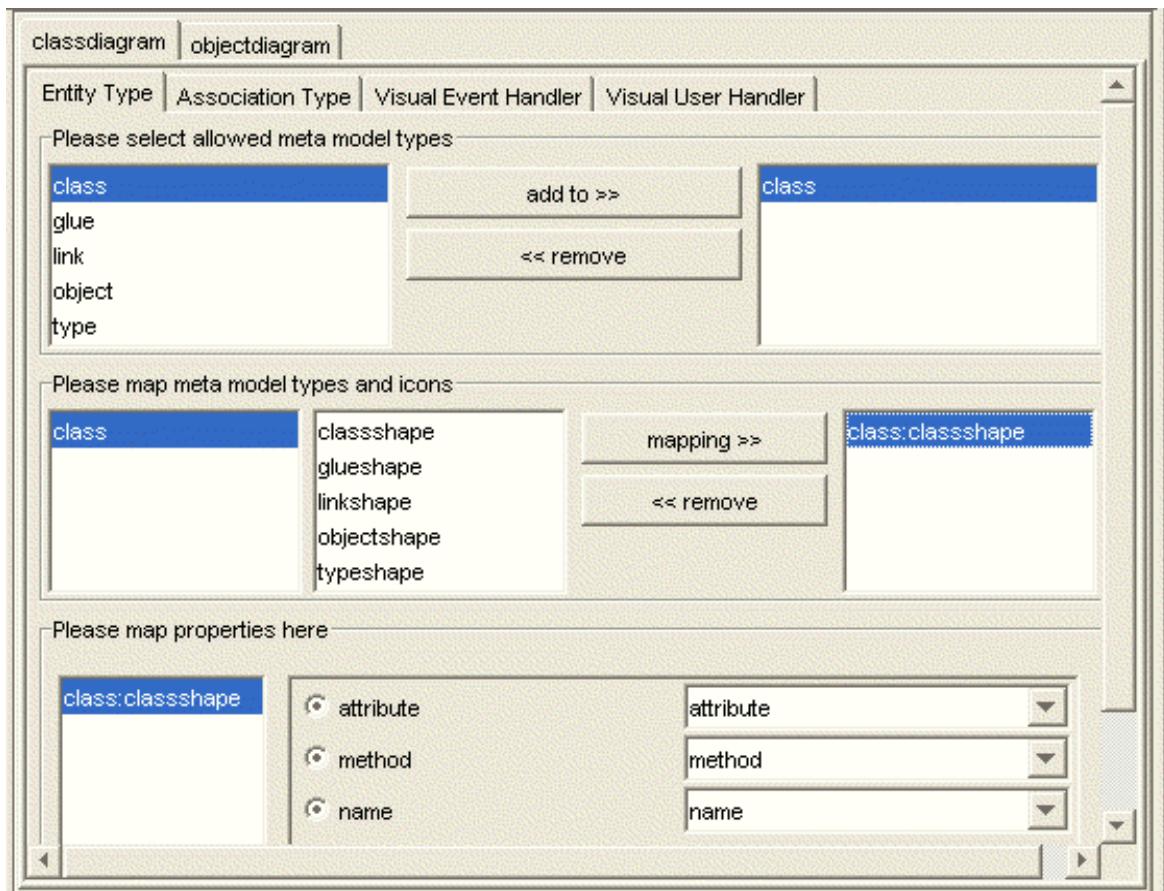


Figure 3.5. Example of the view designer. (Zhu et al, 2007)

The Event Handler Designer is used to define complex behaviour to a tool using an Event-Condition-Action (ECA) model (Kiringa, 2002; Liu et al, 2005). It allows tool designers to choose predefined event handlers from a library or to write and dynamically add new ones as Java plug-in components. Event handlers can be used to add:

- view editing behaviour e.g. “if shape X is moved, move shape Y the same amount”;
- view and model constraints e.g. “all instances of entity Z must have a unique Name property”;
- user-defined events e.g. “check model is consistent when user clicks button”;
- event-driven extensions e.g. “generate C# code from the design model instance information”; and
- environment extension plug-ins e.g. “initialise the collaboration plug-in to support synchronous editing of a shared Pounamu diagram by multiple users”.

Event handler code must be developed using Java. A comprehensive API provides access to the underlying Pounamu modelling tool representation, permitting complex querying and manipulation of tool data. A simple example of an event handler being developed is shown in Figure 3.6. Event handler code is compiled on the fly as the tool is specified or when a tool project is opened.

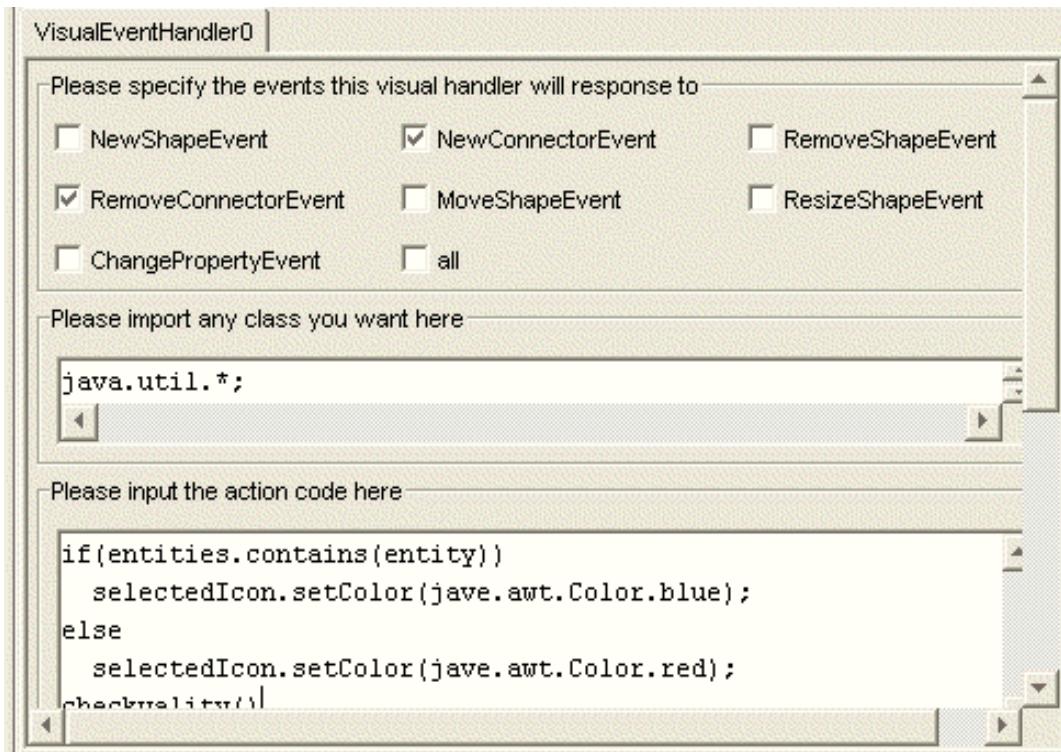


Figure 3.6. Example of the event handler designer. (Zhu et al, 2007)

3.1.2 Tool Usage

Pounamu automatically and incrementally implements tools as they are specified using the Pounamu metatools. This means tools may be tested and evaluated incrementally as they are being developed, avoiding the compile cycle issues noted earlier and creating a live environment. Generation of the tool happens automatically and immediately following specification of any view editor associated with the tool or when a saved tool project is opened. This provides powerful support for rapid prototyping and evolutionary tool development. Changes to a tool specification may, of course, result in information creation or loss in the open or saved modelling projects e.g. when adding or deleting properties or types. Users can create model views using any of the specified view editors. Reuse is supported by allowing shapes, connectors, metamodel elements, and event handlers to be easily imported from other tools or libraries. Multiple tool specification projects may be open when modelling, with specification of parts of the modelling tool coming from different tool specification projects, supporting layered tool development.

Each view editor provides an editing environment for diagrams using the shapes and connectors it supports. Consistency between multiple views is implicitly supported via the view mapping process with no programming required to achieve this, unless complex mappings are required that need event handlers to implement them.

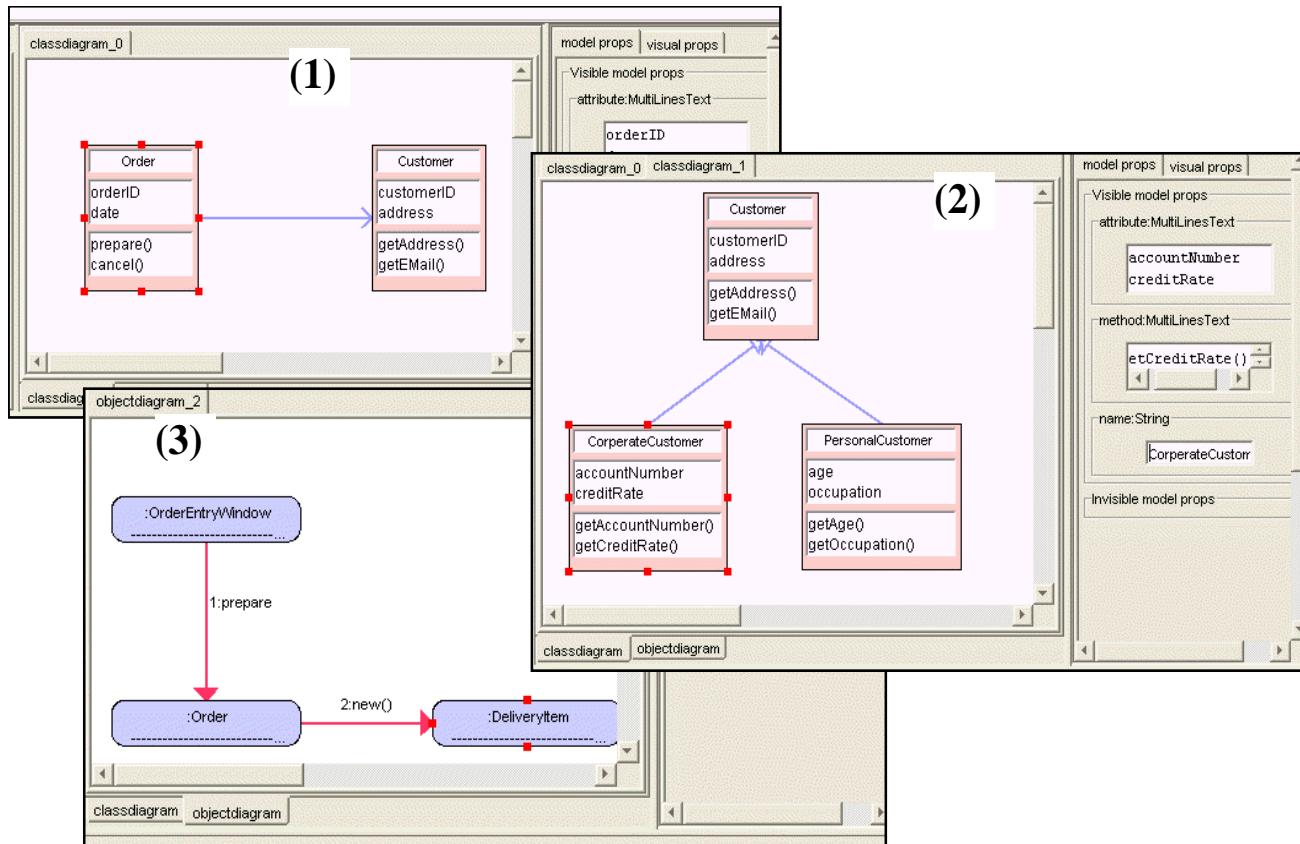


Figure 3.7. Example modelling tool usage. (Zhu et al, 2007)

Figure 3.7 shows the simple UML class diagramming tool in use. View (1) shows a simple class diagram. The user has created a class diagram view from the available view types, added two UML class shapes and an association connector, and set various properties for these, including their location and size. View (2) shows another class diagram included in the same project model, reusing the “Customer” class information. Changes to either view, e.g. addition of a method or change of the class name, are reflected through to the other view. View (3) shows a simplified object diagram view, including an object of class “Order”. Changes to the class name are automatically reflected in this view and only methods defined or inherited by a class may be used in the message calling. The latter is controlled by event handlers managing the more complex consistency requirements.

Having defined a simple tool, and experimented with its notation, additional behaviour can be incrementally added using event handlers to implement more complex constraints. Examples include:

- type checking, e.g. UML associations must be between classes;
- model constraints, e.g. UML class attributes must have unique names for the same class;
- layout constraints and behaviour, e.g. auto-layout of a UML sequence diagram view when edited;

- more complex mappings, e.g. changes to class shape method names automatically modifying method entity properties in the modelling tool information model; or
- back end functionality, e.g. generating C# skeleton code from model instances.

These handlers can be generic for reuse (e.g. a generic horizontal alignment handler) or specific to the tool. As with other meta specification components, adding or modifying a handler results in “on the fly” compilation of handler code and incorporation of that code into any executing tool instances.

As noted above, back end functionality can be implemented by event handlers. In addition, as all tool and model components are represented in XML format, it is straightforward to implement back end processing using XSLT or other XML-based transformation tools. This approach can allow back ends to be developed independently of the editing environment enhancing modularisation. An additional approach for implementing back end functionality is via Pounamu’s web services-based API. This exposes Pounamu’s model representation, modelling commands, menu extension capability, etc, permitting tight and dynamic integration of third party tools, and other Pounamu environments. This API has been used, for example, to implement peer to peer based synchronous and asynchronous collaboration support between multiple Pounamu environments, to implement generic GIF and SVG web-based thin client interfaces, to implement interfaces for mobile device deployment, and to integrate a Pounamu based process modelling tool with a process enactment engine (Zhu et al, 2007).

A wide range of exemplar domain-specific visual language tools have been developed with Pounamu, some of which are illustrated in Figure 3.8. These include (Zhu et al, 2007):

- A full UML tool supporting all UML diagram types. This also provides an import/export facility using the XML Model Interchange (XMI) standard allowing models to be imported from and exported to other XMI-compliant UML tools. A code generator takes XMI models from Pounamu and generates Java code that can be further extended by a programming environment.
- A circuit design tool (Figure 3.8 (a)) providing a CAD-like tool for circuit design.
- A statistical survey design tool SDLTool (Figure 3.8 (b)). SDLTool provides multiple views describing statistical processes, data and analysis steps. This is used by statisticians to design, enact and process complex statistical surveys.

- A software process modelling and enactment tool IMAL (Figure 3.8 (c)). IMAL provides multiple users multiple workflow modelling diagrams. These software process models can be enacted by the tool to help support work co-ordination by multiple developers. External tools are invoked via Pounamu's web service support to provide complex rule processing and XML document display and update. Pounamu itself is invoked by a workflow engine via Pounamu's web services API to support dynamic visualisation of enacted work processes.

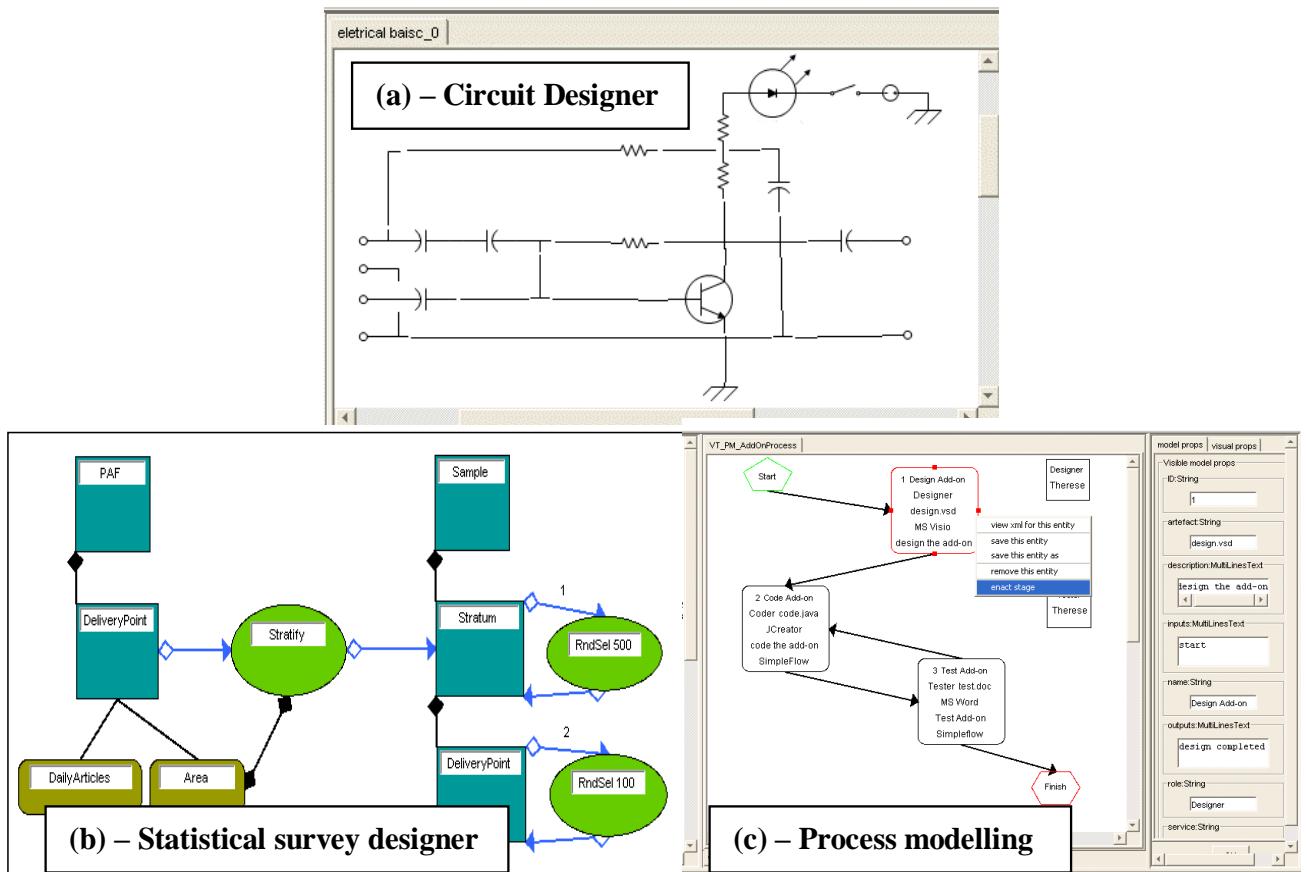


Figure 3.8. Examples of Pounamu DSVL tools. (Zhu et al, 2007)

Pounamu has also been used as a rapid prototyping tool in a range of industrial applications to assist in the design of visual notations and interfaces for client companies. These applications include: a business form designer; a business enterprise modelling tool; a project management tool with Gantt and work breakdown schedule views; and a web services composition tool. In each case the client companies were able to rapidly explore and evaluate a variety of alternative notational approaches with a low level of investment, hence allowing them to lower the risk of development.

3.2 Evaluations of Pounamu

Evaluating a metatool such as Pounamu is not a straightforward task due to the multiple points of view involved (tool developer, end user of developed tool, usability, utility, etc). As a starting point for the research in this thesis, we undertook an evaluation of Pounamu to estimate its strengths and weaknesses. Our approach has been to evaluate Pounamu at several levels and through a variety of mechanisms. These include:

1. Two large group experiments, spaced nearly two years apart, where participants (approximately 45 in each case) constructed a domain specific visual language tool and were then surveyed. Our aim in each case was to use the feedback to help improve the tool, and significant enhancement was undertaken between the two experiments.
2. Qualitative feedback, in the form of experience reports, from a smaller number of developers who used Pounamu to develop more substantial applications, such as the ones in Figure 3.8. These were used to assess whether perceptions altered as more substantial applications (with, for example, more complex back end integration requirements) were developed.
3. Small end user and cognitive dimensions evaluations (undertaken by the developers) of substantial applications developed using Pounamu. These were used to evaluate whether end users found Pounamu generated tools to have good usability characteristics. Many of these have also been reported in detail elsewhere.

3.2.1 Large Group Experiments

In each experiment around 45 participants, who were graduate-level students, were asked to construct a domain-specific visual language tool of their own choosing, but with at least a minimal set of required components, such as numbers of icons, views, handlers, etc so that tools with a realistic level of complexity were designed and constructed. Participants were given two weeks elapsed time (i.e. alongside other obligations) to complete application development; they were then surveyed using a set of open ended questions to qualitatively elicit strengths and weaknesses of Pounamu to construct the desired domain-specific visual language tool. The surveys, one undertaken in August 2004 (46 participants) and the second in May 2006 (45 participants), emphasised elicitation of weaknesses as their primary intention was to provide feedback to be used in improving Pounamu, hence the responses observed tended to describe generic strengths of Pounamu, with more detail on specific weaknesses.

Generic strengths emphasised by respondents in both surveys included: the rapidity with which tools were able to be constructed; the extensibility and customisability of the generated tools; the low learning curve needed to use Pounamu effectively; and the usefulness of being able to update tool definitions on the fly as iterative development was undertaken.

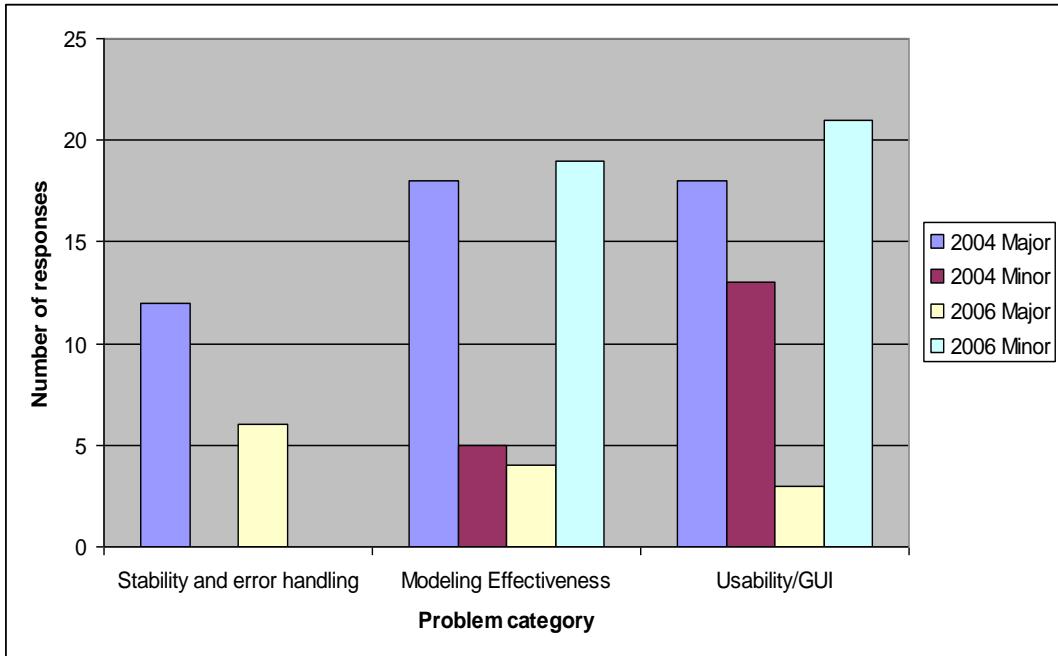


Figure 3.9: Problems identified in 2004 and 2006 surveys

Figure 3.9 charts the number of responses concerning identified weaknesses in each survey aggregated into three categories (classified based on the commonality of the open ended survey data) and subcategorised as “major”, i.e. a significant weakness, or “minor”, i.e. an issue causing irritation but not significantly affecting functionality. General weaknesses identified in the first experiment focused on issues of: stability of the software; inconsistency of the interface compared to other tools; lack of documentation, particularly for the API; difficulty of event handler specification; and weak error handling, all of which might be expected of the, at that stage, proof of concept prototype. Issues of stability, documentation, event handler specification and error handling were largely addressed in changes made to the system before the second experiment, which showed much less concern by participants on these issues. For each category the number of major problems identified was significantly lower in the second survey. Many minor usability issues were also identified by participants, such as the size of text fields for entering Java code for event handlers, some clumsiness around specification of icons, and the response time for some elements of functionality. The second experiment showed increased concern by participants in such issues, primarily, we speculate,

because a lack of concern over major issues permitted them to focus more readily on more minor limitations of the meta-tool. Most issues identified concerned usability of the tool specification components, with very few issues concerns with the usability or efficacy of the generated tools themselves, nor of the representational power of the metamodel.

A key feature for us is the efficacy of our evaluation-cycle based quality improvement approach which has demonstrated significantly enhanced user perception of Pounamu following attention to issues raised in the first large-scale user evaluation.

3.2.2 Developers of Large Applications

A smaller group of 8 developers have used Pounamu to develop more substantial applications, typically as an element of a larger research project, and over an extended period (several months at least). In each case, the developers provided detailed comments on the efficacy of Pounamu as part of a wider ranging implementation report. These qualitative comments were summarised and categorised in a similar manner to the large experiments. The small number of participants makes for less depth in the results, however, they are sufficient to assess any significant change in perception with increased application size. The general strengths identified were the same as for the large group experiments, but with more emphasis on the speed of development and the extendibility and customisability of the generated tools. Far fewer weaknesses were identified; familiarity with Pounamu appears to have mitigated many of minor difficulties identified in the large group experiments. Some issues around stability and performance were identified by those using Pounamu in its early stage of development, in common with the first group experiment, but those using later versions did not report the same issues. In summary, our results suggest that developers using Pounamu to construct larger applications are somewhat more favourably inclined than those developing small applications as in the former case the benefits of efficiency of construction significantly outweigh minor usability issues.

3.2.3 Usability of Substantial Tools Constructed Using Pounamu

For many of the exemplar systems described earlier (and others) the developers have carried out a combination of survey-based end user evaluations of the application visual language environments and cognitive dimensions-based evaluations of the visual environment interaction and information presentation features. These evaluations have each been reported as described in (Zhu et al, 2007).

3.3 Analysis of Evaluation Results

The above evaluations implicated that while specifying static modelling structure using Pounamu saves a huge amount of time, specifying behavioural extension, particularly for more complex modelling tools, adds an overwelcoming cost overhead.

The event handler specification for Pounamu provided a good vehicle for adding dynamic modelling behaviour such as model/diagram constraints, but at the same time, it required Pounamu users to be familiar with Pounamu APIs as well as Java APIs. Without thorough understanding of these, even simple event handlers can be difficult to implement. Both the learning time and the requirements of users' programming competency are not desirable, since Pounamu's target end users are ideally not intended to be programmers.

Pounamu supported a limited set of built-in events including model semantic events and low-level visual events. It should facilitate customised event definitions to be integrated together with the built-in events, such as user-defined call, change, signal and time events.

A library event handler could be added to a tool in an “all-or-nothing” style. Pounamu did not support reuse of event handler code modules. It should support parameterised code modules, as well as allow expert users to add new condition and action building blocks to Pounamu's event handler library for reuse in defining complex event handlers.

It was hard to track and debug the event handling behaviour in Pounamu as there was no utility to support step-by-step visualisation of program flow, nor an indication of affected state changes. Providing mechanisms for debugging the handler code was a highly prioritised task.

Also many Pounamu tools and extensions include some common building blocks for event handling specification. These could be generalised from a range of event-based applications to a common model representation.

To eliminate the need of non-trivial programming, we propose using visual approaches to specify event handlers. A visual event handler definer would provide a high-level visual specification for building both simple and complex event handling functionality for Pounamu tools from a set or

reusable building blocks, and thus reduce the need for end users to use the complex API-dependent Java coding of constraints for tools.

3.4 Summary

We have described Pounamu, a meta-tool for specifying and implementing multiple-view multiple-notation diagramming tools. Pounamu has been used to develop a broad range of domain-specific visual language applications both for academic/research use and industrial purposes and over a large end user base. We have collected the evaluation results of Pounamu and its generated tools. The most major concern for Pounamu users, particularly those developing large Pounamu tool applications, was event handler specification. This was because it requires using complex API-dependent Java coding which is not suitable for our target end users. This has motivated the work described in the remainder of this thesis, which investigates various visual metaphors for event integration specification.

Chapter 4 - Overview of Our Approach

This chapter describes our approach to designing and prototyping a generic event integration framework. This involves developing three exemplar notations/tools and generalising from them to develop a new, generic visual event handling metaphor.

4.1 Introduction

Our approach is to adopt the Three Examples pattern from the Evolving Frameworks Pattern Language (Roberts and Johnson, 1996) as a path to developing a generic event integration specification framework. This pattern suggests choosing two applications that are similar and one that is a little removed. The examples should be explored in either succession or parallel and from which common reusable abstractions should be identified. We have explored three limited-domain exemplars in succession, in which the first two both use a dataflow like icon and connector approach to event handling specification, whereas the third is quite different as it uses a declarative approach. A general metamodel representation that combines atomic primitives (either shared or non-shared) extended by the three examples is then defined.

One increasingly common example event-driven problem domain is web services composition. Web services have become a popular technology for building distributed systems, but there is a lack of languages and tools to specify web service compositions at high abstraction levels, generate lower-level executable process code such as BPEL4WS (IBM, 2003), and visualise, at high abstraction levels, running web services. Most approaches provide basic flow-like BPEL4WS editors or similar (Srivastava and Koehler, 2003; Thone et al, 2002). More abstract approaches (Fensel and Bussler, 2002; Foster et al, 2003) only support limited compositional approaches or do not support generation of BPEL4WS or similar executable forms. We describe a new approach for complex web service composition using a high-level metaphor and visual language called ViTABaL-WS (Liu et al, 2007). This supports higher level design views for service composition that are complementary to current web services composition standards.

The second problem domain that we focus on is visual design tools. These tools have many applications, including software design, engineering product design, E-learning, data visualisation, and tourism. Pounamu (Zhu et al, 2007) is a metatool for building such visual design tools. It incorporates high-level visual specifications of tool metamodels and visual language notations allowing end users to modify aspects of their tools such as appearance of icons and view compositions. However, commonly end users also wish to modify tool behaviour (Morch, 1998; Peltonen, 2000) to specify editing constraints, automated diagram modification, semantic constraints, and computation. Several approaches have been used to support reconfiguration of diagramming tools, including direct modification via an API (Kelly et al, 1996), scripting (Myers, 1997), programming by demonstration (Smith et al, 1995), and Event-Condition-Action rule based languages (Costagliola et al, 2002; Ledeczi et al, 2001). Pounamu currently uses the first approach. Many end users of such tools are not programmers and do not wish to learn or use textual, programmatic scripting languages to tailor their design tools.

Most visual design tools are “event driven”, i.e. when a user modifies a diagram, events are generated and can be acted upon to modify other diagram content, enforce constraints, etc. We have used the event-driven nature of such tools as the basis for an end user domain specific visual language, Kaitiaki (Liu et al, 2005), with which to express both simple and complex event handling mechanisms via visual specifications for their diagramming tools. These include event filtering, tool state querying and action invocation.

MaramaTatau (Liu et al, 2007), our third exemplar, uses a more declarative approach to extend behaviour specification of visual design tools. The focus is to better model relationships in a tool’s metamodel definition. This includes constraining relationships via connector types mapping and multiplicities, and specifying formulae for calculating property values and enforcing constraints. Formula construction is similar to a spreadsheet but expressed at a type rather than instance level. Formulae are all interpreted as one way constraints with Java event handler code generated and realised at a model instance level. Error and to-do list critics provide notification to the user. Visualisations of formula effects are achieved via runtime visual debugging and master-details tabular model instances data views.

Based on the in-depth exploration of the three preceding visual event-based metaphors, our overall aim is to generalise to a metaphor and a language/framework that can provide support for generic event integration specification. The generalised approach should incorporate compositional

primitives as building blocks together with a variety of different communication relationships between them. It also should contain mapping/integration schemes to support interchange between the three approaches.

As stated above, our aim is to generalise from three exemplars to produce a generic event handling specification visual language and supporting environment. In the remainder of this chapter we provide an overview of each exemplar before over-viewing directions for their generalisation.

4.2 ViTABaL-WS

Our aim for the first exemplar was to develop a metaphor to effectively describe the composition of web services and support the development of a visual language and modeling environment. The web services compositional relationships can be very complex and a range of compositional building blocks are required. We chose to use the “Tool Abstraction” (TA) paradigm (Grundy et al, 1995) as our metaphor for web service compositions and to support reasoning about different relationships between compositional primitives. The TA paradigm is a message propagation-centric approach describing interconnections between “toolies” (the encapsulation of functions) and “abstract data structures” (ADSs: the encapsulation of data) which are instances of “abstract data type” (ADTs: typed operations/messages/ events). Connection of toolies to other toolies and ADSs is via typed ports. The TA paradigm supports modeling data flow, control flow and event flow relationships. Reusability, extensibility and expressiveness are key advantages possessed by TA.

ViTABaL (Grundy et al, 1995) is a hybrid visual programming environment for designing and implementing TA-based systems. It uses the TA paradigm to compose systems by integrating, and coordinating toolies and ADS components. TA paradigm appeared to us to be well suited for the web services composition domain by permitting specification of an abstract model involving a series of coordinated invocations to web services operations. Accordingly we adapted this earlier work to develop a new visual language and environment, ViTABaL-WS, which specialises the ViTABaL visual composition language to the domain of web services composition. It supports modelling of both event-dependency and dataflow in designing complex web service compositions. Figure 4.1 is a ViTABaL-WS diagram illustrating examples of compositional primitives in the Tool Abstraction paradigm. Toolies (web services, shaded green ovals) encapsulate data processing and interact with each other through both direct and indirect operational invocations using shared data structures (message ADT instances: rectangular, shaded icons); and event-driven dependencies indicating state

changes to a Data Store ADS (data storage service). A system of typed input and output ports on toolie and ADS services provide message sources and sinks. Services are wired together using these ports with ports supporting only certain kinds of connection and message ADTs. Messages generated by a service output port are distributed to connected web service input ports. Many interconnection schemes are supported including one-way flow, request-response, asynchronous flow, and subscribe-notify. Additional controls support conditional flow, dynamic type checking, synchronisation, iteration etc.

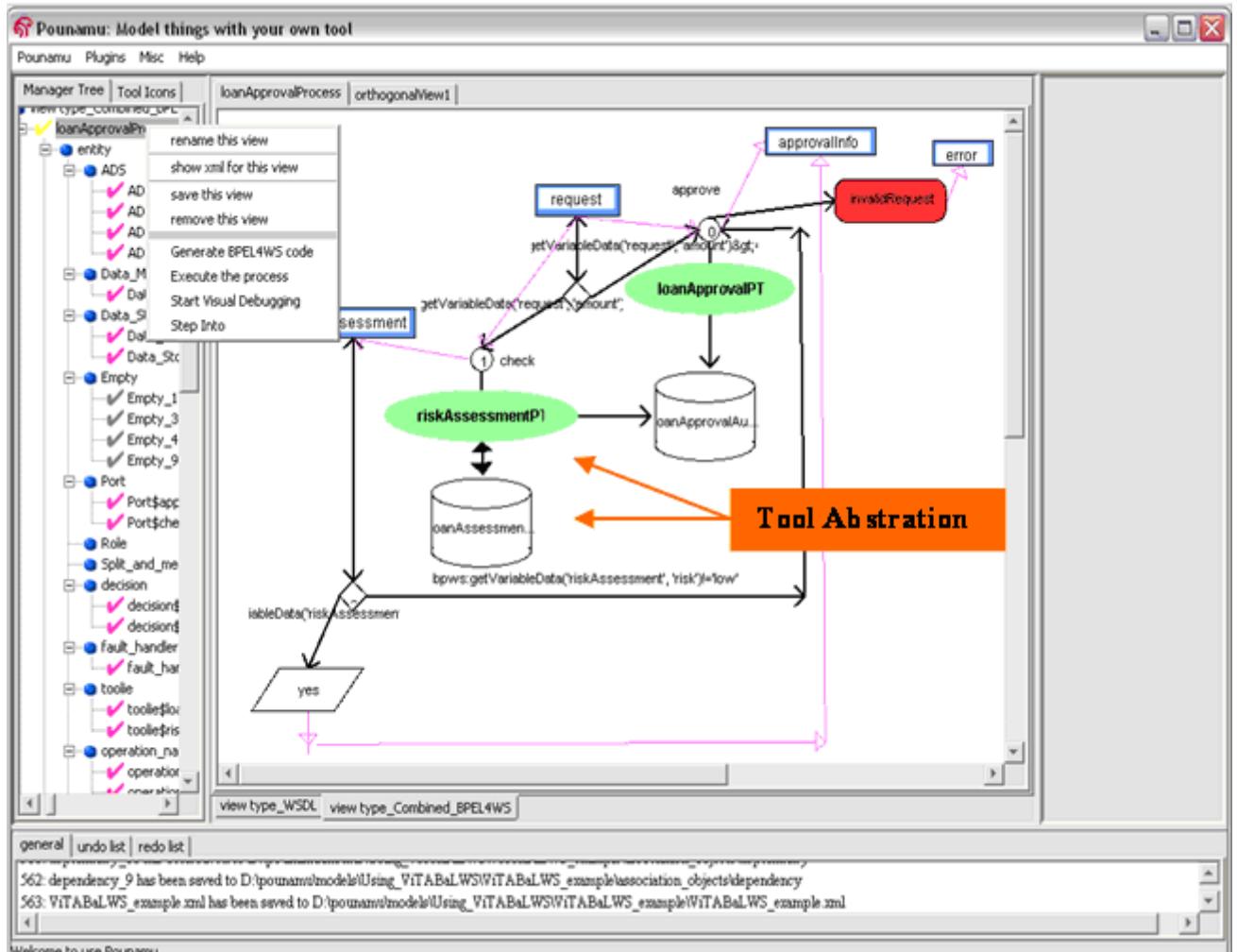


Figure 4.1. ViTABaL-WS editing in Pounamu.

The specified web services are linked together by composition rules enforced in the ViTABaL-WS tool. ViTABaL-WS supports generation of WSDL and BPEL4WS from its abstract composition model. We use the Business Process Execution Language for Web Service Java Run Time (BPWS4J) as the deployment engine for generated BPEL4WS processes. A deployed process is provided with a SOAP interface and a WSDL file, and thus can be invoked by a requesting web

service client. BPWS4J is tightly integrated with ViTABaL-WS: a ViTABaL-WS process can be directly deployed and step-by-step visualisation of its execution can be obtained, with running process state information shown in ViTABaL-WS diagrams.

4.3 Kaitiaki

The Pounamu meta-tool provides a textual code-based event handler specification tool unsuitable for end users. We wanted to replace this with one using a visual language suitable for non-programmer end users. To develop this replacement visual language, Kaitiaki, and its specification tool we carried out an analysis of Pounamu event handlers from a wide range of tools to identify key constructs used to specify different tool behaviours. All had aspects of (1) specifying the event(s) of interest; (2) querying the tool state in various ways; (3) filtering event/query results and making decisions; and (4) performing state changing actions on filtered objects. We also looked at the metaphors used in existing rule-based and event-condition-action event handler specification tools to see how these manifested the behavioural specifications and how suitable these were for end users. From this analysis and survey, we developed a set of key requirements and design approaches for our new Kaitiaki visual event handler designer:

- A need to represent key “building blocks” of state query, data filtering and state modification (actions).
- A need to represent event objects and their attributes; various objects from the Pounamu tool state (both view and model); and query results (typically collections of Pounamu state objects).
- A need to represent “data” propagation between event, query, filter and action representations.
- A need to represent iteration and conditional data flow.

The metaphor used by Kaitiaki is an Event-Query-Filter–Action (EQFA) model conceptually interpreted as: an end user selects an event type of interest; adds queries on the event and Pounamu tool state (usually diagram content or model objects that triggered the event); specifies conditional or iterative filtering of the event/tool state data; and appropriate state-changing actions to be performed. Complex event handlers can be built up in parts and queries, filters and actions can be parameterised, and reused. Ordering is handled by dependency analysis in the code generator. Domain specific tool icons are also incorporated into the visual specification of event handling as placeholders for the Pounamu state, to annotate and make the language more expressive (as shown in Figure 4.2). Step-by-step visualisations of EQFA element invocation and data propagation are supported for incremental development and debugging of visual event handler specifications.

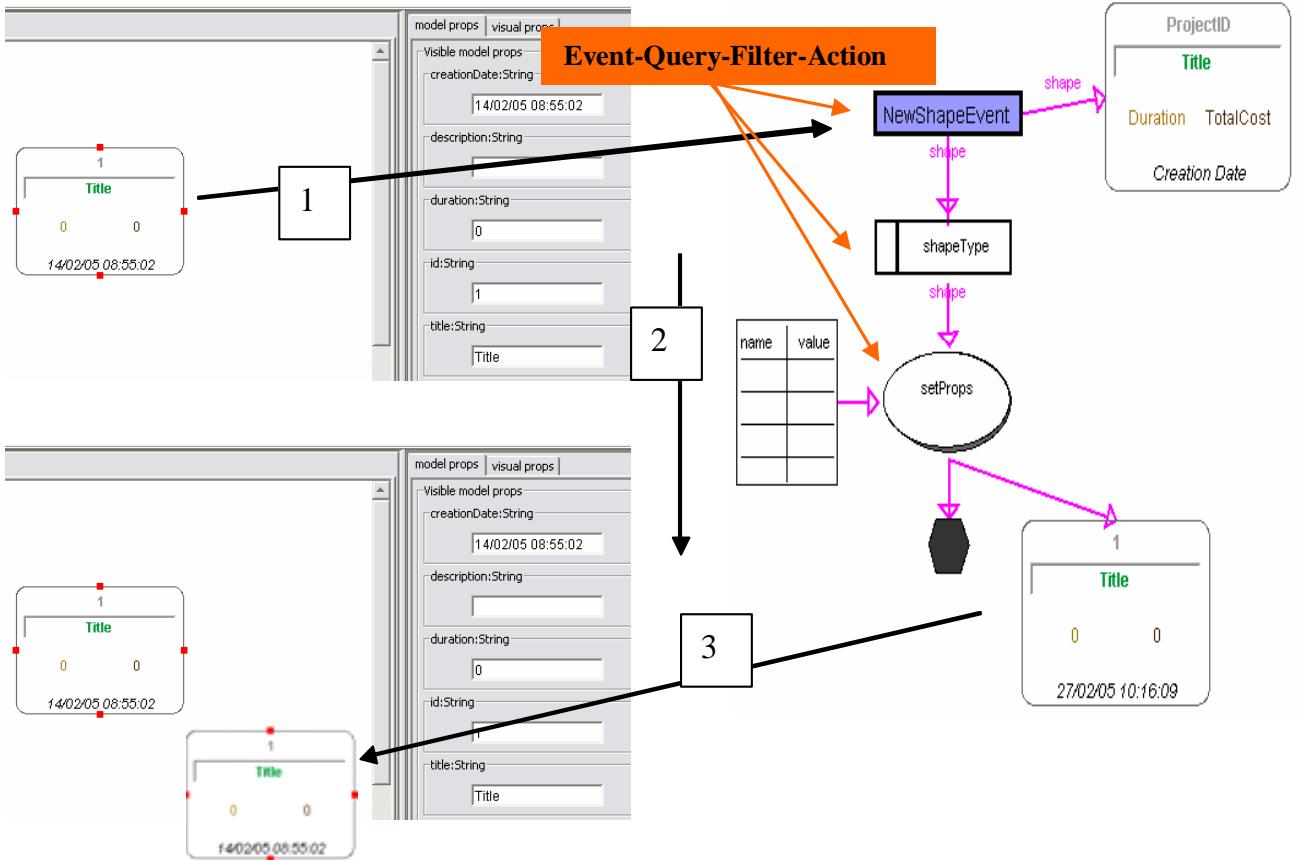


Figure 4.2 Kaitiaki storyboard.

4.4 MaramaTatau

We adopt a spreadsheet-like metaphor to construct metamodel formulae as another approach to specifying visual design tool event handling. A formula is constructed by clicking on entity-relationship metamodel elements (i.e. entity type, association type, and attribute) in a metamodel view and a list of library provided functions as shown in Figure 4.3. Formulae can be attached to an element in the metamodel and detached or removed from it. Context and dependency relationships regarding a constructed formula are automatically inserted/updated reacting to user's clicking actions. Constraints on clicks are also enforced to complement design time semantics. Users can choose to show or hide selected formulae in the view. Consistency between a visual formula and the corresponding textual entity-relationship formula is maintained. Cycle detections are possible while a formula is constructed and de-cycle options at design time are provided to aid error handling.

We adopt the same runtime visualisation technique (i.e., visual debug and step into) as in ViTABaL-WS and Kaitiaki to visualise formula effects, with a complementary tabular display of instance

values as in a spreadsheet. Master-details of related data are shown in the spreadsheet with formulated columns non-editable and non-interpreted formulae shown by tool-tips.

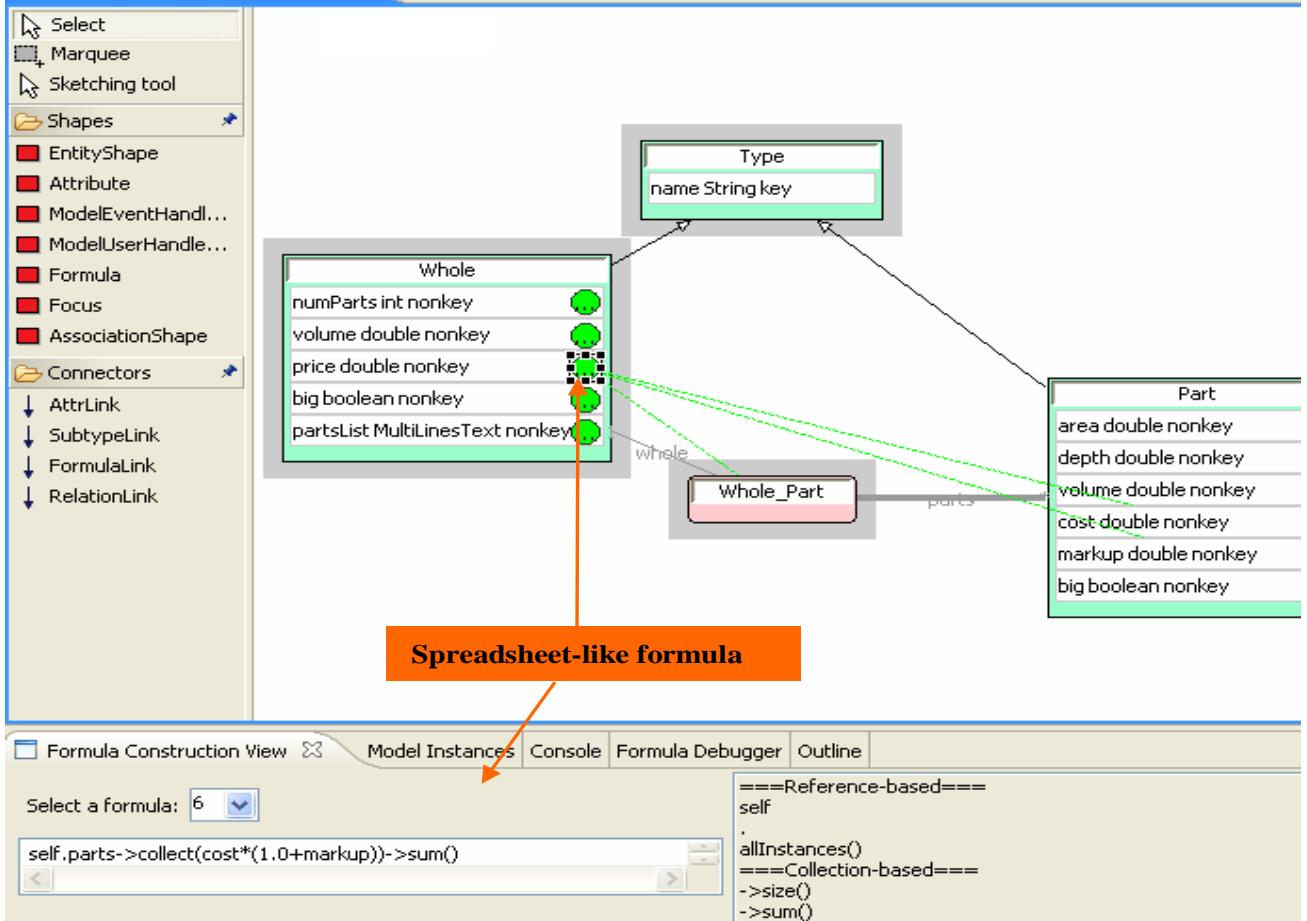


Figure 4.3 MaramaTatau editing in Marama.

4.5 Generalisation

We have generalised from the above three exemplar approaches and developed a metaphor and a language and provided tool support for generic event integration specification. By abstracting from the three exemplars, a general metamodel representation that combines atomic primitives (either shared or non-shared) extended by the three visual languages were defined. This common model supports multiple metaphoric views in the style of the three exemplars and will support generation to a range of underlying implementation technologies for execution or interpretation (OCL (OMG, 2003), RuleML (RuleML Initiative, 2006), stylesheets etc.). As shown in Figure 4.4, a ViTABaL-WS view (a) is used to specify high-level event propagations between components; a Kaitiaki view (b) is used to specify high-level event handling performed by the event consumer component; a MaramaTatau view (c) is used to specify high-level dependency/constraints among components. The

combination of the three metaphoric views allows seamless event integration specification and execution in the Marama (Grundy et al, 2006) environment.

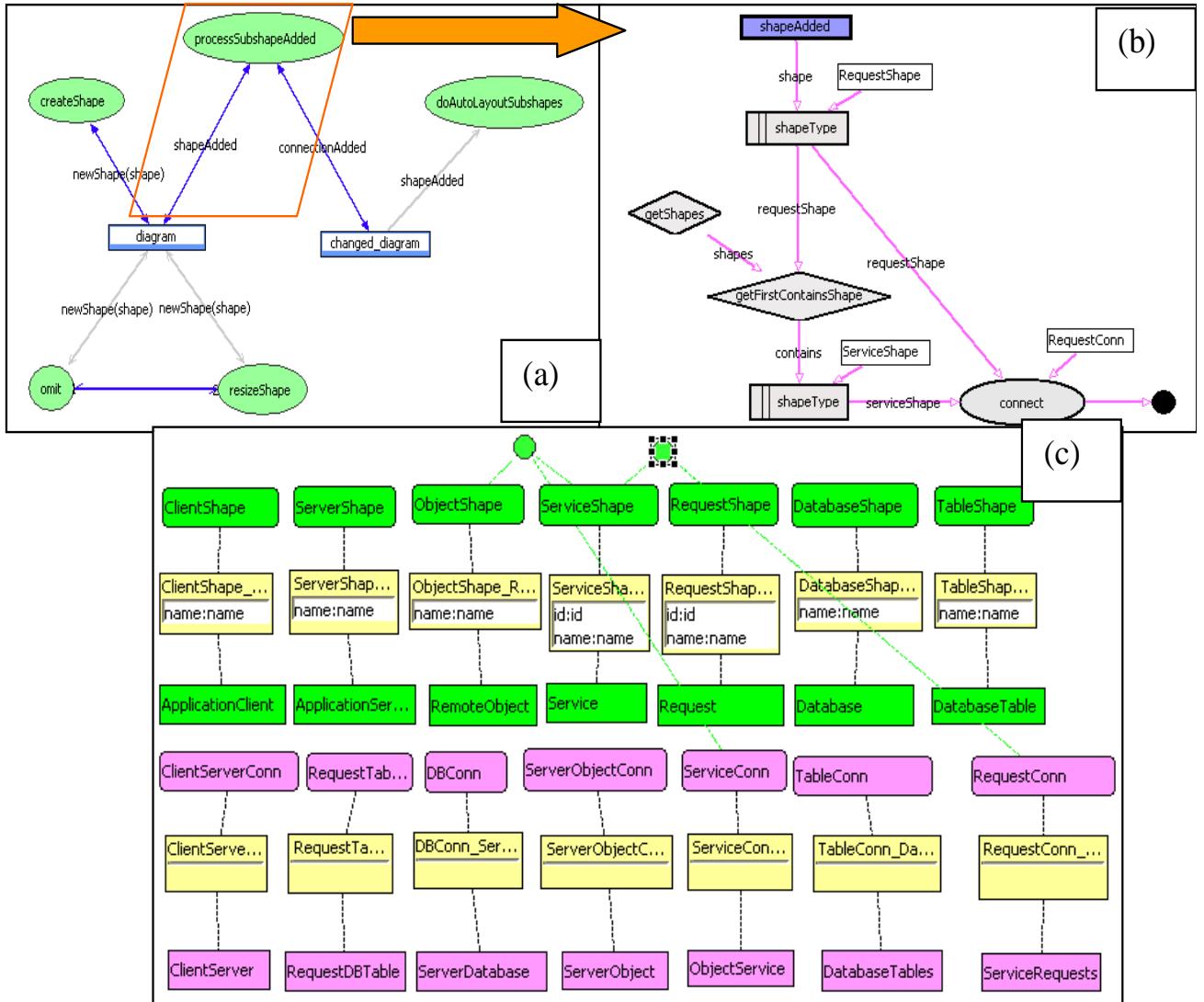


Figure 4.4. Event integration specification in Marama meta-tools.

4.6 Summary

We have overviewed three exemplar visual event-driven system metaphors to specify event-handling support; they are Tool Abstraction in ViTABaL-WS, Event-Query-Filter-Action in Kaitiaki and Spreadsheet in MaramaTatau. We have also overviewed a generalisation from the three exemplars to produce a generic high-level visual event handling metaphor with a visual environment for specifying event-based system integration. In the following chapters, we will address each of the metaphors, including the generalisation, in more detail including elaboration with examples.

Chapter 5 - Visual Web Services Composition

The first problem domain that we focus on towards generating a generic event handling framework is web services composition. This chapter elaborates the metaphor used in this application domain with examples. It is largely based on the ViTABaL-WS (Liu et al, 2005) paper in Proceedings of the 2005 ACM/IEEE International Conference on Automated Software Engineering.

Implementing complex web service-based systems requires tools to effectively describe and coordinate the composition of web service components. We describe a new domain-specific visual language called ViTABaL-WS and its prototype design tool to support modelling complex interactions between web service components. ViTABaL-WS uses a Tool Abstraction metaphor for describing relationships between service definitions, and multiple-views of data-flow, control-flow and event propagation in a modelled process. The tool supports the generation of Web Service Description Language and Business Process Execution Language definitions from a ViTABaL-WS model and directly deploys a generated process model to a workflow engine. Our approach supports specification of both fine-grained, detailed views and more abstract views of business process protocols, message exchange rules and sequencing, and service invocation. ViTABaL-WS also supports visualisation of running processes to support architecture understanding and visual debugging of specified protocols.

5.1 Introduction

Web services are reusable, extensible, platform- and language-independent components that are used over web protocols. An abstract definition of a web service contains two parts: messages and operations (W3C, 2001), each service is described using the Web Services Description Language (WSDL). Running web service operations are bound to ports and run on a host. Web services composition is an approach that integrates individual services to make up a web service-based distributed system. Web services composition combines several existing, published web services and in turn potentially becomes a new web service itself. A web service composition language (either textual or visual) is needed to specify a composite web service, using existing service components

defined in or looked up from a services registry. The composed web service can then be described using WSDL, registered and invoked, and thus added to the network as a new web service component. One common web service composition language is the Business Process Execution Language for Web Services (BPEL4WS (IBM, 2003)), an XML-based service composition language. It describes web services compositions, or orchestration, by defining a set of service partnerships and structured invocation schemes. It also supports specifying concurrency and transaction failure recovery schemes for composed web service components.

While web services have recently become a popular new technology for building distributed systems, there is a lack of languages and tools to specify web service compositions at high levels of abstraction, generate lower-level executable process code such as BPEL4WS, and visualise, at high abstraction levels, running web service processes. Most current approaches provide basic flow-like BPEL4WS editors or similar (Srivastava and Koehler, 2003; Thone et al, 2002). More abstract approaches (Fensel and Bussler, 2002; Foster et al, 2003) only support limited compositional approaches or do not support generation of BPEL4WS or similar executable forms.

We describe a new approach for complex web service composition using a high-level metaphor and visual language. Our approach supports higher level design views for service composition that are complementary to current web services composition standards. We aim to visually represent processes' control-flow, data-flow and event subscription using this metaphor so as to make web services design and implementation easier and less error-prone. To this end we applied a Tool Abstraction paradigm (Garlan et al, 1992; Grundy and Hosking, 1995) to web services composition, characterising different kinds of services (data retrieval, data processing, fault handling, etc) and their interaction (data flow, control flow, event subscribe/notify, synchronised, etc). We then designed a visual language for describing web service co-ordination and built a proof of concept environment that supports modelling with this language. BPEL4WS specifications are generated from our model which can be run in a 3rd party web service orchestration engine to implement the specified web service composition. Events are sent back to the modelling environment and used to animate the composition and allow fine-grained developer control of the running web service for debugging and analysis.

We firstly provide a motivation for this research and a survey of related work. We introduce the Tool Abstraction metaphor and our visual language based on Tool Abstraction for specifying web service composition. We describe our proof of concept modelling tool with a simple example and discuss its

design and implementation. We present results of evaluating our tool, its strengths and limitations, and areas for future research.

5.2 Motivation

Consider a simple loan approval process, as used in the description of IBM's Business Process Web Service for Java (BPWS4J) process execution tool (IBM, 2002). This loan approval process is composed of two main web services: a Loan Assessor web service and a Loan Approval web service. As illustrated in Figure 5.1, when a loan request is received, the new Loan Approval process firstly needs to determine whether the requested amount of the loan is under one thousand dollars or not. If the amount is under one thousand dollars, the Loan Assessor web service is invoked; otherwise the Loan Approver web service is invoked. After the Loan Assessor web service is invoked, the process continues by determining whether the risk for the request is low or high: if the risk is high, the control flows to the Loan Approver web service; otherwise an approval message is generated as the response to the user's loan request. Additional web services might also be used e.g. to provide Loan Assessment Criteria (from a persistent storage mechanism), and to record a Loan Approval Audit trail (storing the loan and approval information in a persistent form for later reporting). Relationships between services in such business process models can become very complex: some send messages and wait for replies; some send messages and continue execution; some provide data while others consume it; synchronisation between concurrently executing services may be needed; service failure may occur and needs to be handled appropriately; and transactional behaviour may be required over services.

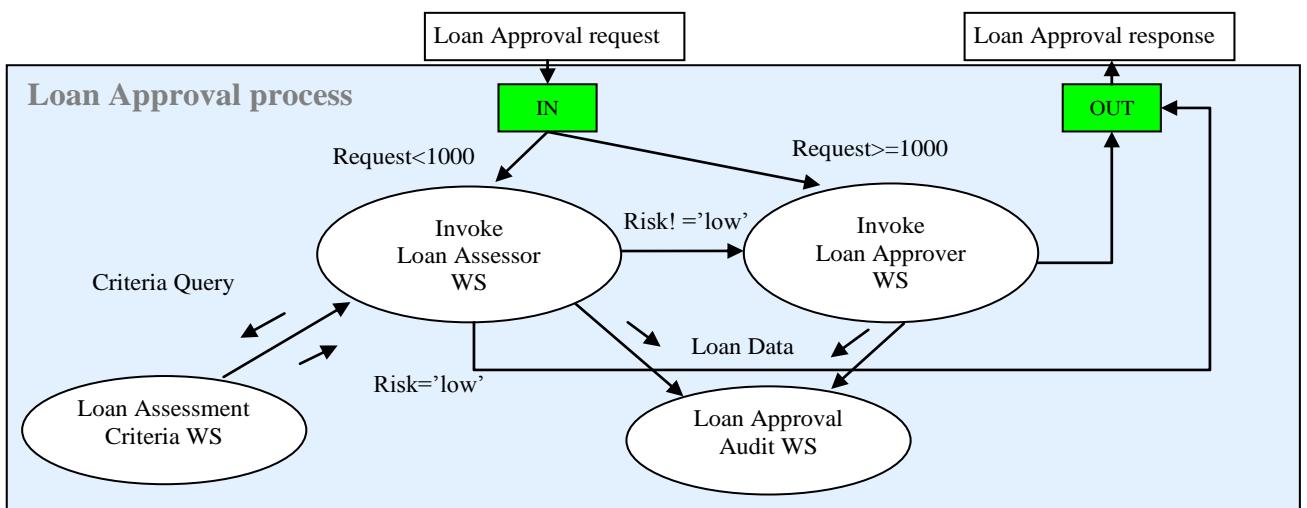


Figure 5.1: Conceptual model of the loan approval process.

5.3 Requirements

Specifying such a composition of existing web services to form a new, reusable web service is the role of web service composition languages, such as BPEL4WS (IBM, 2003). BPEL4WS is an XML-based standard for composing web services to implement business processes that provides a set of structural activities to specify data manipulation, sequence of service invocation and fault handling. Since BPEL4WS is a textual script-based language interpreted by a process flow engine, it inherits the drawbacks of using non-visual techniques, particularly for potential non-programmer users such as business analysts (Pautasso and Alonso, 2005). These include an abstract XML-encoded specification language that is difficult to read, error-prone specifications that are run-time checked, and difficulty in debugging a running specification. It would be highly beneficial to users if a visual modelling language and tool support were provided to specify web service composition models and generate high-quality BPEL4WS from the model. Our work is motivated by these needs. Such a language and tool should meet the following requirements:

- A visual metaphor for composing web services that fits users' mental models of service interaction;
- The visual language for composition must be able to specify: web service interfaces, i.e. abstract message types and operations; variables; and different types of connections (i.e. data flow, control flow, and event flow) between web services in a process;
- A support tool should permit modelling of specifications using the metaphor/visual language; generation of WSDL and BPEL4WS (or other executable business process modelling languages), and easy deployment of generated process models using 3rd party process engines, such as BPWS4J;
- The support tool should permit visualization of running systems by annotating high-level visual specification views from events generated by the process engine, to support debugging of compositions and to assist understanding of others' specifications.

5.4 Related Work

Web service composition is a form of dynamic, component-based architecture. A service description in WSDL publicises the web service's messages, operations and ports, as in Figure 5.2. Web services are "wired together" with messages from one passed to another to build a composition. These web service compositions are commonly called "business processes" or "workflows" (IBM, 2003; Pautasso and Alonso, 2003; Srivastava and Koehler, 2003; Wirtz, 1993). Workflow metaphors are typically used in much recent research on web services composition. Simple workflows are,

however, insufficient to describe the integration and co-ordination of web service components as service composition may be quite complex (Benatallah et al, 2003; Fensel and Bussler, 2002). For example, in Figure 5.2 conditional execution is needed; some links are sequential data-flow from one to another; some asynchronous (such as storage of loanInfo); services may subscribe to events (such as loanInfo storage events); conversion of messages may be needed (such as approvalInfo to loanInfo for storage); and so on.

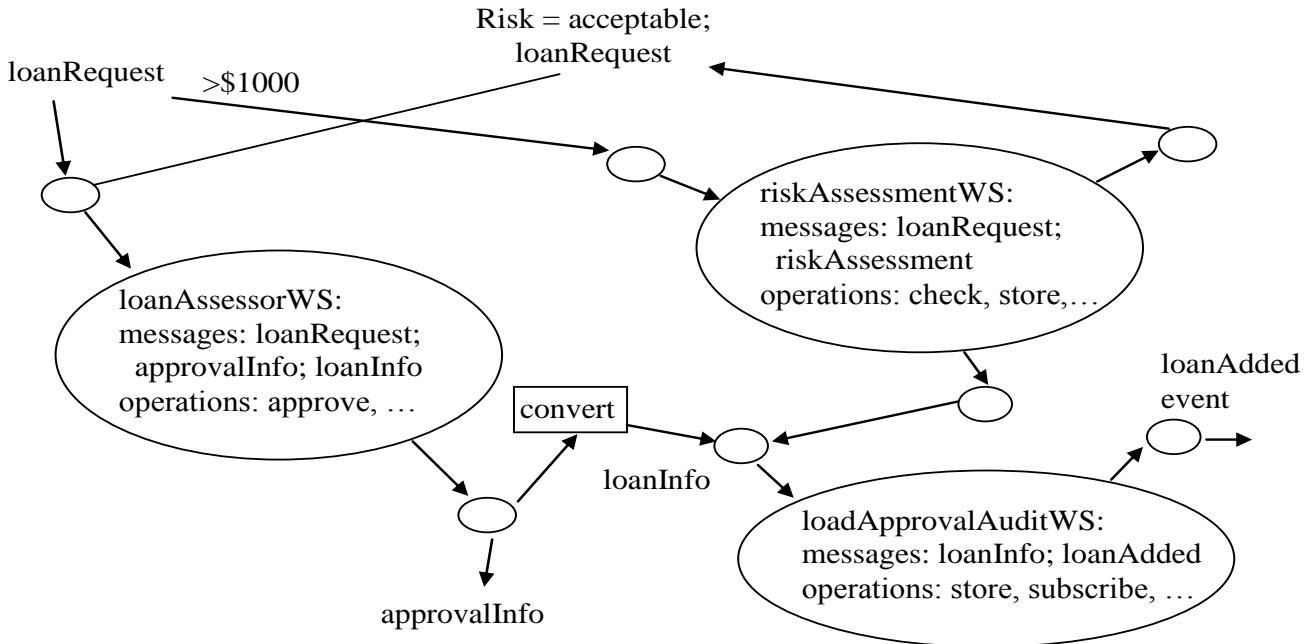


Figure 5.2: Web service composition.

A variety of languages to specify web service composition have been developed. Many are described as “business process modelling languages”, although most web service composition languages, being executable, are quite low-level. Examples include the Business Process Execution Language for Web Services (BPEL4WS) (IBM, 2003), the Business Process Modelling Language (BPML) (Baker, 2002), and jBPM (Baeyens, 2007). Different web service composition languages support different levels of abstraction, fault-recovery, transaction modelling, and service inter-relationships. Most are textual scripts that are interpreted at run-time by workflow or business process flow engines. Such textual scripts are often challenging to read, error-prone to write, and reusability can be limited.

Various visual modelling notations have been developed to support web service composition. Using UML with various extensions is common. UML state-charts can be used to specify implementation aspects of a service composition (Benatallah et al, 2003). These incorporate event handling schemes

where states represent services, transitions are constrained by Event-Condition-Action rules, and an occurrence of an event fires a transition to execute a target action. An argument is that state-charts are based upon finite automata and ECA rules which are easy to comprehend. However this approach provides no means to explicitly specify data flow among services and is cumbersome to use for service compositions. UML-WSC (Thone et al, 2002) uses class diagrams with stereotypes to model static structure and activity diagrams to model dynamic aspects of web service compositions. Service states call operations from components and transform states perform structural transformation on messages. This is limited to modelling request-response and one-way operations only, and lacks fault handling and partner roles.

Message Sequence Charts (MSCs) are compiled into a Finite State Process notation (FSP) to concisely describe and reason about concurrent programs (Foster et al, 2003). This technique provides a high-level metaphor but not web service composition language generation. Petri-Nets have been used to model both offline analysis tasks, such as web service composition, and online execution tasks, such as deadlock determination (Hamadi and Benatallah, 2003; Narayanan and McIlraith, 2002). These approaches describe the capabilities of web services in terms of a first-order logic language. Service descriptions are encoded in an extended Petri-Net formalism with typed arcs, hierarchical control, durative transition, parameterization, typed (individual) tokens and stochasticity. While Petri-Nets are a powerful formalism they are a very general approach, can not be applied to all web service compositions, such as modelling abstract web service interfaces and data flow, and don't provide a visual language easily understood by target end users (Pautasso and Alonso, 2005).

Biopera Flow Language (Pautasso and Alonso, 2003) is a generic visual flow language for coordinating software components, with a development tool tailored for web service composition. This focuses on data flow, execution sequence and fault handling and all can be specified with a simple visual syntax. However it lacks modelling capability for event subscription and various other service relationships like call-backs. The visual syntax is verbose as both data and data bindings must be specified. Web Service Modelling Framework (Fensel and Bussler, 2002) is a methodology for describing and developing web services and their compositions. The integration framework defines a conceptual model for the web services integration (complex web services) and provides services for mediating differences in data structures and message exchange patterns among services.

Many current approaches to modelling web service compositions lack full modelling capability: i.e. are not able to model all types of operations (one-way, request-response, solicit-response,

notification). A common drawback is that a web service interface can not be fully expressed; some model web services operations only; and some can not model invocation constraints in control flow. Most of them use static binding rather than event-based mechanisms to integrate services. Many cannot separate or combine control-flow and data-flow for modelling.

5.5 Metaphor

We wanted a metaphor to effectively describe the composition of web services and support the development of a visual language and modelling environment. As described in Section 5.2, these compositional relationships can be very complex and a range of compositional building blocks are required. We chose to use the Tool Abstraction (TA) paradigm (Garlan et al, 1992, Grundy and Hosking, 1995) as our metaphor for web service compositions and to support reasoning about different relationships between compositional primitives. The TA paradigm is a message propagation-centric approach describing interconnections between “toolies” (the encapsulation of functions) and “abstract data structures” (ADSs: the encapsulation of data) which are instances of “abstract data types” (ADTs: typed operations/messages/ events). Connection of toolies to other toolies and ADSs is via typed ports. The TA paradigm supports modelling data flow, control flow and event flow relationships. Reusability, extensibility and expressiveness are key advantages possessed by TA (Garlan et al, 1992).

ViTABaL (Grundy and Hosking, 1995) is a hybrid visual programming environment previously developed for designing and implementing TA-based systems. It uses the TA paradigm to compose systems by integrating, and coordinating toolies and ADS components. We have found that the TA paradigm is well suited for web services composition domain by specifying an abstract model involving a series of co-ordinated invocations to web services operations. We adapted the earlier work to develop a new visual language and environment, ViTABaL-Web Services (ViTABaL-WS). ViTABaL-WS specialises the ViTABaL visual composition language to the domain of web services composition. It supports modelling of both event-dependency and dataflow in designing complex web service compositions. Figure 5.3 and Figure 5.4 show ViTABaL-WS diagrams illustrating examples of compositional primitives in the Tool Abstraction paradigm. Toolies (web services - shaded, green ovals) encapsulate data processing and interact with each other through both direct and indirect operational invocations using shared data structures (message ADT instances: rectangular, shaded icons); and event-driven dependencies indicating state changes to a Data Store ADS (data storage service). A system of typed input and output ports on toolie and ADS services provide

message sources and sinks. Services are wired together using these ports with ports supporting only certain kinds of connection and message ADTs. Messages generated by a service output port are distributed to connected web service input ports. Many interconnection schemes are supported including one-way flow, request-response, asynchronous flow, and subscribe-notify. Additional controls support conditional flow, dynamic type checking, synchronization, iteration etc.

The specified web services are linked together by composition rules enforced in the ViTABaL-WS tool. ViTABaL-WS supports generation of WSDL and BPEL4WS from its abstract composition model. We use the Business Process Execution Language for Web Service Java Run Time (BPWS4J) as the deployment engine for generated BPEL4WS processes. A deployed process is provided with a SOAP interface and a WSDL file, and thus can be invoked by a requesting web service client. BPWS4J is tightly integrated with ViTABaL-WS, so that a ViTABaL-WS process can be directly deployed and step-by-step visualisation of process execution can be obtained, with running process state information shown in the ViTABaL-WS diagrams.

5.6 Notation

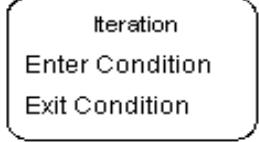
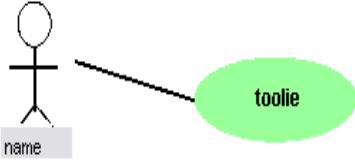
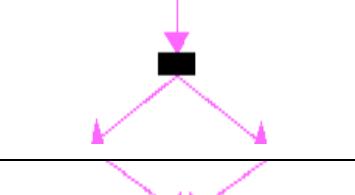
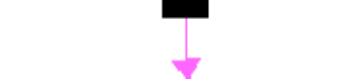
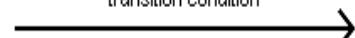
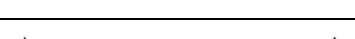
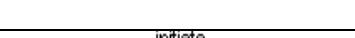
ViTABaL-WS defines a visual notation to specify data flow, control flow and dynamic event flow in a web service composition process. It also provides constructs for distinguishing different kinds of web services, synchronisation, fault handling, message splitting/ composition, dynamic type checking, sub-process composition, and various control flow models.

ViTABaL-WS includes various toolie representations, including data processing services, subprocesses, type checking and fault handling toolies. Data storage/retrieval services are represented by data store Abstract Data Structures. We use the tool abstraction paradigm's ADTs to represent typed web service interaction: data being sourced from, sinked to and transmitted between web services; event messages; and other kinds of toolie interaction. Web service operations are explicitly specified by ports attached to toolies and ADSs, Web service compositions are thus constructed from toolies, ports, message ADTs and ADSs.

Table 5.1 overviews elements in the ViTABaL-WS visual notation. Each toolie specifies a web service interface as a part of the process model. Interaction is via data and control-flow dependencies and event propagations. ViTABaL-WS is not only able to specify basic control-flow elements such as dependencies, decisions, fault handling, iterations and concurrent executions in a process model,

but also, using the Tool Abstraction paradigm, it can specify event propagations between services, producing event-driven compositions. A mix of data-flow, control-flow and event-driven interaction between services is possible.

Element	Visual Representation	Semantics
Message ADT		WS data
Data Store ADT instance		Data Store WS
Partner/Performer/Role		Coordinating service
Fault handler		Exception/fault solution
Toolie		Processing WS; process activity, Atomic activity
Port binding (attached to WS toolies)		Generic operation
		One-way operation
		Request-response operation
		Notification operation
		Solicit-response operation
Dynamic type checking		Type checking and transformation
Sub-process		Complex-activity
Data manipulation		Assign/Copy data value

Iteration		For/while/until
Partner Link		Partner/Performer coordinating
Data flow	   	Input/output flow Parameter flow Split message - parameter decomposition Merge message
Control flow	       	Transition Conditional Branching Asynchronous flow Control Dependency Concurrency - Parallel execution Initiate Wait-for Iterate
Broadcast		One-way communication
Request		Request-response communication
Listen_before		Solicit-request communication
Listen_after		Notification communication

Subscribe notify		Event registration
Callback		Event callback

Table 5.1: ViTABaL-WS notation overview.

ViTABaL-WS permits multiple views for complex processes and sub-processes, allowing a service in one process to invoke via ADS messages and ports another service or a sub-process. Different views allow both static specification of web service interfaces and dynamic specification of messages between processes in different views, with consistent references managed by the specification environment. Orthogonal views allow different kinds of interaction e.g. event-driven and data-flow, to be modelled separately if desired.

5.7 Loan Approval Example

To illustrate our ViTABaL-WS visual web service composition language in use, we use the loan approval business process model example from Section 5.2 (IBM, 2003). Two main web services (loan approver service and loan assessor service) need to be coordinated to synthesise a new process service (loan approval service) which is then exposed to other web service clients. The composite process defines roles performed by all participating services, i.e. “loan approver” service fulfils an “approver” role and “loan assessor” service fulfils an “assessor” role.

This exemplar comprises two main information processing toolies (suffixed by “PT”): loanApprovalPT and riskAssessmentPT. The input and output message types to these processing toolies we characterise as ADTs consumed or produced by the processing toolies. We may additionally characterise key fault handling and dynamic type checking behaviours associated with these toolies.

Figure 5.3 shows some of the toolie specifications used in the loan approval process. Figure 5.3 (a) and (b) show the interfaces for the loanApprovalPT and riskAssessmentPT processing toolies. An abstract web service interface is visually represented using input/output dataflow links, parameter decomposition links, and transition links to support association of a toolie’s web service port types and message ADSs. We attach operations to a port type to represent the port bindings of a web service. For example, in the “loan approver” web service definition in Figure 5.3 (a) the “loanApprovalPT” toolie has one port providing an “approve” operation with

“creditInformationMessage” as input message type (indicated by a data flow link with the arrow pointing to the operation) and “approvalMessage” as output message type (indicated by a data flow link with the arrow pointing out of the operation). The approvalMessage contains one message part, “accept” (shown by the parameter decomposition link). In the case of an error occurring when the toolie is invoked, the operation “approve” transits to the “loanProcessFault” fault handler (via a one-way operation link) which generates a fault message of type “loanRequestErrorMessage”. The “loanApprovalPT” toolie may also invoke a “loanApproval Audit” ADS (via another on-way operation link) to record an audit trail of loan approvals. Toolies may provide multiple ports for other toolies to bind too. Bindings may be data flow in/out, subscribe/notify event-based interaction, one-way asynchronous invocation, bi-directional synchronous invocation and so on. Toolies may also have more than one fault handler for operations.

Multiple views are used to specify toolie interfaces (Figure 5.3 (a) and (b)), complex message decomposition (Figure 5.3 (c)) and toolie usage contexts (Figure 5.3 (d)). A toolie’s interface can be collapsed to just show its port types for other client toolies to bind to, as in Figure 5.3 (d).

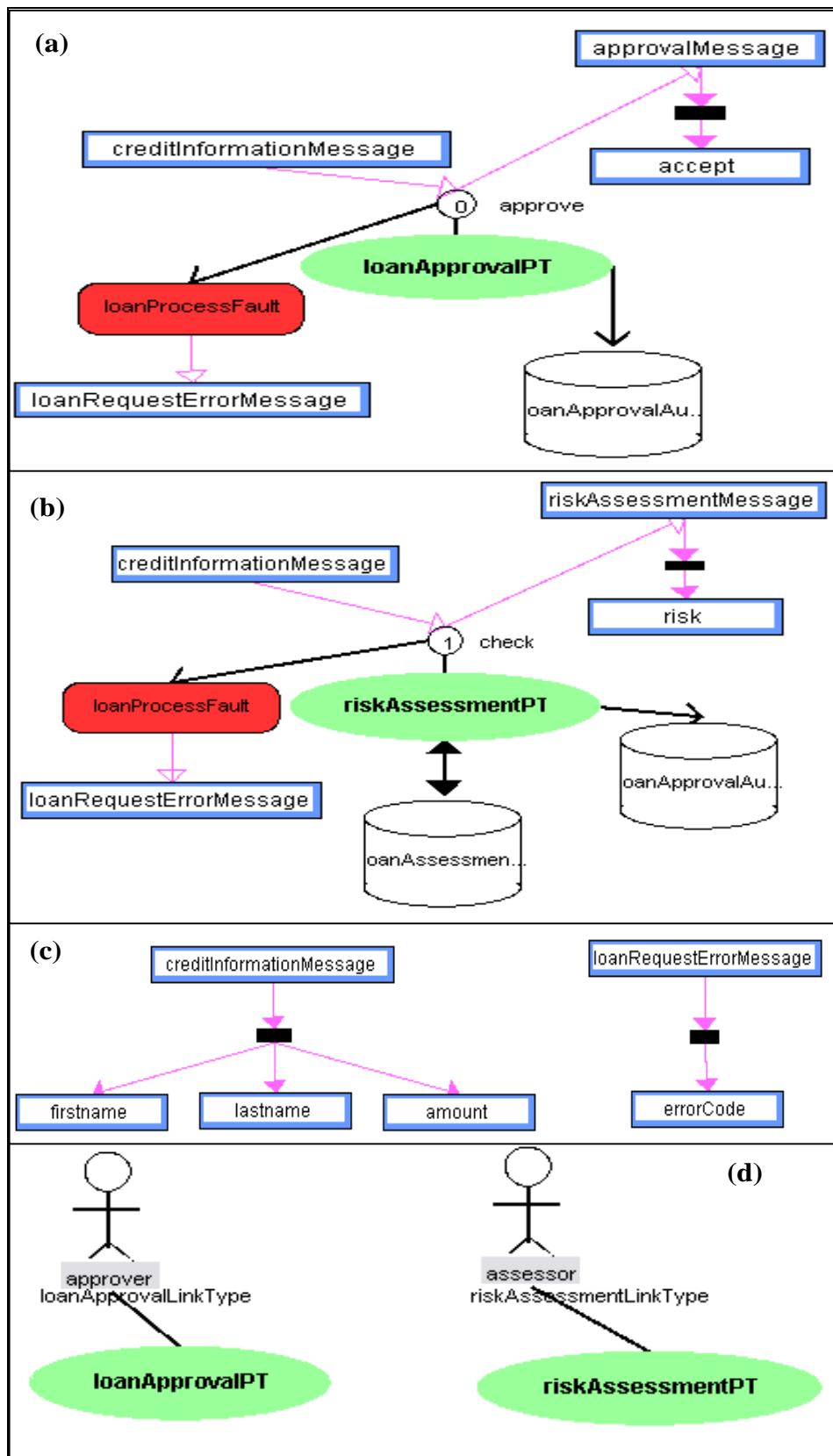


Figure 5.3: Various web service toolies and their interfaces involved in the loan approval process.

A business process model is built up by composing web service toolies using appropriate link types. Figure 5.4 (a) shows the basic loan approval process. Note other overlapping views can be defined to add extra information about a process model e.g. extra: toolie links driven by event notification of asynchronous message flow; fault handling; message data storage/retrieval and so on. The “loan approver” process defined in Figure 5.4 (a) expresses the following semantics: the “loan approver” service receives a loan request. The process’ control flows to a decision point, which retrieves the value of the amount of the loan requested. The conditional is specified by labelling the outgoing links with an XPath expression specifying the comparison (including retrieval of appropriate message content, in this case

```
“bpws:getVariableData('request','amount')&gt;=1000” or  
“bpws:getVariableData('request','amount')<1000”;
```

These expressions, as can be seen, are long and work is needed in our tool implementation to express them in a more satisfactory way, e.g. as a tooltip). If the requested amount is less than \$1,000 the process control invokes the “risk assessment” service, else it flows back to invoke the “approve” operation of the “loan approval” service. The “risk assessment” service takes the loan request as input and decides if the loan is a low risk. It retrieves loan criteria information from the “loanAssessmentCriteria ADS” to be used in the assessment task. If the risk is low the loan is approved, otherwise the process model invokes the “approve” method in the “loan approval” service to do a more thorough check. Both toolies invoke data storage activity on the “loanApprovalAuditADS” to record an audit trail of approvals. Once a loan is either approved or rejected, an approvalInfo message is constructed and returned to the invoking client.

Another example is shown in Figure 5.4 (b), illustrating a different approach to the audit trail, with asynchronous flow from the generated “approvalInfo” message via an adapter converting its format to the “loanApprovalAudit” service and the generation of a “loanAddedEvent” notification subscribed to by a “print audit trail” service.

In a ViTABaL-WS process model we use variables of particular message types to specify message flow from one toolie port to one (or more) others. Each toolie process may be stateful with state information stored in such variables. In a composition model an abstract process toolie interface may have additional constructs added, such as dynamic type checking and message storage interfaces, and extra control flows and event propagations for a more advanced process model specification.

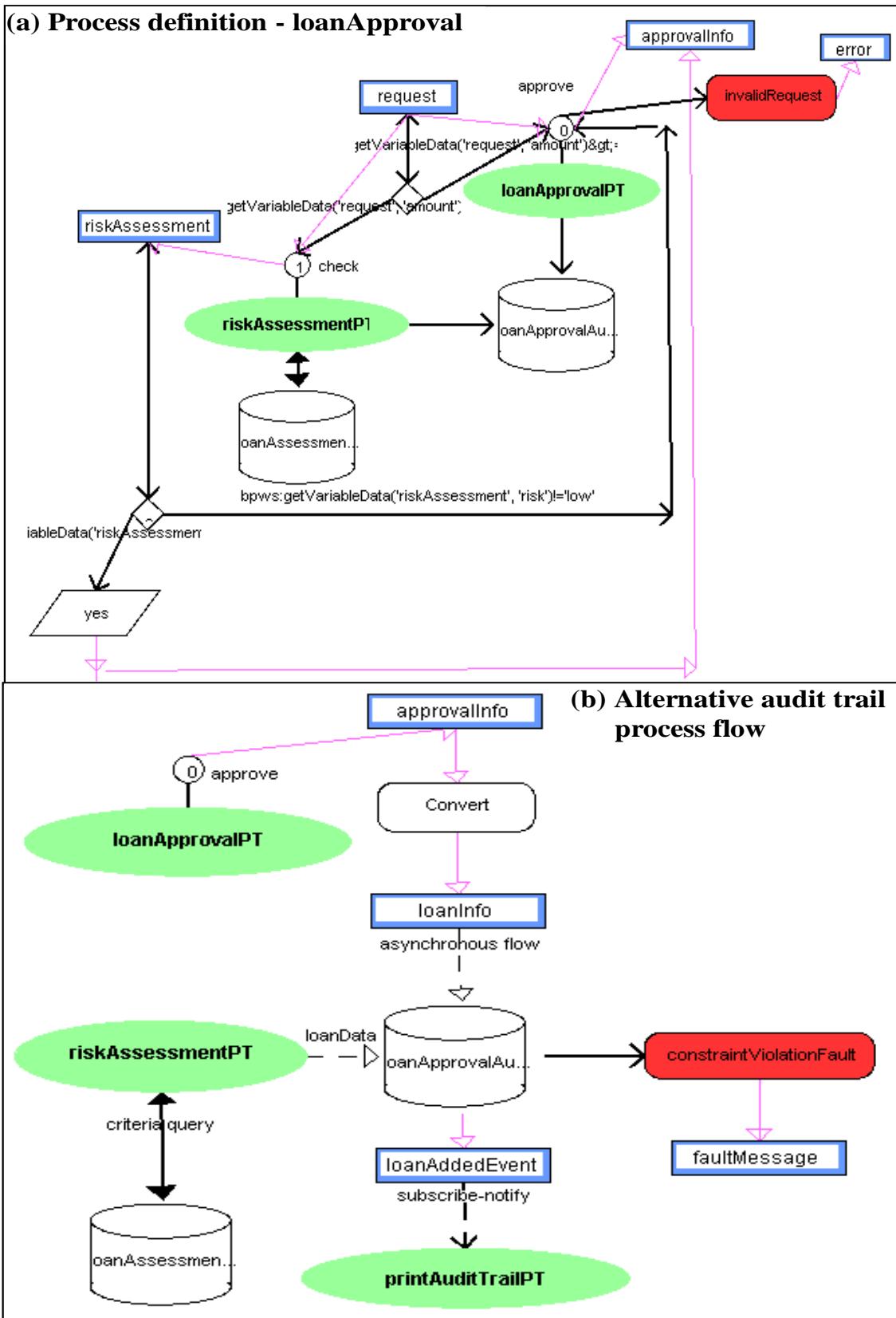


Figure 5.4: Composed Loan Approver web services.

ViTABaL-WS automatically numbers operations to specify an invocation sequence of component web services. The operation numbered zero is the initial task to instantiate the process model and the process is terminated with a reply to its client. Transition links can also specify an implicit order of invocation, with transition dependency constraints specified along with the transition links.

In order to execute our web service process model we needed to translate our model into BPEL4WS. A BPEL4WS composition specification contains XML records specifying web services receiving messages, the service being invoked and reply message being generated (i.e. constructs <receive>, <reply>, <invoke>, <assign> etc). The ViTABaL-WS model contains TA-based modelling constructs that can be mapped onto BPEL4WS constructs. Processing and data storage/retrieval toolies map onto web services, with ADTs in ViTABaL-WS mapping onto BPEL4WS messages. Toolie ports map onto BPEL4WS ports with typing from ADT messages. Fault toolies and links to ports map onto BPEL4WS fault handlers. Synchronisation control, asynchronous message flow and subscribe/notify relationships in ViTABaL-WS map onto BPEL4WS process model script code to implement these behaviours. Concurrent operations in ViTABaL-WS map onto concurrently run BPEL4WS service invocations. Type checking toolies, conditional execution and iteration map onto BPEL4WS script to carry out these operations.

For example Figure 5.5 shows a pair of toolie and port binding visual constructs being mapped to an <invoke> construct in BPEL4WS if the service is not the request receiving service, otherwise, it is mapped to BPEL4WS <receive> and <reply> constructs. The ViTABaL-WS visual links for service invocations and conditional flow/iteration are mapped to BPEL4WS script and links specifying these control flows. The interface for the loanApprovalPT toolie is mapped onto a generated WSDL interface specification for the service, which is used by the generated BPEL4WS process model specification composing an instance of this service in the Loan Approval process. Figure 5.5 also shows BPWS4J deployment view of the composed process.

We use the BPWS4J engine to deploy and debug our generated BPEL4WS models, though any BPEL4WS-compliant engine could be used. ViTABaL-WS allows the user to deploy a generated process model to the engine. The engine checks the host/port specified for each web service in the generated BPEL4WS model has the specified service active. Once a process is deployed, it is assigned a service address by BPWS4J so that it can be called by a client. A web service invocation message sent to this service address (host/port) will now invoke our running loan approval process.

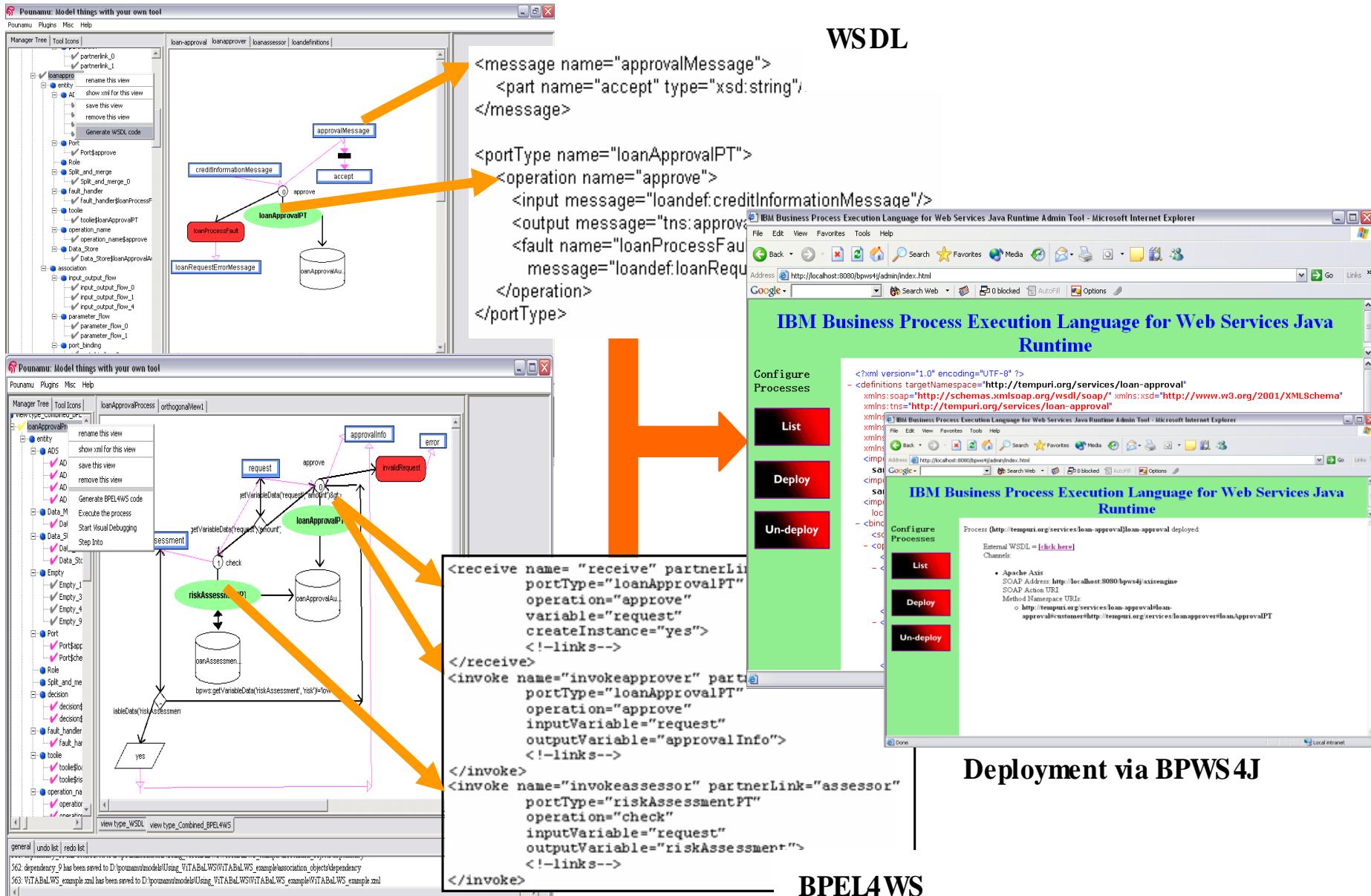


Figure 5.5: Generating WSDL and BPEL4WS specifications from our ViTABaL-WS model.

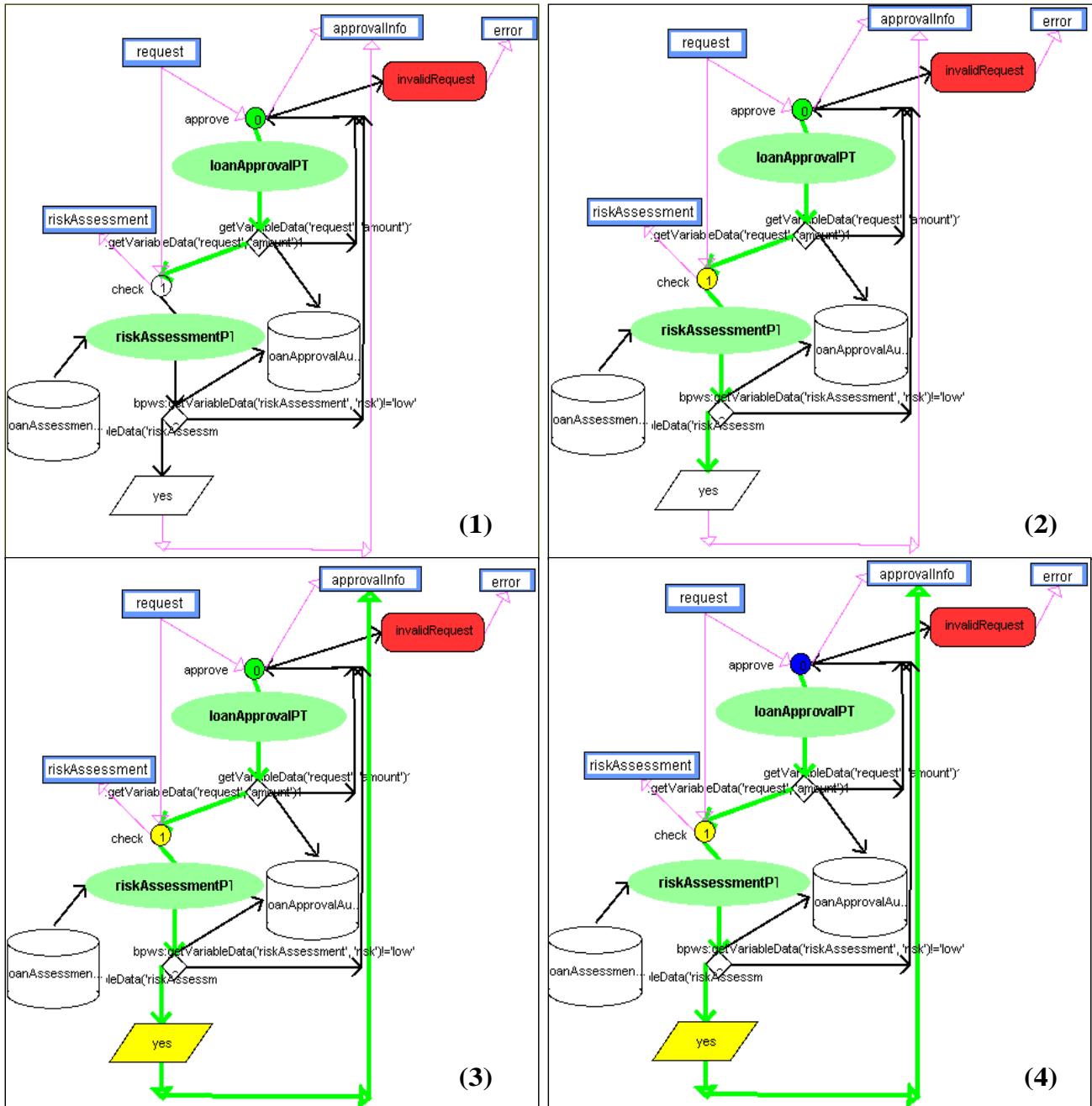


Figure 5.6: Dynamic visualization of a ViTABaL-WS model.

Figure 5.6 shows a running loan approval process (slightly different model to that of Figure 5.4, but similar functionality). The user has asked ViTABaL-WS to generate and deploy a process model then asked ViTABaL-WS to send the running process a loan request of \$100 dollars from a client which returns an approval reply to the client. Messages recording the web service invocations in the process are sent out by each request/reply/invoke step calling a special debug web service implemented by ViTABaL-WS itself. The ViTABaL-WS visual modelling client receives and interprets these BPEL4WS message requests and highlight elements in the ViTABaL-WS views,

providing a dynamic visualisation. The dynamic visualisation includes service invocation (by flashing the service representation node); invocation path into the service (by highlighting the path). The user can double-click on a link or message and see its contents as XML. The traditional “debug and step into” metaphor is used to support step-by-step visualisation. During each step of service execution, the states of all variables (messages) in the process are displayed in a debugging panel. Sub-processes invoked in the process are visualised similarly.

In Figure 5.6, after receiving the request message, the loanApprovalPT passes a riskAssessment message to the riskAssessmentPT (1). The risk is checked (2) and then the loan approved (3). The final process state highlights the request/reply service (node darkened) and the entire invocation path (4). XML messages flowing from/into a stage can be viewed in a property view or via a tool tip.

5.8 Design and Implementation

ViTABaL-WS was implemented using Pounamu, which has been introduced in Chapter 3. We specified, in Pounamu, a metamodel defining ViTABaL-WS visual language constructs and constraints between them, and the graphical notations that are used to visually represent the syntax of the language in each view type. We specified two different kinds of view editors: one for individual web service interface definition, and the other for process definition when composing web service components. The ViTABaL-WS environment allows separation of process logic into data-flows or control-flows, and allows users to define views at different abstraction levels for processes and subprocesses and orthogonal views at the same level for a process. The ViTABaL-WS code generator has been implemented via specialised Pounamu modelling view plug-ins.

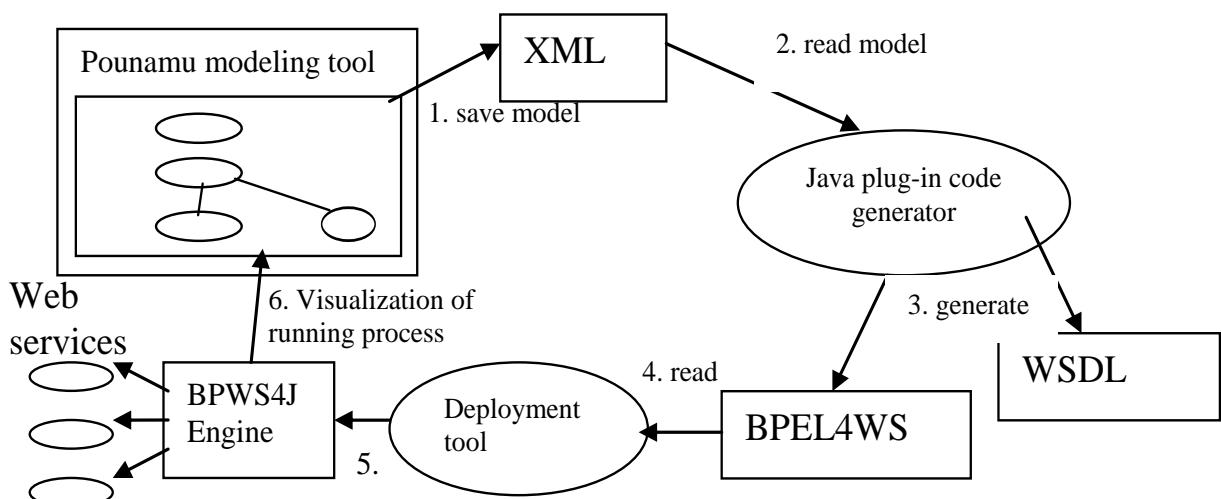


Figure 5.7: Design of our ViTABaL-WS tool.

Figure 5.7 illustrates the high-level design of our ViTABaL-WS environment. Model views are specified using a Pounamu-generated modelling tool with model data stored in an XML format (1). The WSDL and BPEL4WS code generators are plug-ins to the Pounamu-implemented tool. The code generators read model view information from the backend XML files (2), and generate WSDL and BPEL4WS specifications (3). These are derived from the individual web service interface views and composite process model views. WSDL and BPEL4WS XML namespace generation is automated.

Import from other service definitions is automatically handled so that the user need not worry about namespace conflicts. Once code generation is complete, the services and process are deployed, with no need for the user to view or modify any of the generated BPEL4WS code (4). Extra ViTABaL-WS debugger web service calls are automatically instrumented into the generated process model. The BPWS4J engine may be started directly from ViTABaL-WS to deploy and invoke process models (5). When services execute, debug service messages are sent to ViTABaL-WS for visualisation and user controlled step-through via the visual model views (6).

The ViTABaL-WS debugging web service was developed using Java and deployed to the web server using Apache Axis. WSDL for this service is retrieved from the server page and packaged into generated BPEL4WS processes. Generated BPEL4WS web service invocations are wrapped with a pair of debug web service invocations to the start and end ports of the debug service. This also provides access to the XML content of messages sent to/returned from each web service to be displayed in ViTABaL-WS. When development of a ViTABaL-WS process is complete, an optimised BPEL4WS specification without the debugging service calls is generated.

5.9 Discussion

5.9.1 Evaluation

We have carried out three evaluations of our environment: a Cognitive Dimensions (Green and Petre, 1996) evaluation, a user evaluation, and a revisit of the requirements expressed in Section 5.3.

Cognitive Dimensions provides a framework for us (the DSVL designers) to assess usability characteristics of visual languages and their supporting tools using a set of “dimensions”. By examining our environment against the set of Cognitive Dimensions, we understand both the strength of the environment design and the trade-off to mitigate. The dimensions in italics below are

addressed for ViTABaL-WS being those most pertinent for this environment. Our target user group are experienced web service developers hence we have chosen a system with medium *abstraction gradient*. Our primitive visual elements represent a broad range of tool abstraction paradigm components and links which provides a higher level of abstraction than pure BPEL4WS service request/reply/invoke constructs which arguably provides better *closeness of mapping* for our target user group. We have chosen a relatively *verbose* visual formalism, but the language has a core set of constructs which allow a relatively good understanding to be obtained through knowledge of only a *terse* subset of constructs.

Our current mapping tool has limited *juxtaposability* as multiple views can't be open side by side. Thus complex compositions spread over several views/sub-process models may be difficult to navigate and users may lose their context when reading a specification creating *hidden dependencies*. This is addressed in our generic event handling framework which will be discussed in Chapter 8-10.

Users tend to layout their compositions from top to bottom to give an indication of flow in the form of *secondary notation*, though ViTABaL-WS imposes no interpretation of positioning of icons in the diagrams. The usual *viscosity* problems occur when diagrams need to be arranged to insert additional elements. *Progressive Evaluation* is well supported as users can deploy and check execution of specifications at any time with feedback provided at the same abstraction level using debug views.

An informal user evaluation was carried out with several users familiar with the concepts of web services and web service composition. We explained and demonstrated to the users the problem domain of our visual language and its supporting environment and asked them to perform a simple web services composition task. We used three questionnaires focusing on usability, expressiveness and overall capabilities of ViTABaL-WS to obtain feedback, summarized as follows:

- The TA metaphor is easy to understand for advanced users familiar with visual notations/metaphors; but it takes much more time/effort for others to understand.
- The visual language is very expressive to specify web services interfaces and composition. However it is insufficient for specifying dynamic service lookup and invocation.
- It would be helpful to provide intermediate abstraction level specification views to allow more detailed specification to mitigate the abstraction gradient. Other possible improvements would include the visual representation of looping and synchronization.

- The environment needs to provide a means to explain to the user the correct use of notations and generate feedback indicating improper use of notations.

The requirements expressed initially are addressed effectively, given that we have an expressive visual language and proper runtime visualisation support for web services composition tasks. Comparing the generated BPEL4WS process models to hand-coded models (our adopted approach to web services composition prior to ViTABaL-WS), we find that generation of WSDL and BPEL4WS reduces some areas of error-proneness given a quality abstract model. It automatically resolves xml tagging constraints and namespace references, often error-prone in hand-coded BPEL4WS. However, ViTABaL-WS is not as expressive as BPEL4WS for specifying dead-path-elimination, message correlations and transaction rules/expressions.

5.9.2 Strengths and Limitations

From our three evaluations we conclude that ViTABaL-WS provides a generally effective environment for web service composition. The TA paradigm used as the compositional metaphor allows expression of complex web service interactions at a higher level of abstraction than languages like BPEL4WS and most existing BPEL4WS generation tools, which usually provide abstractions directly related to BPEL4WS constructs. We can generate from our ViTABaL-WS specifications complete, executable BPEL4WS models which can be deployed to and run directly from the environment using a BPEL4WS engine. A visual debugger dynamically highlights stages and links in the ViTABaL-WS model providing an interactive debugging and visualisation mechanism with no need change to the BPEL4WS engine.

There are some weaknesses with the use of the general purpose TA paradigm. While it provides an abstract and consistent way to express web service compositions, we found users wanting to express compositions in a notation closer to their target domain (i.e. closeness of mapping requires improvement). E.g. enterprise business systems analysts may not find the paradigm intuitive when they think of composing services to form a new business process model. Addition/use of BPML constructs is likely to improve this. In addition, the choice of icons to represent different types of elements in ViTABaL-WS is arbitrary and based on the previous work with the TA paradigm. Users may find redefining iconic appearance more closely linked to their actual purpose would make composition models easier to read and understand. Our ViTABaL-WS tool is currently weak at pre-BPEL4WS generation analysis, only enforcing simple type and inter-view consistency checks. More

complete model checks for concurrency control, transaction management, dead-lock conditions, etc would improve generated code.

5.10 Summary

We have developed ViTABaL-WS, a hybrid visual programming environment for design and implementation of complex interactions and data exchanges among web service components. It is an exemplar tool implemented using the Pounamu meta-tool. ViTABaL-WS uses the TA paradigm to express complex web service compositions. It provides code generation to BPEL4WS and uses the BPWS4J engine to deploy and execute generated process models. An interactive visual debugger animates running service compositions in ViTABaL-WS by instrumenting debug service calls into the generated BPEL4WS.

This is the first of the three exemplars used to generalise our generic event handling framework. The second exemplar to be described in the next chapter is event handling in visual design tools. It uses a similar dataflow like icon and connector approach to event handling specification.

Chapter 6 - Visual GUI Editing Event Handling

The second problem domain that we focus on towards generating a generic event handling framework is event handling in visual design tools. This chapter elaborates the metaphor used in this application domain with examples. It is largely based on the Kaitiaki (Liu et al, 2005) paper in Proceedings of the 2007 Australasian Conference on User Interfaces.

End users often need the ability to tailor diagramming-based design tools and to specify dynamic interactive behaviours of graphical user interfaces. However most want to avoid having to use textual scripting languages or programming language approaches directly. Our ViTABaL-WS approach specifies high-level tool abstractions, but is not a good approach for GUI event handling metaphor, due to its lack of discrimination of end user objects from abstract queries and state-changing actions, and structured data flow in between. As our second exemplar, we describe a new visual language for user interface event handling specification targeted at end users. Our visual language provides end users with abstract ways to express both simple and complex event handling mechanisms via visual specifications. These specifications incorporate event filtering, tool state querying and action invocation. We describe our language, its incorporation into the Pounamu (as described in Chapter 3) meta-tool environment, examples of its use and results of evaluations of its effectiveness.

6.1 Introduction

Visual design tools have many applications, including software design, engineering product design, E-learning and data visualisation. In Pounamu, for example, high-level visual specifications of tool metamodels and visual language notations allow end users to modify aspects of their tools such as appearance of icons and composition of views. However, both our own and other researchers' experiences indicate that many end users also wish to modify tool behaviour (Morch, 1998; Peltonen, 2000) and reconfigure user interaction with their design tool. This includes: specifying editing constraints, e.g. diagram element layout; automated diagram modification, e.g. auto-add or resize of elements; semantic constraints, e.g. allowing connection of only certain typed elements; automatic

computation, e.g. calculating an attribute value from the values of connected diagram element attributes; and well-founded user interactions, e.g. alerting users to invalid input.

Many end users of such tools are not programmers and do not wish to learn or use complex textual scripting languages to tailor their design tools in these ways. Most approaches for design tool tailoring, however, use just such techniques (Smith et al, 1995; Lewicki and Fisher, 1996; Peltonen, 2000). Some tools support limited configuration via preferences and wizards. But these severely limit the tailoring possible (Morch, 1998). Programming by example has been used for end user configuration, but is limited in power and it is often hard to visualise and modify specifications learnt (Cypher, 1993; Smith et al, 1995).

Most visual design tools are “event driven”, meaning when a user modifies a diagram in the tool, events are generated and can be acted upon to modify other diagram content, enforce constraints, etc. We have used the event-driven nature of such tools as a vehicle to provide end users with a domain specific visual language, Kaitiaki, with which to specify behaviours for their tools. We have added this visual language to the Pounamu meta-tool providing end users with little programming background, a mechanism to detect events and specify actions to take. We first motivate our work and survey related research, then outline our approach and its design and implementation. We finish with an evaluation and conclusions.

6.2 Motivation

Consider a diagram-based design tool for web site and GUI specification, an example of such is illustrated in Figure 6.1. This consists of a web site map view (rear) and a web form view (front). We have built this tool with the Pounamu meta-tool as have many other diagram-based design tools (Zhu et al, 2007). Such applications allow end users to model complex design problems using visual notations appropriate to the domain. As many users of our tools are not programmers, providing ways of specifying behavioural changes is more challenging.

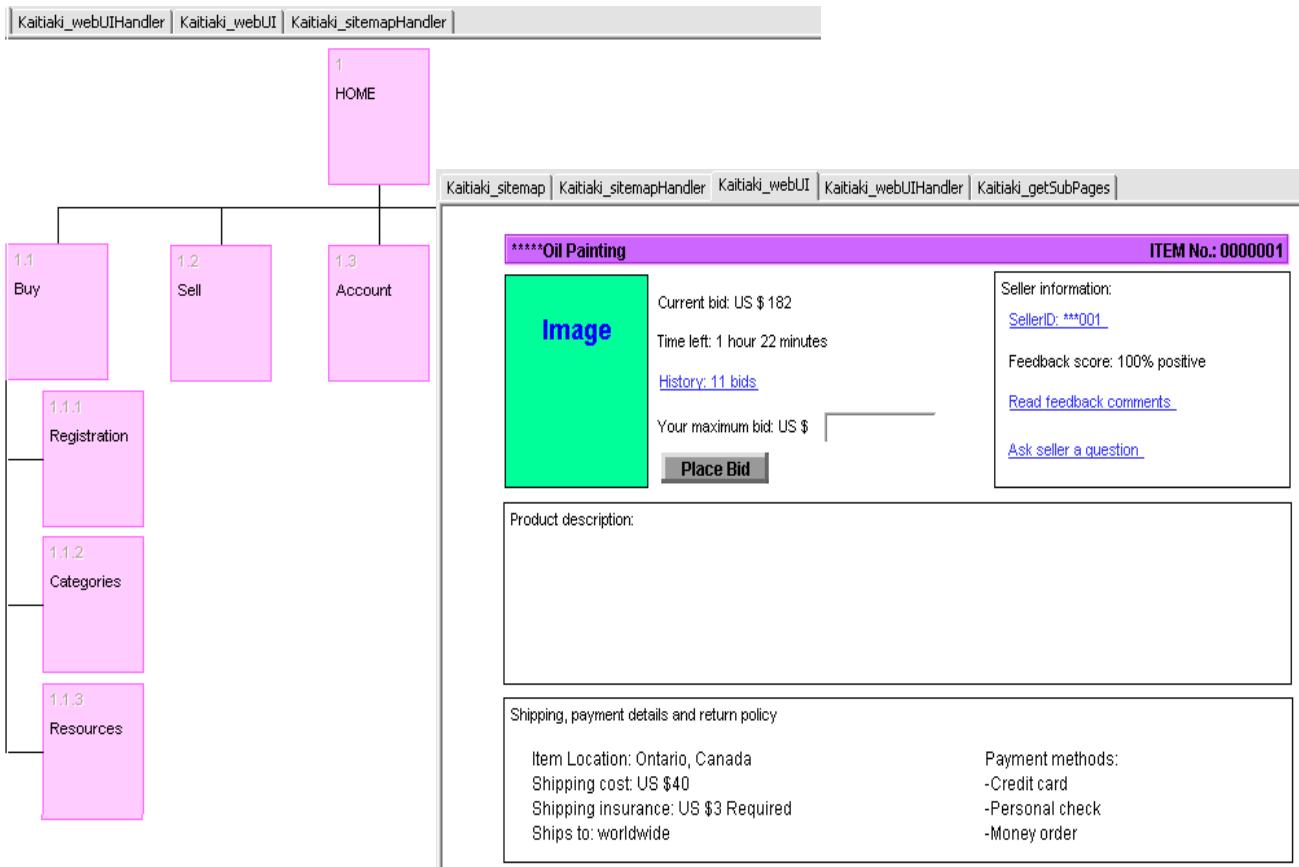


Figure 6.1. Example of a diagram-based design tool.

A variety of approaches have been used to support reconfiguration of diagramming tools. Frameworks, such as Suite (Dewan and Choudhary, 1991), Meta-Moose (Ferguson et al, 1999) and Unidraw (Vlissides and Linton, 1989) require modifications to the tool's code, with an edit-compile-run cycle. Some Tcl/Tk-based tools may be modified while in use (Welch and Jones, 2003), but this requires use of the Tcl programming language. MetaEdit+ (Kelly et al, 1996) and GME (Ledeczi et al, 2001) provide API based code integration facilities, but code must be pre-compiled. Usually only programmers familiar with the tool architecture can make such modifications.

A common alternative approach supporting run-time modification is scripting. This is supported, for example, by Amulet (Myers, 1997) and Peltonen's UML tool (Peltonen, 2000). MetaEdit+ also provides a custom scripting language for report generation while GME uses OCL as a scripting language for constraint specification. These are difficult for non-programmer users to understand and use. Pounamu uses this approach, with event handlers specified using textual Java fragments accessing a defined API and compiled on-the-fly. Figure 6.2 shows a Pounamu event handler for a web site design tool. This is a powerful mechanism for extending Pounamu and very sophisticated

event handling behaviour has been implemented with it. As we reported in Chapter 3, while end users have been very complimentary of Pounamu's visual design tools, they have been less complimentary about the event handler specification as it requires programming skills and knowledge of the Pounamu API even for simple handlers.

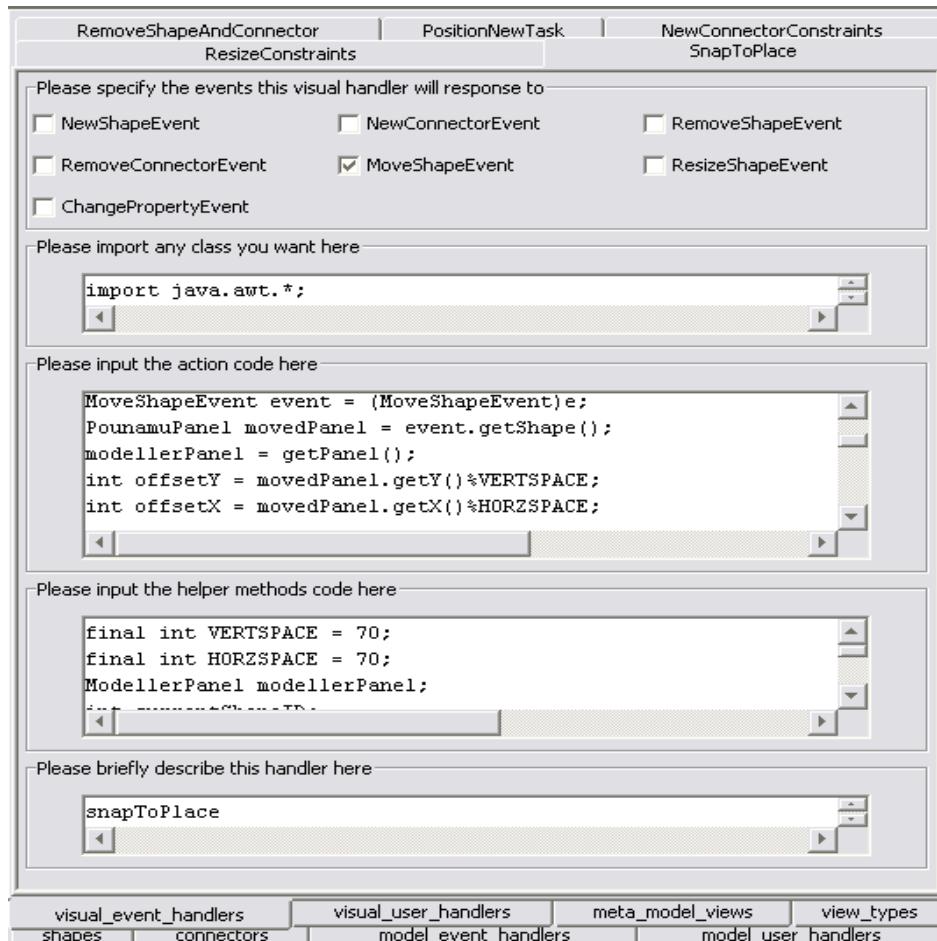


Figure 6.2. Example of event handler textual specification.

Programming by demonstration and rule-based approaches have been used to specify behavioural constraints in some systems, often together and most notably in children's programming environments such as KidSim (Smith et al, 1995) and Agentsheets (Repenning and Sumnet, 1995). Most rule-based approaches exemplify "Event-Condition-Action" based visual languages where the user specifies an event of interest; conditions ("filters") when the action(s) should be run in response to the event; and action(s) to run to modify the tool's state.

Other Event-Condition-Action rule-based languages have been developed for a variety of domains, including building and tailoring design tools (Costagliola et al, 2002; Ledeczi et al, 2001; Lewicki

and Fisher, 1996), user interface event handling (Berndtsson et al, 1999; Jacob, 1996), process modelling (Grundy et al, 1998) and database rule handling (Matskin and Montesi, 1998). However these approaches often suffer from use of inappropriate, textual rule-based languages for end users; reliance on many abstract concepts like control structures and variables; limitations on expressive power of the languages; difficulty in visualising and debugging learned rules from demonstration by the user; and limitations of reconfiguration power, including compile-time rather than run-time changes.

6.3 Requirements

Given the problems noted above, we wanted to replace Pounamu’s textual, Java code-based event handler specification tool with one using a visual language suitable for non-programmer end users. To develop this replacement visual language, Kaitiaki, and its specification tool we carried out an analysis of Pounamu event handlers from a wide range of tools to identify key constructs used to specify different tool behaviours. All had aspects of (1) specifying the event(s) of interest; (2) querying the tool state in various ways; (3) filtering event/query results and making decisions; and (4) performing state changing actions on filtered objects. We also looked at the metaphors used in existing rule-based and event-condition-action event handler specification tools to see how these manifested the behavioural specifications and how suitable these were for end users. From this analysis and survey, we developed a set of key requirements and design approaches for our new Kaitiaki visual event handler designer:

- A need to represent key “building blocks” of state query, data filtering and state modification (actions).
- A need to represent event objects and their attributes; various objects from the Pounamu tool state (both view and model); and query results (typically collections of Pounamu state objects).
- A need to represent “data” propagation between event, query, filter and action representations.
- A need to represent iteration and conditional data flow.

6.4 Metaphor

The metaphor used by Kaitiaki is thus an “Event-Query-Filter–Action” (EQFA) model. This is articulated as “When this event happens, I want these changes made to these things”. This is loosely based on the Serendipity event handling language which has been successfully used by end users in the process enactment domain to express similar kinds of event-driven behavioural models (Grundy et al, 1998). The key visual constructs of our language are representations of events, tool objects,

queries on a tool's object state, state changing actions (including primitives relevant to common event handler requirements), and data flow links between these.

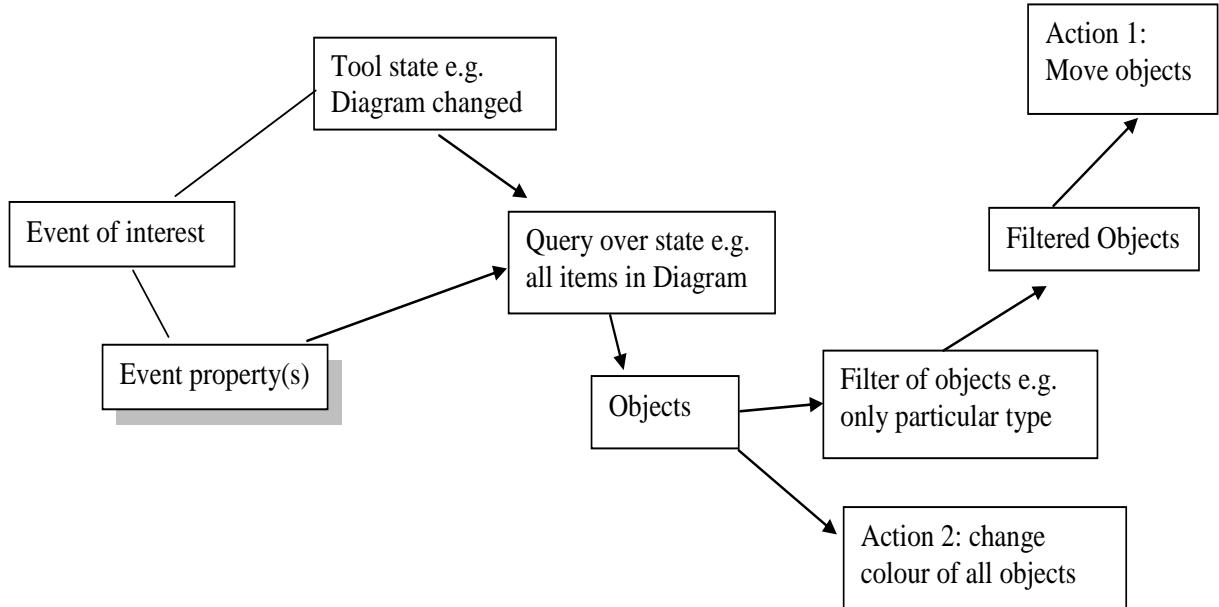


Figure 6.3. The Kaitiaki EQFA metaphor.

A Kaitiaki event specification is conceptually of the form outlined in Figure 6.3. An end user selects an event type of interest; adds queries on the event and Pounamu tool state (usually diagram content or model objects that triggered the event); specifies conditional or iterative filtering of the event/tool state data; and then appropriate state-changing actions to be performed on target tool state objects.

Complex event handlers can be built up in parts and queries, filters and actions can be parameterised, and reused. Ordering is handled by dependency analysis in the code generator. Domain specific tool icons are also incorporated into the visual specification of event handling as placeholders for the Pounamu state, to annotate and make the language more expressive.

6.5 Notation

The design of our Kaitiaki visual language focuses on supporting modularity and explicitly representing data propagation. We have avoided using abstract control structures and adhered to a dataflow paradigm to reduce the user's cognitive load. An overview of the main constructs of Kaitiaki is shown in Table 6.1 with an example Kaitiaki event handler view shown in Figure 6.5. From this we see the visual form of the constructs described in the previous section, i.e. events, filters, tool state queries, and actions plus iteration over collections of objects, dataflow input and output ports and connectors, and concrete iconic forms.

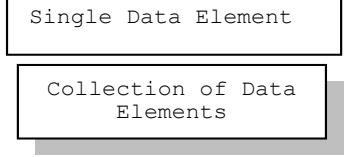
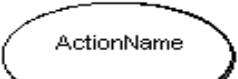
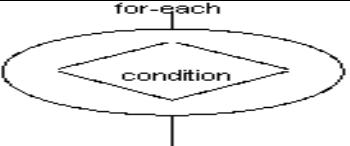
Event representation	
Abstract Pounamu state representation	
Filter	
Query on a tool's state	
State changing action	
Iteration	
Data propagation link	
Data flow ports in and out	
Concrete specification of Pounamu model elements (state)	

Table 6.1. Kaitiaki language key visual constructs.

A single event or a set of events is the starting point for a Kaitiaki event handler specification. From this event various data flows out (event type, affected object(s), property values changed etc). Queries, filters and actions are parameterized with data propagated through incoming connectors. Multiple flows are supported with multiple dataflow connectors pointing to/from a visual construct. Queries retrieve elements and output one or more data elements; filters select elements from their input; actions apply operations to elements passed to them.

Queries and actions are invoked immediately when their actual data parameters are available (data push). If no related data dependency is specified, i.e. no data input parameter flows to the constructs, then queries and actions are invoked on demand when all other parameters to a subsequent flow element have a value (data pull).

Table 6.2 shows some of the predefined primitives for these constructs. These define the core vocabulary for our domain specific language, providing a base set of operations useful for diagram and diagram element manipulation. Typically this involves locating or creating elements, setting their properties, relocating/aligning them, and connecting them.

State querying	
<pre> graph TD name --> getProp{getProp} getProp -- shape --> propValue </pre>	Obtain a named property value of a shape
<pre> graph TD getShapes{getShapes} --> shapes </pre>	Obtain all the shapes in the modeller panel
<pre> graph TD getConnectors{getConnectors} --> connectors </pre>	Obtain all connectors in the modeller panel
<pre> graph TD shape --> getShapeConnectors{getShapeConnectors} getShapeConnectors --> connectors </pre>	Obtain all connectors connected to a shape
Data filtering	
<pre> graph LR shapeType[shapeType] </pre>	Select shapes of specified type from set or test type of single data element input
<pre> graph LR connectorType[connectorType] </pre>	Select a given connector type
<pre> graph LR shapes --> connectedTo[connectedTo] connectedTo --> shapes </pre>	Select all shapes that are connected to a particular shape (i.e. connector source)

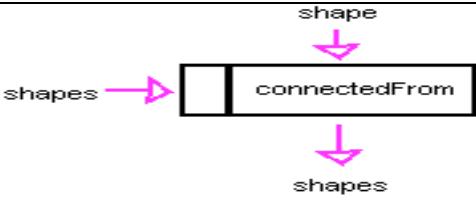
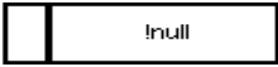
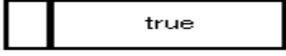
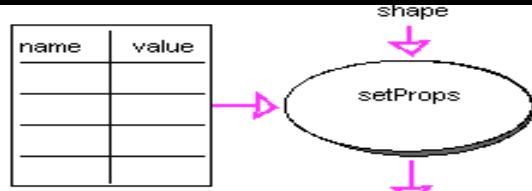
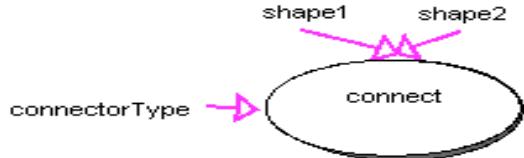
	Select all shapes that are connected from a particular shape (i.e. connector target)
	Filter on a not null value
	Filter on an expression value
State modification	
	Set a list of name-value pair properties for a shape
	Set a value to a named property
	Set a list of values to a named property
	Move a shape by an offset to a location
	Horizontally/vertically align a shape with other aligned shapes
	Create a new shape
	Create a connector of a specified type and connect two shapes using the connector

Table 6.2. Overview of Kaitiaki reusable building blocks.

6.6 Example of Kaitiaki Specifications

To construct a visual event handler specification a user identifies the target affected shape, view or model entity. She specifies the event(s) the event handler should respond to, and then adds building blocks to the handler specification. The concrete representations of Pounamu data, such as the shape icons, allow her to relate her queries, filters and actions to concrete objects in Pounamu. Basic elision support allows the user to show and hide concrete icons, queries, filters and actions to help manage larger specifications. To better illustrate the expressiveness of Kaitiaki, we use an event handler example defined for the web site design tool shown in Figure 6.1. The web site map view (of a simple model like eBay) supports a hierarchical breakdown of web pages for sub-page management. It requires several layout constraints to be enforced.

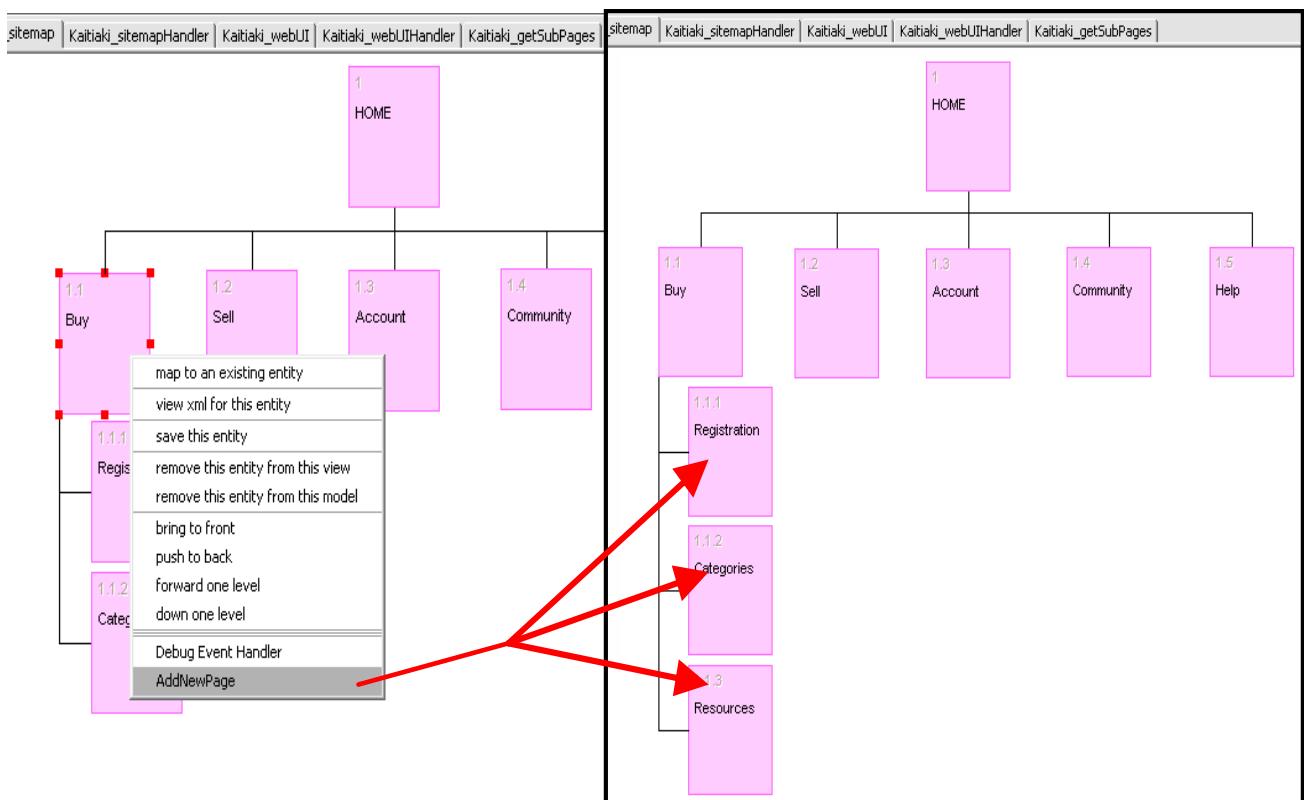


Figure 6.4. Example of addition of a new sub-page.

When creating a page icon for the web site map diagram, several values for its properties need to be set. These are gathered from a range of sources. An event handler is needed to implement one of the layout constraints. Users need to be able to create a new page by a right-click on an existing page; the newly created page is made a child of the existing page and a link is drawn between the old and new pages. The new sub-page and all other sub-pages belonging to this parent are aligned and

repositioned upon arrival of the new page. Figure 6.4 shows the effect of this event handler when a new sub-page is added to the selected.

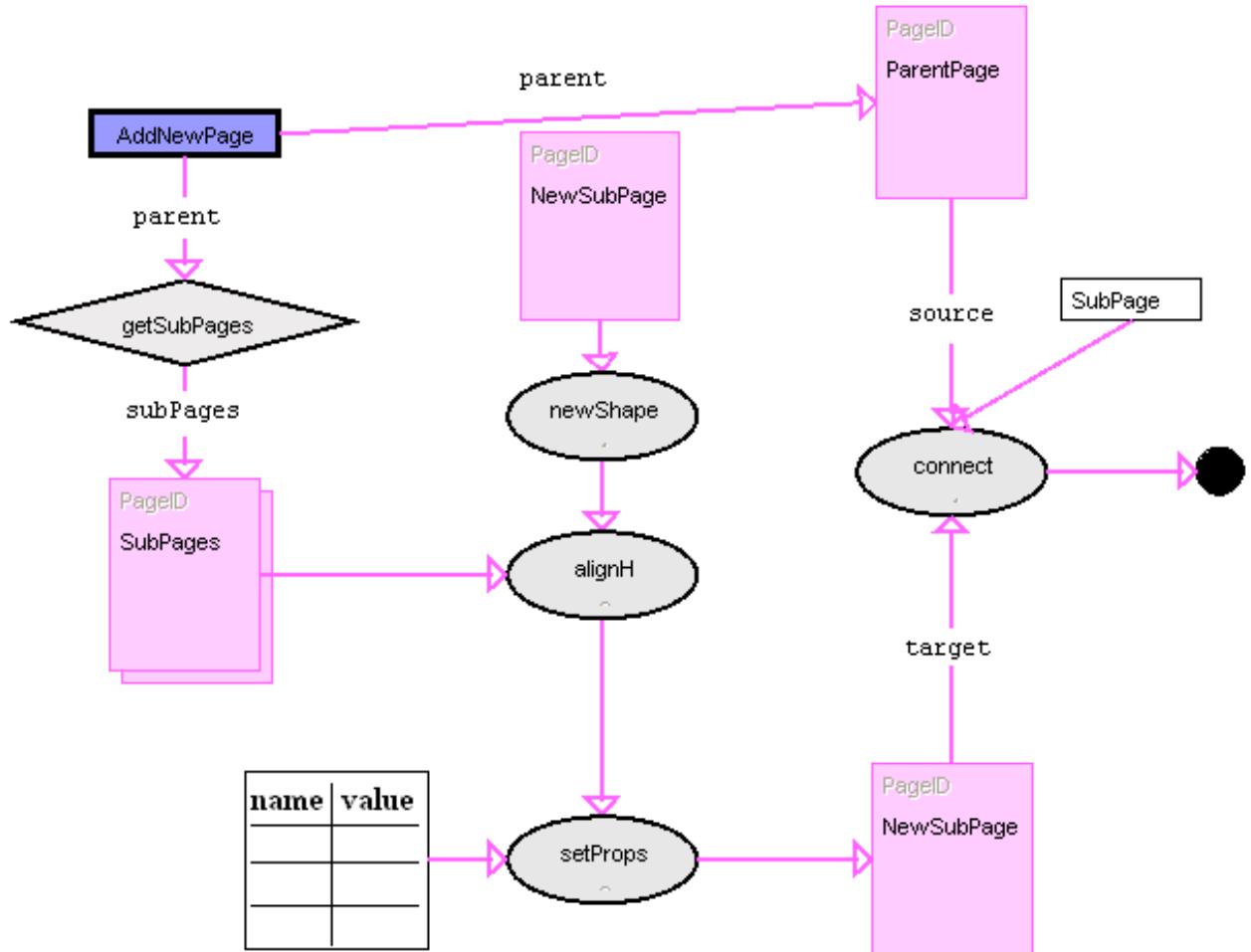


Figure 6.5. Specifying a layout constraint event handler.

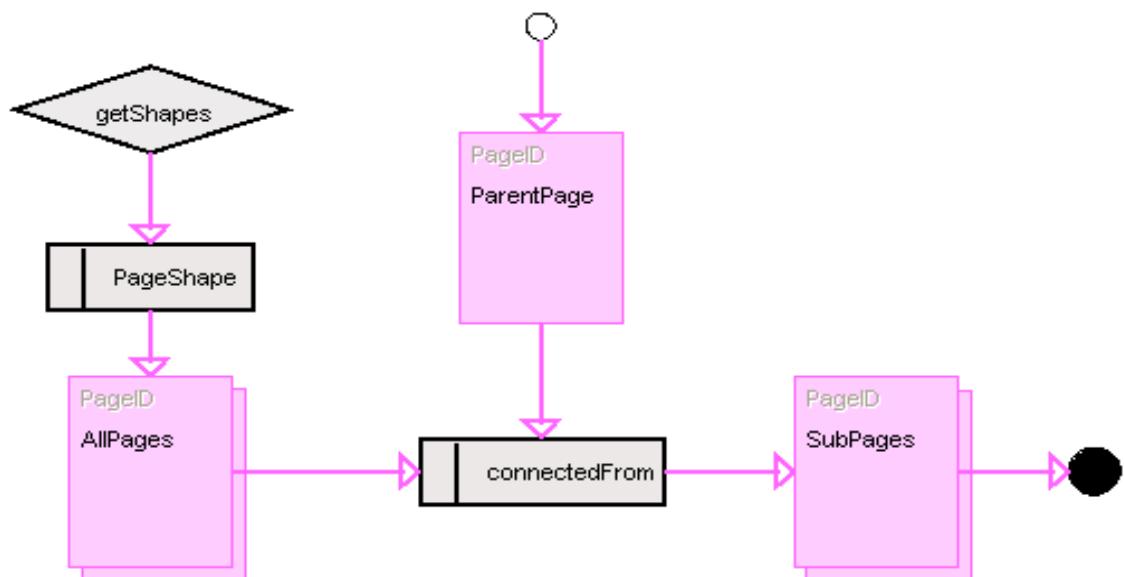


Figure 6.6. An example of a reusable visual query.

The event handler specification for this task is shown in Figure 6.5 which demonstrates the use of predefined Kaitiaki primitives (e.g. create, align and set property and connect shapes). It also demonstrates package and reuse of queries and actions. The modelling constructs contained in this event handler specification include a user defined trigger event (a context-menu event) called “AddNewPage” which has the selected (i.e. Parent) “Shape” flowing from it; a query, “getSubPages” (a packaged query) that locates existing sub-pages of the currently selected page shape (“parent” as propagated to the query); four actions, the “newShape” action creates the new page shape; the “alignH” action does a horizontal alignment (with a user specified vertical distance in between) of the new page shape with the other existing sub-pages; the “setProps” primitive then sets default properties for the newly created page shape; and the “connect” primitive creates a connector of the “SubPage” connector type and connects the new page shape with its parent shape using the connector, now the event handler leads to a final stage, i.e. the end of the event handler specification.

Data sourced from outputs of “source” entities flows through data propagation links to act as input to “sink” entities. Each of the data propagations is statically checked for type compatibility of their data sender and consumer. Also incorporated in the event handler example are some end-user target tool icons, e.g. one on the flow from the “AddNewPage” event to the connect action annotates the flow to visually indicate the type of shape (page) on the flow. Another on the flow from the “setProps” action annotates the flow to indicate that the state change (which sets defaults values) results in modifications to a page shape (the new sub-page). Shadowed icons, such as the one on the “subpages” flow from “getSubPages”, indicate multiplicity in the result. These optional annotations do not affect the semantics and thus are examples of secondary notation augmenting the specification (although their types are checked). They include generic titles (“ParentPage”, “NewSubPage”, etc) to emphasise the reusability of the event handler for other page shapes.

Figure 6.6 shows the packaged “getSubPages” query, which is composed of a number of primitives. We explicitly specify start (data flow in) and end (data flow out) ports for a package. Starting with a parent shape flowing in from the start to the “connectedFrom” filter, the “getShapes” query which gathers all available shapes (via data pull) is invoked. The “PageShape” filter selects all shapes that are of the “PageShape” type. The “connectedFrom” filter then selects only those that are connected from the specified parent shape. The end flow of the composed query indicates that on termination, this query flows out the set of sub-pages of the parent page. This query is invoked in the event

handler in Figure 6.5, but can be reused by other event handlers. Actions and filters can similarly be specified and reused.

6.7 Dynamic Visualisation of Event Handlers

A consequence of introducing a visual language to generate Pounamu event handler code from visual specifications is the need to support their incremental development and debugging. To this end we have developed a visual debugger which dynamically annotates an event handler specification view for a fired event. The viewer exploits the dataflow between event handler building blocks to update a visualisation of event handler execution in its own view.

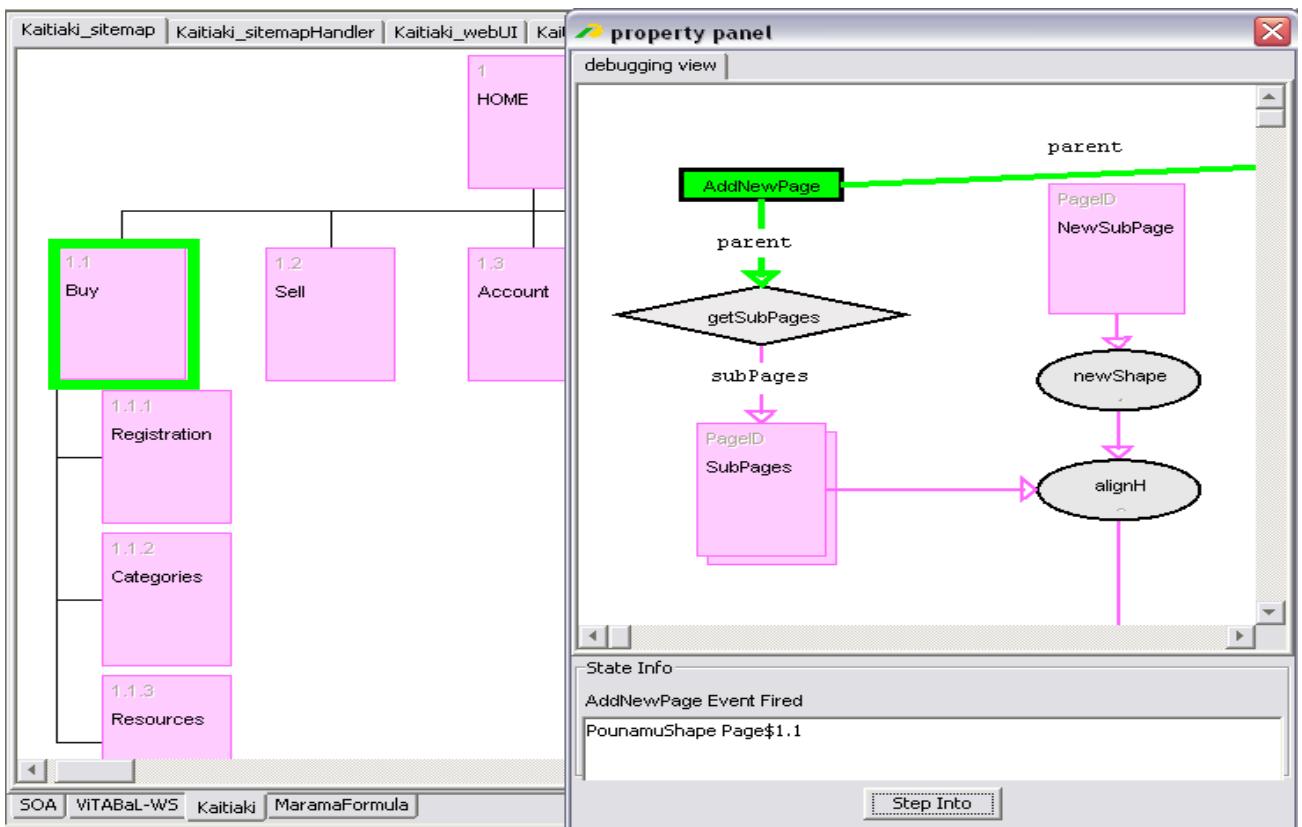


Figure 6.7. Visualising execution of a visual event handler.

The dynamic visualisation of an event handler execution includes the visualisation of EQFA element invocation (by flashing the corresponding node in the graph) and the visualisation of data propagating path to the next construct (by highlighting the dataflow path). The traditional “debug and step into” metaphor is used and step-by-step visualisation controlled by menu command. As seen in Figure 6.7, when the “Step Into” button is clicked, the next element to be invoked and the data propagation path are highlighted and handler execution pauses. The user can then step into the next

element, abort the handler or inspect data values on a propagation path. The final state of the event handler execution highlights all the invoked constructs (nodes coloured in green) and the entire data propagation path. The states of the propagated data are able to be displayed in the debugging state information panel.

6.8 Design and Implementation

We have implemented an environment for Kaitiaki as an extension to Pounamu. As shown in Figure 6.8, the main components added to Pounamu to generate Pounamu event handling code and visualise a running event handler include: Pounamu views and model for specifying visual event handler models; XML-based representation and storage for both library and user-defined queries and actions; and the visual debug viewer.

We have developed form-based specifiers for queries and actions to allow reconfiguration/modification of existing library code modules and creation of new ones by expert users. These are added to the library of reusable building blocks so end users can visually add them to specifications. Query/Action XML DTDs have been defined for Pounamu and XML data files are used for saving to and loading from a library of queries and actions. Visual Kaitiaki nodes are integrated with code modules by the code generator. There is strong coupled mapping of visual components and code components, thus component-based code generation from a specification is achieved. The visual links (connectors) instantiate the visual entity components as they are required i.e. initialise query/action modules and invoke them as needed. The independent use of component-based visual and code components increases the modularity and reusability of the programming constructs.

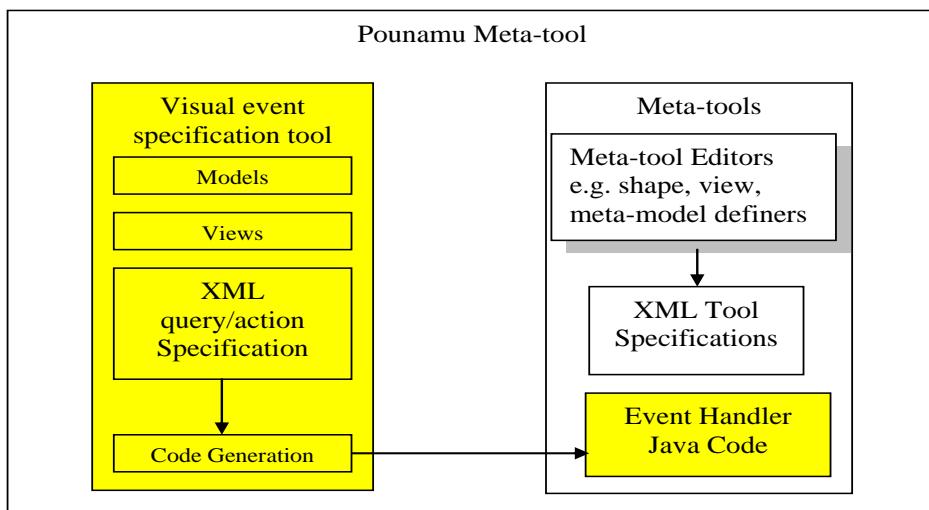


Figure 6.8. Extensions to Pounamu (highlighted).

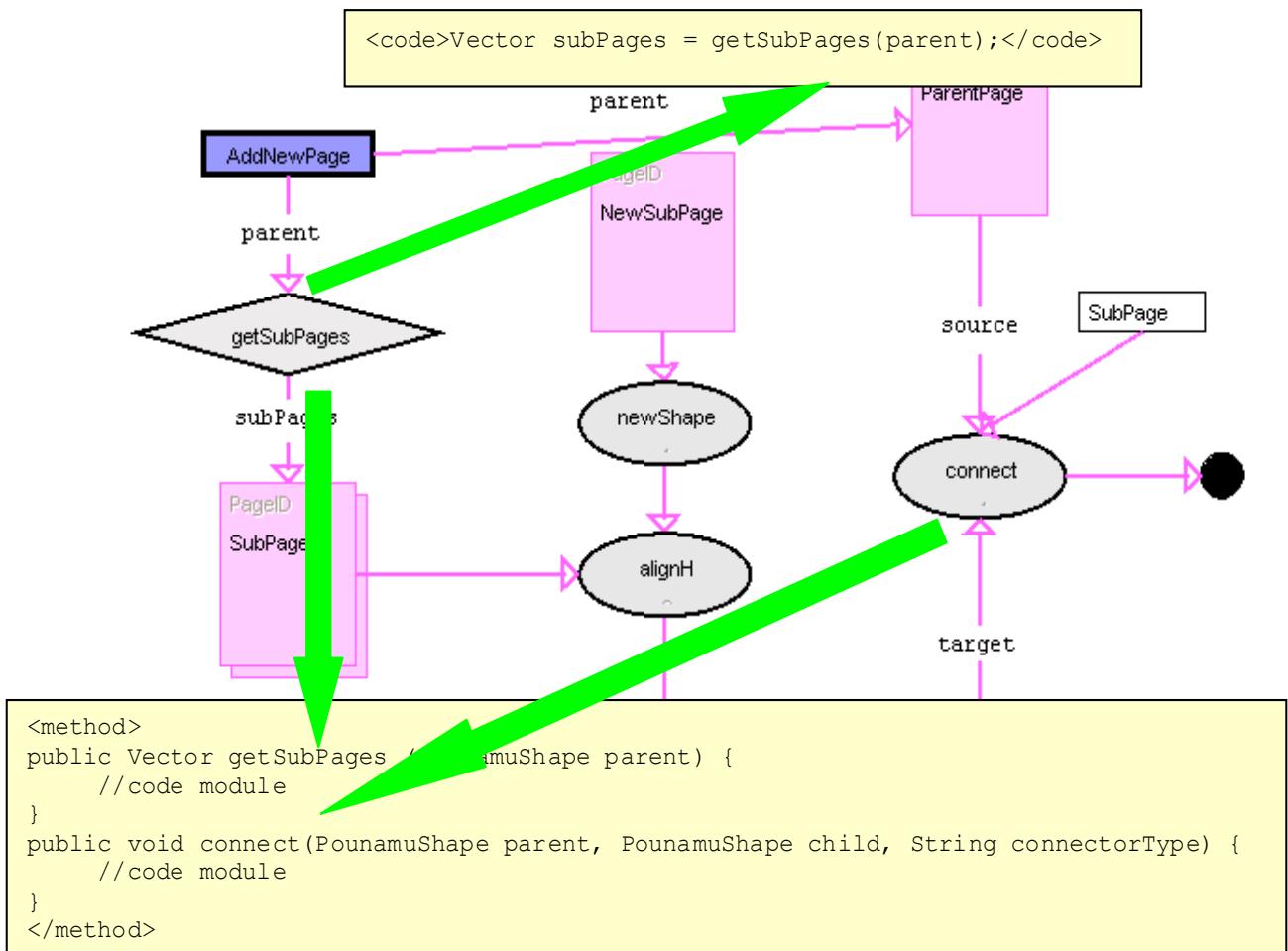


Figure 6.9. Compiling a visual event handler.

The code generator first performs a model (dependency) analysis and then sets module properties obtained from the visual model. It buffers code for creating event instance and query/action invocation, and finally writes the completed event handler code to an XML file. Figure 6.9 shows an example of this translation for the “AddNewPage” specification. Data propagation links instantiate actual method calls to target queries or actions, generating the `<code>` XML construct in the Pounamu event handler XML. Each query and action extracts the reusable, parameterised code from the component. Parameter values are substituted and XML is generated for the `<method>` construct of Pounamu in the event handler XML.

6.9 Discussion and Evaluation

We have carried out a Cognitive Dimensions (Green and Petre, 1996) investigation of our visual event specification language and prototype environment to gauge its effectiveness.

Our target user group are inexperienced Pounamu users. We have chosen a system with a low-to-medium *abstraction gradient*. Abstractions introduced are visual iconic constructs and data flow between them. These abstractions support query/action composition allowing users to specify Pounamu data queries and state changing actions as discrete, linked building blocks in the language. We have chosen a *terse* visual formalism which should allow a relatively good understanding to be obtained. The dataflow metaphor and visual constructs used as primitives in Kaitiaki increases its comprehensibility compared to the Java-based version. Kaitiaki constructs map onto the basic features of our EQFA metaphor, which provides high *closeness of mapping* for our target user group. Concrete representation of Pounamu data elements is supported too. The metaphor is related to the way Pounamu supports event processing and mixes abstract and concrete constructs. The initial abstractions require *hard mental operations* but are mitigated by concrete domain objects.

Kaitiaki allows *secondary notation* to be used to layout, resize and annotate items in the view with iconic and textual labels. End user domain icons (i.e. concrete representation of Pounamu data elements) can be added to mitigate the abstraction of a visual event handler specification, to increase its readability and understandability. Modifying an event handler specification is by direct manipulation and a user can change one module without affecting the rest of the specification. However, it still has *viscosity* problems – that is the nature of dataflow systems – the user typically has to do a bit of rearranging (e.g. connector removal and reattachment, icon shuffling) to insert an element.

The existing Java-based Pounamu event handler designer is very error-prone for both novice and experienced users due to reliance on API knowledge and Java coding. Kaitiaki reduces some areas of *error proneness* by hiding API details and using data flow and visual constructs. However, as the specification is still an abstraction users can still specify faulty behaviour. Kaitiaki allows *progressive evaluation* of a visual event handler specification even when it is partially complete. Modifications to event handlers take effect immediately after re-registration in an end user tool. The visual debugger allows a user to step through a handler's elements and view data, which is not supported by the Java code based event handler. The current tool has reasonable *visibility and juxtaposability*. Information for each element of an event handler is readily accessible. The visualisation of a running event handler is juxtaposed with the modelling view that triggers its execution. But conversely that does not happen when the user is designing the event handler – the user has to switch between the views. *Hidden dependency* is introduced in both Pounamu and its specified tools to manage consistency in multiple views, e.g. between view and shape specifications

and the event handler specification.

An informal evaluation of the visual event handler specification tool has been carried out with experienced Pounamu users and some novice users. Feedback suggests the visual specification approach is greatly favoured for most event handler specification tasks. We plan a more formal evaluation with novice users to better gauge this.

With respect to requirements, our EQFA metaphor captures event generation, state querying, filtering and iteration over query results, and state change actions to describe event handler specifications. The dataflow metaphor describes the composition of these event specification building blocks and seems to map well onto users' cognitive perception of the metaphor. Packing complex parts of a specification into reusable building blocks allows very complex event handlers to be defined with the model. A proof of concept support tool has demonstrated the approach is feasible permitting both simple and complex Pounamu event handlers to be defined visually, code to be generated for them and visual debugging of them supported.

A potential weakness of Kaitiaki is the abstract representation of all events, queries, filters and actions. We have attempted to mitigate this with the addition of concrete iconic representations and are experimenting with elision techniques that allow concrete icons and Kaitiaki elements to be collapsed into a single meaningful icon.

6.10 Summary

We have developed a prototype visual language and proof of concept support environment for specifying diagramming tool event handlers. This uses a metaphor of generating event, tool state queries, filters over query results and state changing actions, with dataflow between these building blocks. The support environment allows users to compose handlers from these constructs and relate them to concrete diagramming tool objects. A debugger uses the visual notation to step through a specification, animating constructs and affected diagram objects. We have added this tool to the Pounamu meta-diagramming tool and specified and generated event handlers for example tools, demonstrating the feasibility of the approach.

This is the second of the three exemplars used to generalise our generic event handling framework. Kaitiaki shares some commonalities with ViTABaL-WS, typically the dataflow used in the

specification to describe propagation of event and data. The major difference between Kaitiaki and ViTABaL-WS is their abstraction gradients. The generalisation of these two visual languages is described in details in Chapter 8. The third exemplar to be described in the next chapter is declarative dependency and constraint specification in metamodeling tools. The third exemplar has attempted a thoroughly different metaphor from the dataflow like icon and connector approach to event handling specification.

Chapter 7 - Visual Relational Formula Specification

Our third exemplar towards generating a generic event handling framework is the declarative specification of dependencies and constraints in metamodels. This chapter elaborates the metaphor used in this application domain with examples. It is largely based on the MaramaTatau (Liu et al, 2007) paper in Proceedings of the 2007 IEEE Symposium on Visual Languages and Human-Centric Computing.

It is increasingly common to use metatools to specify and generate domain specific visual language tools. A common problem for such metatools is specification of model level behaviours, such as constraints and dependencies. These often need to be specified using conventional code in the form of event handlers or the like. Our ViTABaL-WS and Kaitiaki approaches are inefficient for specifying such constraints on metamodels. We report our experience in integrating a declarative constraint/dependency specification mechanism into a domain specific visual language metatool, focussing on the tradeoffs we have made in the notational design and environmental support used. The expressive power of the mechanism developed is illustrated by a substantial case study where we have redeveloped a complex visual tool for architectural modelling, eliminating conventional event handlers.

7.1 Introduction

It is increasingly common to use metatools to specify and generate domain specific visual design tools. Examples of such metatools include MetaEdit+ (Kelly et al, 1996), GME (Ledeczi et al, 2001), Eclipse GMF (Eclipse, 2007), and Microsoft DSL Tools (MSDN, 2005) together with the locally developed Pounamu (Zhu et al, 2007). High-level visual specifications of tool metamodels and visual language notations allow end users to modify aspects of their tools such as appearance of icons and composition of views and metamodels.

However, an area that commonly proves difficult for meta-tool designers is the specification of model level behaviours, such as constraints, dependencies, element initialisations, calculations, etc.

Most approaches for model behaviour specifications use conventional code in the form of event handlers or constraint expressions. For example, Pounamu uses Java-based event handlers, GMF and GME use textual OCL (OMG, 2003) expressions, and MetaEdit+ uses a combination of constraint wizards and external code snippets. The difficulty with all of these approaches is that the resulting behavioural specifications are not strongly integrated with the visual metamodel, resulting in a variety of, in cognitive dimensions (Green and Petre, 1996) terms, hidden dependency, consistency, juxtaposability and visibility issues.

In this chapter we describe MaramaTatau, an extension to the locally developed Marama metatool set, which provides the ability to specify behavioural extensions to Marama metamodels. Although, like GMF and GME, the behaviours have an OCL formula basis, we have attempted in the environment design to mitigate the hidden dependency, consistency and visibility issues noted above. In the following section we motivate and background our work in more detail. We then describe our new approach, using a simple example to illustrate. A more detailed case study follows, showing the reengineering of a previously developed tool, in the process eliminating complex handler code. We discuss the implications of our work then summarise the results achieved and proposing further work.

7.2 From Pounamu to Marama

Pounamu, like other meta-tools, provides a set of editing tools that realise its meta-tool specifications allowing end users to model using the generated domain-specific modelling tools. However, like most other meta-tools Pounamu-generated modelling tools are difficult to integrate with other tools, provide their own look-and-feel and do not produce “commercial quality” IDE user interfaces and support facilities. They rely on custom code generation, plug-in extension and Computer Supported Cooperative Work (CSCW) support mechanisms (Grundy et al, 2006).

Marama has been developed by members of our research group as a set of Eclipse plug-ins that read high-level Pounamu meta-tool specifications and realises multi-view, multi-user graphical editors in the Eclipse IDE. Grundy et al (Grundy et al, 2006) described the Marama approach, its architecture and implementation. We reinstate Marama in this section before we describe the MaramaTatau approach, as MaramaTatau is implemented as an extension to the Marama framework.

Figure 7.1 shows the approach used to realise Eclipse-based DSVL tools with Marama. A tool developer or user creates or modifies a tool specification using the Pounamu meta-toolset (1). This specification is written to an XML-encoded format (2), which is read by the Marama Eclipse plug-in to configure editing tools (3). On reading a tool specification Marama creates a shared model and one or more graphical editors conforming to the Pounamu-generated specification (4). GEF was used to realise the graphical editors and EMF to represent model and diagram state. Model and diagram state are saved and loaded to XML files or an XML database using the OMG XMI common exchange format via EMF's built-in capabilities (5).

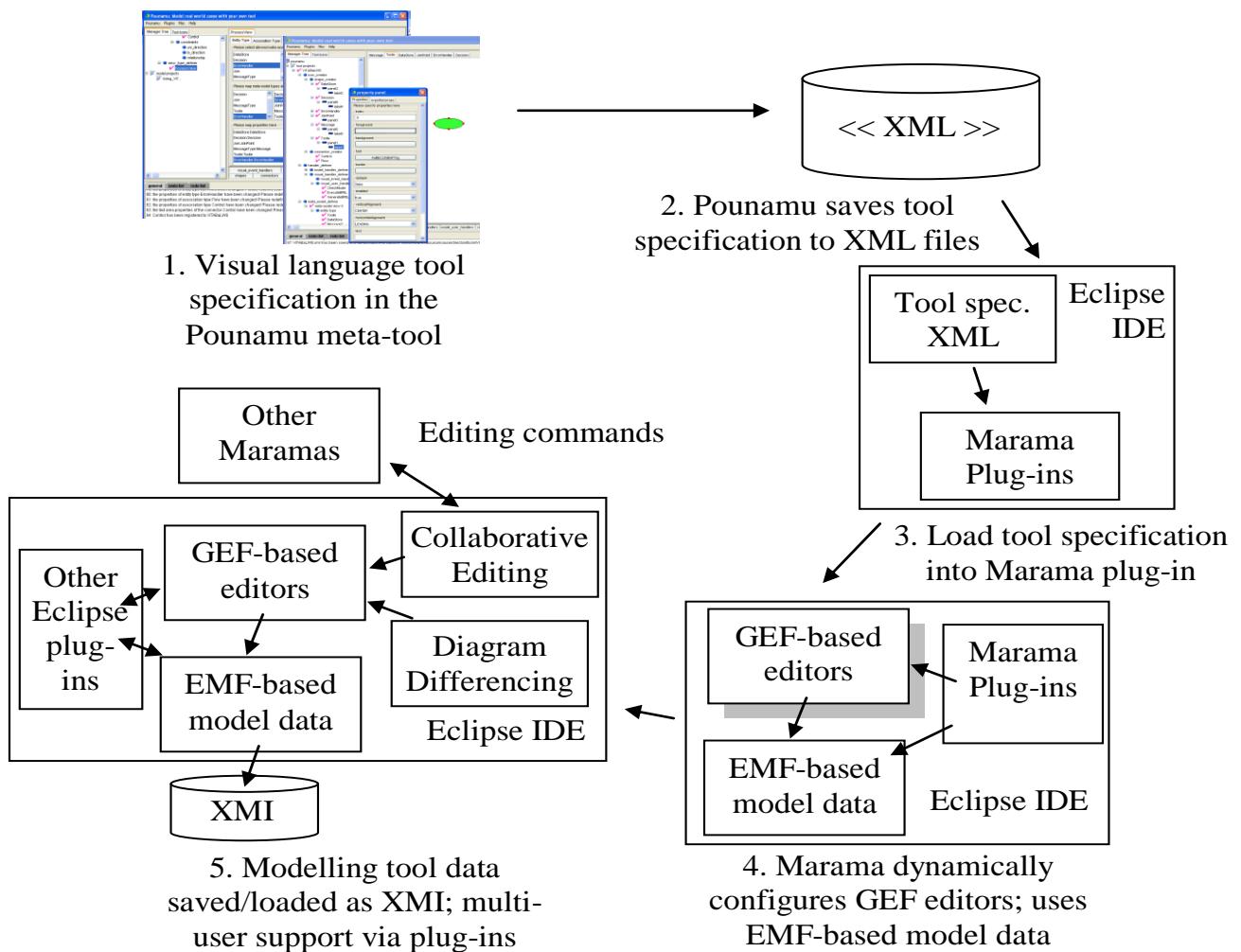


Figure 7.1. The Marama approach to realising Eclipse-based visual language tools. (Grundy et al, 2006)

Figure 7.2 shows a high-level architecture view of the Pounamu meta-tool and Marama Eclipse plug-ins. Pounamu tool specifications represented in XML format are saved to tool projects (1), hierarchically organised directories or ZIP archives. Compiled event handlers are stored as Java

.class files. Users of Marama locate a desired existing Marama project to open or request a project be created via the standard Eclipse resource browser (2). When a project is re-opened or created in Marama, the corresponding Pounamu tool specification files are read and loaded into DOM objects (3). These are parsed and provide an in-memory representation of the Marama tool configuration. This tool configuration is used to configure an EMF-based in-memory model of both model and view (diagram) data (the names and properties of all entities, associations, shapes and connectors). It is also used to produce the editing controls of Marama GEF-based diagram editors (i.e. the allowable shapes and connectors; the rendering of shapes and connectors; the editable attributes of shapes and connectors, etc) (4). When a diagram is opened, Marama configures a GEF editor and renders the diagram (5).

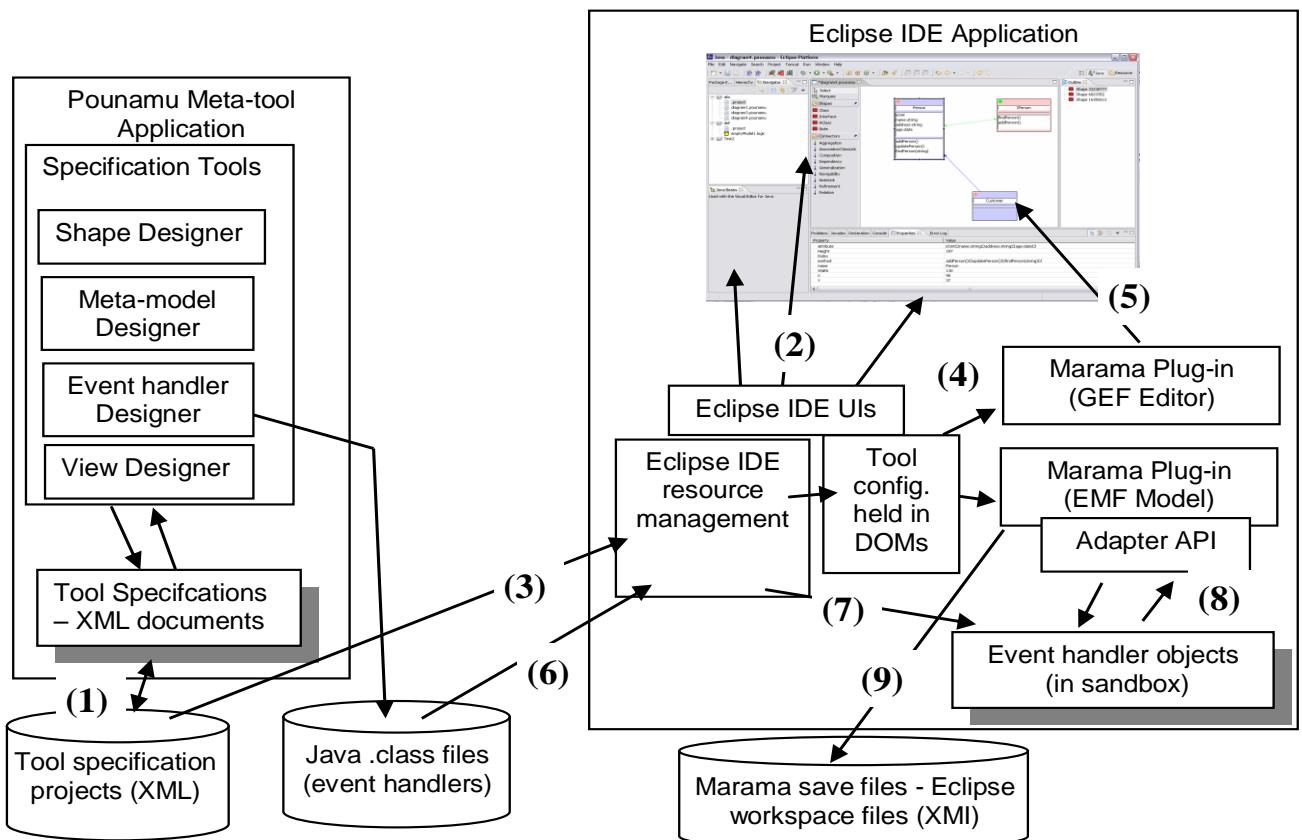


Figure 7.2. The architecture of Marama. (Grundy et al, 2006)

Event handler code is compiled by Pounamu to Java .class files and stored in the tool project directory structure or ZIP archive. Marama loads all event handler compiled classes (6) during tool configuration load time. However, as these classes were compiled to use the Pounamu editing tool's API, they are run in a special sandbox within the Marama plug-in inside Eclipse. A set of adapter classes look to the compiled event handlers like the Pounamu editor API but map Pounamu API calls

onto the Marama Eclipse plug-in APIs (7). When Marama view or model data is updated, the Marama EMF objects are wrapped by Pounamu API adaptor objects and events are sent to the loaded Pounamu-compiled event handler classes. These can then invoke methods on the wrapping adapter classes which are translated into EMF object requests and updates (8). This saves complex conversion of Pounamu event handler code into native Marama form (Grundy et al, 2006).

Marama uses EMF's XMI save and load support to store and load modelling project data (9). Model entity and association instances are written to a .model file, while each diagram and its shape and connector data are written to a separate .view file, all managed within the Eclipse resource workspace. Alternatively an XML database or object to relational database layer can be used for this. Several of these exist for generic EMF model persistency. Stand-alone diagrams can be created and used without a model and a subset of all diagrams for a shared model can be opened at one time. Consistency is supported between views sharing the same information by immediate update if all views are in memory, or differencing and then merging when a view is reloaded (Grundy et al, 2006).

Marama is realised by reusing a number of Eclipse frameworks to implement a dynamic interpreter for the Pounamu-generated DSVL tool specifications. Figure 7.3 illustrates the structure of Marama. Tool specifications are loaded from Pounamu XML files into Document Object Model (DOM) structures. A set of Marama metamodel classes provide an interface to the tool specifications (1). Marama Models use the Eclipse Modelling Framework (EMF) to represent model (entities and associations) and view (diagrams, shapes and connectors) data. When creating or re-opening a Marama project or diagram, these are configured using the DOM derived Marama meta-tool specification objects (2). These define allowed diagram, shape, connector, entity and association types, and their attributes and relationship constraints. When rendering a diagram, Marama EditPart objects create Marama figure objects based on the Marama meta-tool diagram specifications. Figure objects read diagram data and metamodel shape and connector appearance specifications (3) using them to instantiate the diagram via draw2d figures, resulting in a rendered diagram in a GEF window (4). When selected, properties associated with a shape or connector are displayed, with values fetched from the diagram shape/connector and any associated model entity/association, using a standard Eclipse property sheet. Edits to a Marama diagram are processed by GEF edit parts (5). A set of specialised edit part factory, policy and edit parts have been implemented for Marama editors. These generate appropriate figure and outline view renderings and Command objects to modify a diagram's model state (6). Changes to diagram objects generate EMF Notification events. These are

used to determine appropriate changes to make to the underlying shared model entities and associations (7). Updates to model entities and associations also result in generation of EMF events. If multiple views contain shapes or connectors sharing the updated model data, the EMF events are used to trigger appropriate update of diagram model data. The diagrams are then re-rendered to reflect the changes (8). Project and diagram model data is written to and from an XMI format using EMF’s XMI reader/writer support (9). The Pounamu meta-tool compiles Java event handler specifications into Java classes that use the Pounamu editing tool APIs. A mechanism was required to load compiled Pounamu API-using event handlers into Marama as automatic translation to using Marama APIs proved too difficult. A “sandbox” approach was adopted where Pounamu-generated event handler objects are dynamically loaded by Marama into a sandbox providing adaptors between the Pounamu APIs and Marama APIs, making the handlers think they are running in the Pounamu editing tool. EMF Notification objects generated by Marama model and diagram objects are sent to Marama objects representing a proxy to the Pounamu event handler objects (10). Marama model and diagram object changes are wrapped by PounamuEvent objects and sent to these Pounamu-native running event handlers (11). These Pounamu-compiled handlers may then read and update the Marama diagram and/or project model data via a set of adaptor classes between Pounamu API calls and Marama API calls. These calls result in updates to Marama model and diagram objects as appropriate or may invoke other Eclipse tools and plug-ins e.g. the JET code generator. (Grundy et al, 2006)

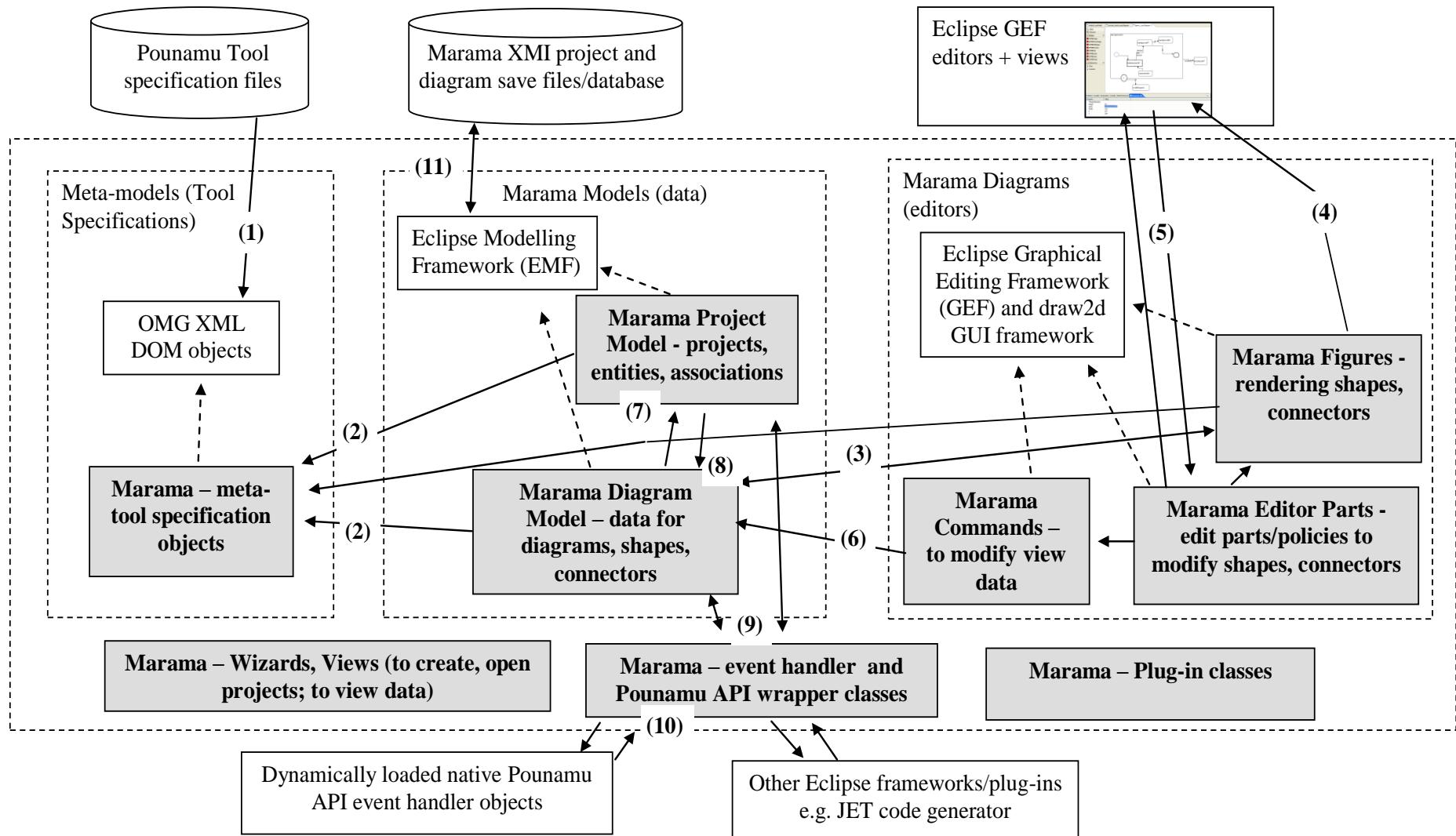


Figure 7.3. Implementation of Marama. (Grundy et al, 2006)

To generalise our work on event handling specifications, we have designed a set of Marama meta-tools to provide a better platform and a vehicle for allowing us to explore event-handling integration. Marama meta-tools provide the visual language design environment similar to Pounamu, but with designer tools realised as a set of refined Marama editors. Marama meta-tools were initially defined using Pounamu, with bootstrapped implementation of a metamodel designer, shape designer and view type designer as the “backbone” of the new metamodelling tool. A complete discussion of the Marama meta-tools development is presented in Chapter 8 and 9.

7.3 Background and Motivation

In our group’s prior work, a variety of frameworks and metatools have been developed to support specification and implementation of multiple view, multiple notation domain specific visual language environments (Grundy et al, 1998; Zhu et al, 2007). In each of these platforms the developers have struggled to find an appropriate means of specifying behaviour, despite having used a variety of approaches. One, used in the JViews framework (Grundy et al, 1998) and Pounamu metatool, was *escape to code* with conventional code accessing tool data structures via an API. This mechanism, also used by MetaEdit+ and DSL Tools, while very powerful is also problematic, requiring much repetitive coding and thorough end user knowledge of the metatool API. It also has significant hidden dependency, visibility and juxtaposability problems due to the differing abstraction levels involved.

A second approach, used in the BuildByWire tool (Mugridge et al, 1998), adopts a concrete visual specification of interface component constraints for use in the JViews framework. This works well for shape and editor constraint specification. Kaitiaki (Liu et al, 2005), described in Chapter 6, uses a dataflow metaphor to specify view level behaviour-oriented constraints for the Pounamu metatool. This is more abstract than BuildByWire, but uses exemplars of user interface components to make the specifications more concrete. These two metaphors do not, however, extend well for model level constraints and dependencies due to the lack of user interface exemplars to “concretise” the model level specifications, and the awkwardness of expressing calculations, common in model level constraints, using these metaphors.

An increasingly common approach is to express model level constraints as declarative formulae. MetaEdit+ uses a combination of wizards to define such constraints and natural language rendering to visualise them. GME and GMF both use OCL expressions to specify constraints and

dependencies. These latter have the advantage of using a standardised and compact notation (OCL) familiar to modellers. These approaches are more successful than “escape to code”, but still involve a large notational and semantic separation between the textual constraint formula and the visually specified metamodel. GME attempts to bridge this gap by annotating visual model elements to indicate constraints apply to them, but editing and understanding a constraint still presents significant hidden dependency and consistency issues.

Formulaic constraints and dependencies are common in spreadsheets (Engels and Erwig, 2005; Burnett et al, 2001). Spreadsheet formulae permit declarative specification of system behaviours and automatic evaluation of them. A highly concrete metaphor is used, however, with the grid structure reused for both formula programming and execution, providing good preservation of the end user’s mental map of the application. This approach is thus not immediately adaptable to the domain of metamodellers as there is necessarily a separation between the metamodel specification and its end user realisation as a set of view editors in a generated application. However, approaches such as ClassSheets (Engels and Erwig, 2005) and Forms/3’s prototype approach (Burnett et al, 2001) provide some indication of how aspects of this metaphor could be adapted to suit the metamodelling domain. Of particular interest are hidden dependency mitigation approaches, such as dependency link views, and the ease of formula construction afforded.

As an adjunct to the redevelopment of Marama, we took the opportunity to address Pounamu’s difficulties in expressing model-level constraints and dependencies. These constraints and dependencies are event triggered (e.g. property change event), to be implemented via event handlers. Both Pounamu and Marama adopt an extended entity relationship (EER) model as the metamodel specification mechanism. The EER model contains definitions of a set of entities, relationships, and attributes. We saw a possibility to extend this simple representation with declarative constraint/dependency specifications. We were attracted to a formulaic approach but wanted to minimise/mitigate the cognitive dimensions tradeoffs involved. This led to the following set of requirements for the constraint representation mechanism:

- Aim for target end users who are programming literate and familiar with modelling concepts
- Ability to represent model level constraints, dependency calculations, and initialisations
- A compact representation
- Use of a standardised notation familiar to the target end users for accessibility of use

- Ability to minimise/mitigate hidden dependency and visibility issues between the constraint specification and the visual metamodel specification
- Ability to rapidly compose constraints
- Ability to simply visualise execution behaviour

In the next section, we introduce MaramaTatau, our approach to implementing these requirements.

7.4 MaramaTatau

MaramaTatau is strongly focussed on structural constraints. The primary notation for constraint representation in MaramaTatau is declarative OCL expressions, a representation chosen for the following reasons:

- OCL expressions are relatively compact (certainly in comparison to Java event handler code).
- OCL is specifically designed as a language to express model level constraints. It thus has primitives for common constraint expression needs, e.g. navigation of relationships, set and list manipulation (including aggregation), and common calculation operations of various types (arithmetic, string, boolean).
- While designed for OO metamodels, OCL is equally applicable to Marama's EER metamodels.
- OCL is a standardised language, likely to be familiar to our intended end users.
- The quality of OCL implementations is increasing.

Providing an OCL expression editor, similar to those in GME and GMF, covers the first four requirements of the previous section. What differentiates our approach, however, is the way we address the other requirements. Our approach is to combine the advantages of the textual OCL formulae with the ease of formula construction afforded by spreadsheets, together with a lightweight, yet robust mechanism to mitigate hidden dependencies.

Figure 7.4 shows the Marama metamodel editor with MaramaTatau extensions. The metamodel shown is for a simple aggregate system modeller, comprising wholes and parts, represented by the Whole and Part entities (1), both generalising to a Type entity and related by a Whole_Part relationship (2). The entities have typed attributes, such “name”, “area”, and “volume”. Below is the formula construction view (3). This allows OCL formulae to be selected, viewed and edited. A list of available OCL functions (4) is used for formula construction. The formula shown “self.parts->

`collect(cost * (1.0 + markup))->sum()`" specifies that the "price" attribute of a whole is calculated by adding the products of its parts' "cost" and "markup" values.

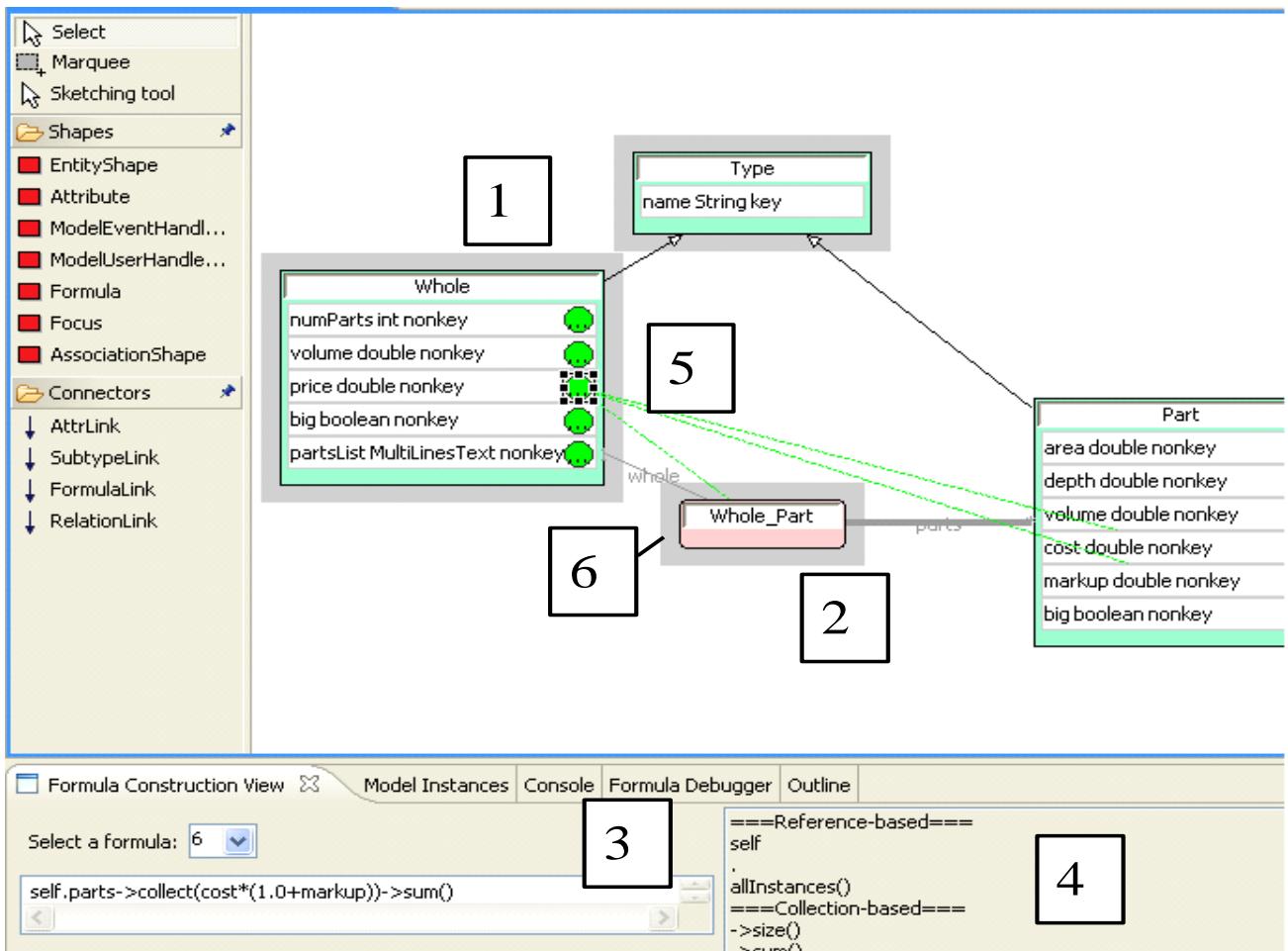


Figure 7.4. MaramaTatau visual notation.

Also shown in the visual metamodel view are various annotations (5) indicating the presence of constraints. Coloured circles placed on attributes or entities indicate that an OCL formula has been defined to respectively calculate their value or provide an invariant constraint over them. All of the attributes of the Whole entity have such formulae, as do the "volume" and "big" attributes of Part. The annotation is coloured differently (red) if the formula is semantically incorrect. Dependency link annotations provide more detailed information about a selected formula by connecting its annotation to other elements used in the formula. For example the formula for the "price" of a Whole entity is selected (selection handles showing). The dependency links show that the price formula is dependent on the "cost" and "markup" attributes of the Parts connected to the Whole by the Whole_Part relationship. Entities and connection paths that are directly accessible when constructing a formula (Whole, Type, Whole_Part) have grey outline borders around them (6, see below).

We have carefully defined the interaction between the two views to enhance visibility and minimise or mitigate hidden dependency issues. Visibility and hidden dependency issues are addressed by the following mechanism:

- The OCL and metamodel editors are juxtaposed together to improve visibility
- Simple annotation of the model elements indicates formulae related to them are present and whether they are semantically correct. This is similar to the GME constraint annotations.
- Formulae can be selected via either the metamodel view annotation or from a selection list in the OCL view. This means constraints can be navigated to/accessed from either view. Selection in one view causes selection in the other.
- The dependency link annotations in the metamodel view provide, at a glance, more detailed understanding of attributes and entities used in the formula. This visualisation extends beyond that of GME, providing a more detailed, constraint specific understanding of dependencies involved. The annotations are modified dynamically as formulae are edited maintaining consistency between the views. The extra annotations are deliberately made visible only when a constraint is selected to minimise clutter, permit scalability, and provide task focussed information to the end user. This approach is similar to dependency visualisations provided in some spreadsheets, linking cells with formulae to those they depend on, but applied to a graphical modelling metaphor rather than a spreadsheet grid. Coloured dependency links and textual element references – as done in some spreadsheets – is a straightforward extension to provide even finer-grained indication of dependencies.

The rapid composition requirement is addressed by several techniques, also adapted from common spreadsheets usage. These assist with hidden dependency and visibility issues. Formula construction can be done either textually, via the OCL view, suitable for those highly OCL fluent, or “visually” via direct manipulation of the metamodel view and function selection list to automatically construct entity, path, and attribute references and function calls. Clicking on attributes in the metamodel view places an appropriate reference to that attribute into the formula. Path references are constructed by clicking on the relationship and then an attribute in the entity referenced by that relationship. A function selected from the list in the OCL view is inserted as a function call into the formula being edited, similar to formula selection in spreadsheets.

A difference from spreadsheet formula construction is that when constructing a formula, only certain elements are semantically sensible at a particular stage of editing whereas in spreadsheets, any cell

may be referenced (circular references excepted). For example, clicking on a Part attribute, without first constructing a relationship reference via Whole_Part, does not make sense. To guide users, grey border highlighting indicates entities and relationship links valid to select at a given point in formula construction. Should a semantically incorrect formula be constructed, the annotation change in the metamodel view provides immediate visual feedback of the error.

Another area of departure from the spreadsheet metaphor is in model instantiation. In spreadsheet based systems the metaphor used is both very concrete and live. The very nature of metatools, where an abstract conceptual metamodel is defined necessarily separately from the views of that model means concreteness must be sacrificed, and hence there is an additional set of hidden dependencies and visibility issues, between the metamodel definitions (including the OCL formulae) and the model instances, created. In designing MaramaTatau's runtime implementation we have introduced several mechanisms to mitigate these hidden dependency issues. Liveness, however, is already well supported in Marama. Unlike almost all other similar metatools, Marama tool definitions can be modified on the fly, with changes immediately reflected in any open tool instances.

Figure 7.5 (1) shows a modelling tool based on the Whole Part metamodel used to edit an example model (the icons and connector forms, and view-model mappings are defined separately using other Marama metatools). When such a model instance is being manipulated (entities and relationships created, property values edited) relevant formulae are interpreted and the derived values assigned to their contextual model entity/relationship properties. For example the parts list in the whole1 Whole entity, represented as a multi-line list in the visual modelling view, has value [part1,part2] constructed using a formula that collects the name of each linked part into a new list. Properties with values defined by formulae are not editable by the end user.

In Figure 7.5 (1), only a single Whole Part view is shown. Marama supports specification of tools with multiple views and multiple notations; each view being mapped to a common underlying model (specified using the metamodel tools). To allow end users to visualise the shared model, a model instance view is provided. Figure 7.5 (2) shows an example of this view for the Whole Part model. The topmost view contains all entity and relationship types defined in the metamodel view. The same element representation is used as in the metamodel specification to minimise/mitigate hidden dependency issues between the metamodel specification and model instance view. Note that we have chosen not to replicate exactly the same view because a Marama metamodel can itself be specified across multiple metamodel views. The model instance view depicts the union of meta elements in all

such views, so does not follow exactly the same layout. This is an area we are still experimenting with. An alternate approach is to provide a set of model instance views, one for each metamodel specification view.

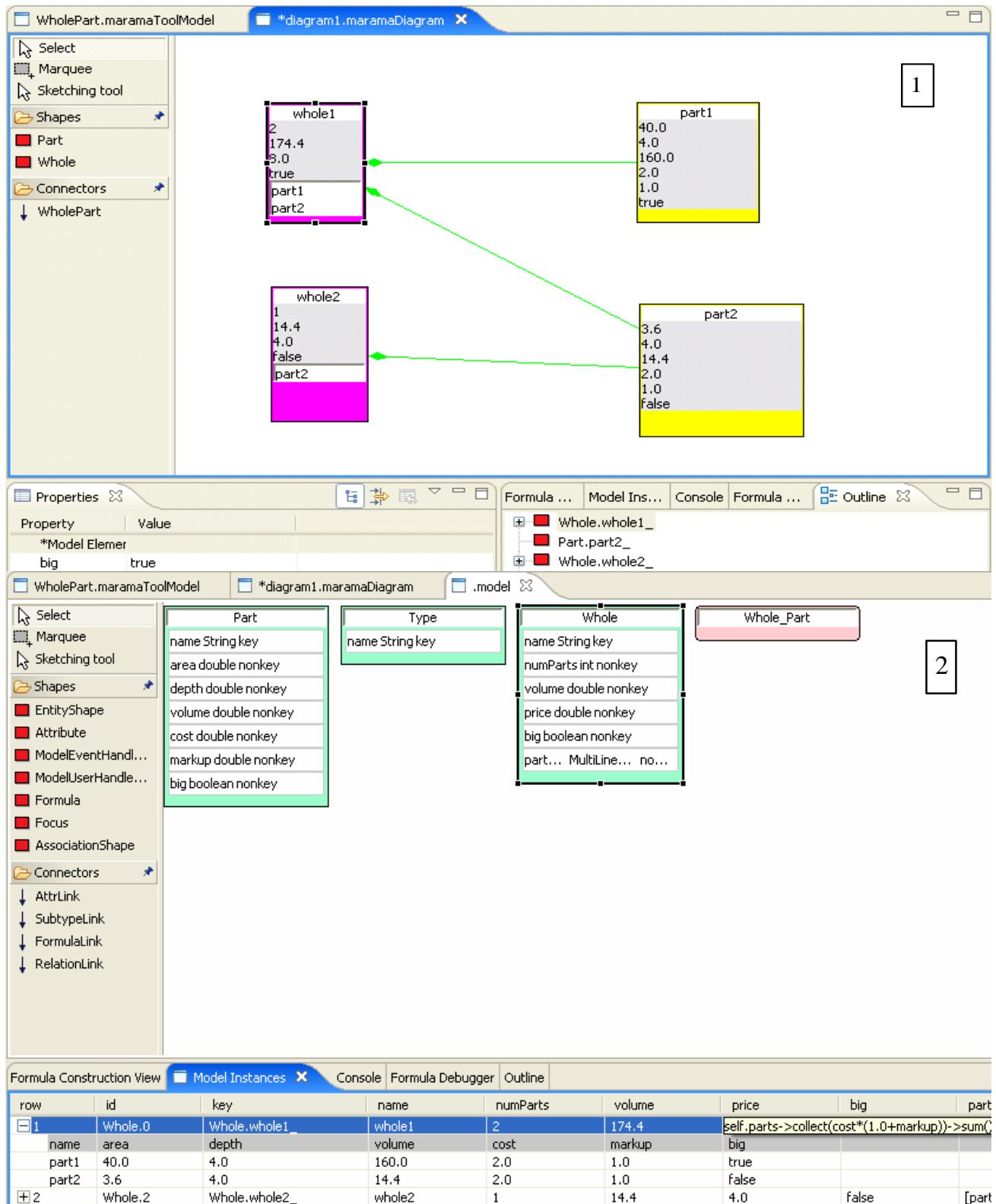
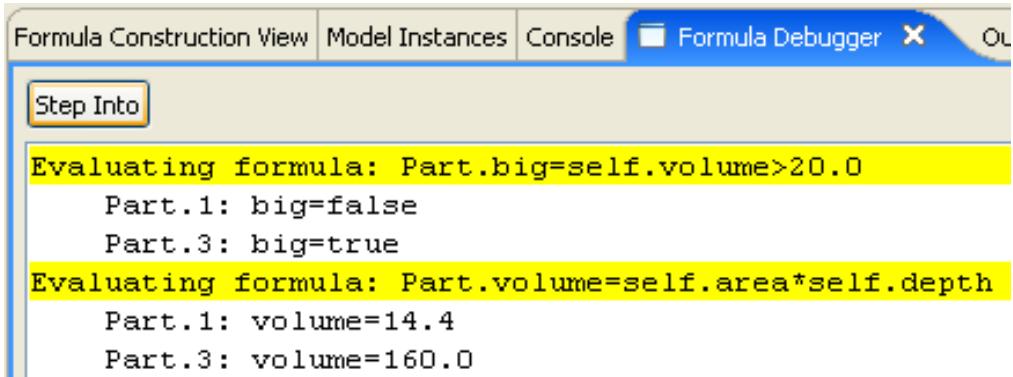


Figure 7.5. (1) Model instantiation view and (2) model instance view.

The table view at the bottom of Figure 7.5 (2) is a spreadsheet like representation of all instances of the element type selected in the top view; the Whole entity in the view shown. Each row details attribute values for an instance of the selected entity. These rows may be expanded, as shown for the first element, to provide details of other elements associated with the chosen element via relationships. In the example shown the two Parts associated with the first of the Whole elements are detailed. This view thus provides a rapid understanding of model elements and related values.

Formulae for calculated attributes are shown by tooltip when the mouse hovers over such an attribute value (as for “price” in Figure 7.5 (2)). This mitigates the hidden dependency between the concrete value and its OCL formula. Further mitigation is provided by a formula debugger view (Figure 7.6). This provides a dynamic, textual visualisation of formula execution, concurrent with changes occurring in the visual views (providing good *visibility* of behavioural changes). These two features together satisfy the final requirement: to simply visualize execution behaviour.



The screenshot shows a software interface with a tab bar at the top containing "Formula Construction View", "Model Instances", "Console", and "Formula Debugger". The "Formula Debugger" tab is active, indicated by a blue border. Below the tabs is a toolbar with a "Step Into" button. The main area displays two sets of OCL evaluation results. The first set, highlighted in yellow, shows the evaluation of `Part.big = self.volume > 20.0`. It lists two parts: `Part.1: big=false` and `Part.3: big=true`. The second set, also highlighted in yellow, shows the evaluation of `Part.volume = self.area * self.depth`. It lists the same two parts with their calculated volumes: `Part.1: volume=14.4` and `Part.3: volume=160.0`.

Figure 7.6. Formula debug view.

7.5 Case study

The previous section introduced the notational features of MaramaTatau plus environment support mechanisms to mitigate hidden dependency and visibility issues. To evaluate the scalability and utility of the approach we present a larger case study reengineering a previously developed Marama tool to replace “escape to code” behavioural specifications with MaramaTatau constraints.

MaramaMTE (Grundy et al, 2006) is a complex visual tool for software architecture design and performance test-bed generation. It provides a number of notational views, including a structural architecture view and a pageflow view for specifying abstract user interface behaviour, all linked to a common underlying model. Figure 7.7 is a screen dump of MaramaMTE in use, with a structural

architecture view describing a three-tier client-server architecture for a travel planning system shown.

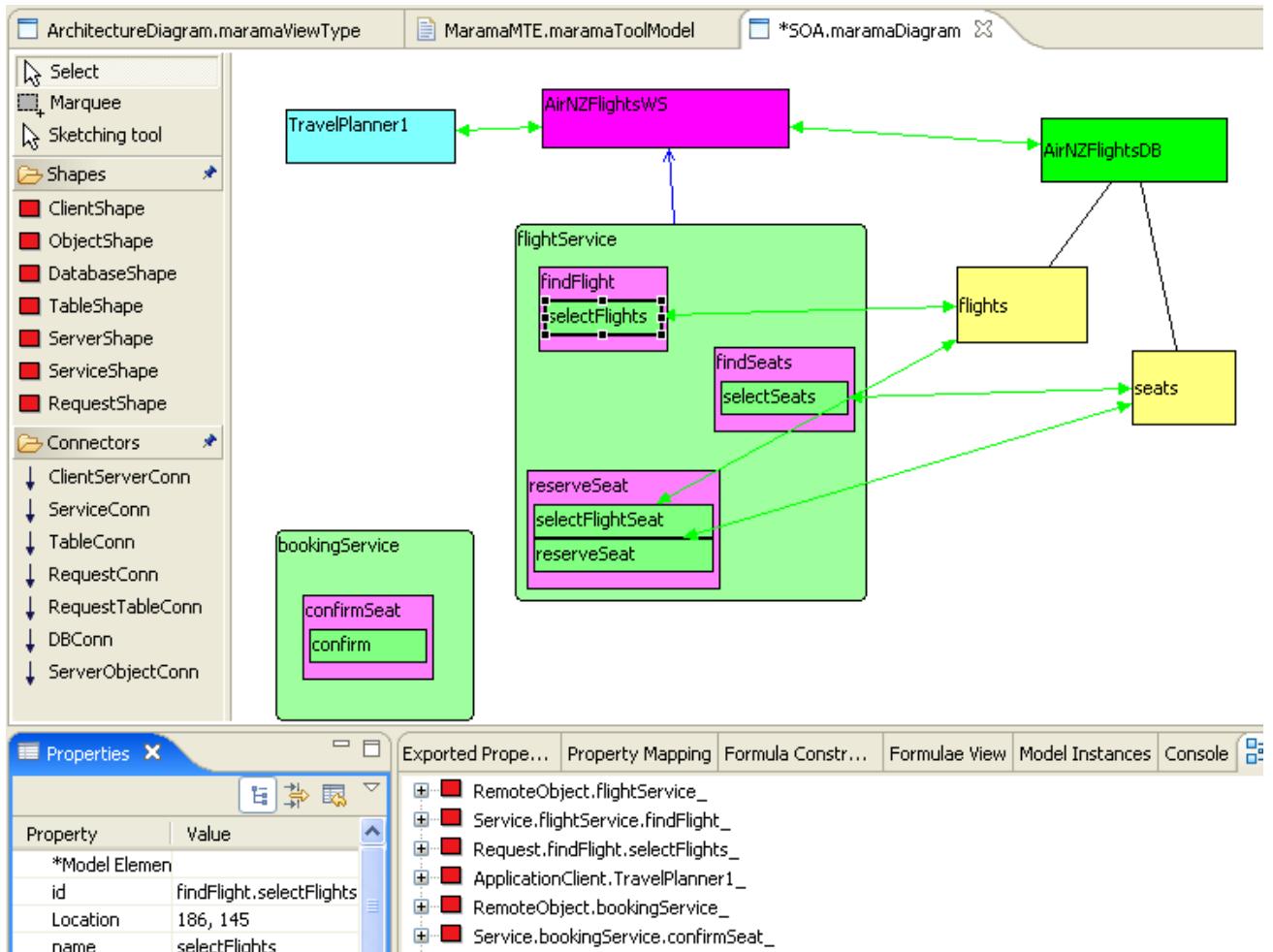


Figure 7.7. A MaramaMTE architecture view

In its original form, the implementation of MaramaMTE required a substantial number of java-based event handlers to implement various event triggered calculations and constraints. Consider remote objects, the rectangular icons containing other icons representing services they provide. For example the “bookingService” remote object has associated a “confirmSeat” service. These remote services have an “id” attribute which is the concatenation of the “name” of the remote object and the “name” of the service (e.g. bookingService.confirmSeat). The handler code implementing this simple constraint is substantial. Part of it is shown in Figure 7.8. Much of the code involved is repetitive or formulaic, manipulating Marama data structures via its API to access attribute values, calculate values, and assign results.

```

/*
public class ManageServices extends SimpleEnclosedShapes {

    String myOwningShapes[] = { "ObjectShape" };
    String mySubshapes[] = { "ServiceShape" };

    public void setDiagram(MaramaDiagram diagram)
    {
        super.setDiagram(diagram);

        OwningShapes = myOwningShapes;
        Subshapes = mySubshapes;

        subshapeConnector = "ServiceConn";

        allowSubshapesOnOwn = true;
    }

    /* (non-Javadoc)
     * @see org.eclipse.emf.common.notify.Adapter#notifyChanged(org.eclipse.e
     */
    public void notifyChanged(Notification notification) {
        super.notifyChanged(notification);

        // auto-generate ID for services
        MaramaConnection conn = connectionAdded(notification, "ServiceConn");
        if(isExecuting() && conn != null &&
           conn.getTarget().getShapeType().equals("ServiceShape") &&
           conn.getTarget().getPropertyValue("name") != null) {
            //System.out.println("added connection "+connectionAddedSource(no
    }
}

```

Figure 7.8. Handler code implementing constraint.

The screen dump in the centre of Figure 7.9 shows a major portion of the metamodel for the reengineered MaramaMTE. A number of formulae have been defined to calculate various attribute values. Below an expanded view of the formulae list shows OCL expressions for each constraint defined. Above an expanded view of part of the metamodel shows the Service and Remote Object entities and the relationship between them plus an OCL formula for the service “id” (formula 8 in the list at the bottom). This expression replaces the complex handler code in Figure 7.8. This specification is not only much more compact, it is also much easier for the end user to understand and reuse.

A range of other constraint expressions are shown in the formula list at the bottom. The first of these is an “id” calculation for service requests similar to the remote object service “id” formula. The next two initialise attributes representing the types of middleware supported by the test bed generator.

These are used in the modeller to constrain the combo-box values selectable by the end user. Those for the “remoteObject” and “remoteService” attributes of Request are moderately complex conditional expressions, which involve tracing a series of relationship paths to derive the names of the remote object and remote service invoking the request. These are thus derived attributes, caching values for more convenient use.

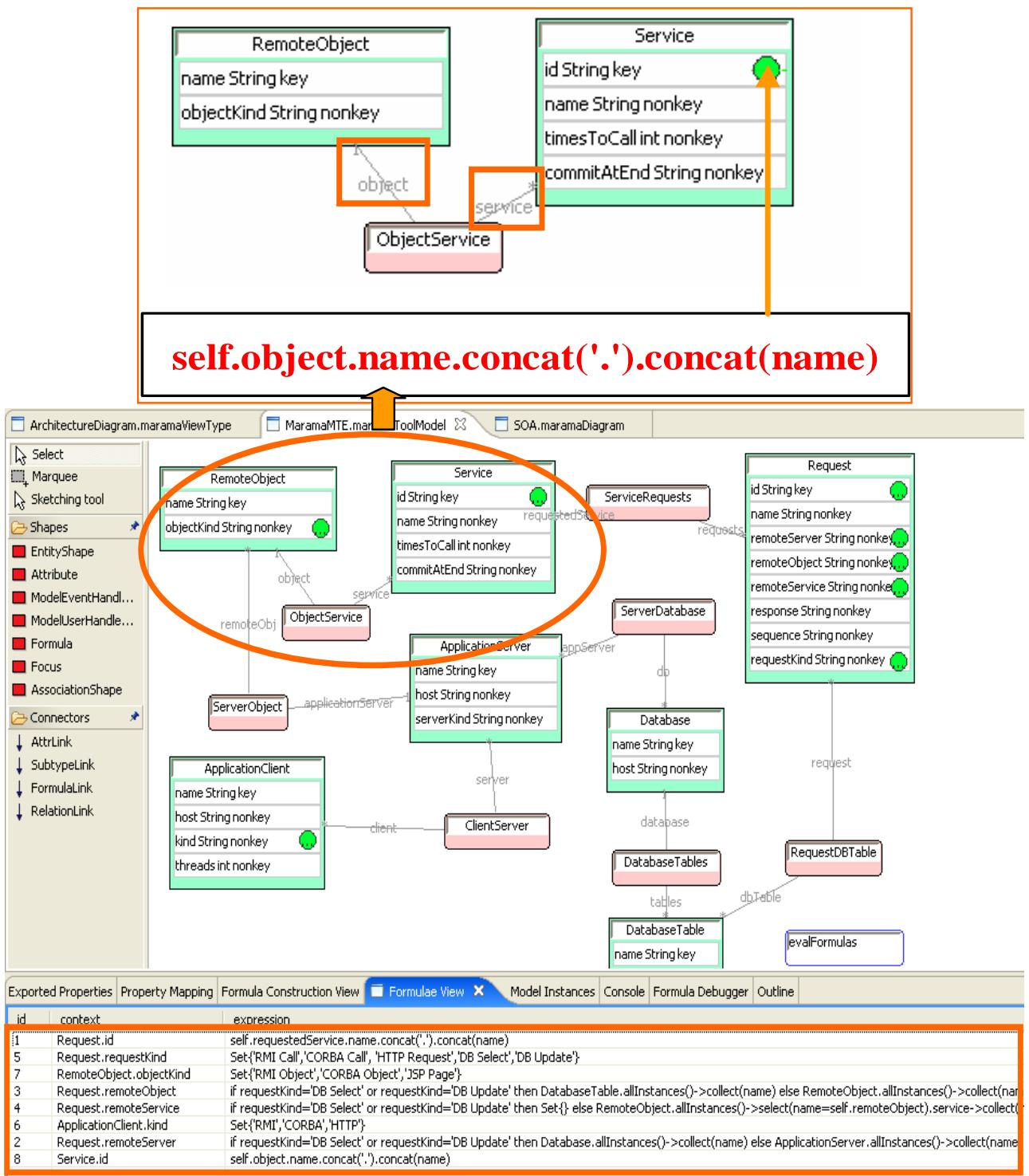


Figure 7.9. MaramaMTE model behaviour specification.

As mentioned in the previous section, formulae can also be placed on entities to specify entity invariant constraints. In Figure 7.10 (a) we have extended the MaramaMTE metamodel with a constraint specifying that every service instance must serve at least one service request. This is expressed as a constraint on the Service entity, with OCL expression “`self.requests->size () <> 0`”, shown in the overlay.

When this formula evaluates false for a service, e.g. the “cancelBooking” service of the “bookingService” remote object in Figure 7.10 (b), a constraint violation error is generated. In this case a problem marker is generated in the Eclipse Problems view (shown below) to provide the user details of the constraint violation. In this case, to solve the identified error, the user needs to add a Request entity for the identified service. When this is done, the constraint evaluates to true and the constraint error is removed from the Problems view.

The developers of the original MaramaMTE applications were provided with a demonstration of the reengineered approach and experimented with using the tool on larger scale modelling examples. Subsequent discussions were conducted with them and their feedback was very positive. Combined together the attribute calculation and invariant constraint formulae were more than adequate to eliminate all event handlers implementing model level constraints in MaramaMTE. The developers felt that the compactness and accessibility of the constraint notation and its environmental support had made the application as a whole much more easily understood and maintained. The notational mechanism also proved to be highly scalable, being unobtrusive when the tool designer’s focus was on understanding metamodel structure, but providing ready ability to focus in and obtain more detailed information about particular constraints without losing the metamodel context they are situated in. The runtime support has proven more than adequate to allow tool users to comprehend the calculations being undertaken and for the tool designer to quickly debug constraints defined.

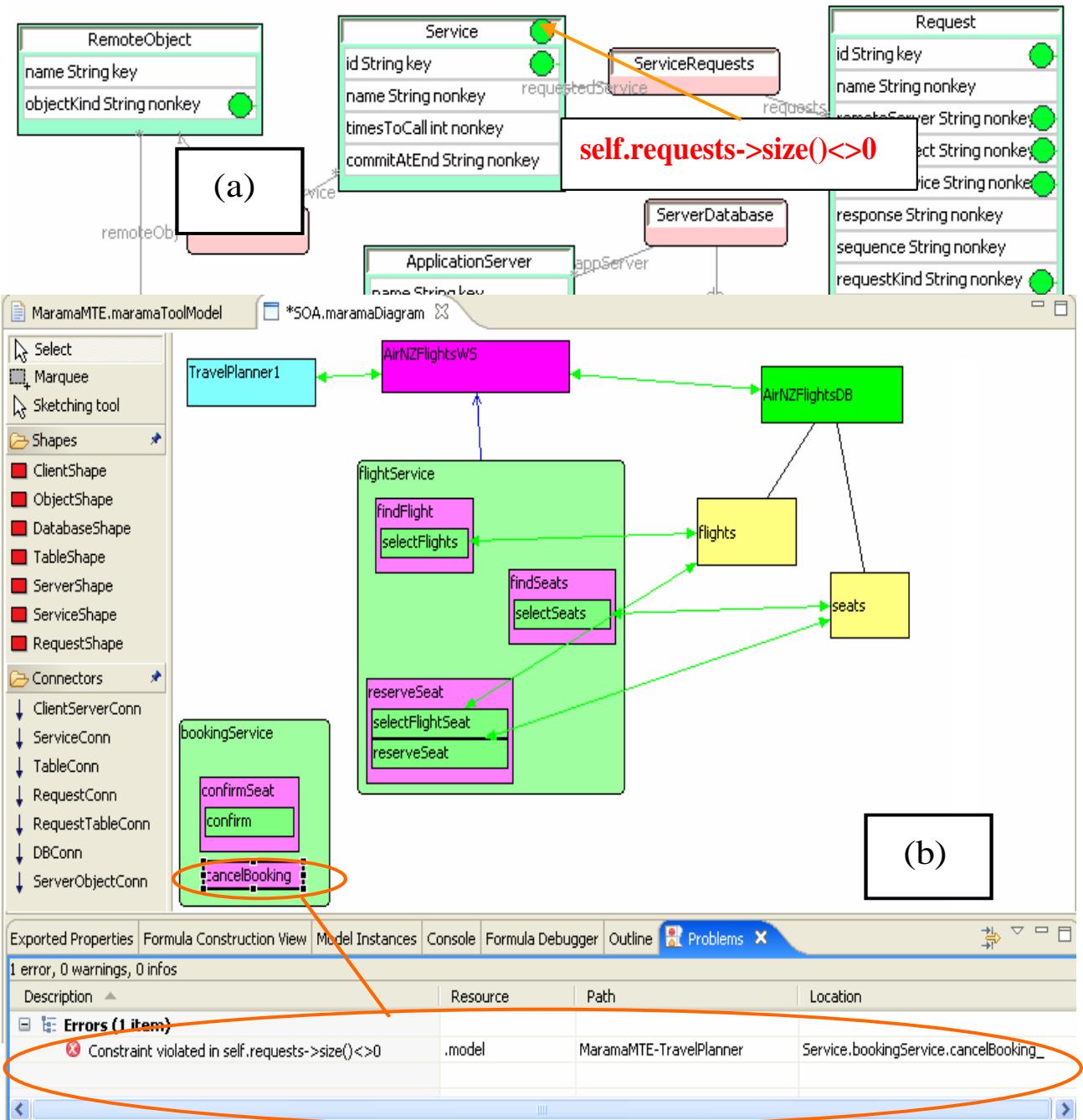


Figure 7.10. Using formulae to constrain entities.

7.6 Implementation

Adapted from the original Marama architecture described in Section 7.2, Figure 7.11 shows a high-level architecture view of the Marama meta-tools and Marama Eclipse plug-ins. The MaramaTatau extension is implemented to the metamodel designer.

MaramaTatau formulae are stored as XML tags together with other metamodel elements. Formulae on the user model are transformed to OCL representations on the Marama EMF model instance. This

process is hidden from the user. To realise MaramaTatau we integrated the EMF OCL (Eclipse, 2006) framework to implement a dynamic compiler and interpreter for MaramaTatau OCL specifications. As Marama view or model data is updated, events are sent and interpreted into EMF object requests and updates, including triggering and executing relevant compiled OCL expressions.

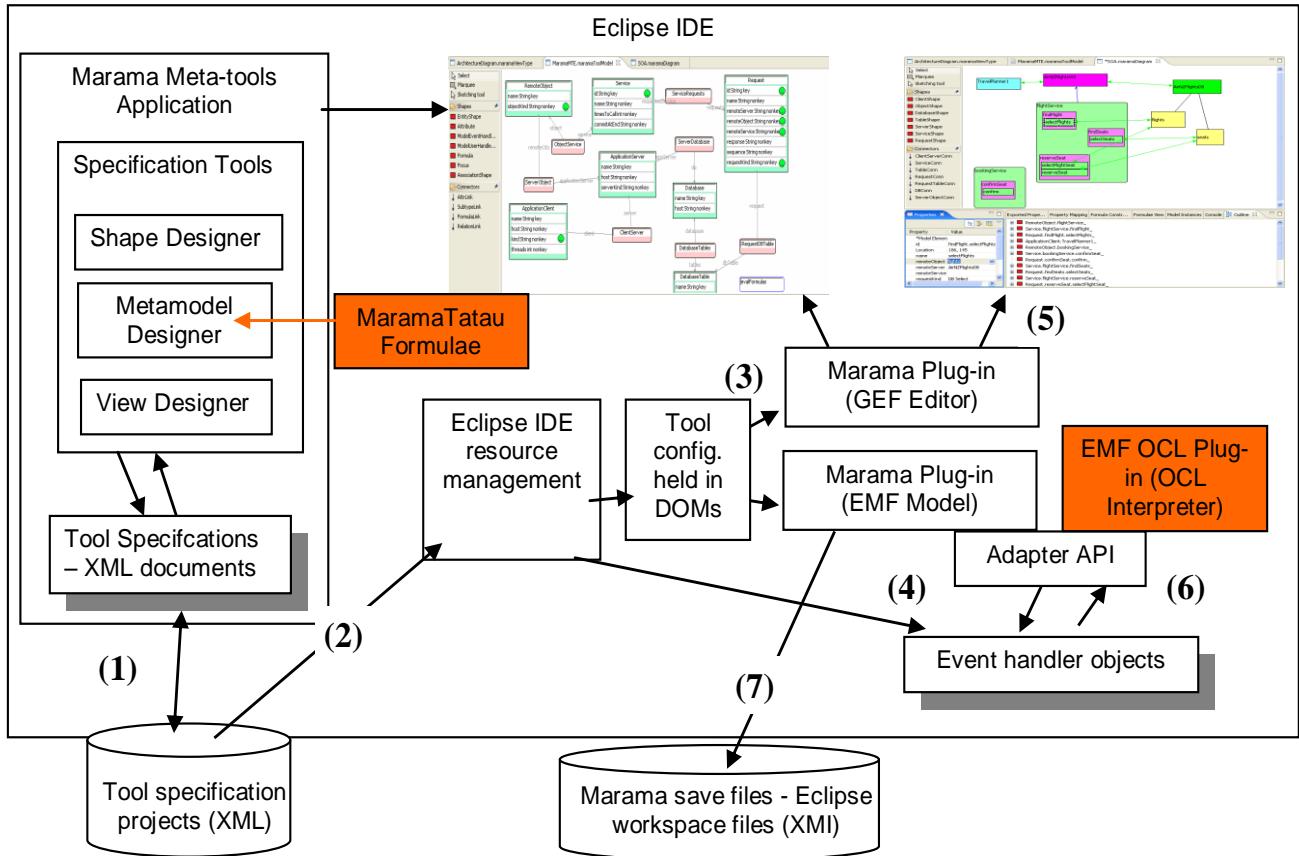


Figure 7.11. The initial architecture of Marama meta-tools. (adapted from Figure 7.2)

7.7 Evaluation

The case study has demonstrated that the approach we have developed is both effective and scalable, and amply meets the requirements we established for it. Informal feedback from the case study developers has been positive. For additional feedback, we have used a focus group approach, presenting and demonstrating case studies to a small group (less than 10 participants) of experienced modellers, to gather qualitative feedback on the MaramaTatau visual notation and environment. Participants found MaramaTatau to be easy to understand and efficient to use to manage constraints and dependencies. We have performed a much more substantial evaluation (122 participants), similar to the one undertaken for the Pounamu tool (Zhu et al, 2007), of the complete Marama environment, including MaramaTatau. We have been sufficiently encouraged by our informal evaluations to

include MaramaTatau in the publicly released version of our Marama tool (Nikau, 2007). Results of this will be presented in Chapter 10.

In developing MaramaTatau, our focus has been on providing a compact and accessible constraint representation for Marama, while minimising *hidden dependency*, *juxtaposability* and *visibility* issues. To understand other tradeoffs that we have made to achieve our primary aims, it is useful to also evaluate MaramaTatau against other cognitive dimensions.

The visual abstractions introduced are visual iconic constructs and data dependency links between them. This is quite a terse (low *diffuseness*) extension to the existing metamodel notation and the abstractions are quite low level, providing a simple overview of constraints and dependencies, and hence have low *abstraction gradient*.

Error proneness has been reduced significantly. The existing Marama Java-based Marama event handler designer is very error-prone for both novice and experienced users due to its reliance on API knowledge and Java coding together with the numerous hidden dependencies with the visual metamodel. MaramaTatau reduces error proneness by avoiding API details and directly using concepts visible in the metamodel.

The verbosity (high *diffuseness*) of the textual OCL, due to its many built in functions, does, however, present similar opportunities for error as does API mastery. The verbosity also introduces some degree of *hard mental operations* as users must remember what function is appropriate for a given purpose. However, the relative familiarity (knowledge of the OCL syntax and experience of using it) of OCL with the target end user group, experienced modellers, mitigates this and also means good *closeness of mapping* for them. The compact nature of the representation, point and click construction, and automatic construction of the visual model annotations, means *viscosity* is low.

MaramaTatau allows *progressive evaluation* of a constraint specification via Marama's live update mechanism. Modifications to formulae take effect immediately after re-registration in an end user tool. A visual debugger allows users to step through a formula's interpretation using the same abstraction level as they were developed in. By contrast, java event handlers require conventional java debuggers and a good knowledge of Marama's internal structure.

The MaramaTatau entity invariant formulae mechanism provides a rudimentary form of design critic (Robbins and Redmiles, 1998). In current work one of our group members is extending this approach to provide a more general critic authoring mechanism integrated with the Marama toolset.

7.8 Summary

We have described an approach for constraint/dependency specification in a domain-specific visual language meta-tool. This borrows much from techniques used to support the spreadsheet metaphor, but in a situation with less concreteness. The innovation lies in combining well known technologies in the form of OCL and spreadsheet interfaces in a simple novel way drawing strength from both while mitigating their weaknesses. MaramaTatau augments the Marama meta-tools' metamodel designers, allowing tool developers to specify formulae over metamodels, combined with a one-way constraint system to compute values during tool usage. This allows for much simpler specification of dependency and constraint handling within Marama meta-tools, compared to both the textual event handlers and Kaitiaki visual event handlers. The approach has some similarity to ClassSheets (Engels and Erwig, 2005), but avoids the grid structure of that approach, and provides more mitigation of hidden dependencies. It considerably extends the visual metamodel annotation mechanism plus OCL expression of GME, providing many additional hidden dependency mitigations. Early developer feedback is very positive.

MaramaTatau adopts a thoroughly different metaphor from the dataflow like icon and connector approach used in ViTABaL-WS and Kaitiaki to event handling specification. We have generalised MaramaTatau together with ViTABaL-WS and Kaitiaki, into a generic event handling framework. ViTABaL-WS provides a visual language for the design and construction of tool abstraction action-event-based architecture. Kaitiaki provides an extensible event-query-filter-action language for responding to propagated events. MaramaTatau provides a static Spreadsheet-like dependency and constraint mechanism to support specification of state-change event propagation and response. The generalisation of these three approaches within the Marama metatool framework provides wider-ranging support for event-based system design and construction. Chapter 8 – 10 elaborates our generalisation approach, the generated event handling framework and its evaluations.

Chapter 8 – Design of the Generalised Event Handling Framework

This chapter discusses the design of a general purpose event handling framework by generalising from the three exemplar approaches described in early chapters. We aimed to develop a visual metaphor and language and to provide tool support for generic event integration specification. Our generalisation approach employs the Three Examples pattern of the Evolving Frameworks Pattern Language (Roberts and Johnson, 1996). By abstracting from the three earlier, limited-domain exemplars, a general metamodel representation that combines atomic primitives (either shared or non-shared) extended by the three visual languages is defined. This common model supports multiple metaphoric views in the style of the three exemplars and will support generation to a range of underlying implementation technologies for execution or interpretation.

8.1 Motivation

Frameworks provide a set of abstract classes and their collaboration relationships for reusable design and implementation of all or part of a software system. Developing a general purpose framework usually requires a considerable amount of effort and time investment, but rewardingly the developed framework can support fast and easy construction of applications in the problem domain.

ViTABaL-WS was developed for the event-based web services composition domain to provide a visual language for the design and construction of tool abstraction action-event-based architecture. Kaitiaki was developed for diagramming-based design tools event handling domain to provide an extensible event-query-filter-action language for responding to propagated events. MaramaTatau was developed to look at general metamodel constraint specifications using OCL with a simple spreadsheet-like interface. There are some similarities identified in our three examples, including a set of event handling modules and the representations of data flows and event dependencies among their visual building blocks. The similarities can be generalised to a common model representation to allow better reuse and easier extension. Based on the in-depth exploration of the three visual event-based metaphors in their different application domains, we aimed to generalise to a metaphor and language for generic event integration specification.

Some event handlers can be specified in multiple ways using ViTABaL-WS, Kaitiaki or MaramaTatau. Users may choose to use their favoured metaphor and may also combine their specifications in multiple ways. We wanted a generalised reusable framework to have the ability to model event handling in several ways for the same system, as well as the ability to generate solutions from a canonical event model.

Multiple tools are useful for developers in that they provide abstractions for separately and progressively modelling a software system using different views and representations. The developers can specify the structure and behaviour of a model in parts then integrate them to generate dynamic environments with various constraints enforced.

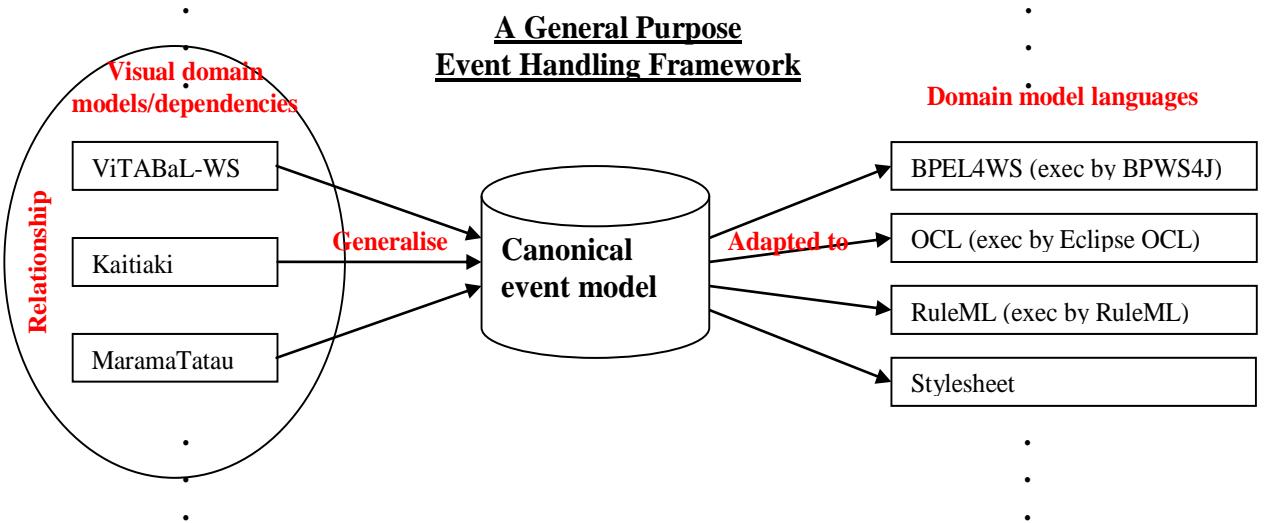


Figure 8.1. A general purpose event handling framework.

The generalisation method that we exploit can be illustrated in Figure 8.1. The three visual domain models ViTABaL-WS, Kaitiaki and MaramaTatau on the left of Figure 8.1 can be integrated based on their metaphorical supplement and their common abstraction and dependency relationship. ViTABaL-WS's tool abstraction metaphor is used to define high-level abstract data and functions and their coordination, where abstract data is further constrained using MaramaTatau's spreadsheet metaphor, and abstract functions are further refined via Kaitiaki's Event-Query-Filter-Action metaphor. The integration of the three metaphors is analogous to the desktop Windowing metaphor that is associated to the both Folder and Tree metaphors. Similar data could be represented in different ways (e.g. Folder and Tree), but maintained in the same highly abstract umbrella representation (e.g. Windows). As a consequence, they can generalise to a common event model representation (as seen in the middle of Figure 8.1). The canonical event model can then be mapped

or adapted to a range of domain model implementation languages to be executed (e.g. BPEL4WS (IBM, 2003), OCL (OMG, 2003), RuleML (Paschke et al, 2006) and stylesheet as seen on the right of Figure 8.1) using appropriate domain engines. The object-oriented framework is readily extensible so more event-based domain models and their dependencies can be added in the future. Our immediate next example being planned is the OMG’s Business Process Modelling Notation (BPMN) (OMG, 2006) in the enterprise modelling domain. The new models to integrate can reuse the canonical model’s components through inheritance or composition, and can add more features and support to evolve the framework.

The generalisation of the integrated event handling framework requires a variety of specialised modules to contribute to the framework capability and complexity. We discuss background and related work in the following section and list the set of gathered requirements in Section 8.3. We then briefly review the three examples described in the previous three chapters and their main approaches and features. The thesis is that the generalisation of these three approaches could provide wider-ranging support for event-based system design and construction. One element of this is to extend the OCL expression language with user-defined function capabilities to provide enhanced expressability.

8.2 Background

Roberts and Johnson proposed a set of patterns that are used together as a pattern language for developing and evolving object-oriented frameworks (Roberts and Johnson, 1996). The patterns are: Three Examples, White Box Framework, Black Box Framework, Component Library, Hot Spots, Pluggable Objects, Fine-grained Objects, Visual Builder and Language Tools. As illustrated in Figure 8.2, these patterns are related to each other with some overlapping usage along the process of generalising a framework.

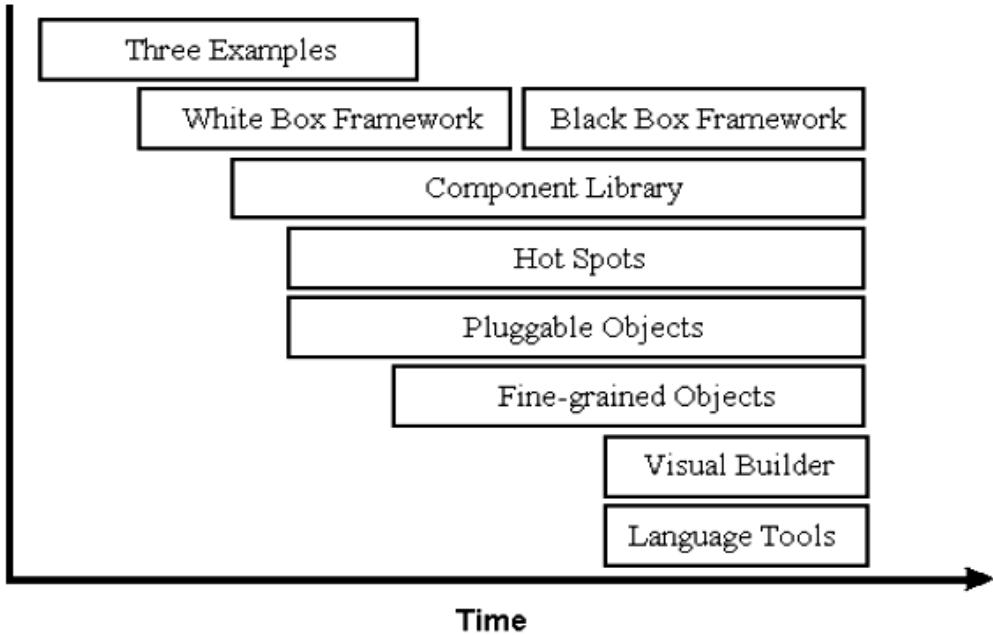


Figure 8.2. Relationship between patterns in the pattern language. (Roberts and Johnson, 1996)

According to Roberts and Johnson (Roberts and Johnson, 1996), abstractions can be well developed by generalising from concrete examples. The *Three Examples* pattern suggests that three examples should be initially used to establish a framework, and more examples are to be explored to make the framework more general. A framework is a reusable design, so we should develop it by looking at examples in either succession or parallel and identifying common, reusable abstractions. Then we can build a *White Box Framework* by generalising from the classes in the individual applications. Common portions are put in abstract classes, and subclasses can be created by inheritance. A collection of concrete classes of exemplar applications accumulate a *Component Library* for the framework. The Component Library begins with all of the concrete classes and in the long run, contains only the classes that are reused by several applications. *Hot Spots* code that changes are separated from those that never change and encapsulated within objects whenever possible. In order to avoid creating trivial subclasses to be added to the Component Library, we can design adaptable subclasses that can be parameterised to create *Pluggable Objects*. We can continue breaking objects into finer granularities (*Fine-grained Objects*) to make them more reusable. Inheritance can be used to organise the Component Library and composition can be used to combine the components into applications. A *Black Box Framework* can then be generated so we can reuse components by plugging them together and avoid programming. A *Visual Builder* can be created to support specifying components and their inter-relationship in a graphical way and generating code from it. Specialised visualisation tools (*Language Tools*) can be built to facilitate navigating and inspecting

the compositions. This evolving framework pattern language is used as our basis for generalising the event handling framework.

The EASY (Event Abstraction SYstem) framework (Grundy et al, 1996) was a prior attempt to generalise a unified event-based software architecture from the synthesis of a set of event handling elements defined in CPRGs (Grundy et al, 1996), ViTABaL (Grundy and Hosking, 1995) and Serendipity (Grundy and Hosking, 1998). CPRGs can effectively describe state-change events and the structural aspects of event-based software architectures. ViTABaL supports visual representation of propagations of action events between software components. Serendipity allows event filtering and response mechanisms to be specified in a graphical way. EASY unifies the handling of CPRGs' state-change events and ViTABaL's action events by incorporating Serendipity's event response abilities. The advantages of the three visual languages, including their visual description capabilities for both structural aspects and dynamic behaviours of event-based architectures, are combined to provide a more general architecture description language that supports wider-ranging event-based architecture design and implementation. Figure 8.3 shows an EASY example which has CPRGs' components and relationships as the backbone, ViTABaL's specification of data and toolie interconnectivity, and Serendipity's specification of event handling using filters and actions.

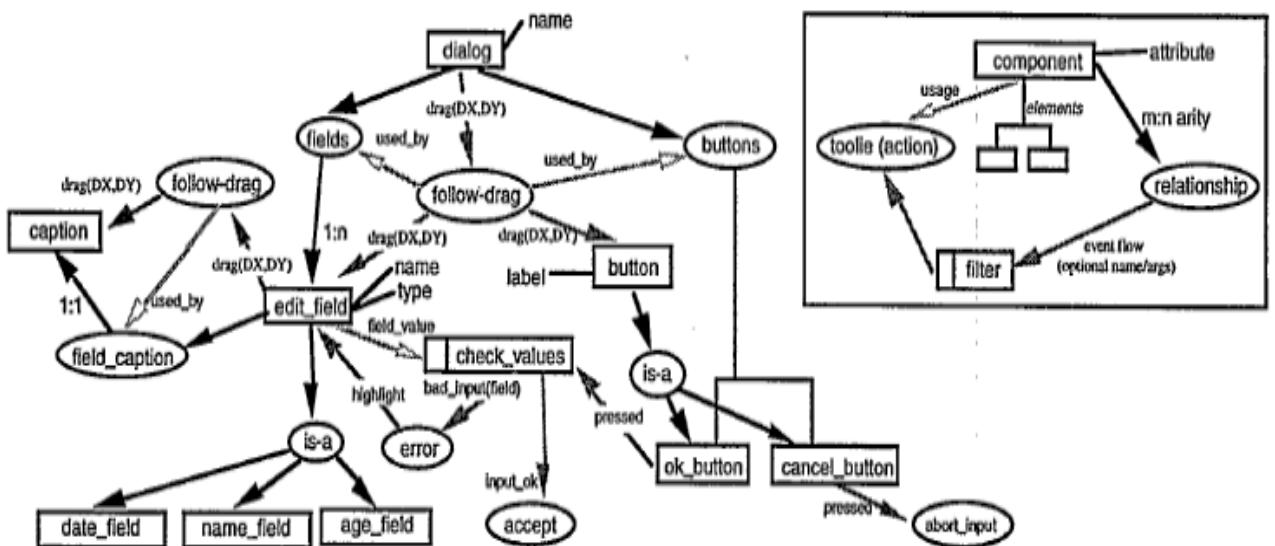


Figure 8.3. Merging CPRGs organisation, ViTABaL event propagation and Serendipity event filtering/action. (Grundy et al, 1996)

We aimed to generalise the event integration framework using the evolving framework pattern language (Roberts and Johnson, 1996) and following the EASY (Grundy et al, 1996) framework as an example.

The Pounamu metamodelling environment that was used to develop ViTABaL-WS and Kaitiaki has many deficiencies as we have seen from Chapter 3. These include:

- stability issues and weak error handling
- inconsistency of user interfaces
- difficulties in expressing model-level constraints
- difficulties in specifying event handlers
- usability issues such as some clumsiness of user interface elements around specification of icons, metamodel and views, and also the response time for some elements of functionality

We wanted a better platform to undertake the integration of event handling specifications. With the requirements for improving Pounamu, we saw an opportunity with the development of the Marama modeller tools to bootstrap development of Marama-based meta-tools. These Marama meta-tools are then a vehicle for allowing us to explore event-handling integration.

8.3 Requirements for Generalisation

A general purpose event handling framework should provide reusable design and implementation for a wide-range of event-based applications. In Chapter 2, we have identified some common issues in the current event handling specification and visualisation techniques and concluded a set of general requirements for our event handling integration framework. Based on the in-depth experiments of our three limited-domain exemplars in the previous chapters, we can now elaborate the requirements for the framework generalisation:

- The generalised framework should incorporate compositional primitives as building blocks and different communication relationships between them. It also should contain mapping/integration schemes as a crossover between ViTABaL-WS, Kaitiaki and MaramaTatau, and possibly other limited-domain event handling models in the future e.g. BPMN (OMG, 2006).
- The common model representation needs to be identified from the specialised modules from ViTABaL-WS, Kaitiaki and MaramaTatau. The relationships among the modules need to be established so that the modules can collaborate with one another. Duplications need to be removed so that the common model is redundancy free.

- The generalised framework needs to offer graphical notations in the style of the three metaphoric exemplars, together with additional textual notations to allow users to escape to code when specifying complex custom behaviours such as code generation.
- The generalised integrated framework must contain reusable designs to allow users to initialise their system and should allow users to specify customised event types, event generators, event receivers and event handling building blocks to enhance the extensibility and flexibility of the framework.
- Multiple views of data, event and behaviour representations must be kept consistent at both the model and user interface levels to ensure the correctness of generated environments.
- The generalised framework should support further tool integration via a canonical data/event model extension and consistent user interfaces.
- The generalised framework should provide mechanisms to allow easy navigation from one view of the specification to another.
- Though visual languages are more self descriptive than textual languages, the framework should still provide support for detailed documentation of modelling elements.
- The generalised framework should allow event propagations to be traced and event handling results to be visualised in running systems based on a user interactive visual debugging model.

8.4 Generalisation

In order to derive a suitable common model we need to be able to represent all of the concepts from the three examples. We also need a way to map between related concepts in each metaphor. This common model supports multiple metaphoric views in the style of the three exemplars and thus is in multiple paradigms. In this section, we briefly review (through Sections 8.4.1 to 8.4.3) the event propagation model features and the building blocks defined for each of ViTABaL-WS, Kaitiaki and MaramaTatau, and then generalise them to a common set of primitives in Section 8.4.4.

8.4.1 ViTABaL-WS Building Blocks

ViTABaL-WS uses a Tool Abstraction metaphor for describing relationships between service definitions. ViTABaL-WS supports modelling of complex interactions between web service components, plus code generation and visualization of running systems. Multiple-views of data flow, control flow and event propagation are specified for a ViTABaL-WS model. Table 8.1 summarises the building blocks defined for ViTABaL-WS.

Metamodel Primitives	Attributes	Semantics
Toolie	Name	Encapsulate data processing and interacts with each other through both direct and indirect operational invocations using shared data structures (message ADT instances) and event dependencies indicating state changes to a data store service ADS
ADS	ADT: name ADS: message name, type	Instance of typed operation/messages/events
Abstract operation port	Name Sequence number	Typed input and output ports on toolie and ADS services that connect toolies to other toolies and ADSs and provide message sources and sinks. Services are wired together using these ports with ports supporting only certain kinds of connection and message ADTs
Data store service ADT	Name	Active data store service
Role	Name	A partner that provides or requests a service
Fault handler	Name	Composed error/exception handler
Decision	N/A	A control flow starting point, following with conditional transitions
Local activity - Type checking (TC) - Type transformation (TT) - Data manipulation (DM)	Activity name TC: isTypeOf TT: fromType, toType DM: variable, value	Atomic or composed activity
Data flow connection	Labels: <<synch>> or <<asynch>> indicator, and <<transaction>> indicator	Flows of data to toolie ports and ADSs - Input/Output flow links - Parameter decomposition links
Control flow connection	Labels: <<synch>> or <<asynch>> indicator, and <<transaction>> indicator	Invocations to toolies - Partner link - Synchronization/Asynchronous flow - Conditional flow - Iterative flow - Transaction flow
Event flow connection	Labels: <<synch>> or <<asynch>> indicator	Event dependencies – indicate event subscribe-notify between toolies and ADSs - One way broadcast - Request-response - Listen-before - Listen-after

Table 8.1. Building blocks defined for ViTABaL-WS.

ViTABaL-WS model views describe the interconnections between toolies and ADSs. These interconnections can be annotated with different type of data flow, control flow, and event flow connections. Different kinds of subscribe-notify event propagations including one way broadcast, request-response, listen-before and listen-after can be used between the connected toolies and ADSs. Toolies encapsulate behaviour in that they respond to events to carry out some system function. ADSs encapsulate data and respond to events to store, retrieve or modify data (Grundy and Hosking, 1995) (Liu et al, 2005).

Modified toolies or ADSs broadcast to all their inter-connected components about the change. Receiving components interpret the change descriptions and modify their state or execute actions accordingly with possible further change descriptions to be generated (Grundy and Hosking, 1995). ViTABaL provides an architecture description language for the event-based tool abstraction paradigm (Grundy et al, 1996). ViTABaL-WS includes a few more building blocks to control event-based behaviour by specifying roles, sequences, decisions, type transformations, iterations, and transactions. Figure 8.4 shows an exemplar ViTABaL-WS event propagation view (generated in Marama meta-tools) that specifies a set of subscribe-notify event propagations between toolies (Marama library functions) and ADSs (Marama shared data structures).

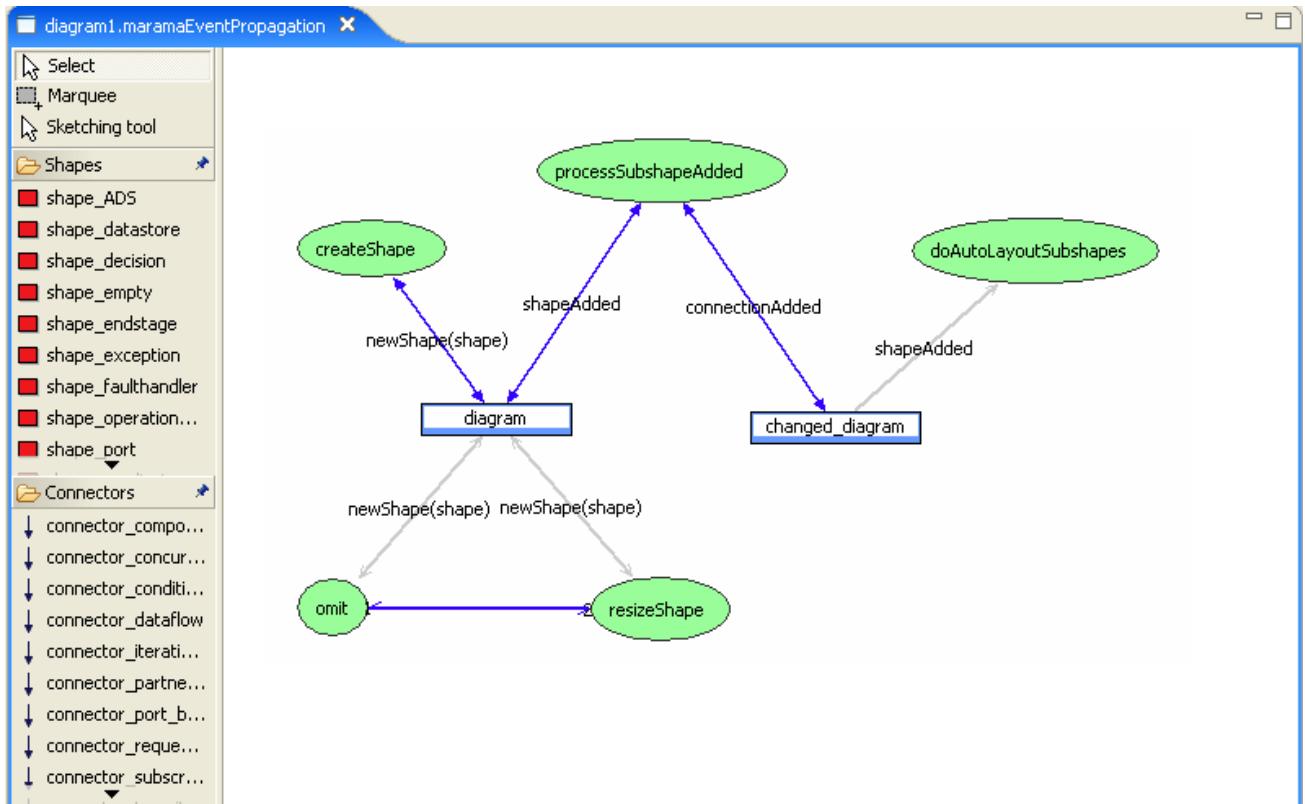


Figure 8.4. ViTABaL-WS event propagation definer in Marama meta-tools.

8.4.2 Kaitiaki Building Blocks

Kaitiaki provides end users ways to express event handling mechanisms via visual specifications. It uses an “Event-Query-Filter-Action” metaphor for describing behaviours for diagramming-based design tools and multiple-views of data flow in a modelled process. Kaitiaki supports building up complex event handlers in parts, providing the representation of:

- key “building blocks” of state query, data filtering and state modification,
- event objects and their attributes,
- data propagation between event, query, filter and action representations, and
- iteration and conditional data flow

Table 8.2 summarises the high-level building blocks defined for Kaitiaki.

Metamodel Primitives	Attributes	Semantics
Event	Event type Event generator object Property values changed	A single event or a set of events is the starting point for a Kaitiaki event handler specification.
Single/Collection object	Name Type Value	Event/tool-state objects/attributes
Query	Parameters (Input) Output	Retrieve elements and output single or collection object. Parameterised with data propagated through incoming connectors.
Data Filter	Parameters (Input) Output	Select elements from their input. Define conditional dataflow. Parameterised with data propagated through incoming connectors.
Action (State modification)	Parameters (Input) Output	Apply operations to elements passed to them. Parameterised with data propagated through incoming connectors.
Iteration	Input: Shape/Connector collection Condition: optional	Iterate through every element in the collection, or iterate while a condition is satisfied.
Data flow ports start and end	N/A	Start/end of a composed building block (event, query, filter, action)
Dataflow	Labels: <<synch>> or <<asynch>> indicator	Data propagation between event, query, filter and action representations - Data push when available - Data pull on demand

Table 8.2. Building blocks defined for Kaitiaki.

While ViTABaL-WS visually describes only the event-based inter-connections between abstract components with the lack of event responses, Kaitiaki's events, filters, queries and actions provide a visual design level notation for specifying event handling mechanisms.

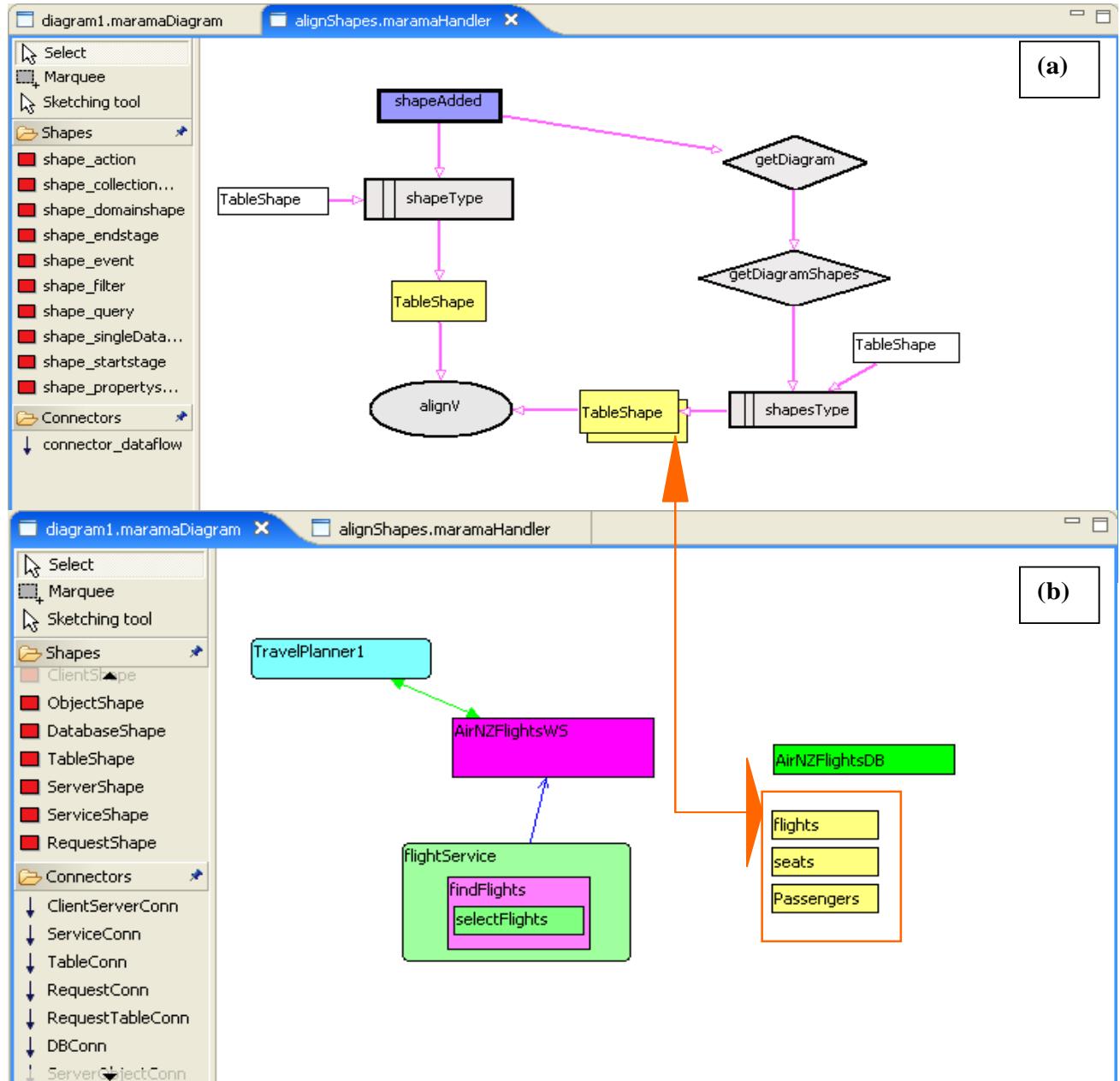


Figure 8.5. Kaitiaki event handler specification (a) and its runtime execution effect (b).

Kaitiaki provides graphical views for specifying handling of both built-in and customised state-change and action events via queries, filters and actions. Queries select data from a common model repository. Filters apply pattern-matching to incoming data, passing matching data to other queries/filters/actions. Actions execute operations which may modify incoming data, display information, or generate new events. Concrete end user domain icons can be added to mitigate the

abstraction and make the specification more readable. Figure 8.5 shows an exemplar Kaitiaki event handler specification (generated in Marama meta-tools) for aligning diagram shapes (a) and its runtime execution effect (b). The handler responds to a “shapeAdded” event, filters out the “TableShape”, and then aligns the newly added “TableShape” with the existing ones that are queried from the diagram.

8.4.3 MaramaTatau Building Blocks

Marama provides a rich structural notation for specifying tool architecture/metamodel via an entity-relationship mechanism. MaramaTatau is used to specify value dependencies and modelling constraints upon these Marama structural specifications. MaramaTatau was initially designed for constraining entity-relationship based metamodels. It has evolved to be usable with any Marama view type specifications.

MaramaTatau uses a declarative spreadsheet-like approach to construct metamodel formulae to extend behaviour specification of visual design tools including the specification of property-change event handling and constraint management. A formula is constructed visually by clicking on entity-relationship metamodel elements (i.e. entity type, association type, and attribute) and a list of library provided functions. Formula construction is similar to a spreadsheet but expressed at a type rather than an instance level. The visually specified metamodel level formulae are interpreted in selected model views. Table 8.3 summarises the building blocks used in MaramaTatau.

Metamodel Primitives		Semantics
Reference-based	Self	Reference to the current instance.
	.	Navigate to attribute, association-end and association class
	Entity type	Reference to an entity type
	Association Class	Reference to an association class
	Attribute	Reference to an attribute
	Association-end (role)	Reference to an association-end
	allInstances()	Get all instances of an entity type or association class
Collection-based	->size()	Get the number of elements
	->sum()	Calculate the sum of all elements
	->collect	Collect a number of elements
	->forAll	Evaluate a condition to true for each element in a collection
	->iterate	Iterate through each element and update an accumulator
	->select	Select elements that satisfy a condition
	->reject	Reject elements that satisfy a condition

	->exists	Evaluate a condition to true for at least one element
	->notEmpty()	A collection is not empty
	->isEmpty()	A collection is empty
	->includes	The collection includes a particular element
	->includesAll	The collection includes all elements in another collection
	->excludes	The collection excludes a particular element
	->sortedBy	Sort a collection
	->union	Union two collections
	->intersection	Intersect two collections
	->first()	Get the first element in a collection
	->last()	Get the last element in a collection
	->at	Get an element at a specified index
	->indexOf	Get the index of an element
Criteria indicator		Condition separator
Arithmetic Function	+,-,*,/,->mod,->abs(), ->floor(),->round(), ->max,->min	Simple mathematics functions
String function	.concat	Concatenate strings
	.size()	Calculate the length of a string
	.substring	Get substring
	.toUpperCase()	Convert to upper case letters
	.toLowerCase()	Convert to lower case letters
Logical Operator	=,>,<,<>	Comparisons
Boolean-based	not, and, or, xor	Boolean operators
	if, then, else, endif	Decision making
	implies	Inference
Type Function	.oclIsTypeOf	Check the type of an element
	.oclAsType	Convert to a type
Dependency links	AttrLink FormulaLink	Define context of an attribute and a formula
Extended view type functions	Contains Encloses Onborder User-defined functions	Define visual shape layout, e.g. a shape is contained/enclosed/on border of another shape. User-defined functions can be added into the library for reuse.

Table 8.3. Building blocks defined for MaramaTatau.

Value dependencies and modelling constraints are state-change events to be handled in MaramaTatau via a uni-directional change-propagation with side-effect extensions to dependent

components. Formulae are used to specify executable query/action constraints. Dependency links are added to explicitly annotate relationships of inter-dependent elements. Values are propagated from sources to dependent targets and interpreted at runtime. Some examples have been demonstrated in Chapter 7.

8.4.4 Generalised Marama Meta-tools

To generalise our work on ViTABaL-WS, Kaitiaki, and MaramaTatau, we have designed a set of Marama meta-tools to provide a better platform and a vehicle for allowing us to explore event-handling integration. Marama meta-tools provide the visual language design environment similar to Pounamu (Zhu et al, 2007), but as an open-source Eclipse plug-in with richer support on event-based visual behaviour modelling. Figure 8.6 illustrates the Marama meta-tools approach, which is an add-on to the Marama framework (Grundy et al, 2006) that includes five sub-tools: Metamodel Definer, Shape Designer, View Type Definer, Event Propagation Definer and Visual Event Handler Definer. The incorporative use of these sub-tools facilitates easy event-based behavioural modelling and integration that is unified with system structural modelling.

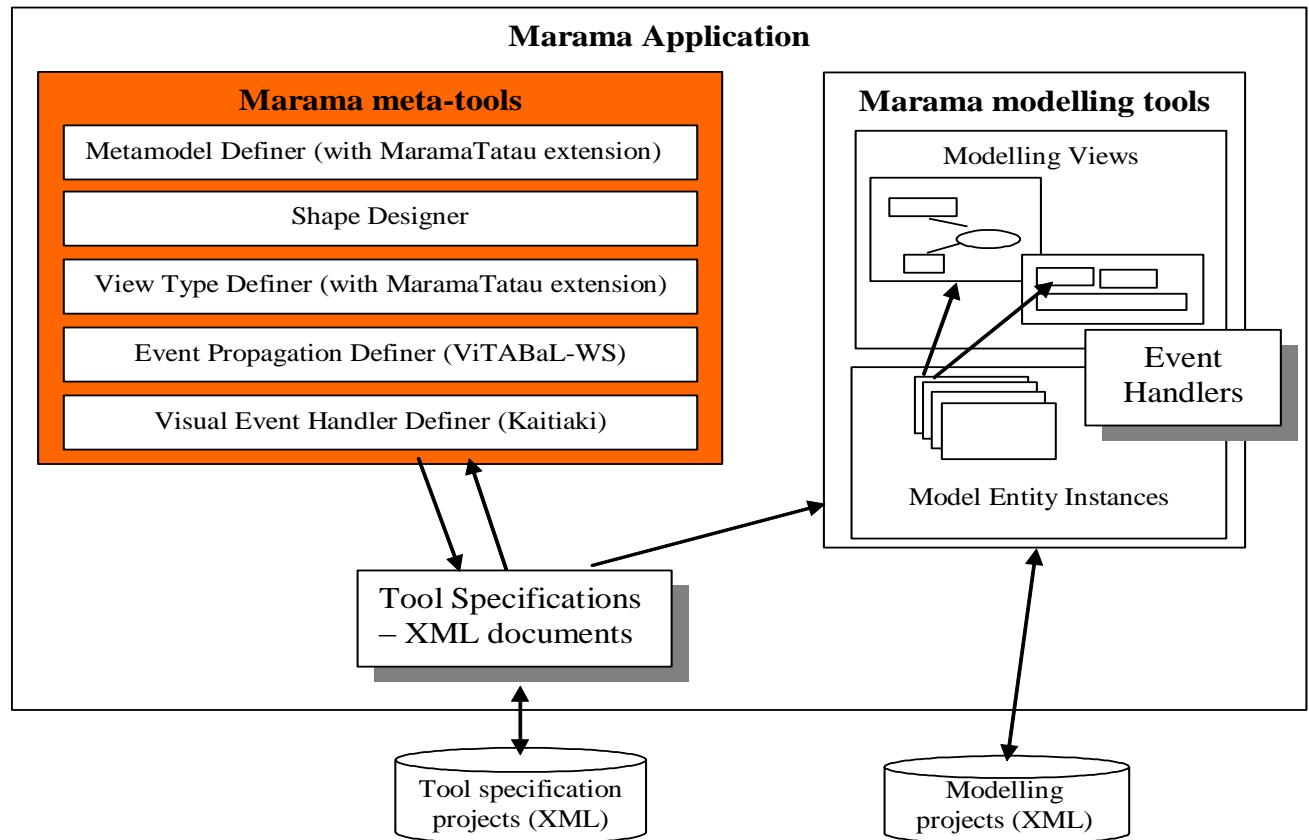


Figure 8.6. The Marama Meta-tools approach. (Adapted from (Grundy et al, 2006))

Apart from static modelling of entities, relationships, shapes, connectors and mappings, which form the backbone of the metamodeling framework in Marama meta-tools, MaramaTatau specifies inter-dependency of Marama static modelling components by adding formulae over both model and view data structures. A one-way constraint system is exploited to compute dependent values at runtime during tool usage. ViTABaL-WS specifies event propagations and other inter-connectivity of toolies (user or library functions) and their shared ADS pool (Marama structural components). The event representations are propagated to the listening toolies which match them to the event patterns they respond to, and the response is invoked (Grundy et al, 1996). Kaitiaki specifies detailed toolie responses via event propagations through a set of library-defined pattern matching queries, filters and actions.

As in the EASY framework (Grundy et al, 1996), Marama meta-tools also permit MaramaTatau state-change events and ViTABaL-WS action events to be handled in a unified manner, via event response modelling capabilities of Kaitiaki as illustrated in Figure 8.7. A detailed example is provided in Figure 8.8, where in the ViTABaL-WS diagram on the left a “shapeAdded” event propagates from the data structure “diagram” to the toolie “processSubshapeAdded” which is an event handler further defined in the Kaitiaki view on the right. MaramaTatau state-change events can also be handled in this extensible manner, via Kaitiaki specifications. This aims to maintain the advantage of MaramaTatau of effective structural dependency and constraints specification, and that of ViTABaL-WS and Kaitiaki of visual representation of event propagation and response mechanisms, while also providing user-defined behaviour extension of MaramaTatau and integrating the three languages to provide unified specifications.

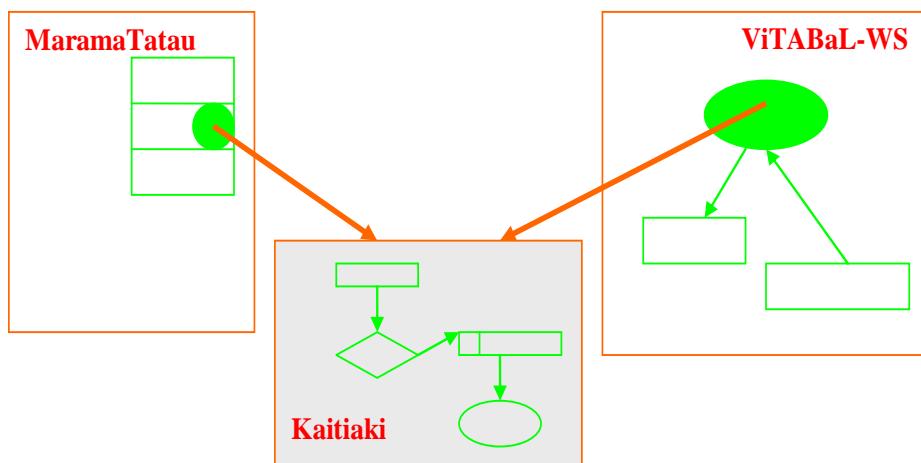


Figure 8.7. Unified event handling in MaramaTatau and ViTABaL-WS using Kaitiaki.

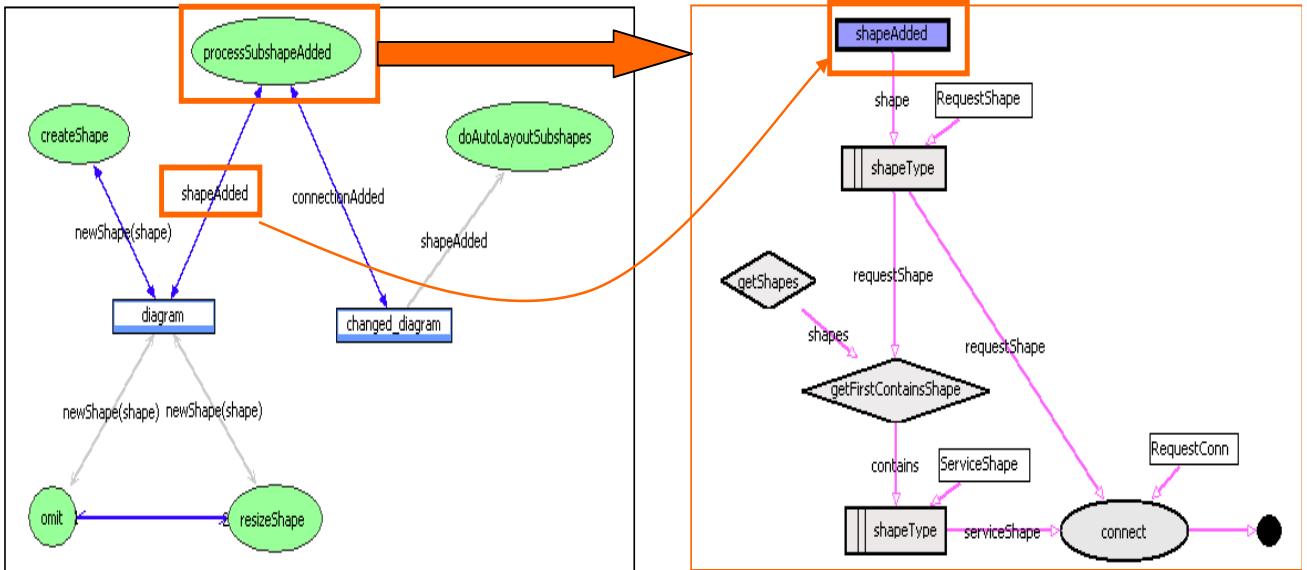


Figure 8.8. Event propagation definition in ViTABaL-WS and event handler definition in Kaitiaki.

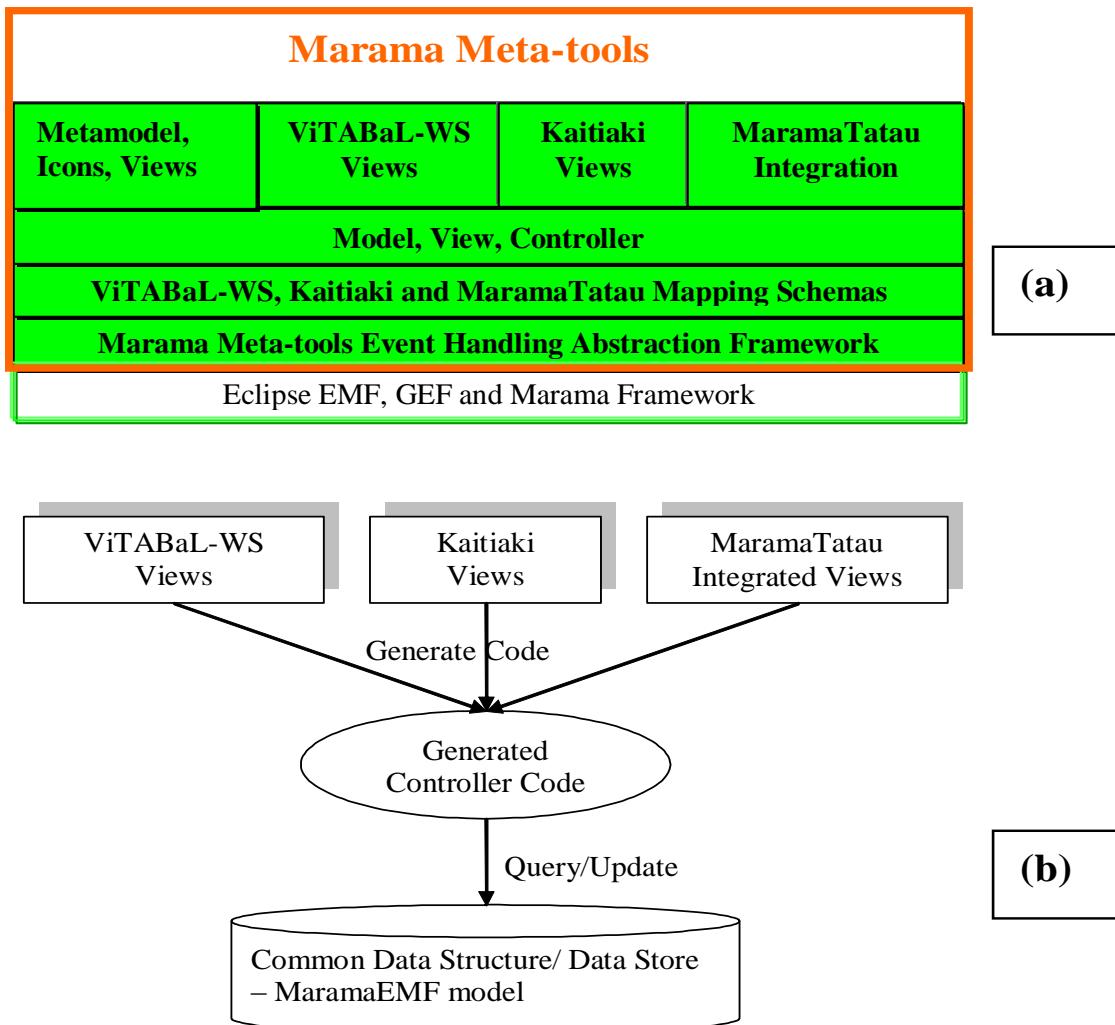


Figure 8.9. Marama Meta-tools Event Handling Abstraction Framework.

By combining the reusable views and building blocks of ViTABaL-WS, Kaitiaki and MaramaTatau, we now have an event handling abstraction framework. Figure 8.9 (a) shows how the event handling abstraction framework is integrated in Marama meta-tools. The Marama framework (Grundy et al, 2006) provides Eclipse-based editors for Pounamu (Zhu et al, 2007) generated domain-specific modelling environments. Marama meta-tools are built on top of Marama and provide visual languages and tools similar to Pounamu but with advanced event-based system integration. Marama meta-tools provide behavioural modelling using the three distinctive yet collaborative metaphoric views and generate from them to a common model implemented in the event handling abstraction framework which in turn accesses class libraries and then interprets them to query and update Marama EMF model and view representations (as seen in Figure 8.9 (b)).

8.4.4.1 Eclipse Framework, EMF, GEF and Marama Framework

The Marama framework is built as an Eclipse plug-in using EMF to represent Marama models and GEF to render Marama views. Detailed Marama architecture and approach have been introduced in Chapter 7. Figure 8.10 shows the Marama EMF representation of its project and diagram elements.

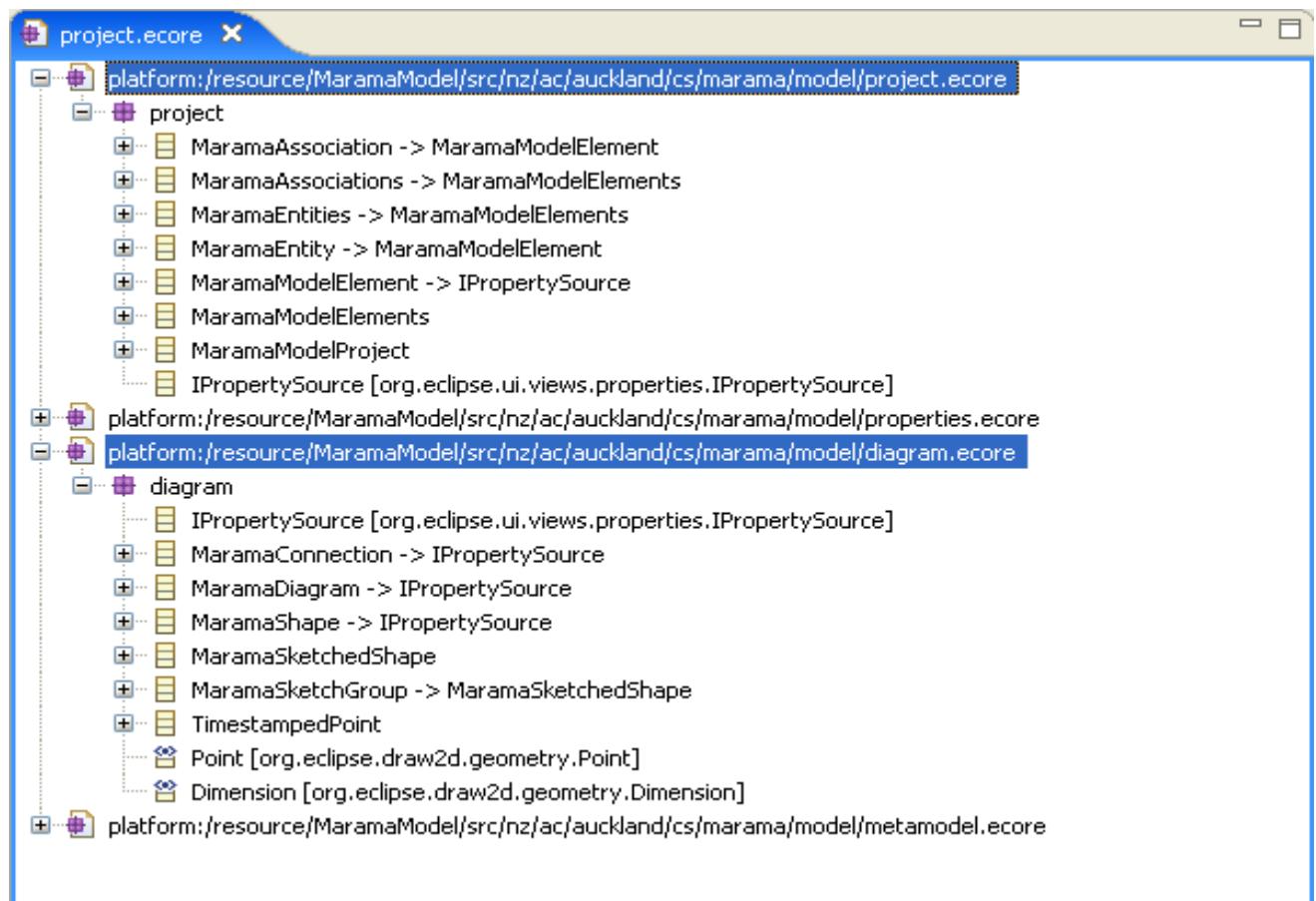


Figure 8.10. Marama EMF specification.

The Marama framework contains a set of classes, attributes, methods, relationships and events. The framework is packaged into three major parts: model, editor and reusable handlers, in the style of a model-view-controller based separation of implementation concerns. These reusable framework elements support easy creation, modification and extension for building domain-specific modelling environments.

8.4.4.2 Marama Meta-tools Event Handling Abstraction Framework

The event handling abstraction framework provided with Marama meta-tools contains a canonical metamodel representation (generic model) of event handling specifications that enables multiple behavioural paradigms to be easily integrated into the framework. ViTABaL-WS, Kaitiaki and MaramaTatau provide visual languages and tools for event handling specification. ViTABaL-WS is used for high-level conceptual modelling of event propagations among Marama components; Kaitiaki is used for intermediate level design of event propagations among a set of user or library defined queries, filters and actions; MaramaTatau is used for implementation level specification of model and view value dependencies and constraints. The three distinctive behavioural modelling views are wired together by their underlying model. The generic event handling model generates Marama XML, EMF notifications and Java event handlers to be interpreted by the Marama framework for dynamic queries and updates of models and views.

The canonical event handling model enables development of general purpose event-based system specifications. The metamodel elements from the three visual languages have been combined with redundancies removed and some bridging elements added. The component library of the event handling abstractions framework is illustrated in Figure 8.11, where mappings of the model elements to those used in the three visual languages are also indicated using coloured boxes. It mainly includes the relationships between event, event generator, event service, event listener and event handler elements. The Event handler is further sub-typed including publish, subscribe-notify, invoke activity, generate event, capture event and custom handler. The connectivity types supported in the framework include structural generalisation, association and composition, and dynamic control, data and event flows. The CompoundActivity interface may take multiple possible roles as event generator, event listener or event handler, may contain the ViTABaL-WS, Kaitiaki and MaramaTatau building blocks and may be involved in a variety of data manipulation and dynamic connectivity operations.

Almost all elements in this common model are defined as extensible. Particular hot spots, or places in the architecture where adaptations for specific functionality should be made (Roberts and Johnson, 1996), include:

- Event

The framework supports a set of system events together with user-defined custom events to be added by either specifying new event details or by sub-typing/composing existing event types.

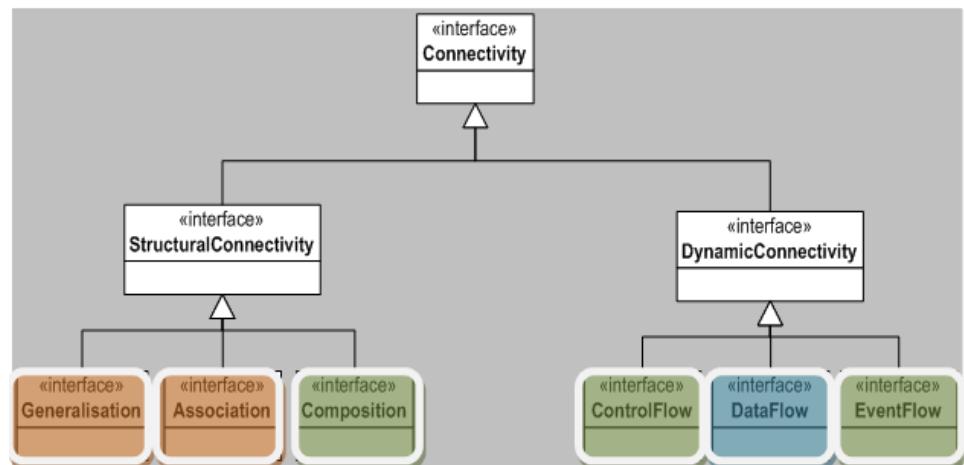
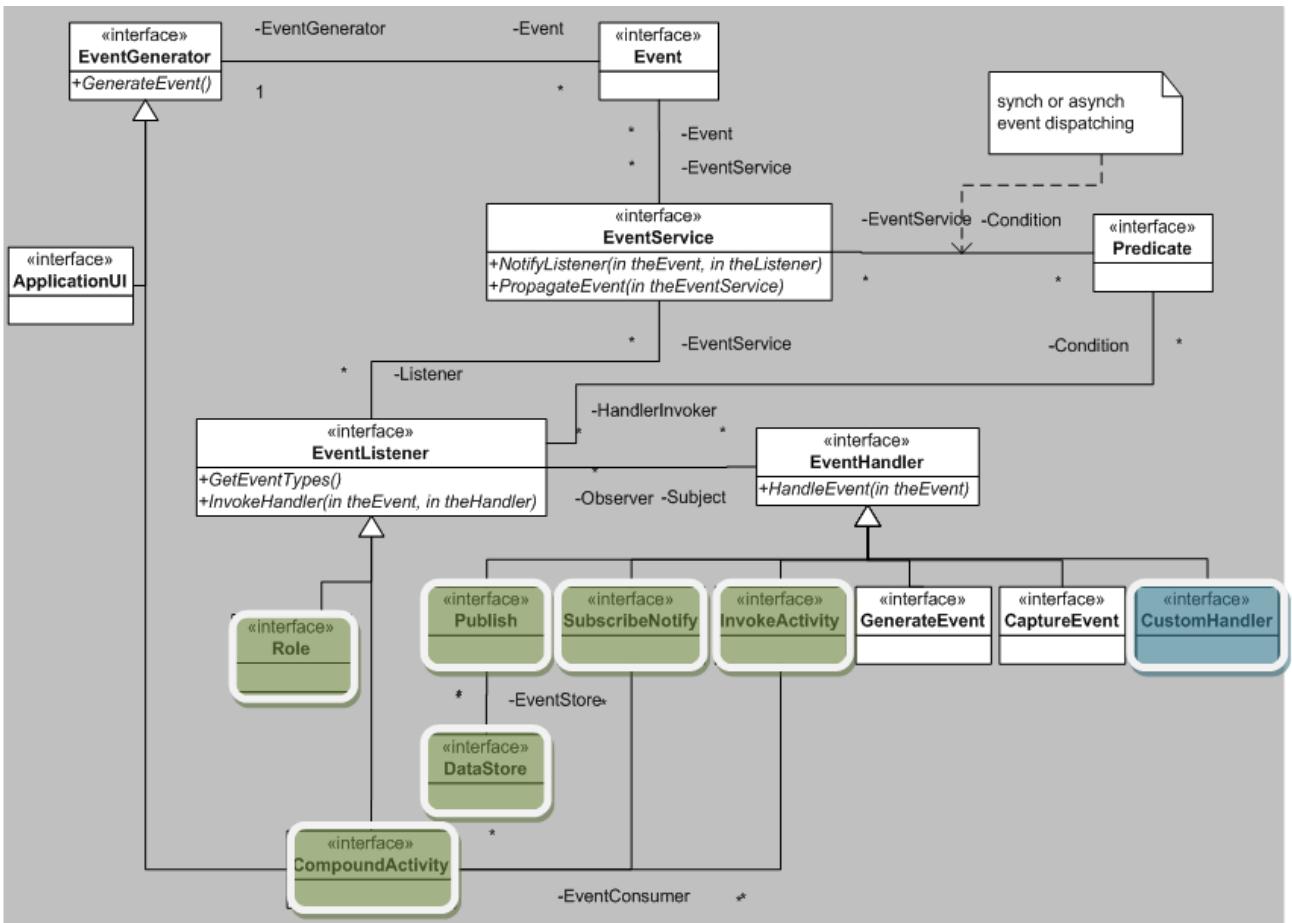
- Event handler

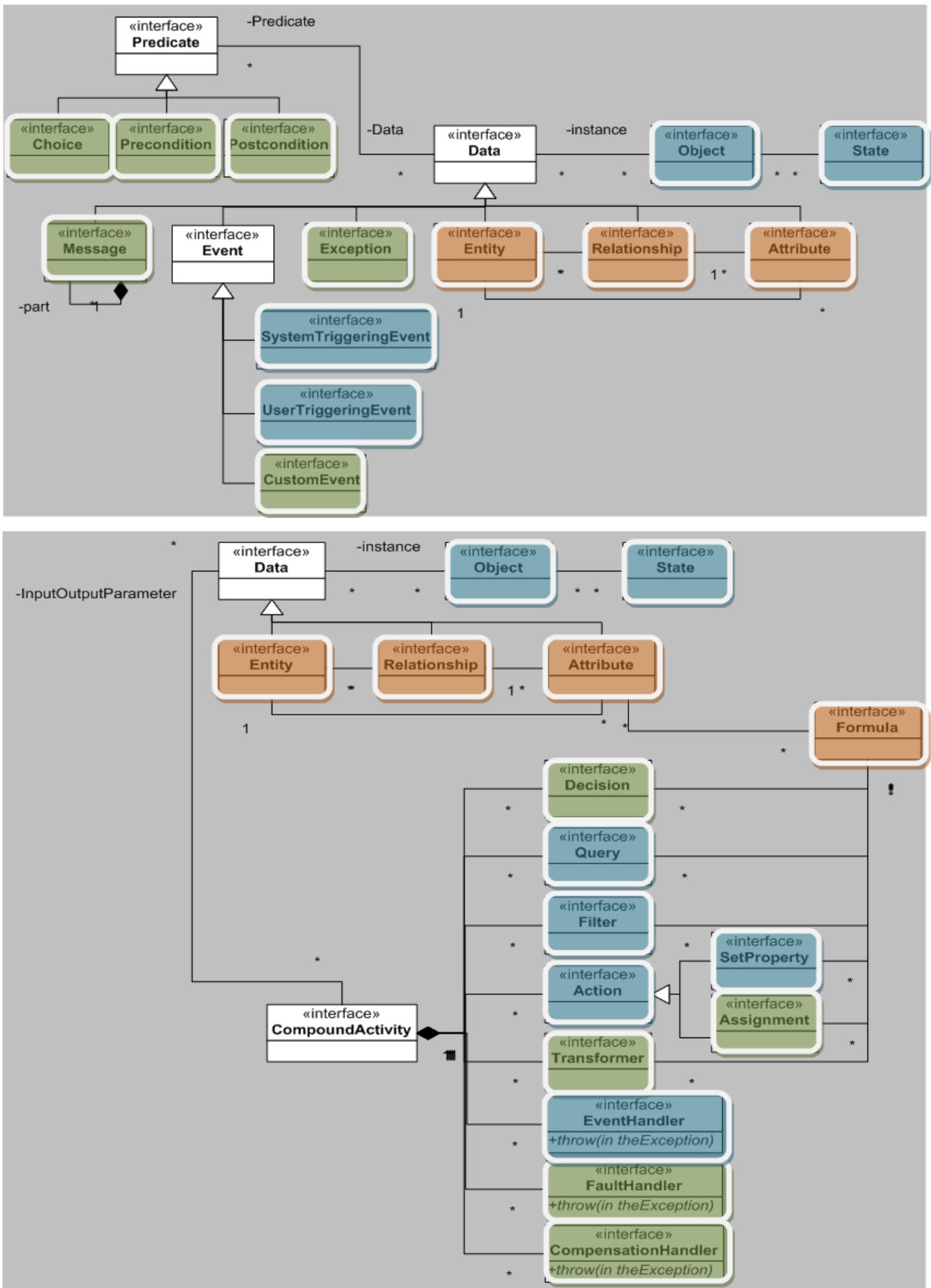
The framework supports a set of system event handler building blocks together with user-defined custom event handlers to be added by either specifying new event handler details or by sub-typing/composing existing event handler building blocks. For examples, the event handler types of a GUI system can include additional “UpdateUI” handler, “AutoLayout” handler, “PromptMessage” handler etc.

- Control flows

Control flows can be stereotypes to specify the transition time requirement (<<synch>> or <<asynch>>), the transition sequence (<<1.1.2>>, <<StartWith>>, <<EndWith>>), etc. Concurrent transitions do not need to be explicitly modelled. When the condition of a transition is met, the transition is invoked immediately. So when an element is associated with multiple transitions, the transitions are concurrent when their conditions are satisfied at the same point of time.

The event service receives all notifications (e.g. entity changed) and forward them in a multiplexed way (Sun, 2005) to any associated event behavioural views – ViTABaL-WS, Kaitiaki or MaramaTatau. Inter-communications of the three behavioural views are monitored by the event service and automatically delegated to Marama processing components.





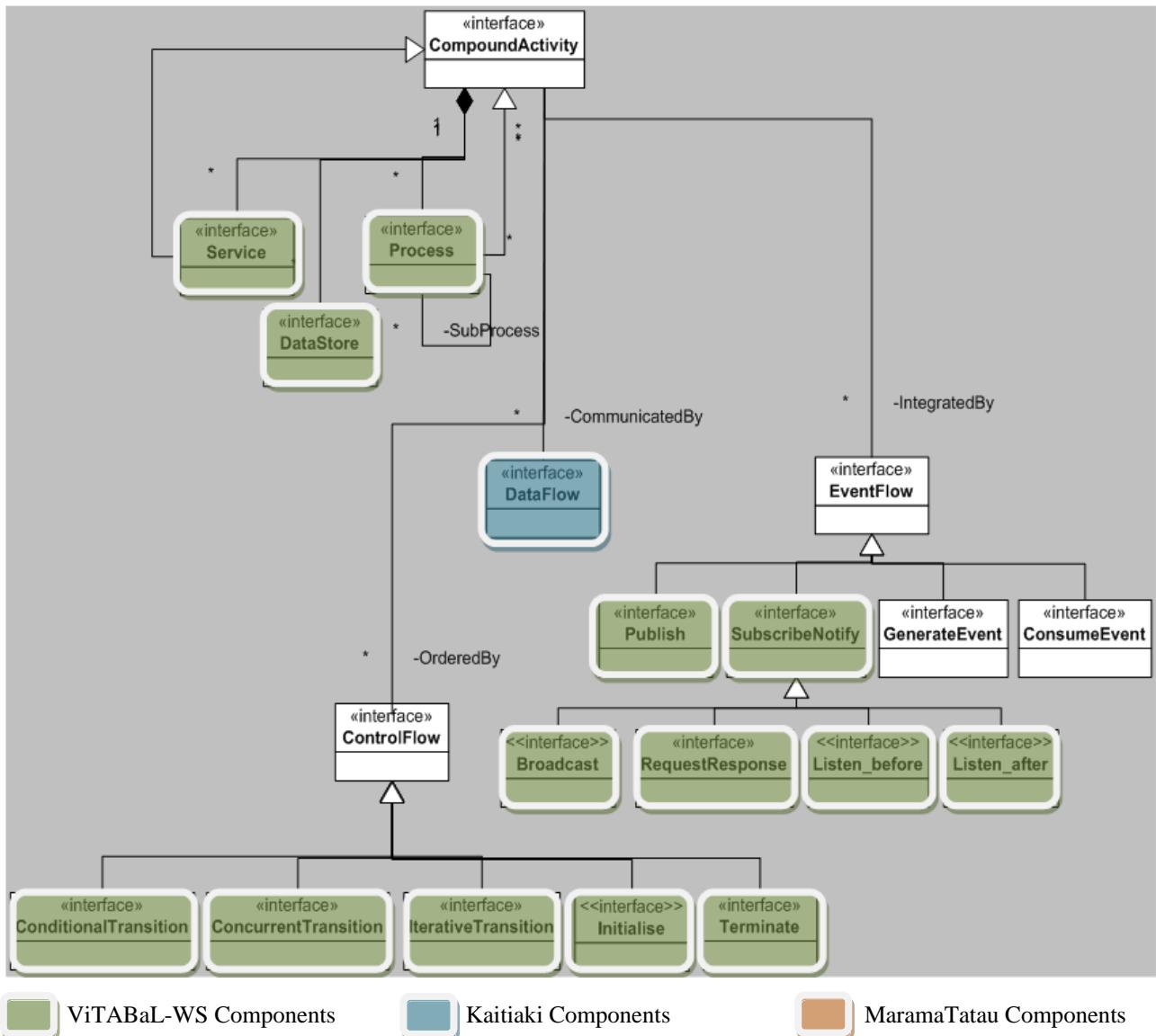


Figure 8.11. Common event handling model.

This canonical model representation is used to instantiate behaviour specifications in ViTABaL-WS, Kaitiaki and MaramaTatau views such as those shown in Figure 8.4 and Figure 8.5. The behaviour model instances are analysed at specification time and are used to generate event handler code to be executed at runtime.

8.4.4.3 Mapping Schemas

Each of ViTABaL-WS, Kaitiaki and MaramaTatau has their own strengths in handling events. They are mainly complementary to one another instead of overlapping. However, there exists the possibility to specify an event handler in multiple ways using ViTABaL-WS, Kaitiaki or MaramaTatau, though one specification may not be as efficient as the other, the required event handling effect can still be achieved.

The connectivity and inter-changeability of the different metaphoric specifications in Marama meta-tools can facilitate mapping concepts in each metaphor and thus provide effective demonstration and model checking.

To allow one specification to generate others with corresponding implementation classes, a set of mapping schemas can be defined in MaramaTorua (Huh et al, 2007) to provide interchanging mechanisms between ViTABaL-WS, Kaitiaki and MaramaTatau specifications. We are currently exploring the mapping specifications to be used for such integration. More details are proposed in the Future Research section of Chapter 11.

8.4.4.4 Model, View, Controller

A sub-model for the event handling building blocks is added on top of the Marama EMF model to support specifying event-based manipulations of Marama structural model elements. The behavioural sub-model contains the definition of all the generalised canonical event model elements and provides a structured way to query and update these element instances. The behavioural sub-model is represented in different metaphoric views in the style of ViTABaL-WS, Kaitiaki and MaramaTatau. Figure 8.12 illustrates the MVC pattern used in Marama meta-tools to synthesise event-based behaviour from multiple views. Model states are manipulated and view representations are synchronously rendered by user interactions (via drag/drop, add, select, delete, move etc. menu events) on the views. This is managed by a set of central Marama controller commands that delegate corresponding actions to the model and views.

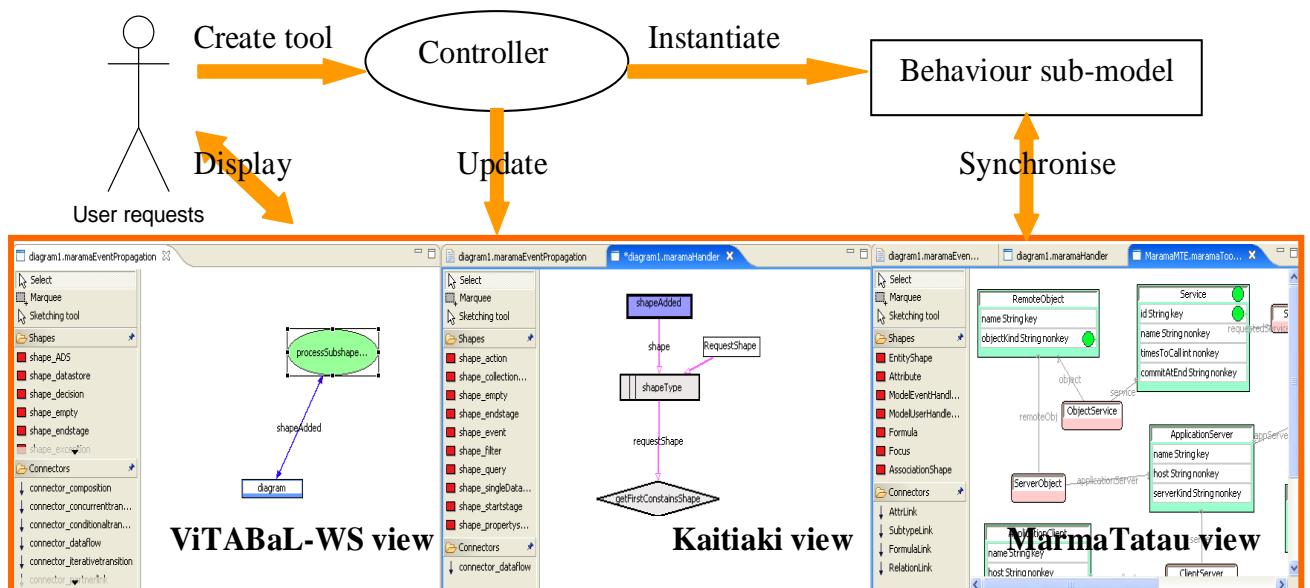


Figure 8.12. MVC of Marama meta-tools.

Visualisation of dynamic event handling behaviour is achieved using a similar MVC approach, where runtime behaviour model states are used to animate the associated diagram elements. The user has full control of running the animation, stepping into the next invocation of a building block and viewing query results or state changes.

8.4.4.5 Metamodel, Icons and Views

The Metamodel, icons and views are the static modelling capability supported by Marama meta-tools. The Metamodel includes semantic entities and relationships of a visual language tool (as shown in Figure 8.13)); icons provide visual shape and connector representations (as shown in Figure 8.14); views specify mappings of metamodel elements to visual elements and filtered displays (as shown in Figure 8.15). These static components are the backbone of a visual language tool being modelled, with the dynamic behaviours to be instrumented using ViTABaL-WS, Kaitiaki and MaramaTatau.

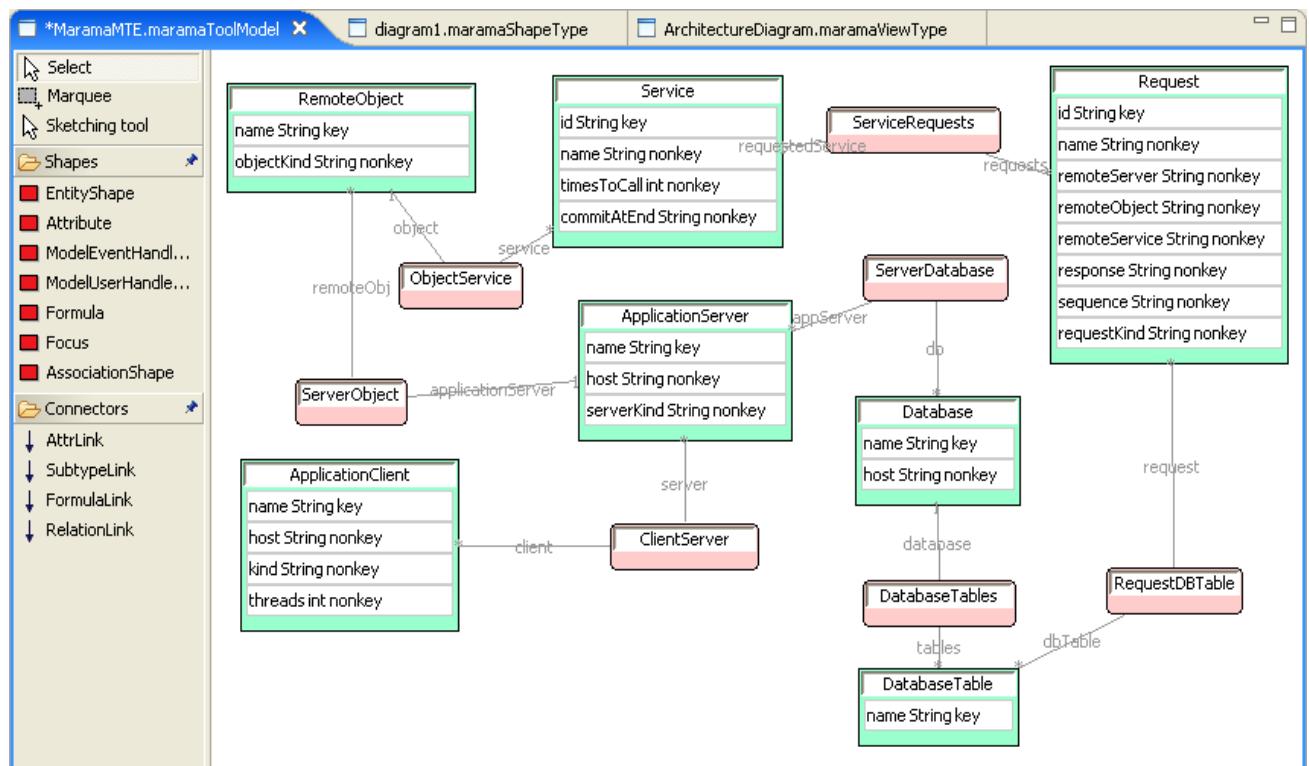


Figure 8.13. Metamodel definer in Marama meta-tools.

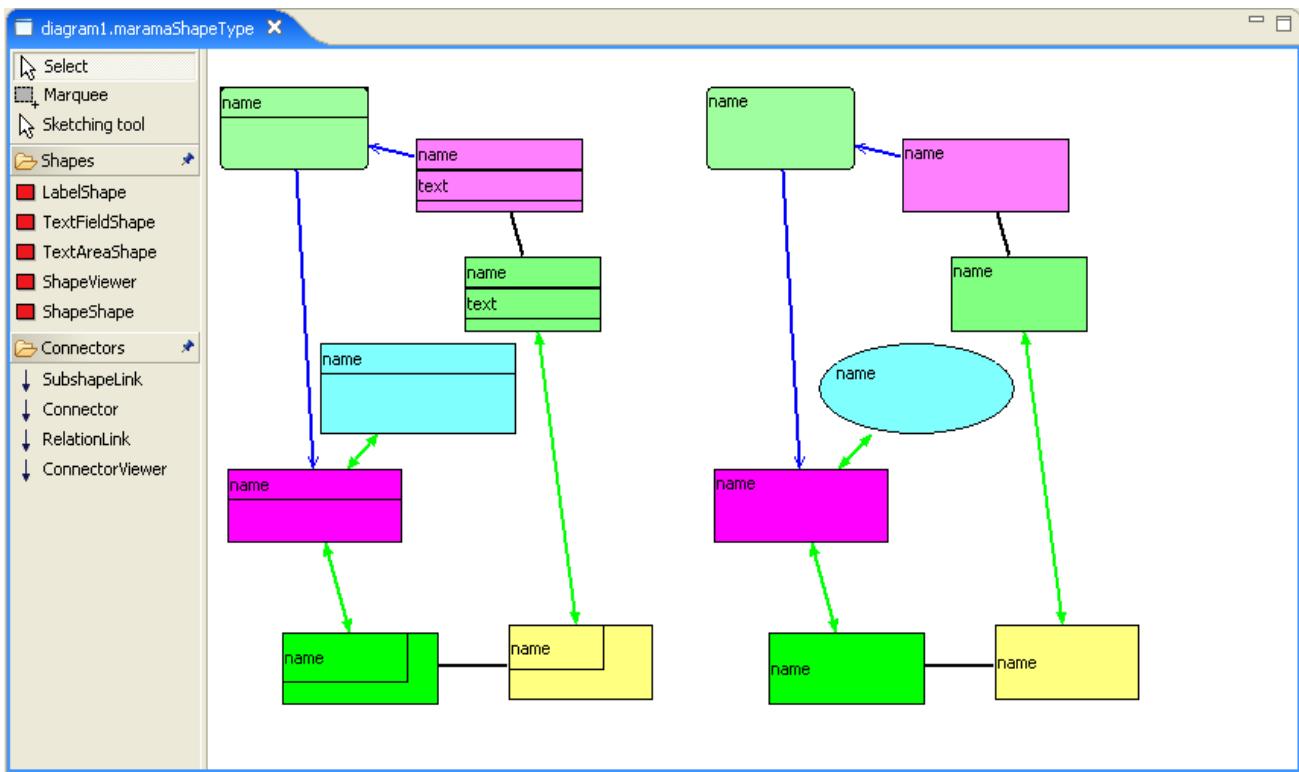


Figure 8.14. Shape designer in Marama meta-tools.

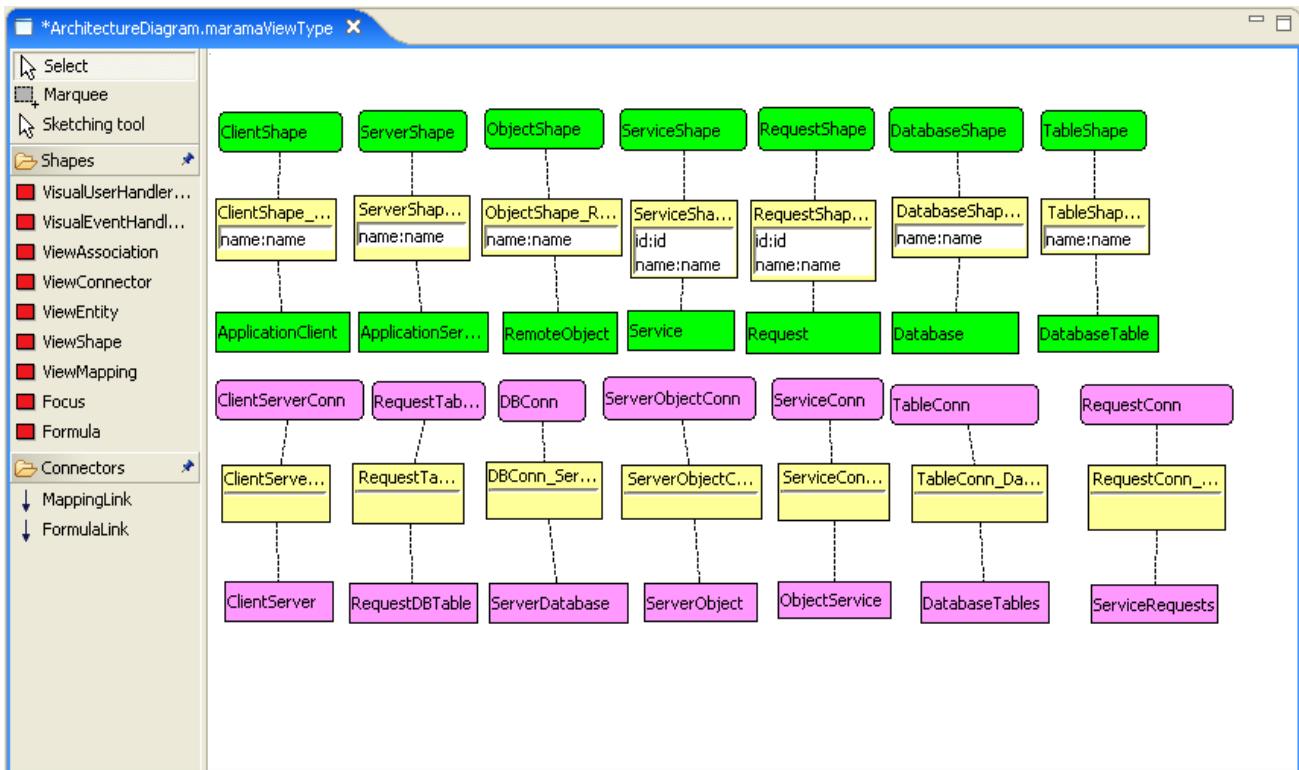


Figure 8.15. View type definer in Marama meta-tools.

8.4.4.6 Event Handling Views

As shown earlier in Figure 8.4, an Event Propagation Definer (ViTABaL-WS view type) has been integrated into the Marama meta-tools to facilitate the specification of model and view instance notification schemes, i.e. event propagations (event flows) and responses/handling among components through inter-component links to maintain consistency between multiple representations and views. For example, a view will be notified of a model property change; connectors are notified of a hiding action of a shape. Allowing users to customise event type, event generators and event receivers greatly enhances the extensibility and flexibility of Marama meta-tools. EMF notifications for Marama tools can also be specified using ViTABaL-WS.

As shown in Figure 8.5, a Marama Handler Definer (Kaitiaki view type) has been integrated into Marama meta-tools to facilitate the specifications of model and view event handlers using a high-level domain specific visual language. High-level event handling specifications are generated to Marama event handler code, registered to the generated metamodels. The generated event handler code is guaranteed to be syntactically correct. Event handlers are executed on model instances at runtime.

MaramaTatau is declarative. The left-hand-side of a formula is the formula context, while the right-hand-side specifies a constraint/query. The query result is assigned to the formula context. We wanted to extend the MaramaTatau language by:

- Adding user defined functions to collaboratively operate with OCL, and to reuse ViTABaL-WS and Kaitiaki building blocks.

MaramaTatau has been extended to be embedded seamlessly into the Metamodel Definer and View Type Definer, providing dependency and constraint specifications on both model-level and view-level semantics. The extended entity-relationship (EER) model allows sub-typing relationships to be specified between both entities and associations, with also inheritance of dependencies and constraints. Besides supporting declarative OCL-based invariant definitions in MaramaTatau, user-defined functions (higher-order/compositional functional definition of reusable formula with parameters) can be added to extend OCL formulae and provide a powerful yet concise mechanism for functional specifications over group objects and to reuse ViTABaL-WS and Kaitiaki building blocks. There are two categories of user defined functions:

- Side-effect-free constraint/query, and

- Side-effect action, i.e. an operation for event handling/constraint handling, e.g. abort, skip, or “do-something”.

Examples of model formulae have been presented in Chapter 7. Figure 8.16 shows an exemplar View Type Definer view with MaramaTatau specified formulae, in which an extended formula “encloses(ObjectShape, ServiceShape, ServiceConn)” specifies the enclosure constraints of the “ObjectShape” and the “ServiceShape”, which enforces a “ServiceShape” to be created inside an enclosing “ObjectShape” and moved together with it.

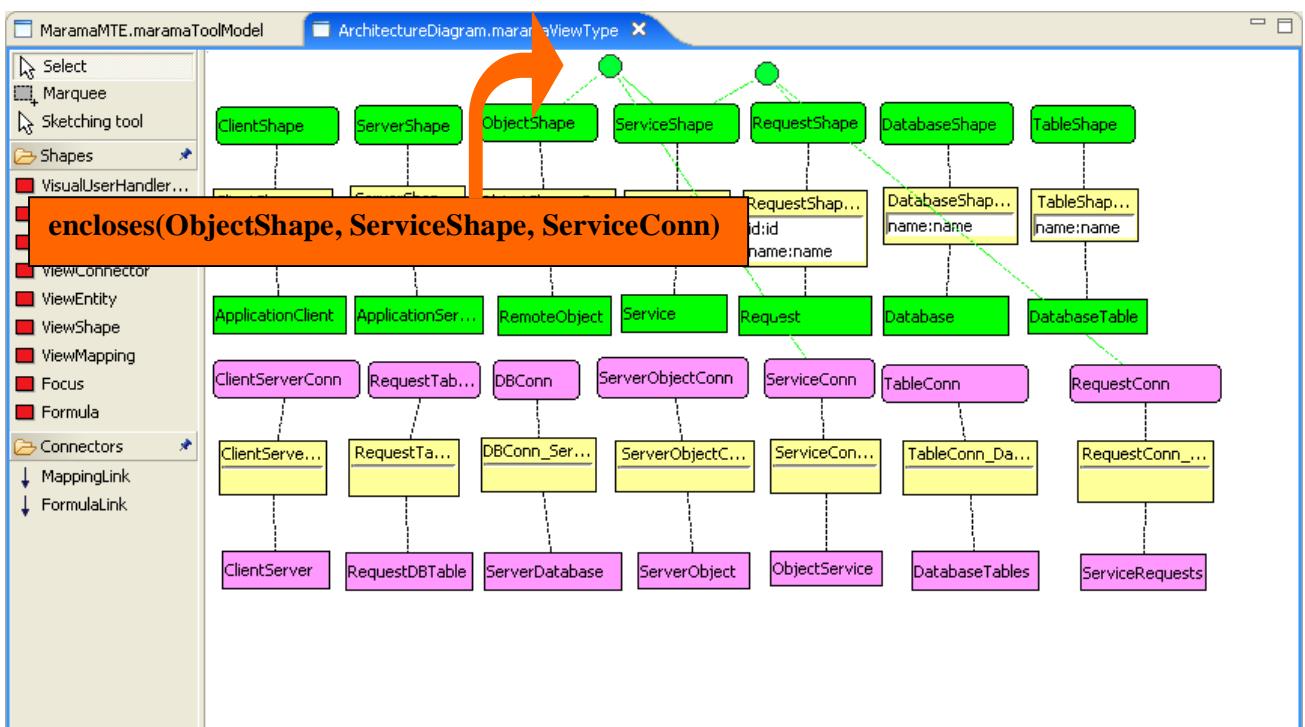


Figure 8.16. MaramaTatau integration in View Type Definer.

8.4.5 Program Visualisation

Marama meta-tools allow users to visualise tool specifications and their executions reusing their metaphoric modelling views, to provide system information at the right abstraction level. The dynamic visualisation system uses the debugging service instrumentation mechanism (Liu et al 2005) initially exploited in ViTABaL-WS to generate low-level tracing events on modelling elements. As illustrated in Figure 8.17, the Marama meta-tools framework handles those events by sending the event data to appropriate modelling elements and annotates them with colours and state information. Marama EMF is the common high-level representation that glues different behavioural views, and supplies dynamic state information to the Marama Visual Debugger. A specialised debugging and

inspection tool is used to allow execution state of event-based systems to be queried, visualised and dynamically modified. The debugger tool provides a common user interface that connects the three metaphoric event specification views with an underlying debug model based on the MVC pattern.

The individual “debug and step into” visualisation of ViTABaL-WS, Kaitiaki and MaramaTatau are now put together to allow cooperative invocation and step-by-step visualisation of execution results at the point of execution of each building block in a particular view.

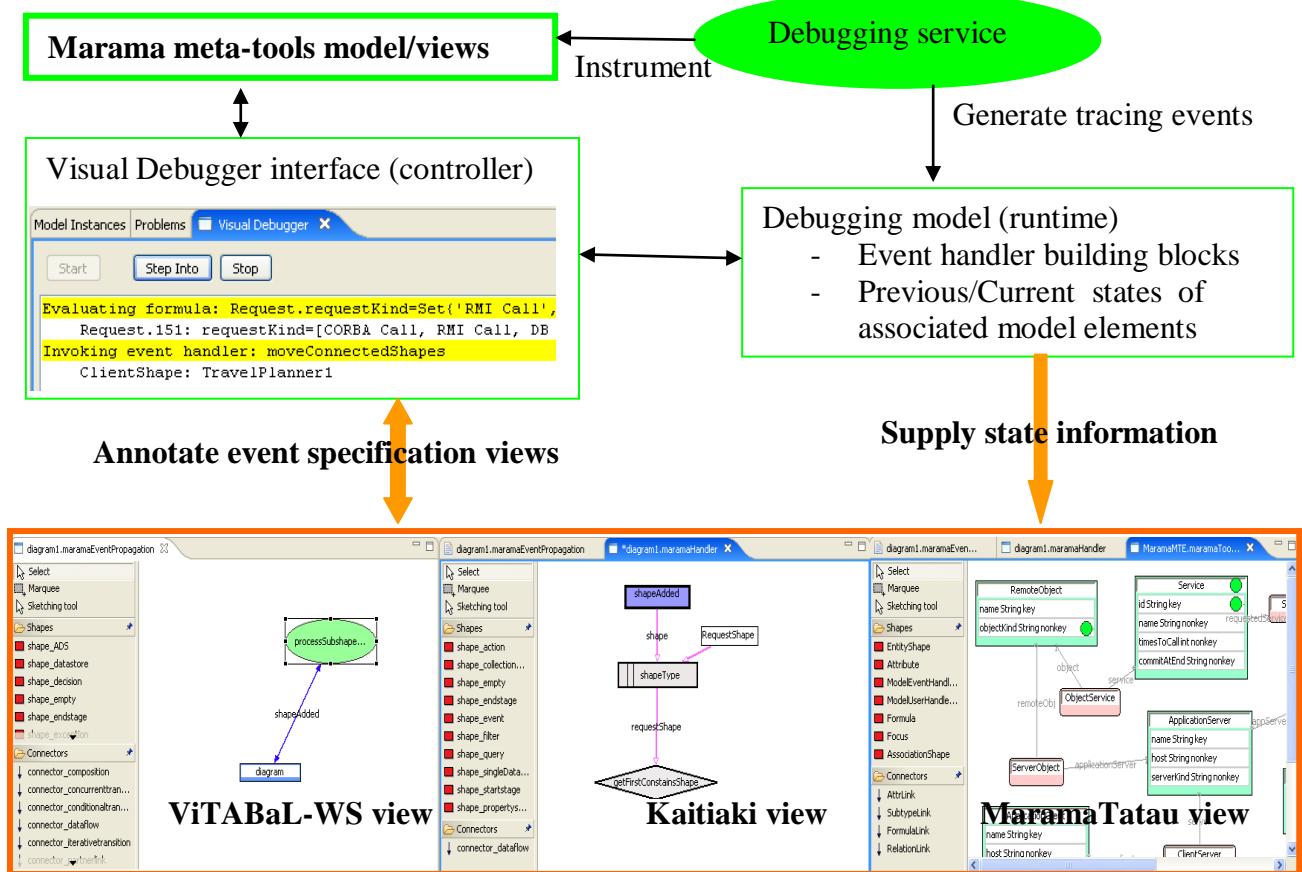


Figure 8.17. Visual Debugger.

Figure 8.18 illustrates the visualisation of runtime interpreted formulae on a Marama model. The Metamodel Definer view with MaramaTatau formula specifications is juxtaposed with the runtime Marama model view. From the Visual Debugger, user has the control over the execution of a formula interpretation. Once a formula is interpreted, the affected runtime model element is annotated (with the yellow background) to indicate the application of the formula, and meanwhile, the corresponding formula node defined in the Metamodel Definer view is annotated in the same manner to show the formula specification.

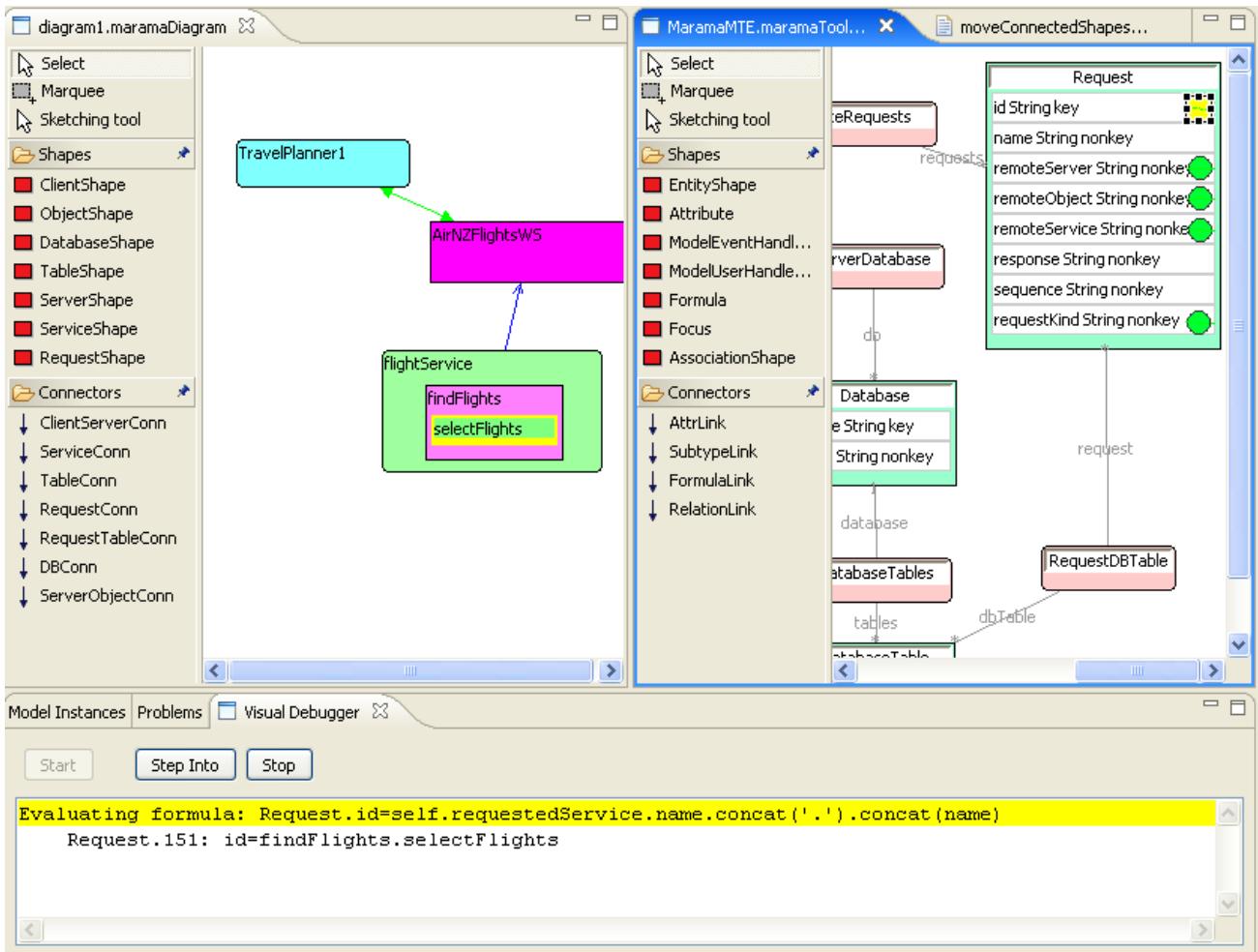


Figure 8.18. Visual debugging MaramaTatau formulae.

Figure 8.19 illustrates the similar visualisation of a Kaitiaki event handler execution on a Marama model. The Kaitiaki view with event/query/filter/action specifications is juxtaposed with the runtime Marama model view. When a user steps into an execution of a Kaitiaki building block, the affected runtime model element is annotated and at the same time, the corresponding Kaitiaki node with data flow links are annotated in the Kaitiaki view to show the event handler execution status.

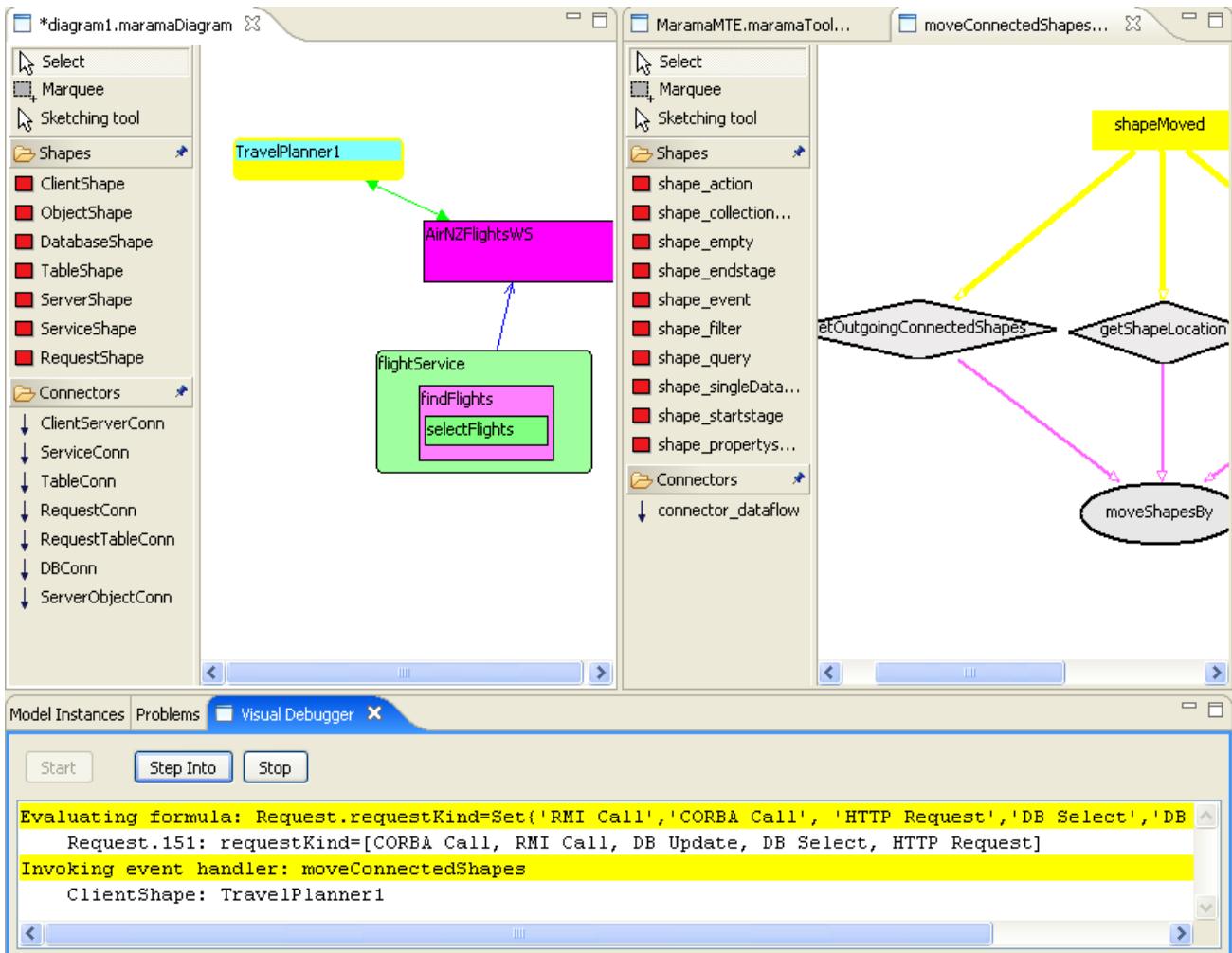


Figure 8.19. Visual debugging a Kaitiaki event handler.

8.4.6 Framework Evolution

The event handling abstractions framework in Marama meta-tools is both black-box and white-box. It provides reuse by both inheritance and composition. Based on the evolving frameworks pattern language (Roberts and Johnson, 1996), our framework will be evolved by abstracting from additional examples to make the framework more general in the future.

Subsequent exemplars are to be developed based on the white-box framework. Our next planned exemplar to be used in generalising more of the event handling abstractions framework is by integrating a BPMN view (OMG, 2006) into the Marama meta-tools. We will first examine what abstractions from the canonical model's component library (as shown in Figure 8.11) can be reused (through either inheritance or composition) by the new BPMN model, and then examine what new features and support can be added to evolve the framework.

The integration of MaramaTorua (Huh et al, 2007) can also provide another view type, with event-driven mechanisms to allow translation of one event handler view to another (e.g. generating the event handlers/formulae to keep view/model consistent from MaramaTorua specifications) to be specified internally and automatically without the need to invoke it as an independent non-event-driven third-party tool. Our generalised model should support new metaphors/models to be further sub-typed or composed.

8.5 Summary

Our research has focussed on providing visual specification and runtime visualisation support for the design and construction of complex event-based systems. We have integrated three event handling specification languages based on a canonical event model. ViTABaL-WS provides a Tool Abstraction language for the design and construction of action-event propagation architectures. Kaitiaki provides an extensible Event-Query-Filter-Action language for both action and state-change event propagation and handling. MaramaTatau provides a static Spreadsheet-like dependency and constraint mechanism to support specification of state-change event propagation and response. A synergy of these languages and their generalisation in the Marama meta-tools environment provide wider-ranging support for event-based system design and construction.

Chapter 9 - Prototype of the Generalised Event Handling Framework

We have implemented the Marama metamodeling environment to support generalised event handling abstraction. Multiple views can be provided that use the ViTABaL-WS, Kaitiaki and MaramaTatau metaphors, allowing both system structure and behaviour to be modularised vertically via partially overlapping views, or horizontally using hierarchical views, similar to the EASY (Grundy et al, 1996) framework approach. A ViTABaL-WS-like visualisation allows users to examine the execution of event-based architectures. A Kaitiaki-like visualisation allows users to reuse or extend the functionality of the language by defining queries, filters and actions using the building blocks of the event handling abstractions framework. MaramaTatau-like formulae can be added in both metamodels and views to specify structural dependencies and constraints, also extensible with user-defined functions. In this chapter we describe and illustrate the key features of our prototype.

9.1 Introduction

As described in Chapter 7, Marama provides a collection of object-oriented classes written in Java and Eclipse Modelling Framework. These classes provide abstractions for specifying representation of language structure and semantics, with multiple textual and graphical view support that allows manipulation of model and view information.

We are motivated by the need for a flexible metamodeling environment as a vehicle to experiment with the generalisation of our work to an event handling integration framework, and also to provide proof of concept implementation of the set of requirements for the generalised framework described in Chapter 8.

We have developed the Marama meta-tools to replace the Pounamu tool specifications, even though Pounamu specified tools are still loadable and function within the Marama environment. With this extension, Marama integrates the features of both meta-tools and modelling tools. Marama uses the Marama Metamodel Definer view to specify the tool metamodel with MaramaTatau formulae for

model dependencies and constraints, Marama Shape Designer views to specify tool shapes and connectors, Marama View Type Definer views to specify mappings of meta-elements to visual representations with MaramaTatau formulae for visual dependencies and constraints, the Marama Event Propagation (ViTABaL-WS) definer to specify event propagations among model and view components, the Marama Visual Event Handler (Kaitiaki) definer to specify visual event handlers, and Marama Diagram views to create model instances or independent views of selected view type.

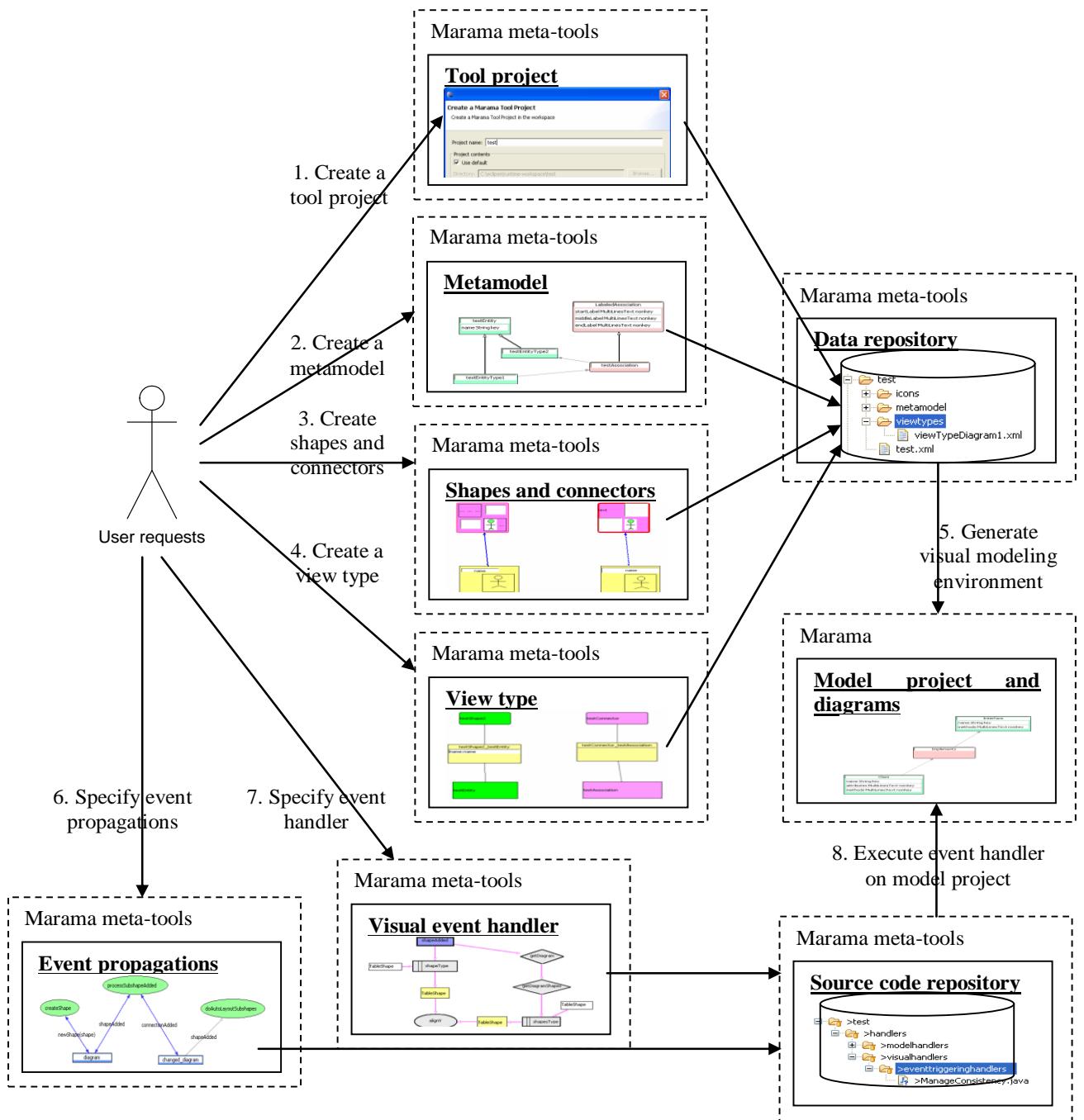


Figure 9.1. Overview of tool creation with Marama meta-tools.

A tool project created using Marama meta-tools packages a tool definition that comprises the tool metamodel, shapes and connectors, view types, dynamic event propagators and handlers. The backend XML files of the tool, metamodel, shapes and connectors, view types, and constraints are generated into the tool project data repository. Visual event propagation and event handler specifications are generated into Java code in the tool project source code repository. A model project can be instantiated using the tool definition which automatically generates a modelling environment in Marama. The model project contains the definitions of the tool metamodel and its instance diagrams. The tool dependencies, constraints and dynamic event-based behaviours specified by MaramaTatau, ViTABaL-WS and Kaitiaki are realised at runtime in model instances. Figure 9.1 illustrates the tool creation process using Marama meta-tools. The following steps are involved in creating a tool with both structural and behavioural modelling capabilities:

- (1) The user creates a tool project using the Marama Tool Project Wizard. This wizard will guide the user through an empty tool project creation, allowing the user to enter a name for the tool and the preferred storage location. A tool XML file is automatically generated into the tool project data repository. A Metamodel Definer view is automatically provided with the newly created tool project.
- (2) The user can define the tool metamodel by specifying entity and association types and their attributes, sub-typing relations and formulae for defining value dependencies and constraints. Metamodel XML files are automatically generated into the tool project data repository.
- (3) The user can create a Shape Designer view (or multiple views) by using the Marama Shape Designer Wizard. A visual editor for creating/modifying shapes and connectors is loaded. The user can visually construct the shape representation for a metamodel entity type, and a connector representation for a metamodel association type. Shapes and Connector XML files are automatically generated into the tool project data repository.
- (4) Having the metamodel and visual representations, the user can then create a View Type (or multiple views) by using the Marama View Type Definer Wizard. A View Type definer editor is loaded so that the user can define the view type specific shapes, connectors, entities, associations, view-model mappings and formulae for defining visual representation constraints. The view type XML file is automatically generated into the tool project data repository.
- (5) The tool project now has static modelling capabilities and can be used to instantiate model project instances using the specification.

- (6) The user can add further a ViTABaL-WS view (or multiple views) by using the Marama Event Propagation Definer Wizard. Event propagations can be specified to facilitate specification of an event-based tool architecture. Event handler Java code is automatically generated into the tool project's source code package.
- (7) The user can further add a Kaitiaki view (or multiple views) by using the Marama Visual Event Handler Definer Wizard. An event handler can be specified visually by composing a set of library building blocks. Event handler Java code is automatically generated into the tool project's source code package.
- (8) The event handler Java code is executed dynamically on model project instances.

In Section 9.2 and 9.3, we extend the MaramaMTE architecture design and performance test-bed generation tool, the complex example used in both Chapter 7 and 8 to describe and illustrate in detail the key facilities of the Marama meta-tools.

9.2 Structure Specification

Structural elements (including metamodel entity and association types, shapes and connectors, and view type elements) of a modelling tool specification form the backbone of the tool. In the following subsections, we illustrate how to use the Marama meta-tools to build structural elements to generate a modelling tool.

9.2.1 Marama Tool Project

Marama provides a wizard (Marama Tool Project wizard) for creating a modelling tool project. As shown in Figure 9.2, a short cut menu called “Marama Tool Project” brings up the tool creation wizard. A tool project contains the definitions of a metamodel, shapes and connectors, and view types. The backend XML files of the metamodel, icons and view types are generated into the tool project folder. A metamodel definer view is automatically generated with the tool creation. Shape designer and view type definer views need to be created by using their own wizards.

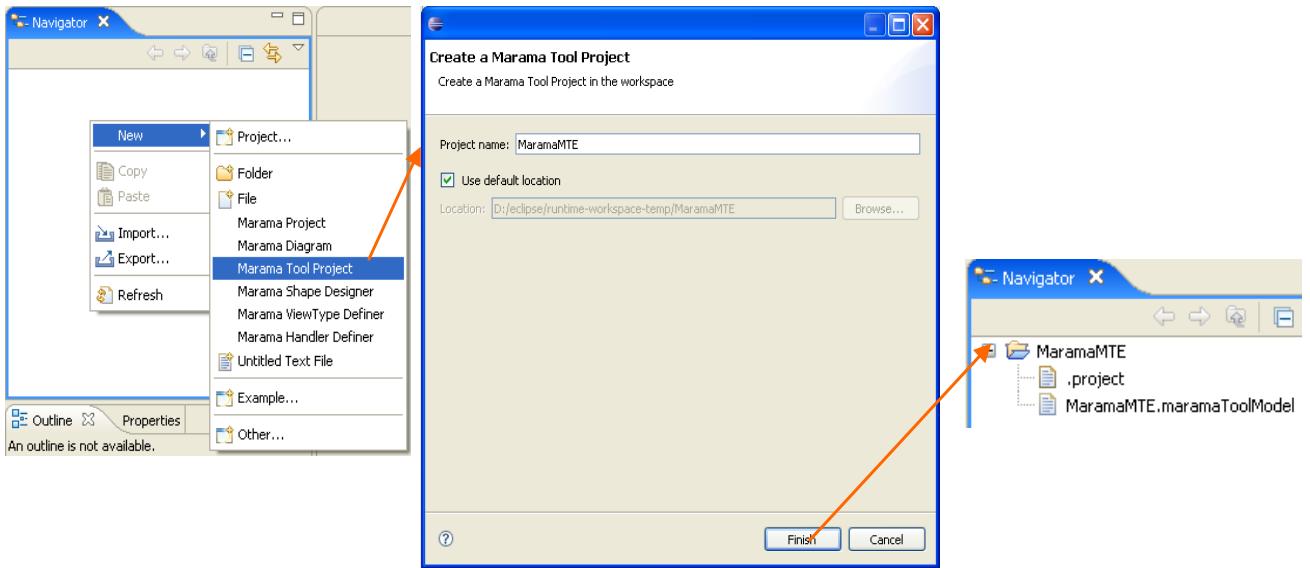


Figure 9.2. Tool creation in Marama meta-tools.

9.2.2 Marama Metamodel Definer

Marama adopts an Extended Entity-Relationship paradigm for its metamodeling. Entity types, association types and attributes of a tool specification are defined in a Marama Metamodel Definer view. MaramaTatau formulae can be specified to constrain metamodel elements and compute dependent values.

An entity type can be created by dragging and dropping an EntityShape from the editor palette. We name the entity type by changing the name property of the EntityShape from the Properties view.

An association type can be created by dragging and dropping an AssociationShape from the editor palette. We name the association type by changing the name property of the AssociationShape from the Properties view. Association end types and association end multiplicities must also be specified. As seen in the Properties view of the ServiceRequests AssociationShape in Figure 9.3, the user needs to select an “end1” (e.g. Service) entity type and an “end2” (e.g. Request) entity type; two relation links are automatically added between the Service entity type and the association type and between the association type and the Request entity type. The user can also add to the relation links a role name for each association end entity type. This constrains that only the ServiceRequests association type can be used to relate the Service and Request entity types. The user then selects “end1Multiplicity” (e.g. 1) and “end2Multiplicity” (e.g. * representing many) to define the allowed number (one-to-many, many-to-one, and many-to-many) of Service and Request instances that can be connected via ServiceRequests associations.

A metamodel validation process generates error messages against any association type that omits property settings for the association end types or multiplicities. We use association types to constrain connectivity based on the entity types and their multiplicities. Such an association type specification prevents users from creating invalid connections between entity instances in the runtime model. The Marama runtime modelling environment automatically removes any association instances that violate the constraints, e.g. if the user adds a ServiceRequests association between a Service entity and a Database entity, the association will be automatically removed from the model, because such an association is only allowed to connect a Service entity to a Request entity as it is defined in the metamodel. This facility is needed to automate model checking and enforce valid model specifications.

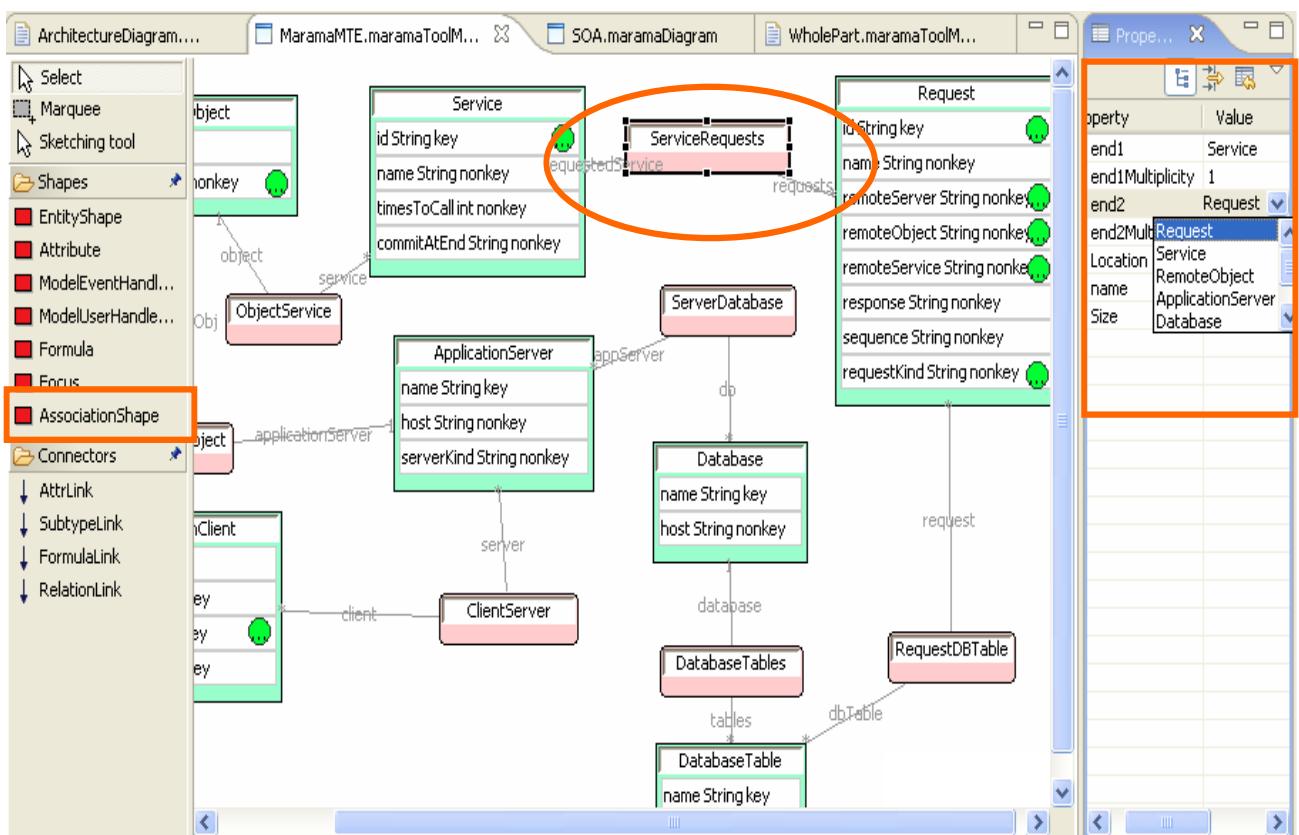


Figure 9.3. Association type specification in Marama meta-tools.

The user may add a number of attributes to an entity type or association type by dragging and dropping an Attribute from the editor palette to a parent shape containing location. The “name”, “iskey” and “type” property of the attribute need to be specified. Figure 9.4 shows an example of Attribute properties specification. Both the “iskey” and the “type” property of the attribute can be selected from a dropdown list of values. Marama meta-tools support six basic built-in attribute types,

they are String, MultilinesText, int, double, boolean and Object types. The available metamodel entity types are also included and can be used as attribute types.

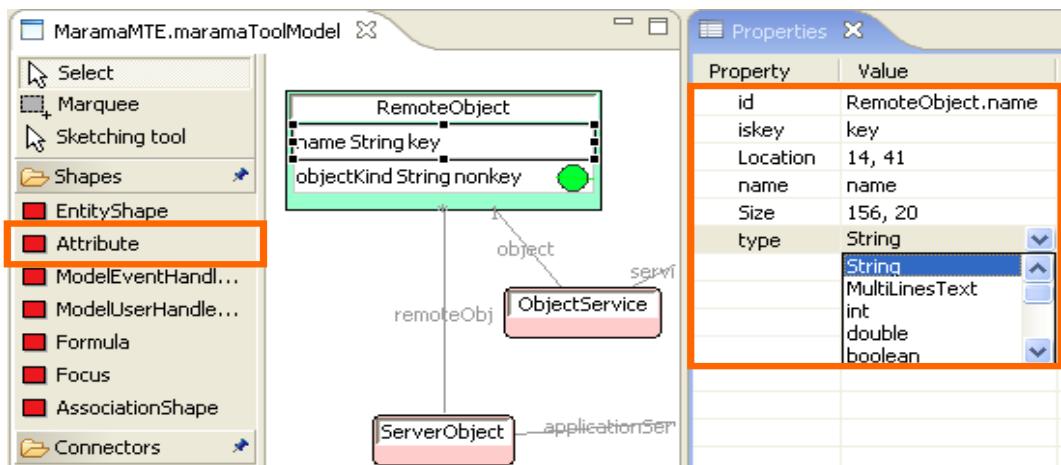


Figure 9.4. Attribute specification in Marama meta-tools.

Marama supports sub-typing entity types and association types. The sub-typing relationship is established by adding a SubtypeLink connector from a child to a parent meta-element. The subtype elements inherit all attributes defined in their parents. Figure 9.5 shows that the ObjectService and ServerObject association types are the subtypes of the LabelledAssociation association type, and thus inherit all of its attributes.

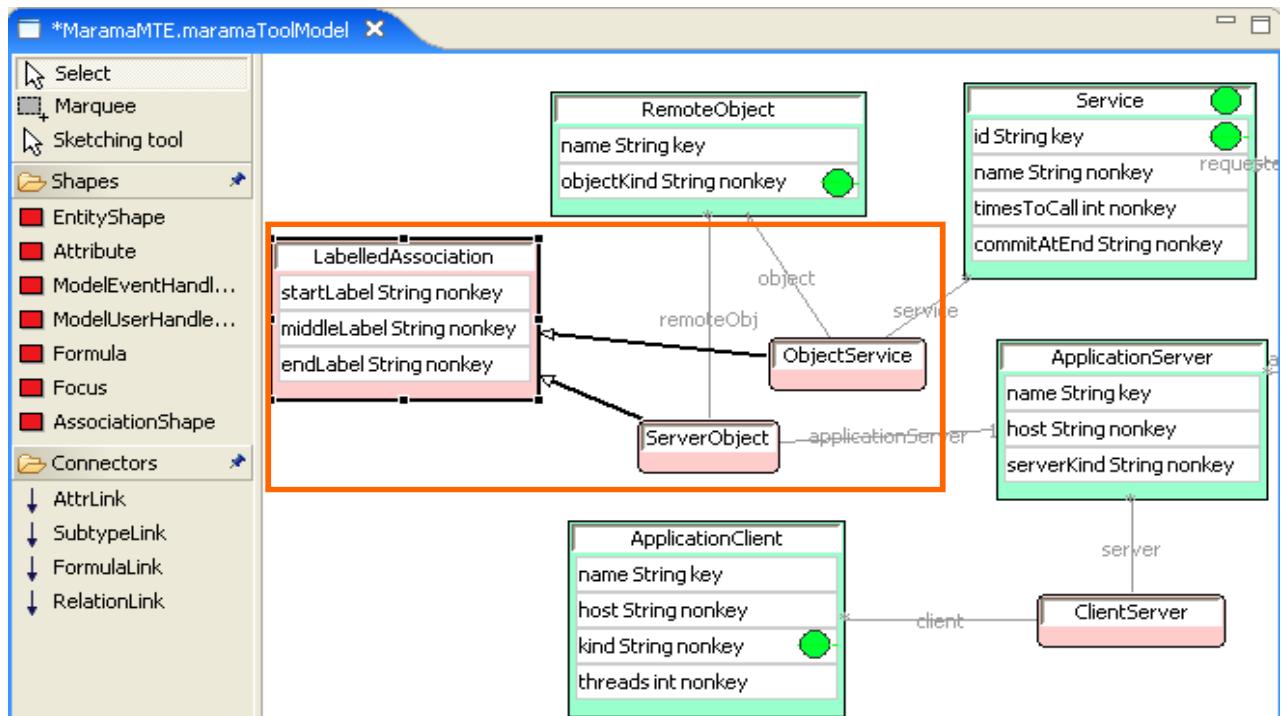


Figure 9.5. Sub-typing in Marama meta-tools.

Saving the metamodel diagram (a Marama Metamodel Definer view instance) from the Eclipse Workbench tool bar or right clicking on the “Register the Model Type” context menu equally generate backend XML files for this metamodel (as shown in Figure 9.6). The metamodel XML files are generated into the tool project folder. Refreshing the folder shows the generated folders and files.

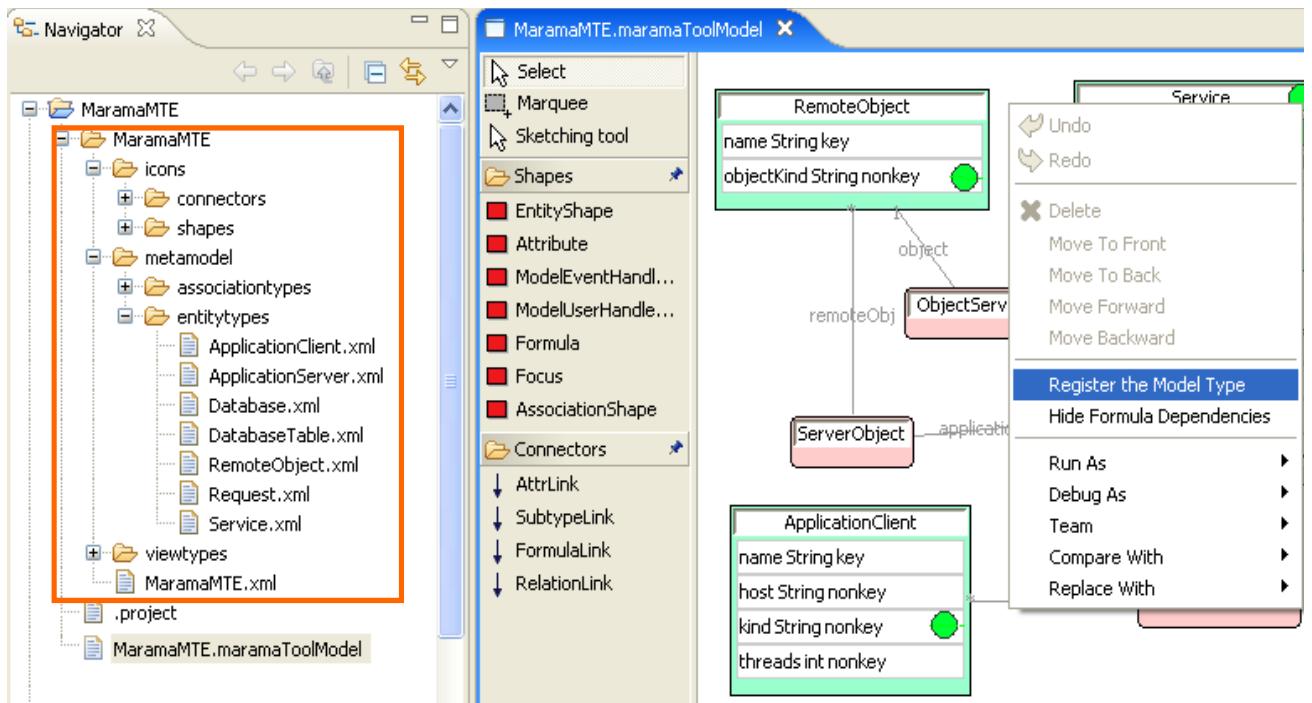


Figure 9.6. Generation of Metamodel XML files in Marama meta-tools.

9.2.3 Marama Shape Designer

Shapes and connectors are visually designed in Marama Shape Designer views. A Shape Designer view (or multiple views) can be created using the Marama Shape Designer Wizard. We can design multiple shapes and connectors in a Shape Designer view. An abstract shape/connector design is accompanied by a concrete viewer instance for immediate design feedback. Figure 9.7 shows a set of shape designs on the left of the view, and their corresponding concrete viewers on the right of the view.

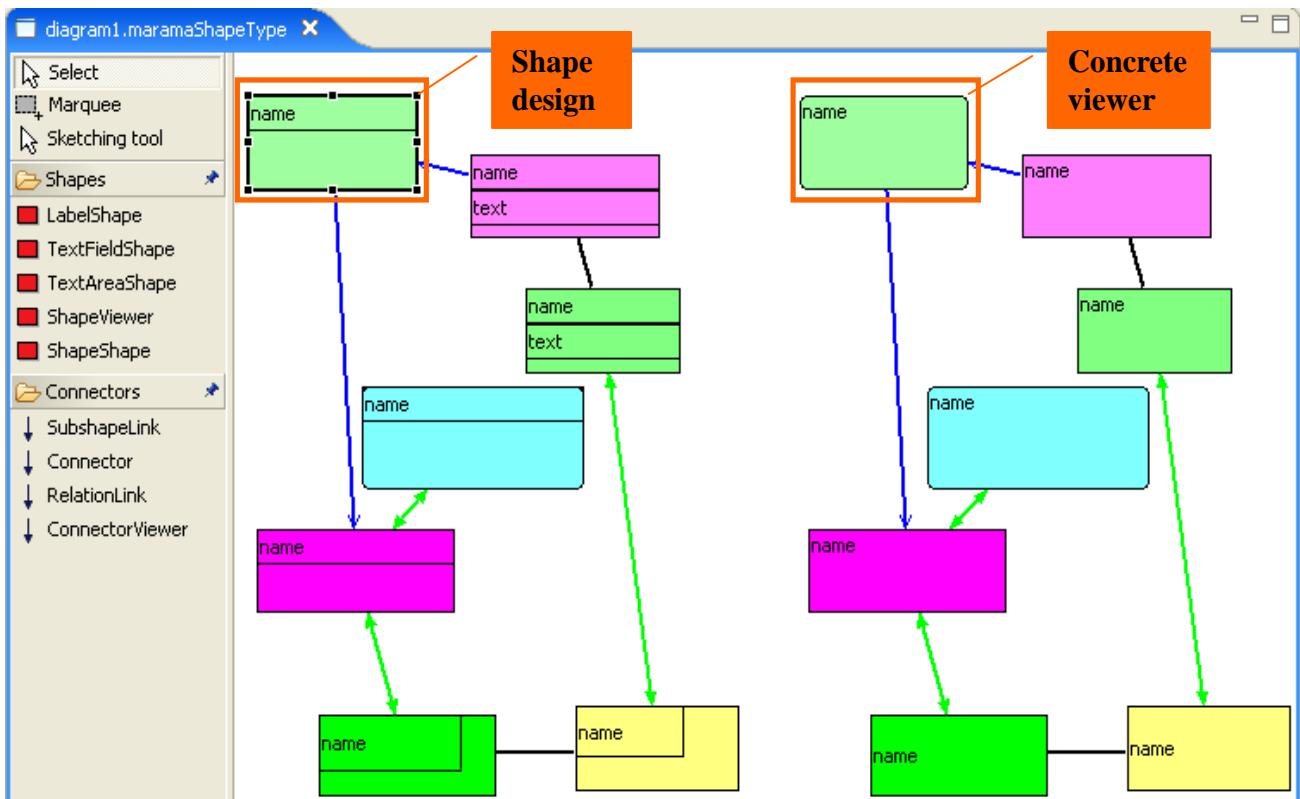


Figure 9.7. Shape and connector design with concrete viewers in Marama meta-tools.

The ShapeShape tool from the palette is used to create owning (base/container) shapes. Each owning shape instance is provided with a shape viewer to visualise the shape design. A “name” property needs to be assigned for a ShapeShape as its identifier.

The LabelShape, TextFieldShape, TextAreaShape and the ShapeShape tool (reusing shape designs) from the palette are used to create sub shapes of an owning shape. A sub-shape is automatically assigned a unique identifier (the name property) when it is added to an owning shape with the specified container layout constraint (e.g. null layout, flow layout, vertical layout, border layout and grid layout). A ShapeShape can be created as a sub-shape as well as an owning shape. A ShapeShape can be moved into an owning shape to become a sub-shape or moved out of an owning shape to become standalone.

There are a number of visual properties that we can set to change the shape’s look:

ShapeShape	<ul style="list-style-type: none"> • fillColor – the filled colour of a shape • shapeOpaque – either the shape is opaque or transparent. The shapeOpaque
------------	--

	<p>property must be set to true in order to show the filled colour</p> <ul style="list-style-type: none"> • <code>lineColor</code> – the colour of the border of a shape • <code>lineVisible</code> – either the border is visible or not • <code>type</code> – four shape types are supported, they are Rectangle, Rounded Rectangle, Oval, and Rombus • <code>stroke</code> – the border line style, e.g. dashed, thickened etc. • <code>layoutManager</code> – five types of layout are supported, they are null layout, flow layout, vertical layout, border layout and grid layout.
<code>LabelShape</code>	<ul style="list-style-type: none"> • background and foreground colour • border – five types of label border styles are supported, they are empty border, line border, etched border, shared raised bevel, shared lowered bevel. • enabled - either enable or disable the label editing • opaque – either the label is opaque or transparent • horizontalAlignment and verticalAlignment – the label's content layout • text – the text content of the label • font – the font of the text • imageResource – an open file dialog used to locate an image resource
<code>TextFieldShape</code>	<ul style="list-style-type: none"> • background and foreground colour • border – as per <code>LabelShape</code> • enabled – as per <code>LabelShape</code> • opaque – as per <code>LabelShape</code> • horizontalAlignment and verticalAlignment – as per <code>LabelShape</code> • text – as per <code>LabelShape</code> • font – as per <code>LabelShape</code>
<code>TextAreaShape</code>	<ul style="list-style-type: none"> • background and foreground colour • border – as per <code>LabelShape</code> • enabled – as per <code>LabelShape</code> • opaque – as per <code>LabelShape</code> • multiLinesText – multiple lines of text content • font – as per <code>LabelShape</code>

The Connector tool from the palette is used to design connectors. Each connector instance is provided with a connector viewer to visualise the connector design. A name property needs to be assigned for a Connector instance as its identifier.

There are a number of properties that we can set to change a connector's look:

Connector	<ul style="list-style-type: none">• basicStroke – the connector's line style, e.g. dashed, thickened etc.• startShape – the start arrow shape, one of the nine arrow types can be selected, they are no shape, half open, full open, half closed empty, half closed fill, full closed empty, full closed fill, diamond empty and diamond fill• endShape – the end arrow shape• lineColor – the connector's line colour• showStartLabel, showMiddleLabel and showEndLabel – options to show a label at the start/middle/end portion of a connector• startLabelContents, middleLabelContents and endLabelContents – set the multiple lines of content to the start/middle/end label of the connector• textColor – the colour of the label text• font – the font of the label text
-----------	--

A number of shape and connector properties can be exported. Exported properties are those that can be mapped to metamodel properties and set at runtime in instances of a shape or connector by the user. We have added an Exported Properties view that is in the same style of the Eclipse Properties view but used to set exported visual properties. The Exported Properties view can be loaded by selecting the Eclipse menu Window → Show View → Exported Properties. With the Exported Properties view in focus and a shape/sub-shape or a connector selected, the view shows the selected icon's properties list. Entering an exported property name makes the property exported. Figure 9.8 shows the process of exporting the text property of a label sub-shape using "name" as the exported property's name.

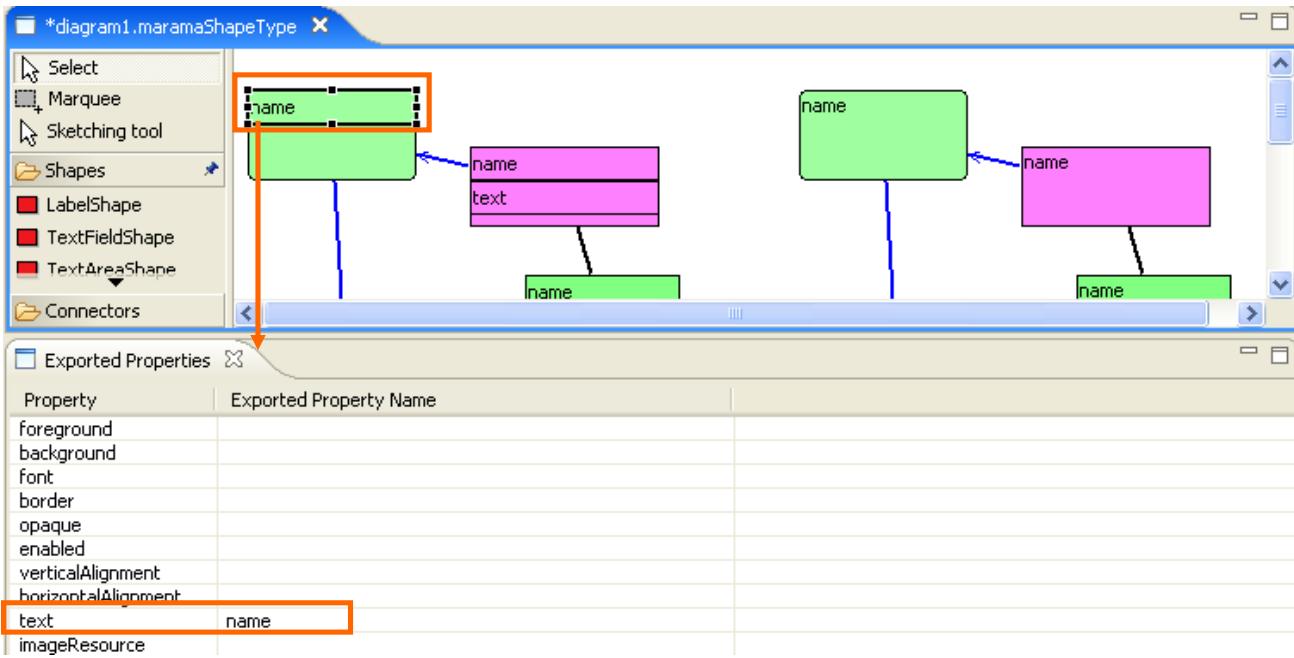


Figure 9.8. Exporting visual properties in Marama meta-tools.

Saving the Marama Shape Designer diagram from the Eclipse workbench tool bar generates backend XML files for the shapes and connectors being designed in the Marama Shape Designer view instance. The icon XML files are also generated into the tool project folder.

9.2.4 Marama View Type Definer

Mappings of metamodel entity types and shapes, metamodel association types and connectors, and properties of them are specified in a Marama View Type Definer view. A view type can be dependent on a metamodel, or standalone. MaramaTatau formulae can be applied to add interconnectivity and constrain visual layout of view elements.

Using the Marama View Type Definer wizard, the user can create a View Type Definer view for an existing project. As shown in the example ArchitectureDiagram view type of the MaramaMTE tool in Figure 9.9, the ViewShape tool from the palette is used to add shapes to the view type. The ViewConnector tool from the palette is used to add connectors to the view type. Available shapes and connectors for the tool project are automatically loaded and selectable to map to a view shape or view connector. The user needs to set the name property of a view shape or view connector by selecting it from a dropdown list in its properties window.

The ViewEntity tool from the palette is used to add entity types to the view type. The ViewAssociation tool from the palette is used to add association types to the view type. Available

entity types and association types for the tool project are also automatically loaded and selectable to map to a view entity or view association. The user needs to set the name property of a view entity or view association by selecting it from a dropdown list in its properties window.

The ViewMapping tool from the palette is used to add mappings of meta-elements and icons to the view type. Existing icons (view shapes and view connectors) and meta-elements (view entities and view associations) from the view type diagram are automatically loaded and selectable for setting the mappings.

With a ViewMapping shape selected in the editor, the user can set the “iconName” property of the ViewMapping shape by selecting an item from its property dropdown list. Similarly, the user can set the “metaElementName” property of the view mapping shape. Mapping links are automatically added between the view shape/connector, view mapping shape and the view entity/association. The “name” property of the view mapping shape is automatically set as its “iconName” property value followed by an underscore and followed by the “metaElementName” property value.

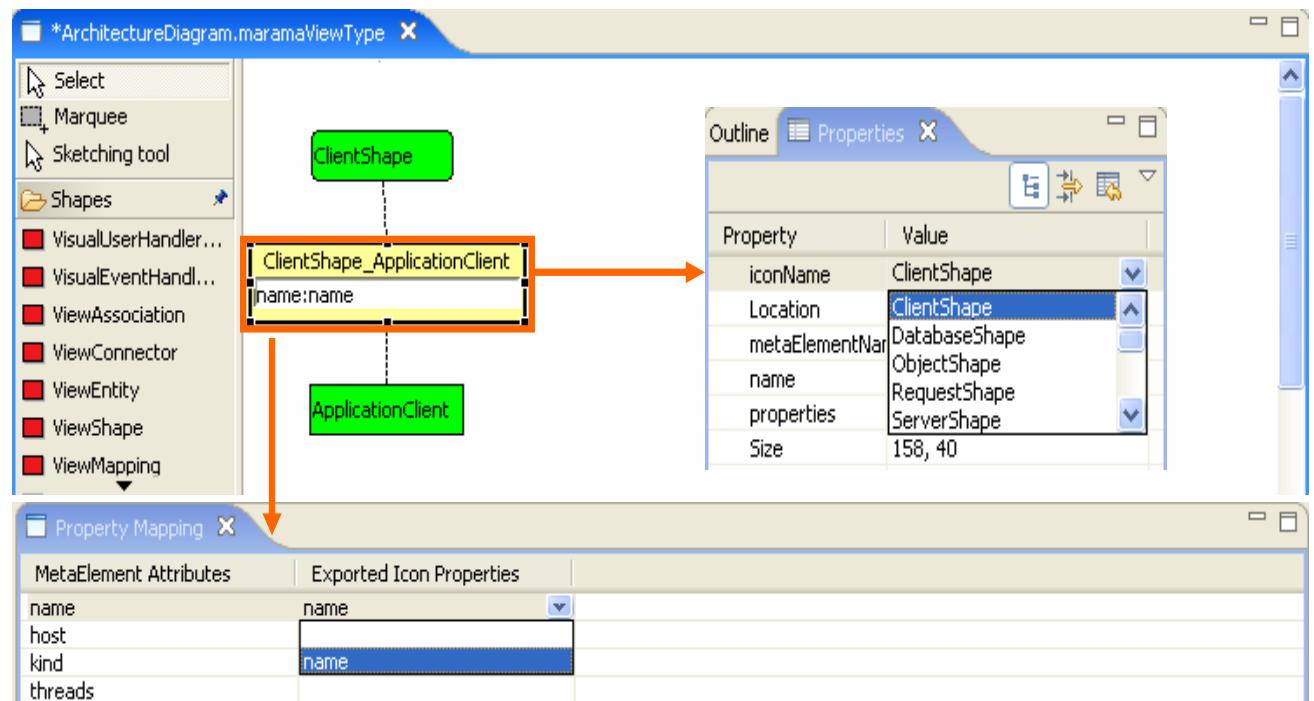


Figure 9.9. View Type Definer in Marama meta-tools.

Property mappings between a view shape and a view entity or between a view connector and a view association are specified in the Property Mapping view. We have added a Property Mapping view that is in the same style of the Eclipse Properties view but used to specify property mappings. The

Property Mapping view can be loaded by selecting the Eclipse menu Window → Show View → Property Mapping. Property mappings can be added by selecting an icon property from the dropdown list to map to a metamodel element property in the Property Mapping view, as shown in Figure 9.9.

Saving the Marama View Type Definer diagram from the Eclipse Workbench tool bar generates the backend XML file for the view type. The view type XML file is generated into the tool project folder.

9.2.5 Marama Model Project and Marama Diagram

The user can create a model project based on a selected modelling tool definition using the Marama Model Project wizard. A model project contains a model instance with multiple view instances. Views can be created using the Marama Diagram wizard, in a similar style to other Eclipse-based wizards. Both the model and view instances can be changed via user interaction. A diagram (view) instance contains all the visual element (e.g. shapes and connectors) creation tools that are defined in its view type in the tool project. As per the mappings of visual elements and model elements specified in the view type, a shape/connector instance in the diagram automatically generates a model entity/association, with appropriately mapped property values. Unmapped visual/model properties of a visual/model element are persisted independently. Figure 9.10 shows an example ArchitectureDiagram view – a view instance of a TravelPlanner model which is created using the MaramaMTE tool. It contains a set of diagram element creation tools such as the ClientShape visual element type, an instance of which maps to an instance of the ApplicationClient model element type. Creating a ClientShape instance e.g. “TravelPlanner1”, as shown in Figure 9.10 generates an ApplicationClient instance with the same “name” property set as “TravelPlanner1” (because the “name” property is a mapped property).

The model project contains a model (file extension “.model”) file, which stores the runtime model state. A view of this model can be used to display the entity types and association types as per the tool’s metamodel specification. An associated/embedded Model Instances view can be loaded by selecting the Eclipse menu Window → Show View → Model Instances. The Model Instances view can then be used to display auxiliary model element information for a selected entity/association type. As seen in Figure 9.11, when the ApplicationClient entity type is selected from the model in the model project, the Model Instances view displays all instances of this entity type as master-details tabular records. Based on the diagram created in Figure 9.10 , an instance of the ApplicationClient

entity type, called “TravelPlanner1”, has been created, hence the record of this instance displays in the Model Instances view. All its associated entities are also displayed as sub-records to this. Expanding the parent record displays all of its related sub-records underneath it.

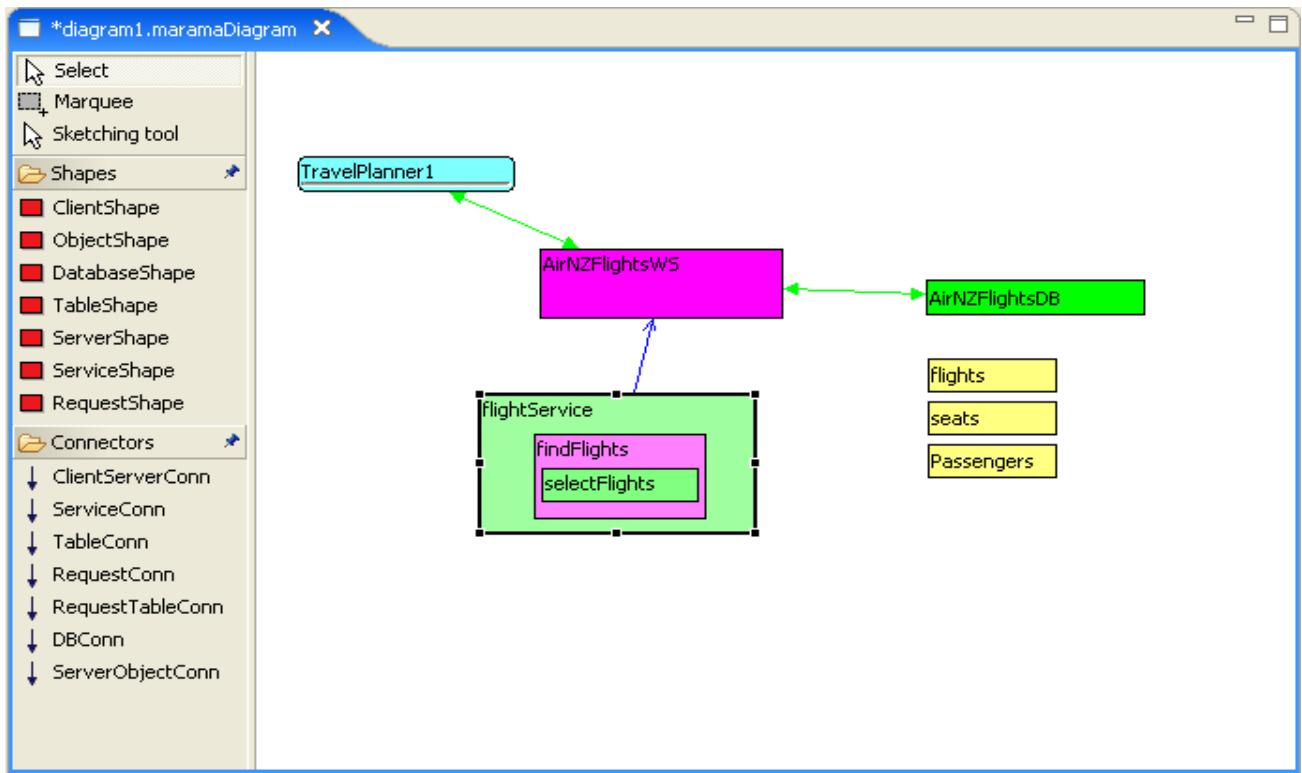


Figure 9.10. A diagram (view) of a Marama model project.

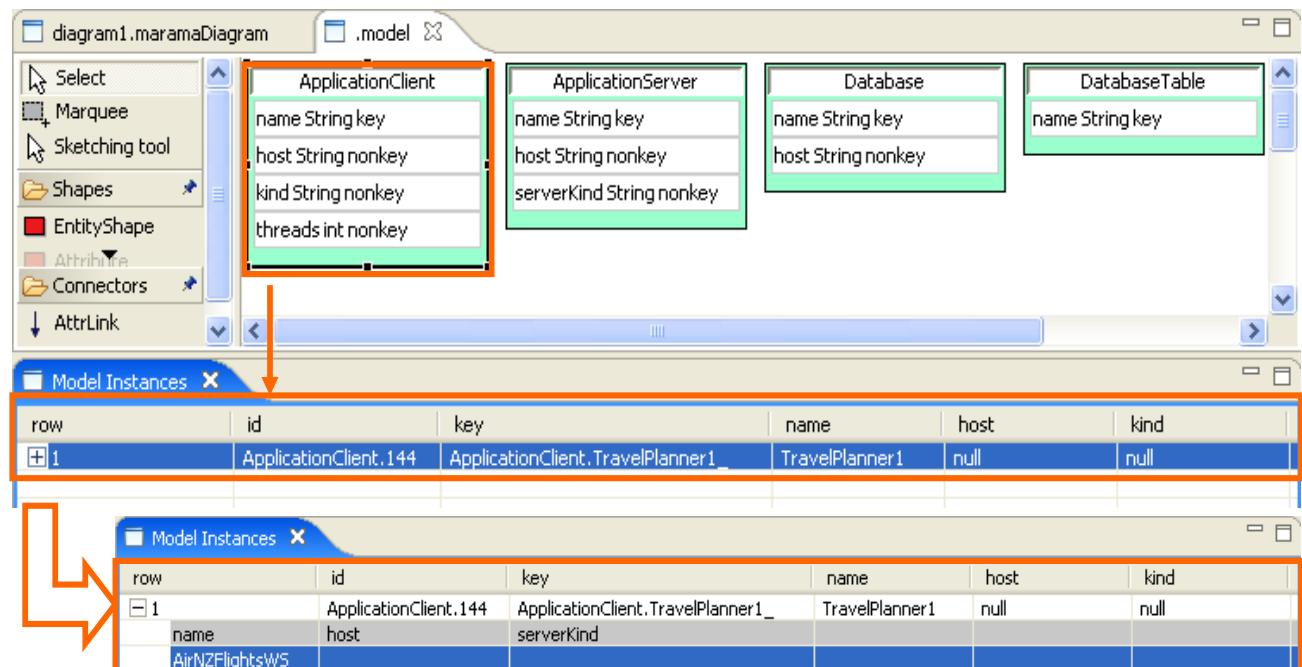


Figure 9.11. The Model Instances view associated with a Marama model project.

9.3 Behaviour Specification

Behaviour specification adds dynamic features such as interactions and constraints to the structural elements of a modelling tool. Behaviours can be specified in a number of ways in the Marama meta-tools, via:

- MaramaTatau formulae on metamodels or view types.

Specifying behaviour using MaramaTatau formulae is easy and efficient in a similar manner to a spreadsheet system. As described in Chapter 7 and 8, it allows definition of value dependencies and constraints on structural elements of a Marama metamodel. Hidden dependencies are mitigated in various ways when specifying formulae.

- ViTABaL-WS visual event propagations.

Specifying behaviour using ViTABaL-WS allows visual definition of event propagations between shared model data structures and functions. As described in Chapter 5 and 8, an event-based architecture can be visually defined for a modelling tool in an efficient way using ViTABaL-WS.

- Kaitiaki visual event handlers.

As described in Chapter 6 and 8, Kaitiaki allows visual specification of event handler behaviour by composing instances of a set of pre-defined building blocks via dataflow. End user domain icons can be added to mitigate Kaitiaki's abstraction and make a graphical event handler specification easy to understand.

- Escaping to code.

Escaping to code is also allowed to enable behaviour specification via flexible coding with Java APIs that are not black-box supported in the Marama meta-tools event handling framework.

In this section, we use further the MaramaMTE (Grundy et al, 2006) tool example to explain where the user would want to use each of the above behaviour specification approaches.

9.3.1 MaramaTatau Formulae

The MaramaMTE tool can be used to model detailed software architecture. Figure 9.12 demonstrates MaramaMTE in use modelling a partial architecture for a travel planner system. The diagram includes a flights web service (“AirNZFlightsWS”) which provides a finding flights (“findFlights”) service. The flights web service has a relationship to a database (“AirNZFlightsDB”) which contains three database tables: “flights”, “seats” and “passengers”. A request (e.g. “selectFlights”) that is made to the finding flights service can be specified with dynamic properties, based on the following

request kind: “CORBA Call”, “RMI Call”, “DB Update”, “DB Select”, and “HTTP Request”. A request’s “remoteServer” and “remoteObject” properties can be set based on a selected request kind. As Figure 9.12 illustrates, if the “requestKind” is set to “DB Select” in Step 1, the “remoteServer” property can be set by selecting one of the available database servers (currently only “AirNZFlightsDB”) in the property list box in Step 2, and then the “remoteObject” property can be set by selecting one of the available database tables in the property list box in Step 3. The information set in the diagram is used to generate a performance test-bed for the system in a later stage.

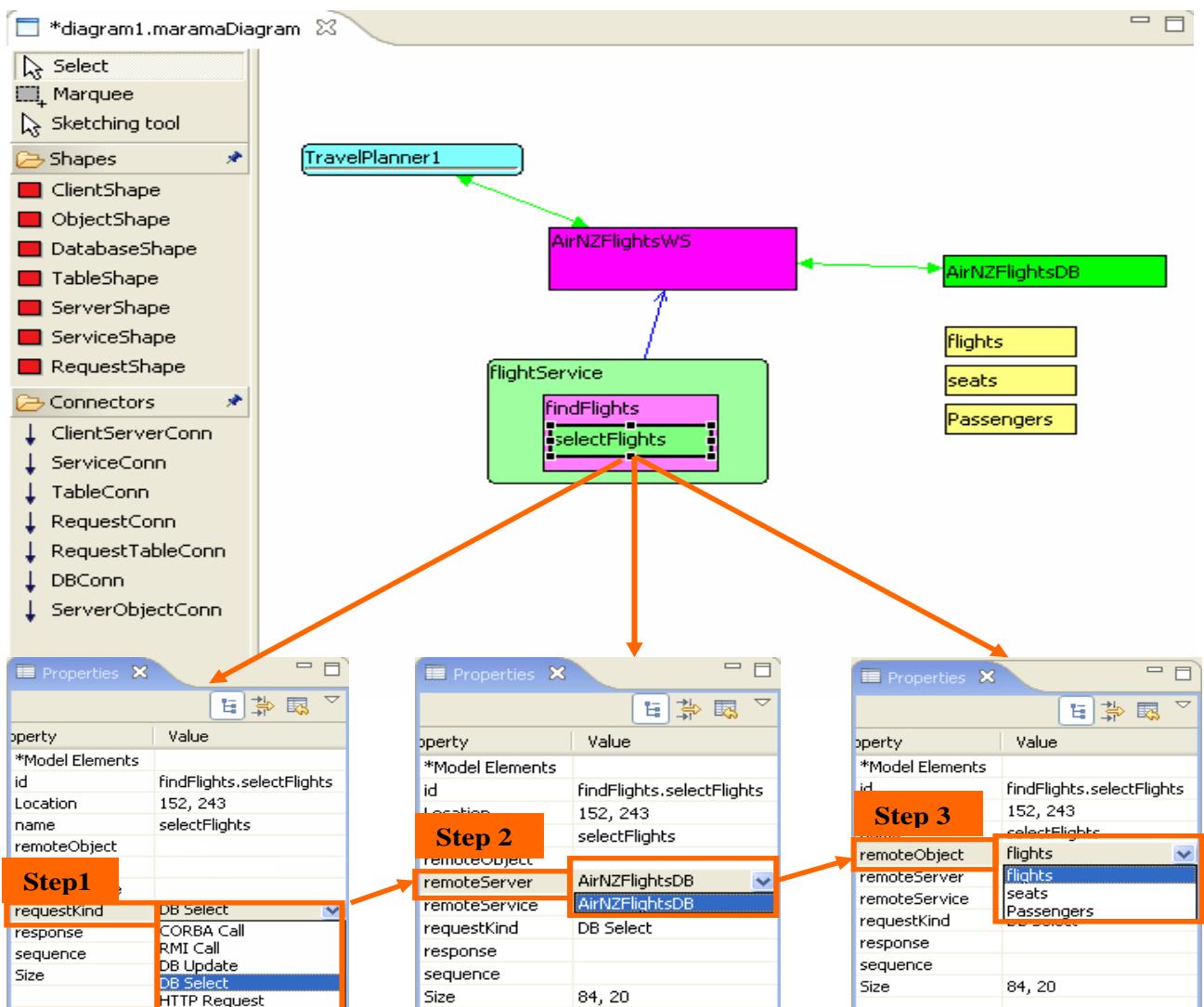


Figure 9.12. Runtime tool behaviour (i.e. derived value updates) enabled using Marama meta-tools.

It would ordinarily require significant coding in order to achieve such value-dependent runtime behaviour. However, this can be easily resolved by using MaramaTatau, as terse OCL expressions

(or user-defined functions) can be applied effectively on the metamodel (or view types) of the tool, and with MaramaTatau’s support, derived property value updates and constraint checks will automatically take place based on the evaluated formula results.

9.3.1.1 Declarative OCL Formulae for Setting Properties and Checking Constraints

As already described in Chapter 7, MaramaTatau facilitates specifying property dependencies as declarative OCL invariants. To achieve the value-dependent runtime behaviour as described in Figure 9.12, we just need to specify three formulae:

- The formula,

```
Request.requestKind=Set{'RMI Call', 'CORBA Call', 'HTTP Request',
'DB Select', 'DB Update'}
```

defines that the “requestKind” property of a Request entity can be any value from the set “{‘RMI Call’, ‘CORBA Call’, ‘HTTP Request’, ‘DB Select’, ‘DB Update’}”.

- The formula,

```
Request.remoteServer=if requestKind='DB Select' or requestKind='DB
Update' then Dabtabase.allInstances()->collect(name) else
ApplicationServer.allInstances()->collect(name) endif
```

defines that if the “requestKind” property is “DB Select” or “DB Update”, the “remoteServer” property of a Request entity can be any value from a set which is calculated by obtaining all the Database entity names, otherwise, all the ApplicationServer entity names.

- The formula,

```
Request.remoteObject=if requestKind='DB Select' or requestKind='DB
Update' then DabtabaseTable.allInstances()->collect(name) else
RemoteObject.allInstances()->collect(name) endif
```

defines that if the “requestKind” property is “DB Select” or “DB Update”, the “remoteObject” property of a Request entity can be any value from a set which is calculated by obtaining all the DatabaseTable entity names, otherwise, all the RemoteObject entity names accordingly.

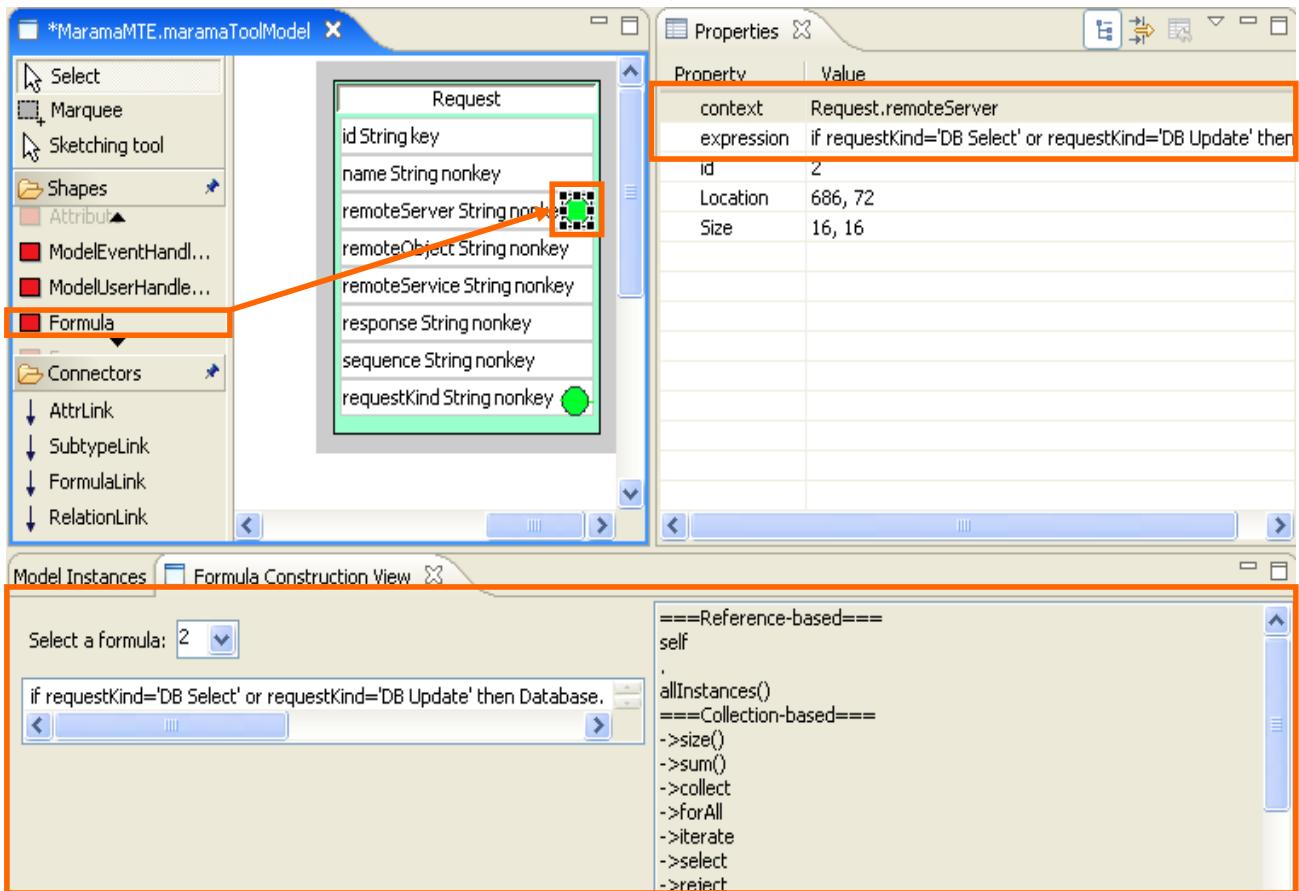


Figure 9.13. MaramaTatau behaviour specification

A formula can be placed on an attribute/entity/association by dragging and dropping a Formula node from the palette on the left of a metamodel diagram. A formula has two major properties, context and expression. The context records the attribute/entity/association to which the formula is attached. The expression stores the formula text, which can be directly entered using the Properties view or semi-automatically generated by clicking on metamodel diagram elements and functions listed in the Formula Construction View. As shown in Figure 9.13, a formula node is placed on the “remoteServer” attribute of the “Request” entity. The Formula Construction View can be loaded by selecting the Eclipse menu Window → Show View → Other ... → MaramaEditor → Formula Construction View.

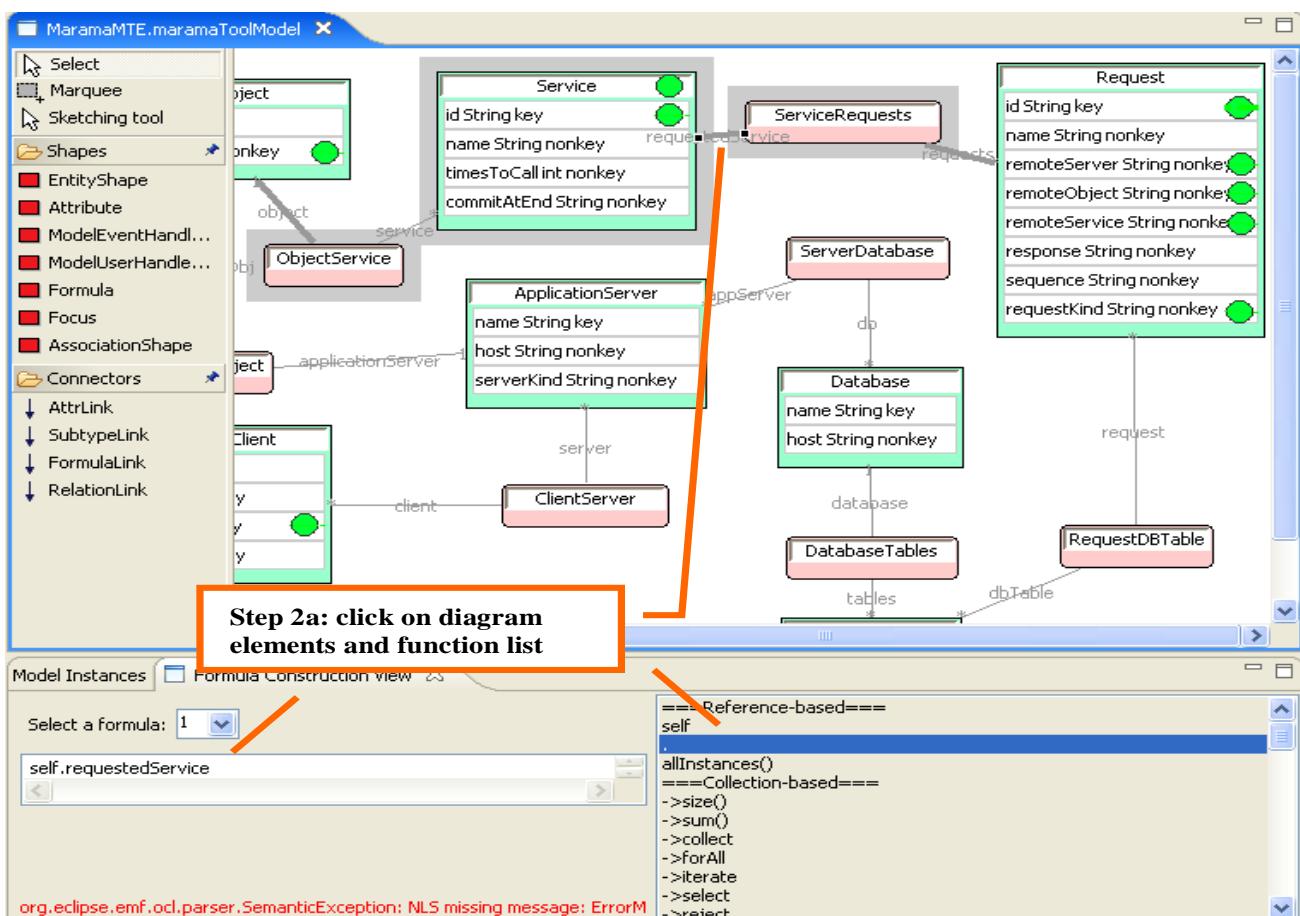
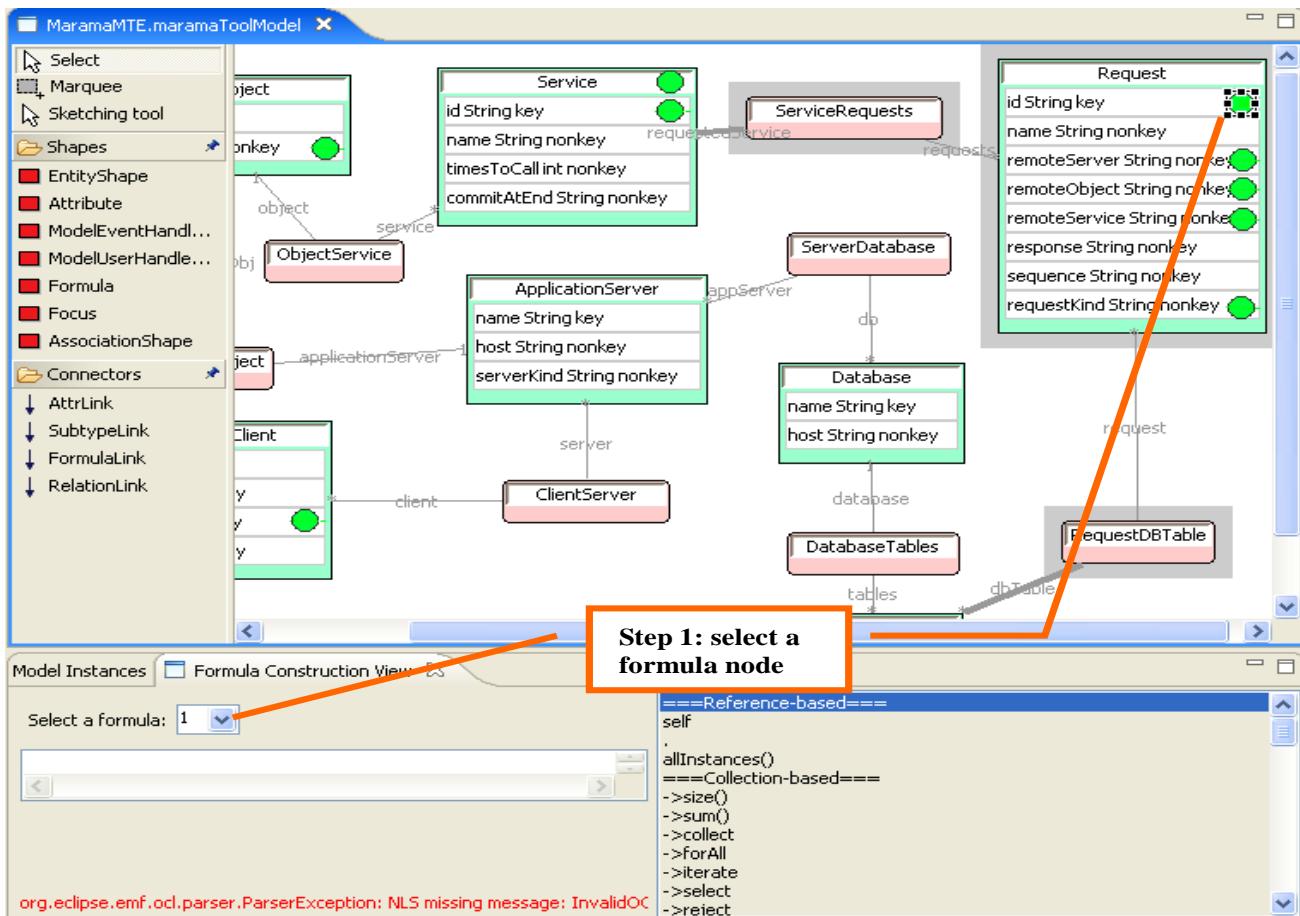
9.3.1.2 Visual Formula Specification and Semantic Highlighting

MaramaTatau formulae can be specified either visually by clicking on entity/association/attribute elements on a metamodel diagram or shape/connector elements on a view type diagram joined by functions from the given function list, or textually by directly entering formula text to the expression property of a formula node.

When a formula is visually specified, the diagram generates visual semantic highlights, where relevant entities/associations/attributes/association-ends are highlighted at each user's focus point in specifying the formula. This dynamic support guides the user towards a semantically correct specification, as shown in Figure 9.14. In Step 1, the user selects a formula node from either the diagram or the Formula Construction View. The semantic highlights at this point of formula selection include the Request entity type, the ServiceRequests association type and the requestedService association end role that implies the Service entity type. Clicking on an unhighlighted diagram element may indicate the invalidity of the formula specification. In Step 2a, the user selects “self” and “.” from the function list, and then clicks on the requestedService association end role on the diagram which triggers the Service entity type with its linked association and association end role elements to be highlighted at the selection point. In Step 2b, the user continues clicking on the “.” reference from the function list and then the “name” attribute of the Service entity type. A formula link is automatically generated pointing from the formula context (i.e. the “id” attribute of the Request entity type) to the attribute of its dependency (i.e. the “name” attribute of the Service entity type). After a sequence of clicks on the diagram elements and functions from the function list, in Step 2c, the user finishes constructing the formula.

The formula is syntactically checked in the MaramaTatau environment when it is being constructed, with a visual indication of any syntax error in the Formula Construction View until the formula is specified as being error free. If a formula is problematic, i.e. either syntax or semantically incorrect, the formula node is turned to red (as shown in Figure 9.15). Correcting the formula specification will turn the formula node into the normal mode. When the user has finished constructing a formula, i.e. when the user saves the diagram, the formula is compiled.

We are planning, in future work, to implement mutual synchronisation for the visual and textual specifications, to allow synchronised visual and textual editing, in other words, updating the formula via textual input will create the same visual highlighting effects in the diagram.



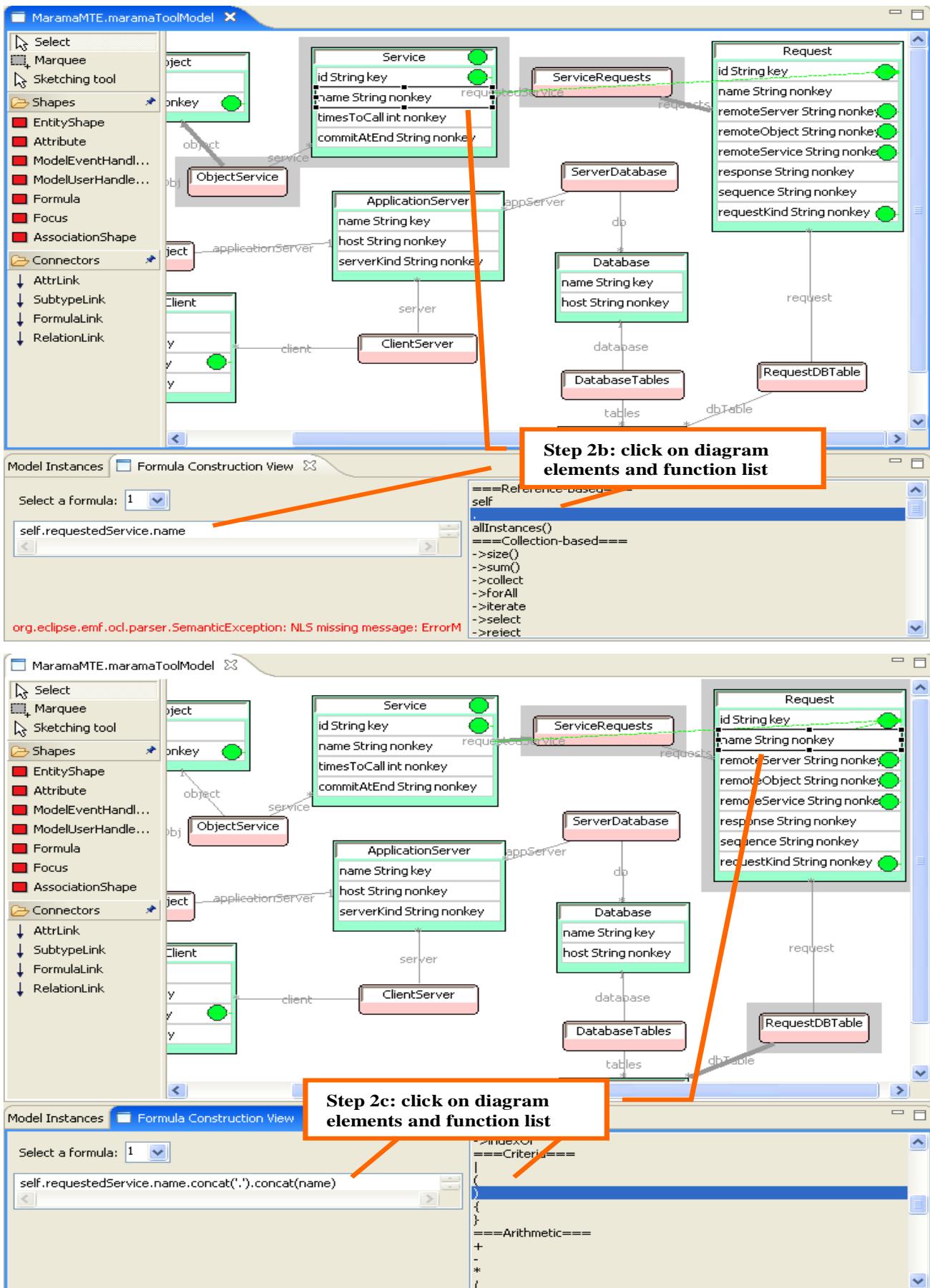


Figure 9.14. Visual formula specification via clicks and highlights

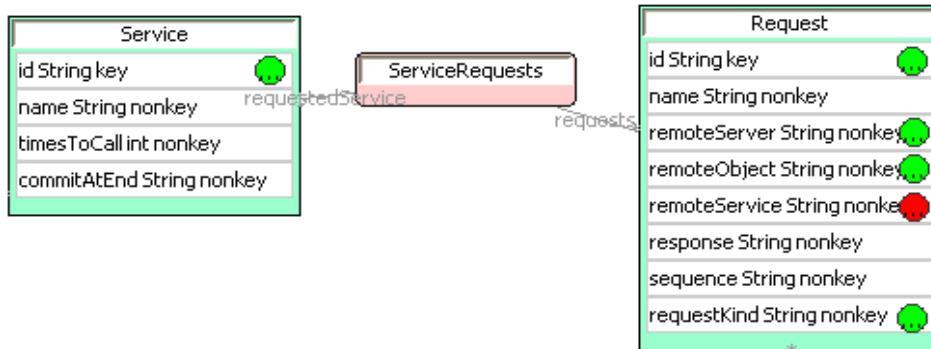


Figure 9.15. Indication of erroneous formula compilation

9.3.1.3 Extended Formulae with User-defined Operations

Marama meta-tools support user-defined operations to be used in formulae. The user-defined operations can produce side-effects to user models. To define a custom operation, the user clicks on the “Add or remove a function” context menu from the function list of the Formula Construction View, and a mirrored Marama workspace editor pops up for the user to add custom operation code, as shown in Figure 9.16. All Marama APIs are accessible to be used in the implementation of a custom operation.

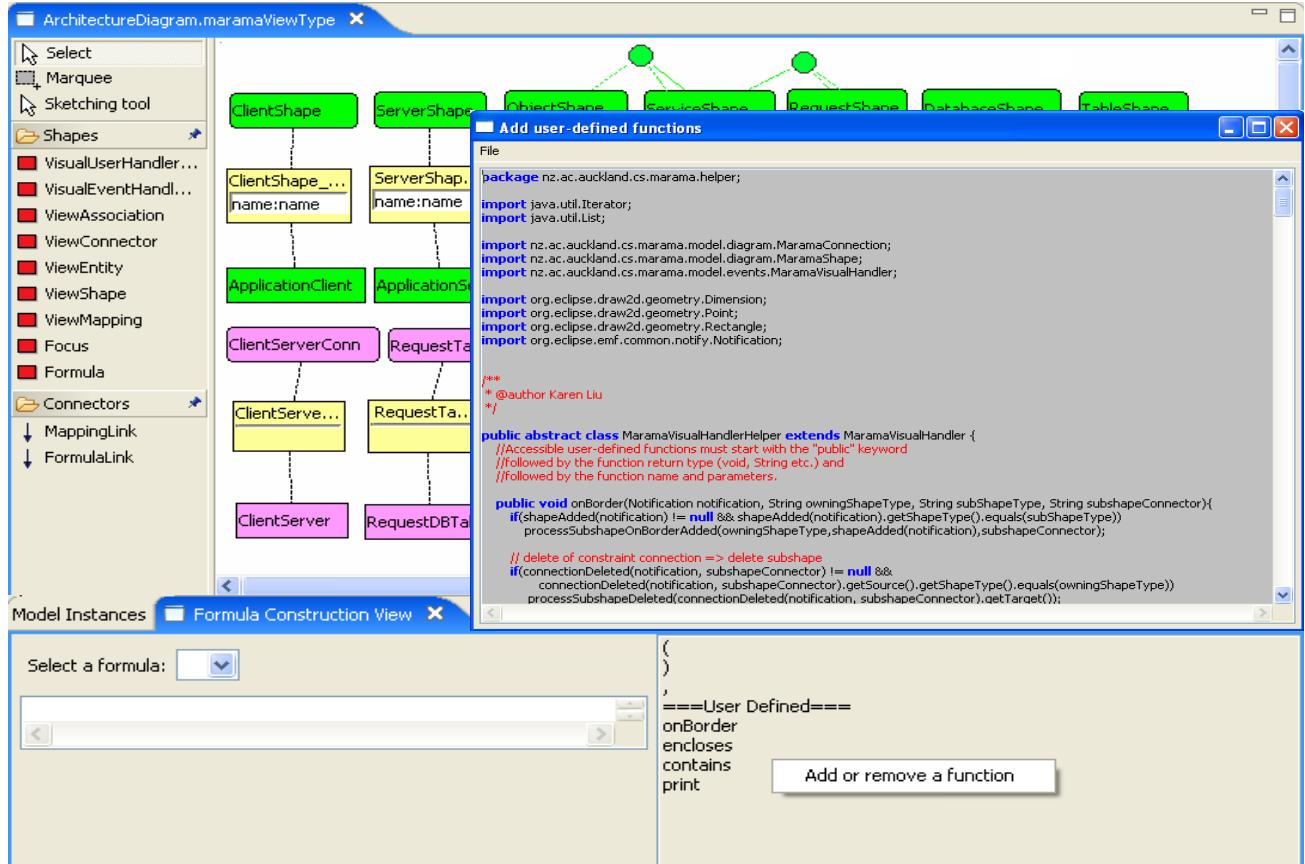


Figure 9.16. Adding user functions

User-defined operations must conform to the following rules:

- A Notification parameter must be defined as the first parameter of an operation in order to facilitate event notifications and event-condition-action responses.
- The rest of the parameters must be of String type to represent a shape type, connector type, entity type or association type.

There are currently three predefined operations in the library that are frequently reused in view types to define diagram elements layout. They are:

Method name	Parameters	Side effect
onBorder	<ul style="list-style-type: none"> • Notification notification • String owningShapeType • String subShapeType • String subshapeConnector 	Enforce an on-border layout constraint for a pair of shapes, i.e. a sub-shape is attached to the border of an owning shape. Moving the owning shape causes the sub-shapes to move together.
enclosed	<ul style="list-style-type: none"> • Notification notification • String owningShapeType • String subShapeType • String subshapeConnector 	Enforce an enclosure layout constraint for a pair of shapes, i.e. a sub-shape is enclosed inside an owning shape. Moving the owning shape causes the sub-shapes to move together, but the enclosed shape may be moved within the parent shape.
contains	<ul style="list-style-type: none"> • Notification notification • String owningShapeType • String subShapeType • String subshapeConnector 	Enforce a containment layout constraint for a pair of shapes, i.e. a sub-shape is contained and vertically aligned with other sub-shapes inside an owning shape. Moving the owning shape causes the sub-shapes to move together.

The user can define custom operations to be added to the library in a similar style. Upon completion of implementing an operation, the list of all available user-defined operations is updated; the user can then use the new operation in a formula composition.

The list of user-defined operations presented in the Formula Construction View is specialised for view type specifications. A formula for the view type is added and specified in a similar way to the metamodel formulae, i.e. via drags and drops, and clicks on diagram elements and functions, but instead of being attached to a particular diagram element, it can be located anywhere on the diagram.

As shown in Figure 9.17, the example formula, “contains (ServiceShape, RequestShape, RequestConn)”, has been constructed using the user-defined “contains” operation. Dependency links are automatically generated as a consequence of user’s click actions to compose a formula. But notice that the actual arguments provided to the “contains” operation call exclude the Notification object and only include the remaining matching arguments.

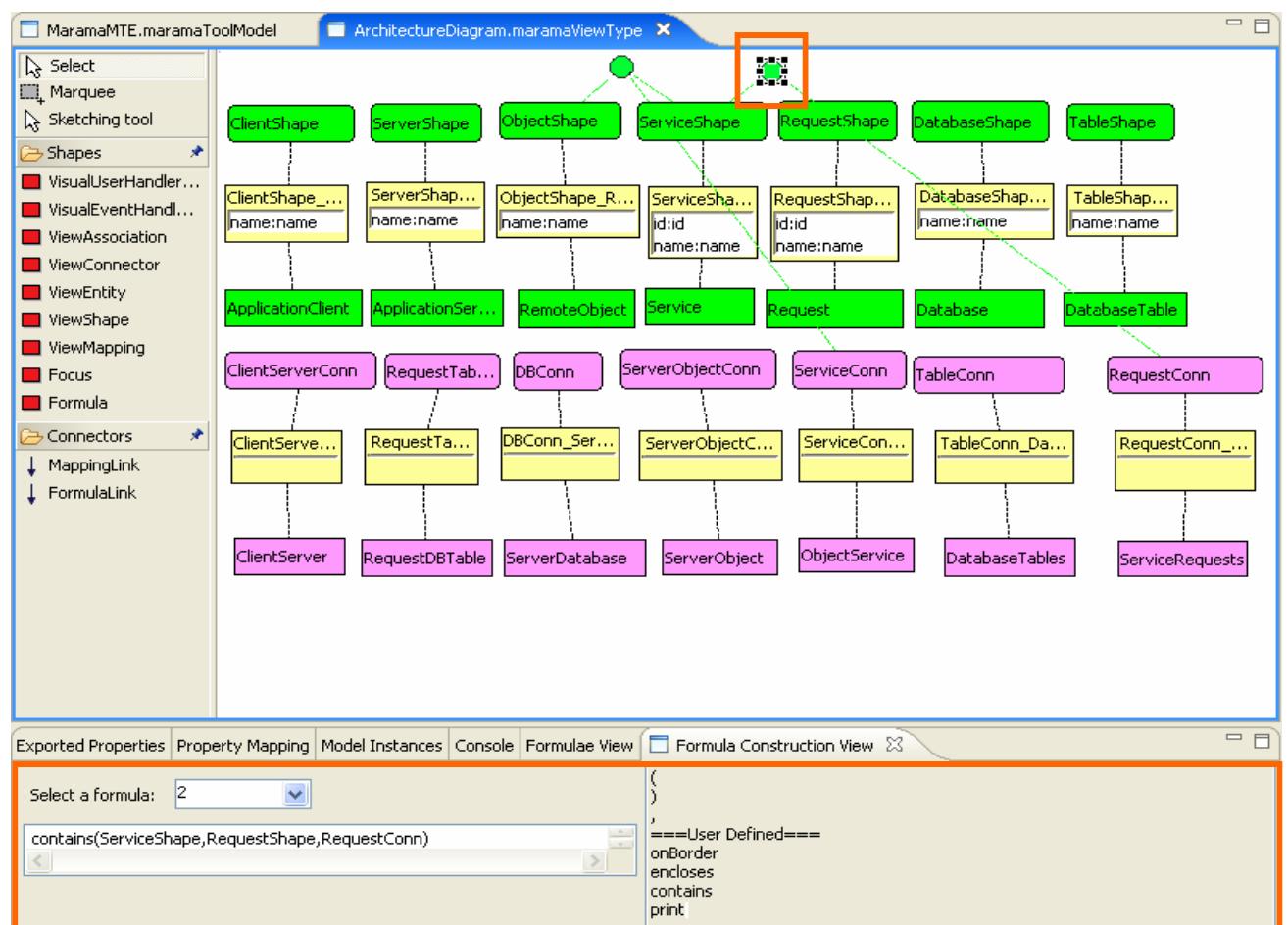


Figure 9.17. Defining view type formulas

The example formula in Figure 9.17 defines that a set of RequestShapes can be contained and vertically aligned in a ServiceShape. Moving the ServiceShape means that all the contained RequestShapes move together with it. This formula is translated and generated to a visual event handler as:

```

package nz.ac.auckland.cs.marama.userdirectory.tools.MaramaMTE.handlers.visualhandlers.eventtriggeringhandlers;
import org.eclipse.emf.common.notify.Notification;
import nz.ac.auckland.cs.marama.helper.MaramaVisualHandlerHelper;
public class Formula2 extends MaramaVisualHandlerHelper {
    public void notifyChanged(Notification notification) {
        contains(notification, "ServiceShape", "RequestShape", "RequestConn");
    }
    public String getName() {
        return "Formula2";
    }
}

```

Figure 9.12 also demonstrates the runtime reflection of this formula, where the ServiceShape named “findFights” and the RequestShape named “selectFlights” are constrained by this containment layout.

All view type formulae are generated into view level event handlers in the same manner, which immediately take effect on the view instances at runtime.

9.3.2 ViTAbL-WS Event Propagations

Apart from the reusable built-in events from the Marama library, the user may want to define tool-specific events and specify responses to them. Neither Kaitiaki nor MaramaTatau is suitable for implementing this task, but ViTAbL-WS is effective for specifying user-defined action events and responses.

Visual event propagations in ViTAbL-WS notation can be defined using the Marama Event Propagation Definer. Figure 8.8. shows user-defined events and their notifications among various Marama event handling toolies and structural components. This example defines that when an “ArchitectureDiagram” instance is deleted from a MaramaMTE model project, all the mapped view data are deleted from other views of the model project, and all the mapped model data are deleted from the model project, so that the views and the model are still synchronised with consistent data.

Three Marama structural components are involved in this ViTAbL-WS specification, they are:

1. diagram – the deleted diagram of a model project

2. views – all the multiple views of the model project
3. modelProject – the model project instance created using the MaramaMTE tool

A condition is added to this ViTABaL-WS process initially. Only an “ArchitectureDiagram” instance is concerned. The process goes to the end stage if the diagram being deleted is not of the “ArchitectureDiagram” type. The “processDiagramData” toolie generates a “diagramDeleted” event to be propagated to the “deleteMappedViewData” toolie and the “deleteMappedModelData” toolie, which define the event handling responses. A further “viewUpdated” event is propagated from the “deleteMappedViewData” toolie to the “views” data structure, and a “modelUpdated” event is propagated from the “deleteMappedModelData” toolie to the “modelProject” data structure. The toolies’ responses generate side-effects on the shared data structures. The “views” and “modelProject” data structures are “synchronised” with each other via the propagation of the “synchronised” action event.

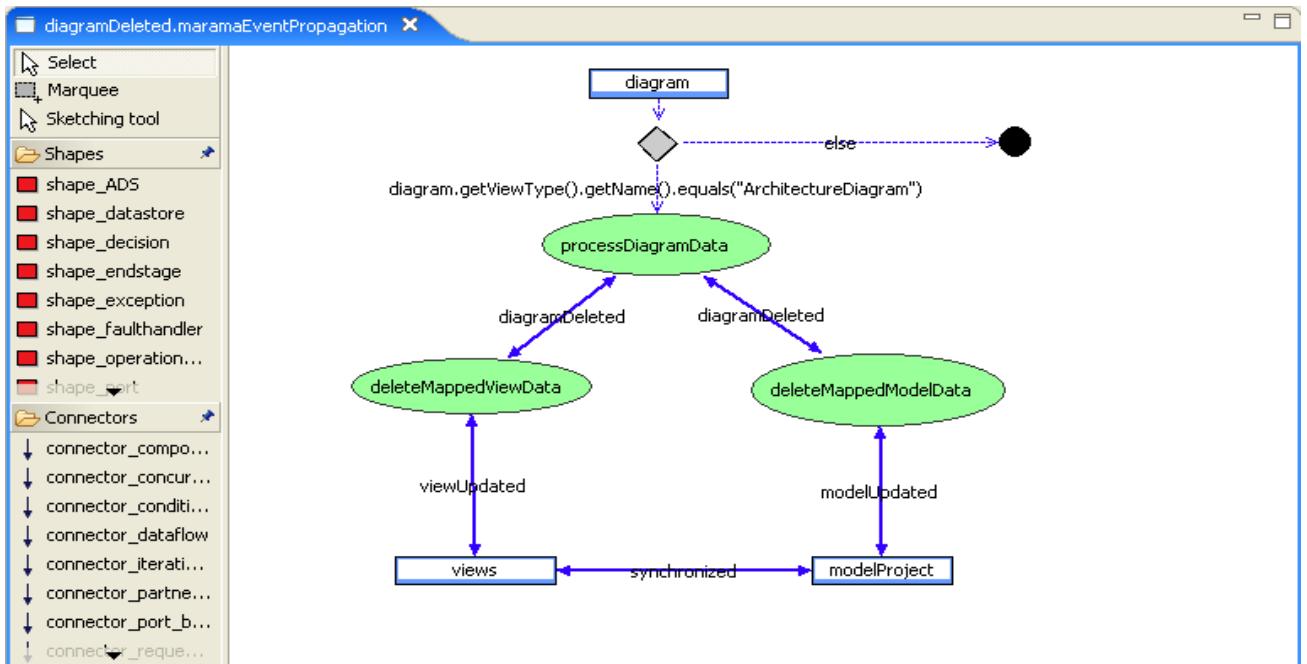


Figure 9.18. A ViTABaL-WS event propagation view

Consistent with the user interfaces of MaramaTatau, the palette tools in the Marama Event Propagation Definer can be used to create instances of the ViTABaL-WS building blocks. The visual specification is compiled into the following Java code for execution.

```

package nz.ac.auckland.cs.marama.userdirectory.tools.MaramaMTE.handlers.visualhandlers.eventtriggeringhandlers;
import org.eclipse.emf.common.notify.Notification;
import nz.ac.auckland.cs.marama.helper.MaramaVisualHandlerHelper;
import nz.ac.auckland.cs.marama.model.diagram.MaramaDiagram;

public class diagramDeleted extends MaramaVisualHandlerHelper {
    public void notifyChanged(Notification notification) {
        setEnabled(false);
        MaramaDiagram diagram = getDiagram();
        if (diagram.getViewType().getName().equals("ArchitectureDiagram")) {
            processDiagramData(new CustomEvent("diagramDeleted"),
                deleteMappedViewData(new CustomEvent("viewUpdated"), views));
            processDiagramData(new CustomEvent("diagramDeleted"),
                deleteMappedModelData(new CustomEvent("modelUpdated"), modelProject));
            new CustomEvent("synchronized", views, modelProject);
        }
        setEnabled(true);
    }
    public String getName() {
        return "diagramDeleted";
    }
}

```

9.3.3 Kaitiaki Visual Event Handlers

There are usually many common activities that are reused when defining visual event handlers, such as querying diagram state, filtering diagram elements and changing the state of an element or a list of elements. Kaitiaki features a set of such reusable modules (building blocks) and provides the ability to compose event handlers using a domain-specific dataflow metaphor – the Event-Query-Filter-Action metaphor. Kaitiaki is more suitable than MaramaTatau and ViTABaL-WS to define visual event handlers as by using these existing building blocks and the dataflow based Event-Query-Filter-Action metaphor, specifications are easier to develop and understand.

Visual event handlers in Kaitiaki notation can be defined using the Marama Visual Event Handler Definer, to handle either Marama built-in events or user-defined action events specified in ViTABaL-WS views. The previously presented Figure 9.12 also demonstrates the runtime execution

effect of a Kaitiaki event handler for aligning diagram shapes, where a “TableShape” is added and then aligned with the existing “TableShapes” that are queried from the diagram.

Figure 9.19 illustrates how this Kaitiaki event handler is specified. The handler responds to a “shapeAdded” event; it filters out the “TableShape”, and then uses the “alignV” action building block to align the newly added “TableShape” with the existing ones that are queried from the diagram, via the “getDiagram” and “getDiagramShapes” query building blocks, followed by the “shapesType” filter building block to filter out the “TableShape” shapes. A visual component is added by dragging and dropping a Kaitiaki building block type (e.g. shape_action, shape_filter etc.) from the palette tool, and the name of the component can be selected from the drop down list of available library building blocks of that type. A domain specific shape (an end user tool icon) can be added by dragging and dropping a shape_domainshape tool from the palette, and the shape’s appearance is dynamically updated when the user selects a shape type from the drop down list of previously defined shapes. Domain shapes mitigate the abstract Kaitiaki specification and make the visual language easier to understand.

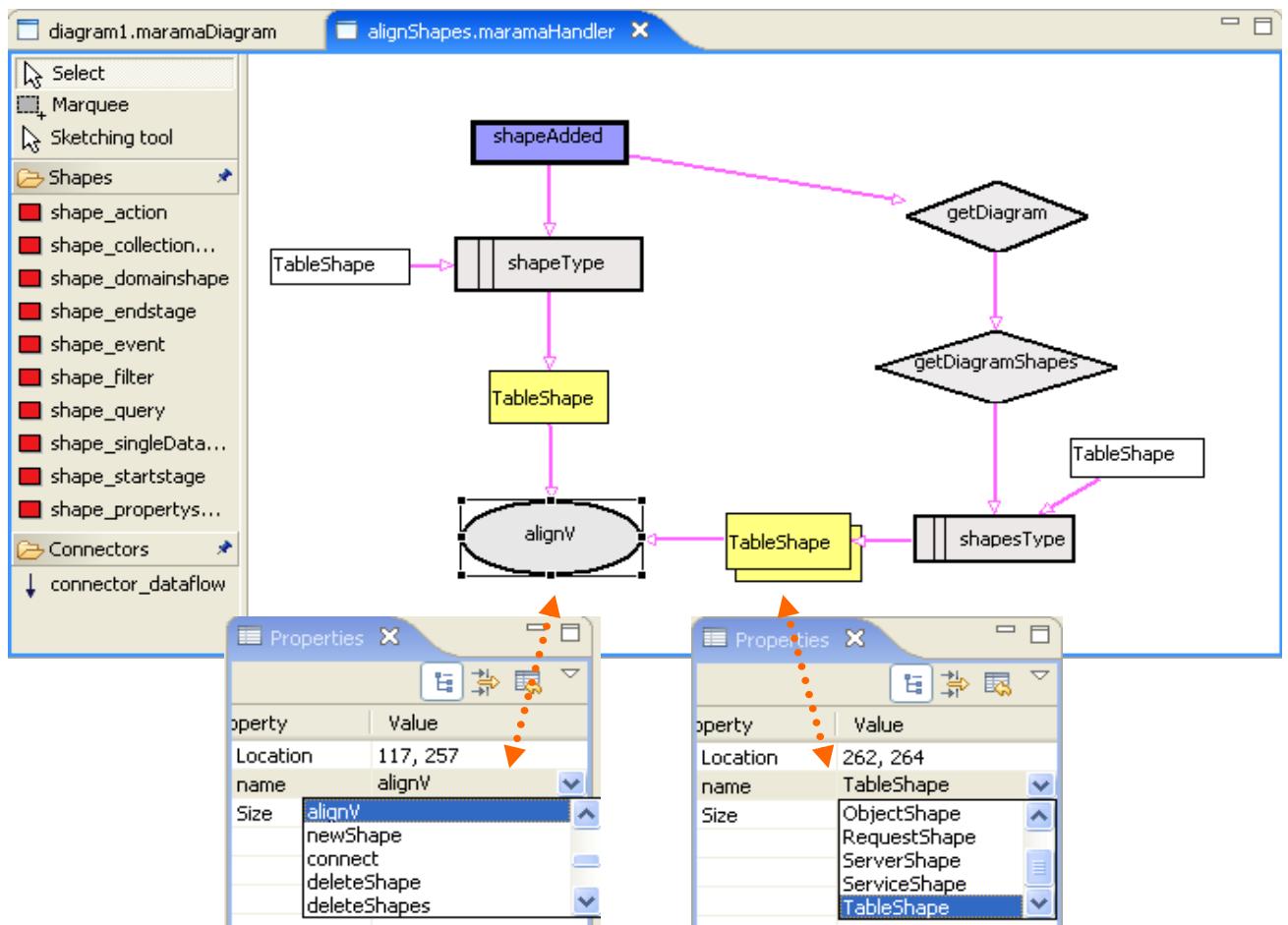


Figure 9.19. Kaitiaki event handler specification.

The Kaitiaki visual event handler is compiled into the following Java code for execution.

```
package nz.ac.auckland.cs.marama.userdirectory.tools.MaramaMTE.handlers.visualhandlers.eventtriggeringhandlers;

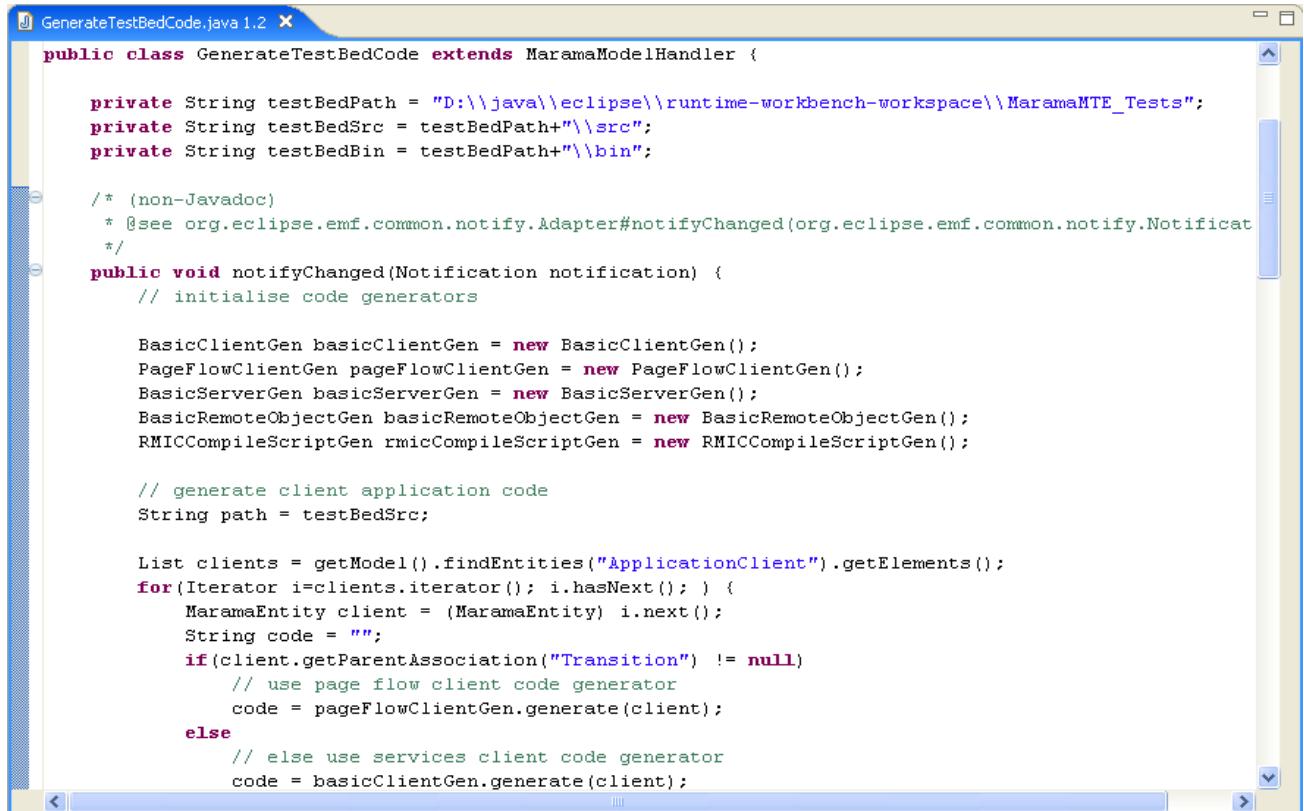
import org.eclipse.emf.common.notify.Notification;
import nz.ac.auckland.cs.marama.helper.MaramaVisualHandlerHelper;
import nz.ac.auckland.cs.marama.helper.QueryLibrary;
import nz.ac.auckland.cs.marama.helper.FilterLibrary;
import nz.ac.auckland.cs.marama.helper.ActionLibrary;

public class alignShapes extends MaramaVisualHandlerHelper {
    public void notifyChanged(Notification notification) {
        setEnabled(false);
        if (shapeAdded(notification) !=null) {
            ActionLibrary.alignV(
                FilterLibrary.shapeType(shapeAdded(notification)),
                new String("TableShape")),
            FilterLibrary.shapesType(
                QueryLibrary.getDiagramShapes
                    (QueryLibrary.getDiagram(shapeAdded(notification))),
                new String("TableShape")));
        }
        setEnabled(true);
    }
    public String getName() {
        return "alignShapes";
    }
}
```

9.3.4 Escaping to Code

Although MaramaTatau, ViTABaL-WS and Kaitiaki facilitate easy and effective visual event handling specification by their modelling capabilities and library support, there are still domain-specific event-based behaviours that can not be specified fully using them. For example, MaramaMTE generates performance test-beds, and the complex domain-specific code that needs to be generated is not able to be specified by MaramaTatau, ViTABaL-WS or Kaitiaki. The user needs to escape to code to implement such freely customisable event handler behaviours. Therefore, the previous Marama support for event handling based on custom code writing is still necessary. The

user needs to code within the context of Marama EMF and its implementation. Figure 9.20 shows a Marama code editor (a.k.a. the Eclipse Java editor) with Java code implementing the performance test bed code generation.



```

Generate TestBedCode.java 1.2
public class GenerateTestBedCode extends MaramaModelHandler {

    private String testBedPath = "D:\\java\\eclipse\\runtime-workbench-workspace\\MaramaMTE_Tests";
    private String testBedSrc = testBedPath+"\\src";
    private String testBedBin = testBedPath+"\\bin";

    /* (non-Javadoc)
     * @see org.eclipse.emf.common.notify.Adapter#notifyChanged(org.eclipse.emf.common.notify.Notification)
     */
    public void notifyChanged(Notification notification) {
        // initialise code generators

        BasicClientGen basicClientGen = new BasicClientGen();
        PageFlowClientGen pageFlowClientGen = new PageFlowClientGen();
        BasicServerGen basicServerGen = new BasicServerGen();
        BasicRemoteObjectGen basicRemoteObjectGen = new BasicRemoteObjectGen();
        RMICCompileScriptGen rmicCompileScriptGen = new RMICCompileScriptGen();

        // generate client application code
        String path = testBedSrc;

        List clients = getModel().findEntities("ApplicationClient").getElements();
        for(Iterator i=clients.iterator(); i.hasNext(); ) {
            MaramaEntity client = (MaramaEntity) i.next();
            String code = "";
            if(client.getParentAssociation("Transition") != null)
                // use page flow client code generator
                code = pageFlowClientGen.generate(client);
            else
                // else use services client code generator
                code = basicClientGen.generate(client);
        }
    }
}

```

Figure 9.20. The custom code writing approach in Marama to define event flows.

Marama incorporates event notifications and event handlers. Model handlers are used to specify reactions to model events (e.g. entity/association changes), whereas visual handlers are used to specify reactions to visual view-based events (e.g. shape/connector changes). Both model and visual handlers are sub-typed further by specialising them to event triggering and user triggering (via user menu-click action) natures. In summary, Marama handlers are categorised into model handlers and visual handlers and their sub-types as seen in the tree hierarchy in Figure 9.21 below.

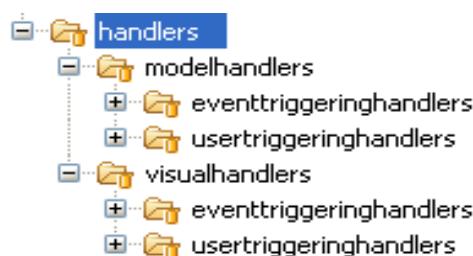


Figure 9.21. Marama handler hierarchy

A model event/user handler is defined as a subclass of MaramaModelHandler (in package nz.ac.auckland.cs.marama.model.events) and saved as a Java file in the corresponding directory (i.e. eventtriggeringhandler/usrtriggeringhandler directory).

```
public class AModelEventHandler extends MaramaModelHandler {
    public void notifyChanged(Notification notification) {
        /** Reaction code goes here. */
    }
    public String getName() {
        return "A model event handler"; // handler name/description
    }
}
```

A visual event/user handler is defined in the same way but extends MaramaVisualHandler (in package nz.ac.auckland.cs.marama.model.events).

```
public class AVisualEventHandler extends MaramaVisualHandler {
    public void notifyChanged(Notification notification) {
        /** Reaction code goes here. */
    }
    public String getName() {
        return "A visual event handler"; //handler name/description
    }
}
```

Events are notified by the event generators and propagated to all the event handlers at runtime. The method “public void notifyChanged(Notification notification)” contained in the Marama handler code receives all the event notifications and implements the reaction behaviour for a filtered event or list of events. The user-defined handler code being called from the “notifyChanged” method can include events of interests, queries of model/diagram states, filters on a collection of data, and state changing actions on Marama objects. Figure 9.20 contains a model user handler (a model handler reacting to a user’s menu-click action). So the “notifyChanged” handler code is fired whenever the user right clicks on a context menu item named “Generate Test Bed Code” from a view of a MaramaMTE model instance, and as it is being implemented in the handler method, both the server side test bed code and the client side simulation code are generated based on the runtime model information.

A model event/user handler must be first defined in a Marama Metamodel Definer view by dragging and dropping a ModelEventHandler/ModelUserHandler icon from the palette to the metamodel diagram as seen in Figure 9.22, and then coded as a handler class and saved in the corresponding handlers' folder of the tool's source code repository. The name of the diagram handler icon and that of the Java class must be consistent in order to get the handler registered and fired correctly. Visual event/user handlers are all defined in a similar way but in a Marama View Type Definer view.

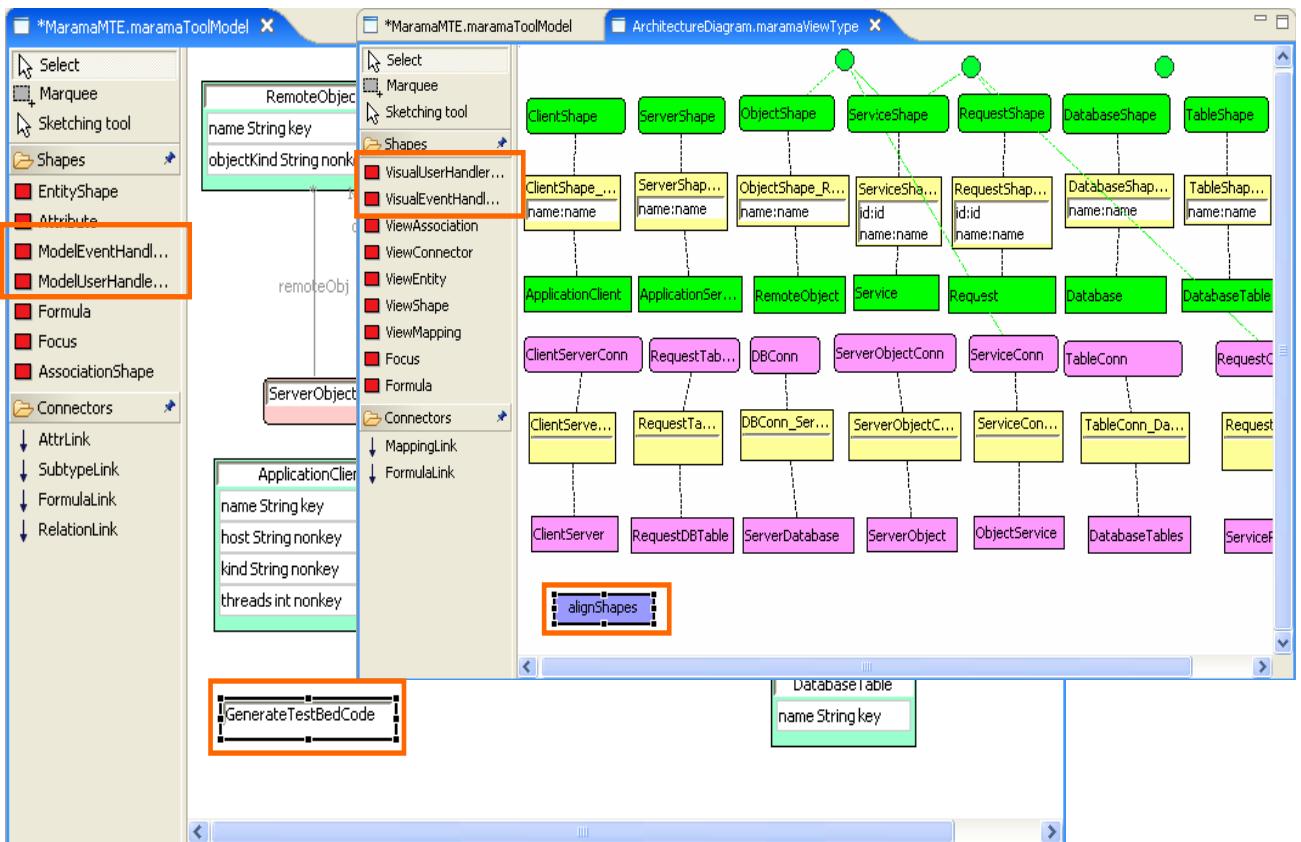


Figure 9.22. Registering a handler to the metamodel or a view type in Marama meta-tools.

9.4 Prototype Implementation

Marama was implemented mainly using the Eclipse EMF plug-in for modelling component class generation and the Eclipse GEF plug-in for diagram component rendering. Eclipse UI, SWT and JFACE packages were also used to provide Marama with menus, toolbars, and a set of Eclipse views including Properties view, Outline view, Problems view, etc., and some Eclipse views were extended with Marama domain-specific features (such as the various tool creation wizards, Model instances view, Formula Construction view and Visual Debugger) to provide Marama with a rich set of user interface components that are consistent with the Eclipse environment. Marama meta-tools were implemented inside Marama using the same plug-in libraries.

We used Pounamu to specify the meta-metamodel for Marama meta-tools, i.e. the modelling elements of the Metamodel Definer, Shape Designer, View Type Definer, Kaitiaki Visual Event Handler Definer and ViTABaL-WS Event Propagation Definer. These components are realised in Marama editors using consistent graphic modelling interfaces and packaged to synthesise the set of Marama meta-tools.

The prototype implementation was heavily based on the Model-View-Controller pattern. Data representation of entities, associations, shapes, connectors, views, properties and events together with interactive controller command objects were implemented as an Eclipse plug-in named “MaramaModel”; Eclipse-based graphical representations of model instances were implemented as a model dependent plug-in named “MaramaDiagram”. Reusable library operations were defined in another model dependent plug-in named “MaramaBasicHandlerLibrary”. Marama meta-tools used the existing Marama APIs, and as an outcome they extended the Marama class library with the addition of event-based behaviour abstractions.

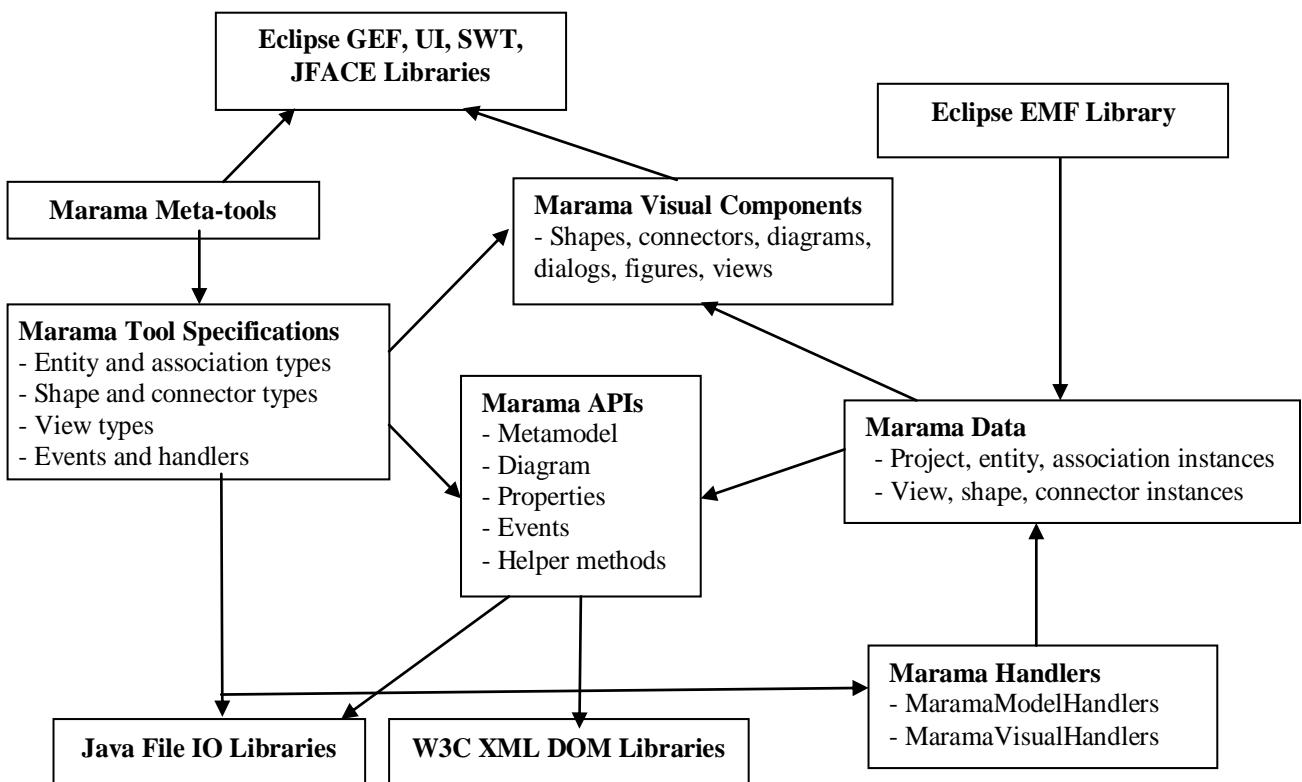


Figure 9.23. The component structure of Marama meta-tools.

Figure 9.23 illustrates the component structure with which Marama meta-tools are implemented. Based on Pounamu’s implementation, the Java API for XML and W3C Document Object Model (DOM) framework were used in Marama meta-tools for representing tool specification data as in-

memory XML data structures. Marama meta-tool instances map to Marama EMF diagram instances with shapes and connectors as property sources. Modelling tools specified using Marama meta-tools are persisted as XML files that are automatically generated on a save of the modelling view instances. These XML files are read by model projects using DOM parsing. The Java file IO APIs are used for tool specification data storage and retrieval to/from the XML format. The XML-based repository enables easy exchange and integration with other tools.

Various code generators have been implemented to analyse diagram and model elements and translate visual programs into Marama Java programs with appropriate APIs and reusable user library method calls. The code generation processes are hidden from the user in order to hide their complexity. From the user's perspective, the visual programs are executed as efficiently as textual Java programs, with the advantages of easy and efficient specification and visualisation. The implemented canonical event handling framework allows inter-communication of fine-grained modules to be used crossing over different event handler specifications.

Both ViTABaL-WS and Kaitiaki should perform static consistency checks to ensure a specification is semantically correct before generating code for execution but this has yet to be implemented. The checks have however been implemented for MaramaTatau. Consistency checks should include checking event propagations are handled by appropriate event responses in receiving components, and ensuring event arguments are passed in the correct type and order, as described in the EASY framework (Grundy et al, 1996).

9.5 Summary

We have developed the Marama meta-tools as a vehicle for exploring integrated event handling specification. We initially constructed the structural modelling facilities in Marama meta-tools to support modelling with generated domain-specific visual modelling environments. The structural modelling views include a Metamodel Definer that allows visual definition of the tool metamodel (entity types and association types), a Shape Designer that allows visual construction of shapes and connectors, and a View Type Definer that allows visual composition of view elements and visual-to-model mappings. We then extended this metamodeling environment by adding event-based behaviour modelling facilities using a set of unified techniques, with the integration of MaramaTatau, ViTABaL-WS and Kaitiaki. MaramaTatau allows visual construction of formulae in a spreadsheet-like style to specify model and view value dependencies and constraints. ViTABaL-WS

allows event-based architecture to be defined for the modelling tool, enabling specification of event propagations and responses. Kaitiaki allows composition of event handlers using a set of visual building blocks and hiding code complexity from the user. The reusable building blocks created by each of the three approaches can be used in an interleaved way collaboratively in the Marama meta-tools environment, since they are generalised to a canonical event model and unified to be used flexibly by the integrated environment.

Chapter 10 - Evaluation of the Generalised Framework

Following our initial prototype development, we have conducted both developer-based and end user-based evaluations of the Marama meta-tools to test their usability and effectiveness for specifying event-based system integration with the aim of identifying potential problems. The evaluation results have been sufficiently positive for us to release the Marama meta-tools as a publicly accessible toolset following a number of enhancements to address tool stability.

10.1 Introduction

It is not a straightforward task to evaluate a substantial environment/toolset such as the Marama meta-tools, as it involves multiple points of views of tool developer, end users of developed tool, usability, utility, etc. (Zhu et al, 2007). Most formal usability evaluation approaches are limited to understanding the effect of one or two variables (Dillon, 2001; Hartson et al, 2003). Controlling for variability is an almost impossible undertaking when assessing the usability of a large environment. Formal evaluation for this type of system is hard. This means we have had to adopt a variety of less formal, but overlapping approaches to obtain usability and efficacy data.

We have evaluated Marama meta-tools at several levels and through a variety of mechanisms in a similar way that evaluations of the Pounamu metatool were conducted (Zhu et al, 2007). These include:

- We, the designers, conducted a cognitive dimensions (Green and Petre, 1996) evaluation focusing on the event handling specifications. The use of three distinct metaphors together in the system has increased the initial learning curve of the Marama meta-tools, but provided effective event handler specifications by addressing identified concerns and allowing tool designers to escape from writing conventional code.
- We, the designers, conducted an evaluation of the Marama meta-tools against the requirements established in the research background and motivation in Chapter 3 and 4, and elaborated in the meta-tools design in Chapter 8.

- A large number of graduate-level student end users (novice short-term research task-oriented users) were involved in an extensive usability study. In the experiments, 122 participants constructed a domain specific visual language tool of their choice, but with a minimum set of tool features that had to be included in their tool, and were then surveyed. The participants were allowed to work either individually, in pairs, or in a team of 3-5. The aim of the experiment was to provide a substantial, realistic tool development situation and obtain qualitative information on user perceptions of the toolset and task completion data (whether the minimum feature set was in fact implemented). The experiments evaluated whether end users found the Marama meta-tools easy and effective for generating their chosen domain specific visual language tool. We aimed to use the end users' feedback to improve the Marama meta-tools, and significant enhancement was undertaken after the experiments.
- A smaller number of developers (experienced long-term research goal-oriented users) in our research team, who used Marama meta-tools to develop more substantial applications, provided qualitative feedback in the form of experience reports. The advanced applications being developed or integrated with Marama meta-tools include a generic mapping tool, a health care visual modelling environment, a business process integration tool, an architecture modelling/mapping tool, and a design critiquing system. These qualitative feedback reports were used to assess whether our perceptions of the Marama meta-tools needed to be altered for more experienced user groups, and whether additional requirements were needed (e.g. more complex back end integration requirements).

We describe the evaluation criteria that we used and the results that we obtained in Section 2. Evaluations need to be conducted iteratively after every progress has been made to improve Marama meta-tools. We describe our continuous evaluation plan in Section 3, and then summarise this chapter.

10.2 Evaluation Techniques

Marama meta-tools have been evaluated based on the following main techniques: the Cognitive Dimensions, the previously established requirements, and the usability from both novice and experienced end users' point of view. Table 10.1 shows the evaluation methods being used versus what information we intended them to provide to us.

Evaluation method	Expected information to obtain
Cognitive dimensions	Cognitive dimensions allow us to understand usability tradeoffs and hence where mitigations need to be placed.
Evaluation against the requirements	The requirements established in the research can be used as the benchmark for evaluating the functional utility of the Marama meta-tools.
Large end user survey	The large study allows us to understand qualitative end user usability perceptions and quantitative task completion data.
Small experienced user group study	The small study allows us to continuously collect qualitative experienced user perceptions and additional requirements as needed.

Table 10.1. The evaluation methods adopted by the Marama meta-tools.

10.2.1 Cognitive Dimensions

We have conducted a Cognitive Dimensions investigation for each of ViTABaL-WS, Kaitiaki and MaramaTatau individually as described previously in Chapter 5, 6 and 7 respectively. Each of the three languages and environments feature easy and effective specifications with some dimensional tradeoffs where we have placed effort to provide mitigations (e.g. to minimise hidden dependency issues in MaramaTatau). In this section, we describe a cognitive walkthrough from the end users' perspective for the Marama meta-tools focusing on the event handling integration. Using Cognitive Dimensions, some points in particular highlight Marama meta-tools suitability, while others are indicative of negative tradeoffs that have been made.

The Marama meta-tools provide users with facilities to define their own domain-specific visual language environment. Both the static structure and dynamic behaviours can be flexibly defined using either the existing component library or their customised extensions, but their use requires an understanding of several moderately complex metaphoric abstractions and the way they can interact. Thus Marama meta-tools have a relatively high *abstraction gradient*. The Marama meta-tools allow users to mix abstractions to specify a visual design system. High-level abstractions (via visual building blocks and their compositions) are used to model both static and dynamic aspects of a system, while low-level abstractions (via property settings and escape to code) are used to model detailed structural constraints and visualise runtime debugging information. This seamless mixture provides flexibility for Marama meta-tools to be used by both novice and experienced developers. The primitive elements in Marama meta-tools represent a broad range of components from the Tool Abstraction, Event-Query-Filter-Action, and Spreadsheet paradigms, and communication relationships between them. The visual metaphors and visual constructs used increase comprehensibility compared to the textual code-based behaviour specifications. The Marama meta-

tools offer low *diffuseness* in that they provide a terse extended Entity-Relationship language to specify metamodels, a terse set of notations for shapes and view type constructions, however we have chosen a *verbose* multi-paradigm formalism in the integrated event handling specification.

The abstractions used in Marama meta-tools require some *hard mental operations* for novice users. The initial learning curve of Marama meta-tools is high for users who are not familiar with metamodelling approaches, but this is not our target end user group. Marama meta-tools' languages and visual metaphors need to be learned, and the learning process typically requires access to the provided documentation and exemplar tools. Once the Marama meta-tools approach is initially learned, users can benefit greatly from its support for fast and easy prototype generations. *Premature commitment* is required when using the Marama meta-tools. Design is supported in an interactive way with dynamic graphical visualisations provided by the Marama meta-tools. However, the user typically has to describe the structural parts before the behavioural; otherwise the user does not have metamodel components that can be referenced when doing the behavioural side. This is not a major issue though as it is a natural progression for software designers. The Marama meta-tools support rapid prototyping to prove both the structural and behavioural design concepts.

Marama meta-tools' structural and behavioural constraint model fits closely to conventional metamodelling concepts and the Model-View-Controller architectural pattern. *Closeness of mapping* is thus high for our target end users and especially for current users of Eclipse, as Marama meta-tools make use of a collection of Eclipse-based views to display integrated information. Close mappings of tool specifications with domain concepts are easily achieved, as the Marama meta-tools provide users with the flexibility to define custom domain-specific concepts and elements. However, the existing GUI elements are a little constraining in this regard, requiring additional types of component such as buttons, sound and video etc.

Consistency is well managed in the Marama meta-tools. Though the different view types support specification of a distinguished modelling aspect of a visual language environment, the interfaces provide a consistent look and feel. ViTABaL-WS, Kaitiaki and MaramaTatau each provide terse language syntax, as well as presenting close interconnectivity. While they are indeed based on a generalised common event handling model representation, users will not feel they are individually irrelevant, and can easily establish relationships between them. The visual notations used in the different behavioural modelling views are consistent subsets of a common representation: Rectangles represent data, Circles represent constraints, and Connections represent relationships.

Error-proneness has been reduced in comparison with conventional code writing in some areas; typically the code generation from visual specifications eliminates syntax errors and some semantic errors. Once Marama meta-tools' step-by-step metamodelling approach is understood and followed, it is not error-prone for structural modelling. Static checking of structural elements is performed before a runtime modelling environment is generated and indications of an incorrect structural specification are presented to the user. Visual behaviour specifications in ViTABaL-WS and Kaitiaki are potentially error-prone at this stage as they directly generate Java code without performing compile-time checking first. Implementation of static type checking for these metaphors is important future work.

The Marama meta-tools have minimal *hidden dependencies* in constructing formulae and constraints. The support for multiple views introduces hidden dependencies, especially when certain behaviour is specified using a set of mixed ViTABaL-WS, Kaitiaki and MaramaTatau elements. To mitigate this, we are exploring using a generated Dependency view to indicate source and target behaviour elements dependencies, allow cascading changes and provide easy navigation mechanisms between these interdependent elements.

Progressive evaluation is well supported. Marama meta-tools allow tool specifications to be evaluated at any stage. Partially completed specifications can be executed as well. View type extensions for both structural and behavioural aspects can be added into a Marama tool at any stage, and previously specified elements can be freely used or updated with new features.

Marama meta-tools specification is well structured, with every component performing a unique set of tasks (*Role-expressiveness*), e.g. the metamodel components define visual language semantics and constraints; the shape components define visual representations of metamodel elements; the view type components compose view elements and their dependencies on the model; the event propagation components define the event-based relationship between model or view elements; and the event handling components define dynamic interactive behaviour.

The Marama meta-tools make use of layout, colour, and domain specific tool icons as *secondary notations* to enhance the syntax and semantics being conveyed from both structural and behavioural specifications. The set of Marama editors used in Marama meta-tools provide easy modification via consistent graphical user interfaces. Changing an element in one view does not affect all other elements unless dependency is specified for consistency management. However, the usual *viscosity*

problems occur for all the diagram types in Marama meta-tools when diagrams need to be arranged to insert additional elements. Provision of automatic layout support could potentially alleviate this (at both model and metamodel levels). Providing such facilities is left for future work.

Marama is an Eclipse plug-in; this provides Marama with good mechanisms to support *visibility and juxtaposability*. Marama editors can be freely juxtaposed side-by-side to allow simultaneous visualisations of different views of concern. Marama meta-tools behavioural specification views can especially be juxtaposed with Marama modelling views for debugging purposes.

10.2.2 Evaluation against the Requirements

The set of requirements established in Chapter 8 were our benchmark for evaluating the functional utility of the Marama meta-tools. These requirements have all been met. In the following we describe how each of the requirements has been addressed in the design and implementation of the Marama meta-tools

The generalised Marama meta-tools framework incorporates compositional primitives as event handling building blocks and allows composition relationships between them. The framework contains reusable designs to allow users to initialise their system and specify customised event types, event generators, event receivers and event handling building blocks to enhance the extensibility and flexibility of the framework. The framework supports tool integration via a canonical data/event model extension and consistent user interfaces.

Graphical notations are offered in the style of the three metaphoric exemplars – ViTABaL-WS, Kaitiaki and MaramaTatau, to allow easy and effective event handling specifications and visualisations. Marama meta-tools allow specification of event generators, events, and event receivers using ViTABaL-WS, specification of view-level event handling behaviours using Kaitiaki, and specification of model-level and view-level structural constraints using MaramaTatau. The integration of the three languages enables Marama meta-tools to handle complex events in a straightforward way. Textual notations are also permitted so that users can escape to conventional code when specifying complex custom behaviours such as code generation. Both the visual languages and the textual languages can be documented thoroughly in the Marama meta-tools environment.

Multiple views of data, event and behaviour representations are kept consistent in both the model and user interface level to ensure the correctness of generated environments. Multiple views can be easily navigated from one to another.

The underlying Marama framework provides the support to realise Marama models together with views and dynamic behaviour when an instance of a tool specified using the Marama meta-tools is realised. In a runtime modelling environment, MaramaTatau formulae and ViTABaL-WS specified event propagations can be traced and Kaitiaki event handling results can be visualised based on a user interactive visual debugging model in a step-by-step fashion.

With regards to the quality of service, Marama meta-tools is a proof of concept toolset which provides good levels of abstraction from high-level conceptual design to low-level implementations. Visual languages exploiting easy-to-understand metaphors are used to simplify the behavioural specification tasks. Users can still escape to code when complex custom tasks need to be implemented. The integrated meta-tools environment provides design guidance support especially in validation of the specifications. The toolset has proof of concept stability issues needing to be addressed. The issues are suggested from the user data in the next section.

Marama has good scalability. The framework provides well structured extensibility via multiple views and by separating concerns of modelling structure from behaviour, as well as integrating them at runtime in a seamlessly unified manner. The framework is readily extensible with the addition of additional generic or domain specific building blocks.

Developing prototypes using Marama meta-tools takes considerably less time than implementing them using a programming language from scratch. The behavioural models generate Java code which is executed as efficient as code implementations.

10.2.3 Large End User Survey

In the user evaluation experiments, 122 participants, who were fourth year Computer Science or Software Engineering students, were asked to construct a Domain-Specific Visual Language (DSVL) tool of their own choice, but with at least a minimal set of required components (so that tools with a realistic level of complexity were designed and constructed) similar to the set of tasks defined in the Pounamu evaluation experiments (Zhu et al, 2007), including:

- At least three metamodel entity types and appropriate associations

- At least three different iconic shapes, possibly of differing complexity (of the shape image)
- At least two different shape connectors
- At least two different view types, i.e. that show different kinds of information within the view types
- A few simple formulae and/or event handlers managing things like diagram layout, editing constraints, model (entity) constraints, mock code generation, data import, etc.

Preparatory training lectures were provided on:

1. General DSVL design concepts, including Cognitive Dimensions
2. General introduction to meta-tools and metamodeling concepts
3. Specific introduction to Marama and the Marama meta-tools

Students were also provided with a demonstration-based tutorial and an online user manual. Three exemplar tools were provided to be used in Marama directly. They are the simple Whole-Part aggregate modeller presented in Chapter 7, a simple UML tool and the complex MaramaMTE tool presented in Chapter 7-9.

Participants were then given three weeks elapsed time (while they were working alongside other commitments) to complete the prototype development together with a survey report containing a set of open ended questions to qualitatively elicit strengths and weaknesses of the Marama meta-tools in constructing their desired DSVL tool. Their experiments and survey reports were collected and analysed. These participants had used many software tools, though few for metamodeling similar to Marama and Pounamu (Zhu et al, 2007). We believe their previous experience had little influence on their experiences and expectations.

Sixty-five tool instances were created (due to the fact that the participants worked either individually, in pairs, or in a team of 3-5), among which fifty were based on the existing application areas (having been explored using the Pounamu tool) from the provided references in the task description and other web references. These included software process modelling, software architecture design, aspect-oriented design, design pattern modelling, entity-relationship modelling, data mapping specification and statistics design tool. Fifteen tools were based on the participants' exploratory development of new DSVLs of their own design, including a wizard creation tool, a DHTML web development tool,

an online banking tool, an enterprise reporting tool, a family tree navigation tool, a restaurant management tool, and a university degree modelling tool.

Figure 10.1 charts the total number of developed tools and the statistics for the minimal task completion (as defined previously, comprising a basic set of metamodel, notation, view and event handling elements). The task completion data is positive showing that tools with realistic level of complexity (usable tools with both static and dynamic features) can be designed and constructed using the Marama meta-tools in a short period of time (three weeks working alongside other commitments).

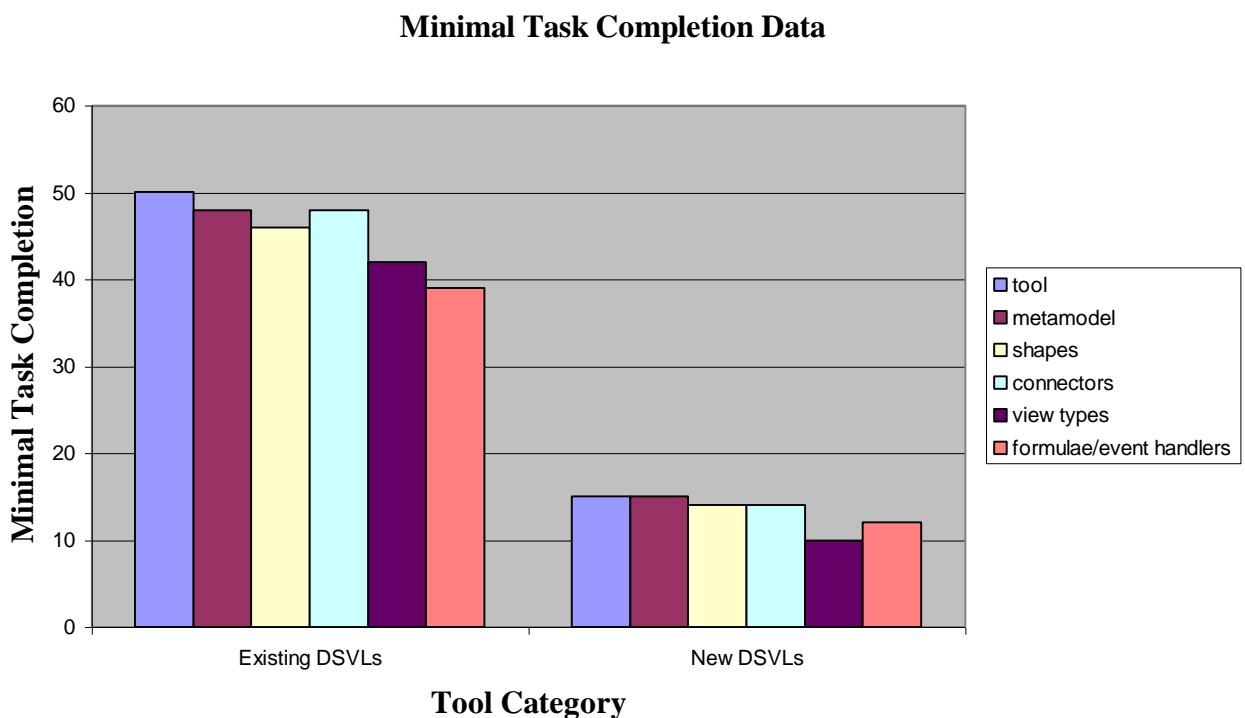


Figure 10.1. Minimal task completion.

The participants responded in the survey that the Marama meta-tools were suited to develop the end user tools in general, but there were still a lot of improvements to make. General strengths emphasised in the survey included: the rapidity of constructing DSVL tools; the simple approach in defining tool data structures and behaviour models; the consistent user interface and the ease of creation and management of multiple views; the low effort and minimum hidden dependency needed to constrain end user model and views effectively; the extensibility and customisability of the generated tools; and the usefulness of being able to sub-type to define reusable metamodel elements and generation of association constraints.

Figure 10.2 charts the number of responses concerning identified weaknesses in three categories (i.e. Stability and Error Handling, Model Effectiveness, and Usability/GUI) and 2 subcategories (i.e. “major” significant weakness, or “minor” issues causing irritation but not significantly affecting functionality, as per the style of the Pounamu survey (Zhu et al, 2007). General weaknesses emphasised in the survey included: the steep learning curve of the Marama meta-tools; the lack of API documentation (users need to have access to API documentation for very complex event processing) and comprehensive user manual; the stability and the ineffective error handling in the prototype; the lack of support for copy/paste specifications; the limited number of reusable building blocks for behavioural specifications; and the difficulty of defining complex formulae due to unfamiliarity with OCL. Some significant stability errors in the Marama meta-tools were also reported during the experiments and were immediately corrected with continuous updates being released during the course of the experiment. The survey result shows the bulk of the issues being in the area of minor stability and minor usability, which were expected of a software prototype at its early proof of concept stage.

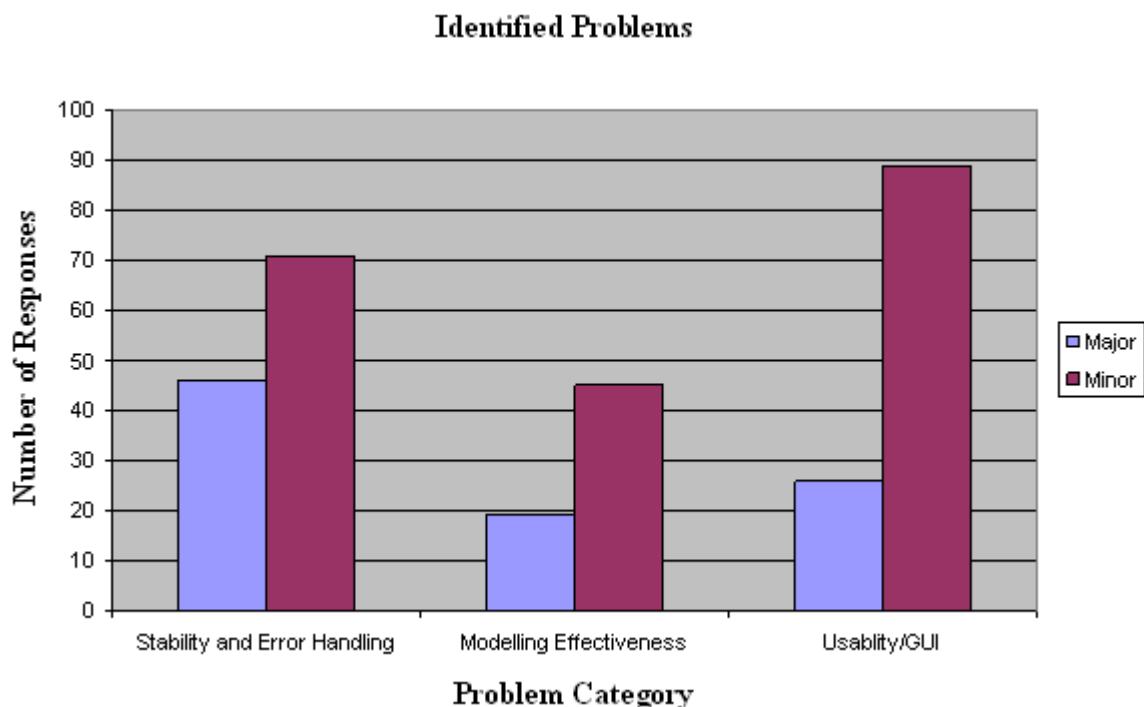


Figure 10.2. Problems identified in the survey.

Figure 10.3 charts the number of distinct suggestions concerning the improvements/extensions of the Marama meta-tools. This gives us insight into what are missing and what the end users found hard to do with the Marama meta-tools. Major suggestions are on the Usability side, targeting the Marama meta-toolset as robust open source software, which should encompass comprehensive

documentation, automated researching and registration, progress tracking, automatic layout, print to file, copy/paste and undo/redo etc. support. Typical suggestions on Modelling Effectiveness include providing an n-nary association type in the metamodel definer, adding more wizard/dialog support for tool/model creation, and providing more comprehensive event handler building blocks for reuse. Typical suggestions on Stability and Error Handling include supporting automatic backup and version control, allowing rollback transactions to a previous stable state, and providing user friendly error messages.

Proposed Improvements/Extensions

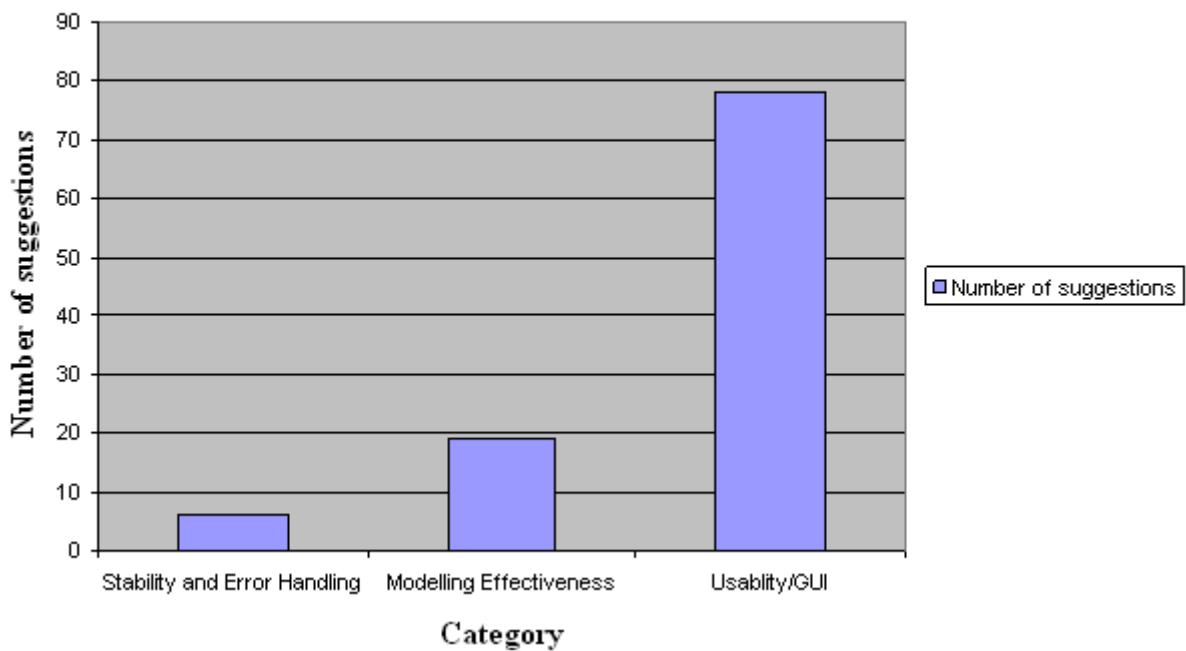


Figure 10.3. Proposed improvements/extensions in the survey.

10.2.4 Small Experienced User Group Study

A set of substantial applications have been developed in our research team, using or integrated with the Marama meta-tools. The developers provided qualitative feedback in the form of experience reports, which were used to assess whether our perceptions of the Marama meta-tools needed to be altered, and whether additional requirements were needed.

A generic mapping tool, MaramaTorua (Jun et al, 2007) has been developed using Marama and later integrated with the Marama meta-tools to provide generic mapping specification support (including model transformations). It has been successfully used in translating BPMN to BPEL4WS code, and importing old Pounamu tool specifications into the Marama meta-tools equivalent. The developer of

MaramaTorua found that Marama meta-tools were substantially easier to use than Pounamu (which he had had significant previous experience with). He felt the consistent views provided for modelling both the static and dynamic aspects of the system were beneficial and the toolset was approaching the quality of a typical commercial software tool.

A health care visual modelling environment, a business process integration tool and an architecture modelling/mapping tool have also been generated using the Marama meta-tools environment. Qualitative feedback from those experienced long-term research goal-oriented users suggested that Marama meta-tools provide a good structural and behavioural modelling and constraining mechanism, while, consistent with the large end user survey, improvements can be made by making the framework robust and error-free.

A design critiquing prototype (Ali, 2007) is under development as an extension to the Marama meta-tools. The developer found that the Marama meta-tools' modelling concept was initially hard to grasp; the Model-View-Controller pattern made it even more difficult to understand; and it is a bit confusing as the extended entity-relationship metamodelling style was partially overlapping with the UML notation, while specifying correct OCL formulae also needed a big effort.

While some event handling building blocks can be used effectively to compose event-based behaviour specifications, all the expert developers needed to escape to code (i.e. use the original custom code writing approach) to define complex backend code generation and user interface extensions (particularly for complex layouts). This indicates to us that the Marama meta-tools need to be further generalised from more examples so that it can provide support for a wider-range of event-based system specifications. The integration of MaramaTorua with the Marama meta-tools (outside the scope of this thesis) is one example addressing these experienced modeller concerns. MaramaTorua can be used for moderately complex code generation and model-model transformation support reducing the need to escape to code in such situations. A new project, also outside the scope of this thesis work, is looking at providing a generic layout specification tool for Marama views. This will build on the three metaphors described here but provide additional layout specific building blocks.

10.3 Further Continuous Evaluation Plan

Substantial efforts have been taken to improve the Marama meta-tools based on these evaluation results. The Marama meta-tools have been made more stable and more resistant to incorrect specifications so that a generated DSVL tool can be error-free for use. Some unnecessarily required user specifications, such as an event triggering handler for interpreting formulae and enforcing constraints, have been replaced by automations enabled by the Marama meta-tools.

A set of JUnit-based test suites are under development. They will be used to perform automatic testing on the Marama meta-tools. This will remove much of the effort of the developers in undertaking white box, black box, unit, integration and system testing, and allow more focus to be placed on end user usability studies.

Our evaluation approach has demonstrated its effectiveness in eliciting weaknesses of a software prototype, so we are reusing the approach to conduct iterative evaluations on the Marama meta-tools. However, from the previous evaluation results, we found that the major barrier for users to effectively use the Marama meta-tools was the initial steep learning curve. To remove this barrier, we plan to provide the end users with more interactive, story-telling examples in a video-format tutorial so that they learn the Marama meta-tools in a more constructive way. We plan to follow the set of guidelines for developing such videos suggested by Plaisant and Shneiderman (Plaisant and Shneiderman, 2005).

10.4 Summary

The Marama meta-tools have been evaluated using a variety of approaches: Cognitive Dimensions, previously established requirements, and substantial formal and informal usability studies. The evaluation results are positive in accepting the integrated approach for event handling specifications but indicate many minor improvements are needed to improve the usability of the Marama meta-tools. While the Marama meta-tools are being improved, we will be conducting similar evaluations to see how effective such improvements are. This is a similar approach to the longitudinal study we undertake with the Pounamu metatool (Zhu et al, 2007). We plan to take an iterative approach in solving the existing problems in the Marama meta-tools and will examine additional metaphors and visual languages to evolve the framework.

Chapter 11 - Conclusions and Future Research

Research contributions of this thesis include design and proof of concept development of: ViTABaL-WS using the Tool Abstraction (TA) metaphor to describe event propagations between abstract components; Kaitiaki using the Event-Query-Filter-Action (EQFA) metaphor to specify event handling behaviour; and MaramaTatau using the Spreadsheet metaphor to specify structural dependencies and constraints to be realised at runtime. The three visual languages and metaphors have also been generalised in the Marama meta-tools environments, unifying the event-based behaviour specifications for a wide range of system behaviour modelling support. In this chapter we elaborate on these achievements and propose future work.

11.1 Research Contributions and Conclusions

We have investigated three exemplar visual event-driven system metaphors to specify event-handling support: Tool Abstraction in ViTABaL-WS; Event-Query-Filter-Action in Kaitiaki; and Spreadsheet in MaramaTatau. We have generalised from the three exemplars and developed a generic high-level visual event handling metaphor and built a proof of concept visual environment for specifying event-based system integration.

ViTABaL-WS was initially designed to support modelling complex interactions between web service components. It uses a Tool Abstraction metaphor for describing relationships between service definitions, and multiple-views of data-flow, control-flow and event propagation in a modelled process. It supports specification of both fine-grained, detailed views and more abstract views of business process protocols, message exchange rules and sequencing, and service invocation, together with generation of Web Service Description Language and Business Process Execution Language definitions from a ViTABaL-WS model for direct deployment. ViTABaL-WS also supports visualisation of running processes for architecture understanding and visual debugging of specified protocols. ViTABaL-WS's event propagation abstraction model has been generalised in the Marama meta-tools environment to facilitate implementation of complex event-based interactions and data exchanges among structural and behavioural components.

End users usually want to specify dynamic interactive behaviour associated with their graphical user interfaces but want to remove the need of having to code these in low level textual programming languages. Our ViTABaL-WS approach specifies high-level tool abstractions, but is not a good approach for GUI event handling, due to its lack of discrimination of end user objects from abstract queries and state-changing actions, and structured data flow between them. Kaitiaki is a visual language for user interface event handling specification targeted at end users. It provides end users with abstract ways to express both simple and complex event handling mechanisms via visual specifications. These specifications use a metaphor of generating events, tool state queries, filters over query results and state changing actions, with dataflow between these building blocks. The support environment allows users to compose handlers from these constructs and relate them to concrete diagramming tool objects. A debugger uses the visual notation to step through a specification, animating constructs and affected diagram objects. Kaitiaki's event handling abstraction model has been generalised in the Marama meta-tools environment to facilitate implementation of complex event handling behaviours by composing a set of reusable graphical building blocks.

A meta-tools approach is commonly used to specify and generate domain specific visual language tools. Specifications of model level behaviours, such as constraints and dependencies, are however very difficult to specify in existing meta-tools. These often need to be specified using conventional code in the form of low level event handlers or the like. Our ViTABaL-WS and Kaitiaki approaches are inefficient for specifying such constraints on metamodels. We integrated MaramaTatau, as a declarative constraint/dependency specification mechanism into the Marama meta-tools. MaramaTatau borrows much from techniques used to support the spreadsheet metaphor, but in a situation with less concreteness. It combines challenged technologies in the form of OCL and spreadsheet interfaces in a simple yet novel way drawing strength from both while mitigating their weaknesses. MaramaTatau augments the Marama meta-tools' metamodel designer, allowing tool developers to specify formulae over metamodels, combined with a one-way constraint system to compute values during tool usage. This allows for much simpler specification of dependency and constraint handling within Marama tools, compared to both the textual event handlers and Kaitiaki visual event handlers. MaramaTatau is generalised together with ViTABaL-WS and Kaitiaki into a generic event handling framework in the Marama meta-tools.

By abstracting from the three earlier, limited-domain exemplars, a general metamodel representation that combines atomic primitives (either shared or non-shared) extended by the three visual languages

is defined. We have developed the Marama meta-tools with this common model to support multiple metaphoric views in the style of the three exemplars for event handling integration. With ViTABaL-WS's focus on providing a visual language for the design and construction of tool abstraction action-event-based architecture, Kaitiaki's focus on providing an extensible event-query-filter-action language for responding to propagated events, and MaramaTatau's focus on providing a declarative spreadsheet-like specification mechanism for model/view level dependencies and constraints, the generalisation of these three approaches within the Marama meta-tools framework provides wide-ranging support for event-based system design and construction.

The generalised Marama meta-tools have been evaluated thoroughly to test their usability and effectiveness for specifying event-based system integration. The evaluation results are positive in accepting the integrated approach for event handling specifications but indicate many minor improvements are needed to improve the usability of the Marama meta-tools. We have released the Marama meta-tools as a publicly accessible toolset following a number of enhancements to address tool stability.

11.2 Future Research

Marama meta-tools framework is still at the prototype stage. We aim to continually develop it to be a robust open source software system to be freely used by interested researchers and organisations. A large range of possible future work directions exist developing from such a platform.

More complete checking of behaviour models, particularly for ViTABaL-WS and Kaitiaki, could catch errors in the specification before code generation and realisation.

Users must currently manually layout ViTABaL-WS and Kaitiaki composition models and automatic layout of views would in some cases be useful, especially when they become large. Automatic layout may be useful to improve a user's ability to show/hide/collapse parts of a specification to manage size and complexity.

Programming by example extensions would be useful in every view of the Marama meta-tools to allow users to make changes to an exemplar modelling tool view and add/remove building blocks to/from it.

A more complex view specification tool is under development, which allows many-to-many mappings between view shapes and connectors and model entities and associations to be specified using formulae. This will again make it easier for tool developers to build more complex view-model mappings without resorting to using complex event-driven handlers.

We plan to extend the MaramaTatau language by adding higher-order functions (HOF) (especially the “map” and “accumulate” functions seen in functional programming languages like Haskell (Haskell, 2007<http://www.haskell.org/>)). For an example of the “map” HOF, we could specify

```
UnconnectedShape = map (TestNoConnection, listOfShapes)
```

where

```
TestNoConnection(MaramaShape): Boolean = if  
MaramaShape.getSourceConnections().size()==0 and  
MaramaShape.getTargetConnections().size()==0 then true else false  
endif
```

The `map (Function, Collection)` function applies the `Function` to each item in the `Collection` to produce the `map` function result (in this example another `Collection` of `MaramaShape` instances being unconnected).

We could also extend the MaramaTatau language by adding function compositions, i.e. using a function as a parameter to another function, another HOF ability. This facilitates reuse of formulae and existing library functions in function compositions. We have attempted to add user defined functions (e.g. compositional functional definition of reusable formula with parameters) to collaboratively operate with OCL. We can extend MaramaTatau to reuse ViTABaL-WS and Kaitiaki specifications either partially or completely, to allow both side-effect-less constraint/query, and side-effect action to be specified in formulae. To this end, hidden dependencies will be an issue that requires mitigation. A new high-level generalisation dependency view could be useful to indicate/annotate cascading references in dependent views.

Run-time monitoring of the Marama meta-tools for performance analysis could be supported via the visual debugging sub-system. The visual debugger could be further enhanced with “watch” controls so that the user can choose to trace a certain event and its response instead of debugging the entire behavioural specification.

The Marama meta-tools are to be evolved by abstracting from more domain-specific examples, such as BPMN (OMG, 2006). The Marama meta-tools' event handling abstraction model can be specified in the MaramaTorua (Huh et al, 2007) mapping tool to facilitate generation to a wide range of implementations for interpretation.

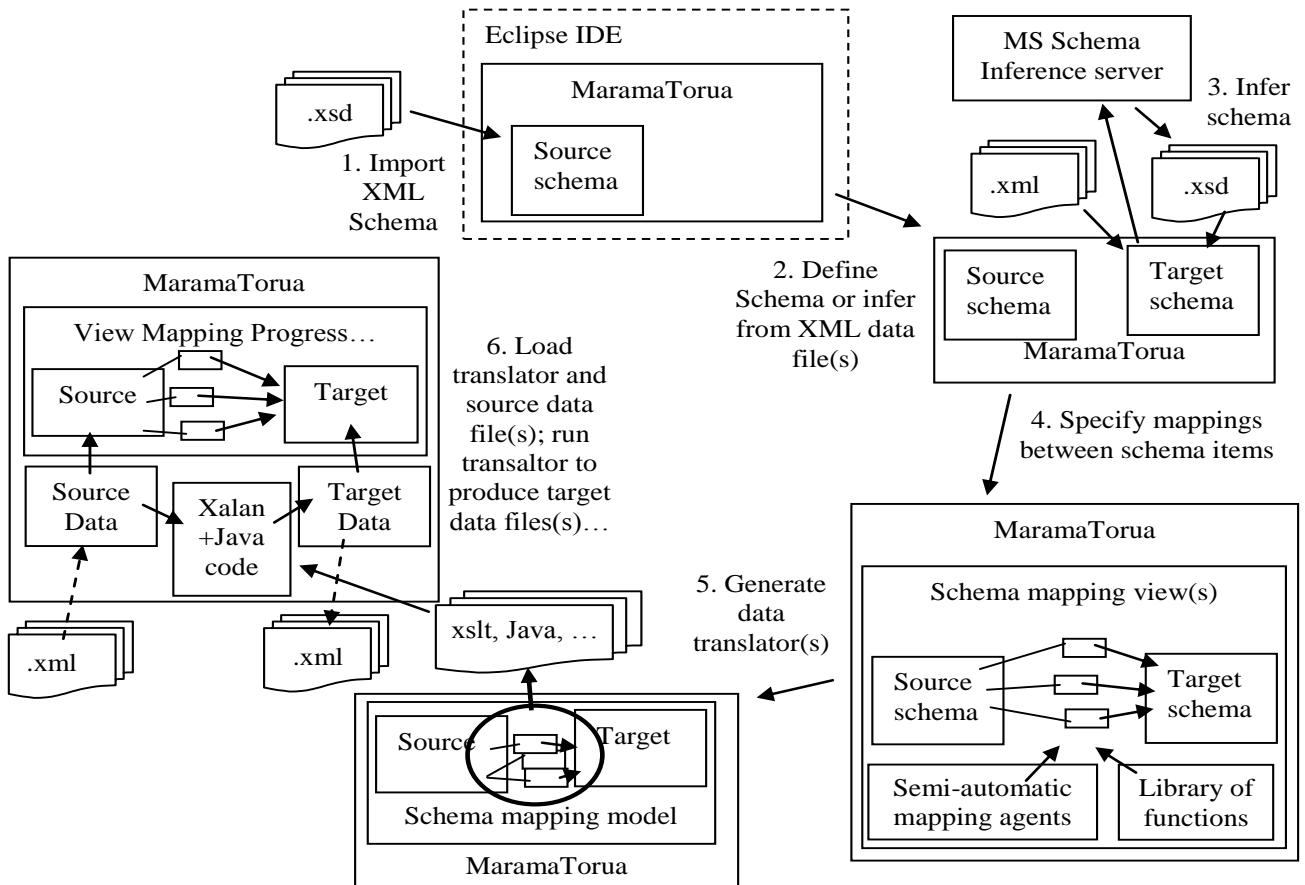


Figure 11.1. Outline of using MaramaTorua. (Huh et al, 2007)

To allow one specification to generate others with corresponding implementation classes, a set of mapping schemas can be defined in MaramaTorua (Huh et al, 2007) to provide interchanging mechanisms between ViTABaL-WS, Kaitiaki and MaramaTatau specifications. MaramaTorua is integrated with the Marama meta-tool and its generated translators can be used directly within new Marama tools to support model integration, translation, and code and script generation. Figure 11.1 illustrates the MaramaTorua approach in specifying mappings. This involves the following steps of tasks:

- (1) Users import XML schemas (either manually created or automatically generated from the Marama meta-tools) into MaramaTorua to provide the source and/or target data format specifications.

- (2) Users can define their own schema using MaramaTorua's schema editor or an existing Eclipse schema editor.
- (3) A schema can also be generated using a remote web service link to the Microsoft schema inference engine.
- (4) Once the schemas are imported into MaramaTorua, users can specify mappings between the source schema and target schema elements. The mapping specifications can be either simple so that users can copy a source data item to a target item, or complex so that users need to iterate over the source collection filtering on specified data item values and create new target data structures.
- (5) On completion of the inter-schema mapping specification, a translator can be generated. MaramaTorua reuses a set of mapping functions to synthesise a data translator implementation.
- (6) Other translator implementation languages (e.g. Eclipse ALT or pure Java code) can also be used. Users can test the translator by executing it with example source data files loaded into MaramaTorua.

Many other Marama extensions are being developed. These include a distributed environment with thin client user interfaces and web service back-end, collaborative support for concurrent team work, sketch-based user interfaces and automatic translations to formal Marama model and views. Once these extensions are fully developed, we will integrate them into the Marama meta-tools thus making the framework more fully-functioned.

11.3 Summary

The research has focussed on providing visual specification and runtime visualisation support for the design and construction of complex event-based systems. We have integrated three event handling specification languages based on a canonical event model. ViTABA-L-WS provides a Tool Abstraction language for the design and construction of action-event propagation architectures. Kaitiaki provides an extensible Event-Query-Filter-Action language for both action and state-change event propagation and handling. MaramaTatau provides a static Spreadsheet-like dependency and constraint mechanism to support specification of state-change event propagation and response. A synergy of these languages and their generalisation in the Marama meta-tools environment provide wide-ranging support for event-based system design and construction.

References

- Ali, N. M. (2007) A Generic Visual Critic Authoring Tool, In Proceedings of the 2007 IEEE Symposium on Visual Languages and Human-Centric Computing, Coeur d'Alène, Idaho, USA.
- Baeyens, T. (2007) The state of workflow, <http://www.bpm.org/state.of.workflow.html>
- Baker, J. (2002) Business Process Modelling Language: Automating Business Relationships, Business Integration Journal, <http://www.bijonline.com>
- Barrett, D. J., Clarke, L. A., Tarr, P.L., and Wise, A.E. (1996) A Framework for Event-Based Software Integration, ACM Transactions on Software Engineering and Methodology (TOSEM), pp. 378-421.
- Bederson, B., Meyer, J. and Good, L. (2000) Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java, In Proceedings of 2000 ACM Conference on User Interface and Software Technology, ACM Press, pp. 171-180.
- Ben-Shaul, I. Z. (1994) Oz: A Decentralized Process Centered Environment. Technical Report CUCS-024-94, Columbia University Department of Computer Science, PhD Thesis.
- Benatallah, B., Dumas, M., Fauvet, M.C. and Rabhi, F. (2003) Towards Patterns of Web Services Composition, In Patterns and skeletons for parallel and distributed computing, Springer.
- Berndtsson, B., Mellin, J., and Hogberg, U. (1999) Visualization of the Composite Event Detection Process, In proceedings of the 1999 International Workshop on User Interfaces to Data Intensive Systems, IEEE CS Press, pp. 118-127.

Box, D., Cabrera, L.F., Critchley, C., Curbera, F., Ferguson, D., Graham, S., Hull, D., Kakivaya, G., Lewis, A., Lovering, B., Niblett, P., Orchard, D., Samdarshi, S., Schlimmer, J., Sedukhin, I., Shewchuk, J., Weerawarana, S., Wortendyke, D. (2006) Web Services Eventing (WS-Eventing), <http://www.w3.org/Submission/WS-Eventing/>

Buchmann, A., Bornhövd, C., Cilia, M., Fiege, L., Gärtner, F., Liebig, C., Meixner, M., Mühl, G. (2004) DREAM: Distributed Reliable Event-based Application Management, In Web Dynamics, Springer.

Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., and Grose, T. (2003) Eclipse Modeling Framework: A Developer's Guide, Addison Wesley Professional.

Burbeck, S. (1992) Applications Programming in Smalltalk-80(TM): How to user Model-View-Controller (MVC), <http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html>

Burnett, M., Atwood, J., Djang, R.W., Reichwein, J. (2001) Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm, Journal of Functional Programming, 11(2): 155-206.

Burnett, M. and Ambler, A.L. (1992) A Declarative Approach to Event-Handling in Visual Programming Languages, In IEEE Workshop on Visual Languages, pp. 34-40, Seattle, Washington.

Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.K. (1994) Composite Events for Active Databases: Semantics Contexts and Detection, In Proceedings of the 20th International Conference on Very Large Data Bases.

Chappell, D. and Associates (2007) Microsoft and BPM: A Technology Overview, <http://www.microsoft.com/biztalk/solutions/bpm/technicalwhitepaper.mspx>

Cohen, D. (2006) AP5 Reference Manual, <http://ap5.com/doc/ap5-man.html>

Conway, M., Audia, S., Burnette, T., Cosgrove, D., and Christiansen, K. (2000) Alice: Lessons Learned from Building a 3D System for Novices, In Proceedings of the SIGCHI conference on Human factors in computing systems, pp. 486-493.

Costagliola, G., Deufemia, V., Ferrucci, F., Gravino, C. (2002) The Use of the GXL Approach for Supporting Visual Language Specification and Interchanging, In Proceedings of HCC'02, Arlington, Virginia, pp.131-138.

Costagliola, G., Deufemia, V. and Polese G. (2004) A Framework For Modeling and Implementing Visual Notations with Applications to Software Engineering, ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 13 Issue 4.

Coupage, T., Roncancio, C. L., Bruley, C. (1999) A Visualization Service for Event-Based Systems, Proc. 15emes Journees Bases de Donnees Avancees, BDA.

Cox, P., Giles, F., Pietrzykowski, T. (1989) Prograph: A step towards liberating programming from textual conditioning, 1989 IEEE Workshop on Visual Languages, Rome, Italy, 150-156.

Cox, P. T., Smedley, T. J., Garden, J. and McManus, M. (1997) Experiences with Visual Programming in a Specific Domain – Visual Language Challenge '96, In Proceedings of the 1997 IEEE Symposium on Visual Languages, pp. 254-259.

Cugola G., Di Nitto E., Fuggetta A. (1998) Exploiting an event-based infrastructure to develop complex distributed systems, In Proceedings of the 20th international conference on Software engineering, Kyoto, Japan, pp. 261-270.

Dewan, P. and Choudhary, R. (1991) Flexible user interface coupling in collaborative systems, CHI'91, pp. 41-49.

Dillon, A (2001), Usability evaluation, In W. Karwowski (ed.) Encyclopedia of Human Factors and Ergonomics, London: Taylor and Francis.

Drumea, A. and Popescu, C. (2004) Finite State Machines and Their Applications in Software for Industry Control, Electronics Technology: Meeting the Challenges of Electronics Technology Progress, 27th International Spring Seminar, Vol.1, pp. 25-29.

ebPML (2002) BPML 1.0 Analysis, http://www.ebpml.org/bpml_1_0_june_02.htm

Eclipse (2007) EMF, <http://www.eclipse.org/modeling/emf/>

Eclipse (2006) EMF OCL plug-in, <http://www.eclipse.org/modeling/mdt/downloads/?project=ocl>

Eclipse (2007) GEF, <http://www.eclipse.org/gef/>

Engels, G. and Erwig M. (2005) ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications, In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, Long Beach, CA, USA, pp. 124-133.

Felfernig, A., Friedrich, G., Jannach, D., Russ, C. and Zanker M. (2003) Developing Constraint-Based Applications with Spreadsheets. In 18th International Joint Conference on Artificial Intelligence. Acapulco, Mexico.

Fensel, D. and Bussler, C. (2002) The web service modeling framework WSMF, Electronic Commerce Research and Applications, vol. 1, no. 2, pp. 113--137.

Fernstrom, C. (1993) Process Weaver: Adding Process Support to UNIX. In Proceedings of the Second International Conference of Software Process. Pages 12-26.

Foster, H., Uchitel, S., Magee, J. and Kramer, J. (2003) Model-based verification of web service compositions, In Proceedings of the 18th IEEE international conference on automated software engineering, Montreal, Canada.

Ferguson R., Parrington N., Dunne P., Archibald J., Thompson J. (1999) MetaMOOSE-an object-oriented framework for the construction of CASE tools, In Proceedings of CoSET'99, LA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

Gatziu, S, Dittrich, K.R. (1993) Events in an Active Object-Oriented Database System, Proc. 1st Intl. Workshop on Rules in Database Systems (RIDS).

Gottfried, H. J. and Burnett M. M. (1997) Programming Complex Objects in Spreadsheets: an Empirical Study Comparing Textual Formula Entry with Direct Manipulation and Gestures, Papers Presented at the Seventh Workshop on Empirical Studies of Programmers ESP'97.

Green, T. R. G. and Petre, M. (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing*, 7, 131-174.

Grundy, J.C., Hosking, J.G. (1995) ViTABaL: A Visual Language Supporting Design by Tool Abstraction, Proceedings of the 1995 IEEE Symposium on Visual Languages, Darmstadt, Germany, IEEE CS Press, pp. 53-60.

Grundy, J.C. and Hosking, J.G. (1996) Constructing Integrated Software Development Environments with MViews. *International Journal of Applied Software Technology*, vol. 2, no. 3-4, 133-160.

Grundy, J.C. and Hosking, J.G. (1998) Serendipity: integrated environment support for process modelling, enactment and work coordination, *Automated Software Engineering: Special Issue on Process Technology* 5(1), January 1998, Kluwer Academic Publishers, pp. 27-60.

Grundy, J.C., Hosking, J.G., Li, L. And Liu, N. (2006) Performance engineering of service compositions, ICSE 2006 Workshop on Service-oriented Software Engineering, Shanghai, China.

Grundy, J.C., Hosking, J.G., and Mugridge, W.B. (1996) Supporting flexible consistency management via discrete change description propagation, *Software – Practice and Experience*, Volume 26, Issue 9, pp. 1053 – 1083.

Grundy, J.C., Hosking, J.G., Mugridge, W.B. (1996) Towards a unified event-based software architecture, in Joint Proceedings of the SIGSOFT'96 Workshops, 1996 International Software Architecture Workshop, Oct 14-15, San Francisco, ACM Press, 121-125.

Grundy, J. C., Hosking, J. G. and Mugridge, W. B. (1997) Visualising Event-based Software Systems: Issues and Experiences. In Proceedings of SoftVis97. Adelaide, Australia.

Grundy, J.C., Hosking, J.G., Zhu N., and Liu N. (2006) Generating Domain-Specific Visual Language Editors from High-level Tool Specifications, In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, Tokyo, Japan, pp. 25-36.

Grundy, J.C., Mugridge, W.B. and Hosking, J.G. (1998) Visual specification of multiple view visual environments, In Proceedings of IEEE VL'98, Halifax, Nova Scotia, Canada, IEEE CS Press, pp. 236-243.

Grundy, J.C., Mugridge, W.B., and Hosking, J.G. (1998) Static and Dynamic Visualisation of Software Architectures for Component-based Systems, In Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering, San Francisco, KSI Press, pp. 426-433.

Gugola G, Nitto E, and Fuggetta A. (2001) The JEDI event-based infrastructure and its application to the development of the OPSS WFMS, IEEE Trans. Software, 2001, 27(9): 827-850.

Haeberli, P. (1988) ConMan: a visual programming language for interactive graphics, In Proceedings of the 15th annual conference on Computer graphics and interactive techniques, ACM Press, pp. 103-111.

Hamadi, R., Benatallah, B. (2003) A petri-net based model for web service composition, Proc 14th Australasian Database Conference, Adelaide, Australia, CRPIT Press.

Hanna, K. (2002) Interactive visual functional programings, Proceedings of the seventh ACM SIGPLAN international conference on Functional programming ICFP '02, Volume 37 Issue 9.

Hanson, J. (2005) Event-driven services in SOA, Java World,
<http://www.javaworld.com/javaworld/jw-01-2005/jw-0131-soa.html>

Hartson, H. R., Andre, T. S., and Williges, R. C. (2003) Criteria for evaluating usability evaluation methods, International Journal of Human-Computer Interaction 15(1): 145-181.

Haskell (2007), <http://www.haskell.org/>

Hill, R. D. (1992) The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Monterey, California, pp.335-342, ACM Press.

Hill, R. D., Brinck, T., Rohall, S. L., Patterson, J. F. and Wilner, W. (1994) The Rendezvous Architecture and Language for Constructing Multiuser Applications. ACM Transactions on Computer-Human Interaction (TOCHI). Vol. 1, no. 2, pp.81-125, ACM Press.

Hirakawa, M. and Ichikawa, T. (1992) Advances in Visual Programming, In Proceedings of the Second International Conference on Systems Integration, pp. 538-543.

Hoof, J.V. (2006) how EDA extends SOA and why it is important, <http://soa-eda.blogspot.com/2006/11/how-eda-extends-soa-and-why-it-is.html>

Hudak, P. (1989) Conception, evolution, and application of functional programming languages, ACM Computing Surveys 21 (3): 359-411.

Huh, J., Grundy, J.C., Hosking, J.G., Liu, N., Amor R. (2007) Integrated data mapping for meta-tool model integration, transformation and code generation, working paper, the University of Auckland.

IBM (2002) Business Processes Web Services for Java,
<http://www.alphaworks.ibm.com/tech/bpws4j>

IBM (2003) Specification: Business Process Execution Language for Web Services Version 1.1,
<http://www.ibm.com/developerworks/library/ws-bpel/>

Inazumi, H. and Omoto, N. (1999) A new scheme for verifying rule-based systems using Petri nets, 1999 IEEE International Conference on Systems, Man, and Cybernetics, Volume 1, Page(s):860 - 865 vol.1.

Jacob, R. (1996) A Visual Language for Non-WIMP User Interfaces, In Proceedings of the 1996 IEEE Symposium on Visual Languages, Boulder, CO, USA, pp. 231-238.

Jia, X., Steele, A., Qin L., Liu, H. and Jones, C. (2005) An Event-Based Framework for Model Integration, In Proceedings of the 2005 IEEE International Conference on Electro Information Technology.

Jin, W. (2003) A structured approach to visualising the event handling specification, Thesis (MSc-Computer Science) – University of Auckland.

Jong E.D. (1997) Multi-paradigm Programming in Large Control Systems, In Proceedings of the 1997 Joint Workshop on Parallel and Distributed Real-Time Systems (WPDRTS/OORTS '97).

Jung, M.C. and Cho, S.B. (2005). A novel method based on behavior network for Web service composition, In Proceedings of the International Conference on Next Generation Web Services Practices, 22-26 Aug. 2005. Page(s):6 pp.

Kelly, S., Lyytinen, K., and Rossi, M. (1996) Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, In Proceedings of CAiSE'96, LNCS 1080.

Kelso, J. (2002) A Visual Programming Environment for Functional Languages, Thesis (PhD – Computer Science) – Murdoch University, <http://www.csse.uwa.edu.au/~joel/vfpe/thesis.pdf>

Kiringa, I. (2002) Specifying Event Logics for Active Database, Description Logics

Koenig, J. (2004) JBOSS jBPM, http://www.jboss.com/pdf/jbpm_whitepaper.pdf

Kraemer, F. A. and Herrmann P. (2007) Transforming Collaborative Service Specifications into Efficiently Executable State Machines, In Proceedings of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007).

Krishnamurthy, B., Rosenblum, D.S. (1995) Yeast: a general purpose event-action system, In Proceedings of Software Engineering, IEEE Transactions, pp. 845-857.

Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., and Karsai G. (2001) Composing Domain-Specific Design Environments, Computer, 44-51.

Lewicki, D. and Fisher, G. (1006) VisiTILE - A Visual Language Development Toolkit, Proceedings of the 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, pp. 114-121.

Li, L., Grundy, J.C. and Hosking, J.G. (2007) EML: A Tree Overlay-based Visual Language for Business Process Modelling, In Proceedings of the 2007 International Conference on Enterprise Information Systems, Portugal.

Li, X., Marin, J. M. and Chapa, S. V., A Structural Model of ECA Rules in Active Database, Lecture Notes in Computer Science, in Proceedings of the Second Mexican International Conference on Artificial Intelligence, Pages 486-493, 2002.

Li, X., Mugridge W. B. and Hosking G. (1997) A Petri Net-based Visual Language for Specifying GUIs, In Proceedings of the 1997 IEEE Symposium on Visual Languages, Isle of Capri, Italy.

Liu, A.F., Chen Z.G.; He H., Gui W.H. (2007) Treenet: A Web Services Composition Model Based on Spanning tree, In Proceedings of the 2nd International Conference on Pervasive Computing and Applications, 26-27 July 2007 Page(s):618 – 623

Liu, N., Hosking, J.G. and Grundy, J.C. (2005) A Visual Language and Environment for Specifying Design Tool Event Handling, In Proc. VL/HCC'2005, Dallas, Texas, USA.

Liu, N., Grundy, J.C. and Hosking, J.G. (2007) A Visual Language and Environment for Specifying User Interface Event Handling in Design Tools, In Proceedings of the 2007 Australasian Conference on User Interfaces, Ballarat, Australia, CRPIT Press.

Liu, N., Hosking, J.G. and Grundy, J.C. (2004) Integrating a Zoomable User Interfaces Concept into a Visual Language Meta-tool Environment, In Proceedings of the 2004 International Conference on Visual Languages and Human-Centric Computing, Rome, Italy, IEEE CS Press.

Liu, N., Hosking, J.G. and Grundy, J.C. (2007) MaramaTatau: Extending a Domain Specific Visual Language Meta Tool with a Declarative Constraint Mechanism, in Proceedings of the 2007 IEEE Symposium on Visual Languages and Human-Centric Computing, Coeur d'Alène, Idaho, USA

Liu, N., Grundy, J.C. and Hosking, J.G. (2005) A visual language and environment for composing web services, In Proceedings of the 2005 ACM/IEEE International Conference on Automated Software Engineering, Long Beach, California, IEEE Press, pp. 321-324.

Matskin, M. and Montesi, D. (1998) Visual Rule Language for Active Database Modelling, Information Modelling and Knowledge Bases IX. IOS Press, pp. 160-175.

Meier, R. and Cahill, V. (2002) Taxonomy of Distributed Event-based Programming Systems, In Proceedings of the 22nd International Conference on Distributed Computing Systems Workshop.

Menon, S., Dasgupta, P., and LeBlanc, R.J. (1993) Asynchronous event handling in distributed object-based systems. In Proc. the 13th Conference on Distributed Computing Systems, pages 383-390, Pittsburgh, Pennsylvania.

MIT (2006) The Alloy Analyzer, <http://alloy.mit.edu/>

Morch, A. (1998) Tailoring tools for system development, Journal of End User Computing, 10:2, pp. 22-29.

MSND (2007) Queues Overview, <http://msdn2.microsoft.com/en-us/library/ms733789.aspx>

MSND (2007) Create Trigger, [http://msdn2.microsoft.com/en-us/library/aa258254\(SQL.80\).aspx](http://msdn2.microsoft.com/en-us/library/aa258254(SQL.80).aspx)

MSDN (2007) Understanding the Event Model, <http://msdn2.microsoft.com/en-us/library/ms533023.aspx>

MSDN (2005) DSL Tools, <http://msdn2.microsoft.com/en-us/vstudio/aa718368.aspx>

Mugridge, W.B., Hosking, J.G. and Grundy, J.C. (1998) Drag-throughs and attachment regions in BuildByWire, Proc. of OZCHI'98, Adelaide, Australia, IEEE CS Press, pp. 320-327.

Myers, B. (1990) A. Garnet: Comprehensive Support for Graphical, highly Interactive User Interfaces. IEEE COMPUTER. 23 (11), 71-85.

Myers, B.A. (1997) The Amulet Environment: New Models for Effective User Interface Software Development, IEEE TSE, vol. 23, no. 6, 347-365.

Myers, B.A., Pane, J.F., and Ko, A. (2004) Natural programming languages and environments, <http://www.acmqueue.org/>

Narayanan, S. and McIlraith, S.A. (2002) Simulation, verification and automated composition of web services. In Proceedings of the 11th World Wide Web Conference.

Neag, I. A. and Tyler, D. F. (2001) Combined Visual and Textual Programming Methodology for Signal-based Automatic Avionics Testing Systems, In Proceedings of the 20th Conference on Digital Avionics System, 9A5/1-9A5/10 vol.2.

Nikau (2007) Marama, <http://www.cs.auckland.ac.nz/Nikau/marama/>

OMG (2006) BPMN 1.0, <http://www.bpmn.org/>

OMG (2004) Event Service Specification, <http://www.omg.org/docs/formal/04-10-02.pdf>

OMG (2003) OCL, <http://www.omg.org/docs/ptc/03-10-14.pdf>

Palanque, P.A., Bastide, R., Dourte, L. and Sibertin-Blanc, C. (1993) Design of User-Driven Interfaces Using Petri Nets and Objects, In Proceedings of Advanced Information Systems Engineering, pp. 569-585, Springer-Verlag.

Paschke, A. (2006) ECA-LP / ECA-RuleML: A Homogeneous Event-Condition-Action Logic Programming Language, Int. Conf. on Rules and Rule Markup Languages for the Semantic Web (RuleML'06), Athens, Georgia, USA.

Pautasso, C. and Alonso, G. (2003) Visual Composition of Web Services, Proc IEEE HCC'03, Auckland, pp. 92-99.

Pautasso, C. and Alonso, G. (2005) The JOpera Visual Composition Language JVLC, 16(1-2):119-152.

Peltonen J. (2000) Visual Scripting for UML-Based Tools. In Proceedings of ICSSEA 2000: Paris, France.

Plaisant, C. and Shneiderman, B. (2005) Show me! Guidelines for producing recorded demonstrations, in Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, Dallas, USA, 171-178.

Rapide Design Team (1997) Program Analysis and Verification Group, Computer Systems Lab, Stanford University, Guide to the Rapide 1.0 Language Reference Manuals.

Repenning, A. and Sumnet, T. (1995) Agentsheets: a medium for creating domain-oriented visual languages, Computer, 28, no. 3.

Robbins, J. E., Medvidovic, N., Redmiles, D. F. and Rosenblum, D. S. (1998) Integrating Architecture Description Languages with a Standard Design Method, In Proceedings of the 20th International Conference on Software Engineering, Kyoto, Japan, pp. 209-218.

Robbins, J.E. and Redmiles D.F. (1998) Software architecture critics in the Argo design environment, J Knowledge-Based Systems 11 (1998) 47-60.

Roberts, D, and Johnson, R. (1996) Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks, In Pattern Languages of Program Design 3, Addison-Wesley, Reading, MA.

RuleML Initiative (2006), <http://www.ruleml.org>

Schiffer, S. and Fröhlich, J.H. (1994) Concepts and Architecture of Vista – a Multiparadigm Programming Environment, in Proceedings of the 10th IEEE/CS Symposium on Visual Languages, St.Lois/USA, pp. 40-47.

Sheth, B.D. (1994) A Learning Approach to Personalized Information Filtering, Master thesis in Computer Science and Engineering, Massachusetts Institute of Technology.

Sliwa, C. (2003), Event-driven architecture poised for wide adoption, Computer World, <http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,81133,00.html>

Smith, D.C., Cypher, A. and Spohrer, J. (1995) KidSim: programming agents without a programming language, Communications of the ACM, vol. 37, no. 7, pp. 54 – 67.

Smith, D. N. (1990) The interface construction set, in Visual Languages and Applications, (T. Ichikawa, E. Jungert, R. Korfhage, eds.), Plenum Pub., NY.

Srivastava, B., Koehler, J. (2003) Web Service Composition - Current Solutions and Open Problems, ICAPS Workshop on Planning for Web Services, Trento, Italy.

Sun (2005) EV - JiniTM Distributed Events Specification, <http://java.sun.com/products/jini/2.1/doc/specs/html/event-spec.html>

Sun (2005) Java BluePrints Model-View-Controller, <http://java.sun.com/blueprints/patterns/MVC-detailed.html>

Sun (2007) The J2EE™ Tutorial, <http://java.sun.com/javase/5/docs/tutorial/doc/>

Sun (2007) The Java™ Tutorials, Lesson: Writing Event Listeners,
<http://java.sun.com/docs/books/tutorial/uiswing/events/index.html>

Taentzer, G. (1999) Adding Visual Rules to Object-Oriented Modeling Techniques, in Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS'99), Nancy, France. IEEE Computer Society.

Tang, Y., Chen, L. He, K.T. and Jing, N. (2004). SRN: an extended Petri-net-based workflow model for Web service composition, In Proceedings of the IEEE International Conference on Web Services, 6-9 July 2004. Page(s):591 – 599

Thone, S., Depke, R. and Engels, G. (2002) Process-oriented, flexible composition of web services with UML, Proc ER-Wkshp on Conceptual Modeling Approaches for e-Business, Tampere, Finland, LNCS, 2002

Tolvanen J. (2006) OOPSLA demonstrations chair's welcome: MetaEdit+: integrated modeling and metamodeling environment for domain-specific languages, Companion to the 21st ACM.

Vlissides, J.M. and Linton, M. (1989) Unidraw: A framework for building domain-specific graphical editors, Proc. UIST'89, ACM Press, pp. 158-167.

W3C (2001), Web Services Description Language (WSDL) 1.1, <http://www.w3.org/tr/wsdl>

Wagner, F., Schmuki R., Wagner T. and Wolstenholme P. (2006) Modeling Software with Finite State Machines: A Practical Approach, Auerbach Publications.

Weber, G. (2003) Semantics of form-oriented analysis, <http://www.diss.fu-berlin.de/2003/72/>

Welch, B. and Jones, K. (2003) Practical Programming in Tcl and Tk, Prentice-Hall.

Wirtz, G. (1993) A Visual Approach for Developing, Understanding and Analyzing Parallel Programs, in Proc IEEE VL'93, IEEE CS Press, pp. 261-266.

Wordsworth, J.B. (1992) Software Development with Z - A Practical Approach to Formal Methods in Software Engineering, Addison Wesley.

Workflow Management Coalition (1999) Terminology & Glossary,
http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf

Zhu, N., Grundy, J.C., Hosking, J.G., Liu, N., Cao, S. and Mehra, A. (2007) Pounamu: a meta-tool for exploratory domain-specific visual language tool development, Journal of Systems and Software 80 (8), Elsevier.