Generating Domain-Specific Visual Language Editors from High-level Tool Specifications

John Grundy[1, 2], John Hosking[1], Nianping Zhu[1] and Na Liu[1]
[1]Department of Computer Science and [2]Department of Electrical and Computer Engineering
University of Auckland, Private Bag 92019
Auckland, New Zealand
{john-g, john, nianping, karen}@cs.auckland.ac.nz

Abstract

Domain-specific visual language editors are useful in many areas of software engineering but developing such editors is challenging and time-consuming. We describe an approach to generating a wide range of these graphical editors for use as plug-ins to the Eclipse environment. Tool specifications from an existing meta-tool, Pounamu, are interpreted to produce dynamic, multi-view, multi-user Eclipse graphical editors. We describe the architecture and implementation of our approach, examples of its use realizing domain-specific modelling tools, and strengths and limitations of the approach.

1. Introduction

Domain-specific visual languages (DSVLs) are popular in many areas of software engineering [2], [8]. DSVLs provide high-level, domain-specific visual notations to describe complex information in a particular domain more efficiently and effectively than general-purpose modelling languages [11], [16]. Examples include special-purpose modelling languages for software architecture modelling and requirements capture [19], UML extensions for modelling e.g. aspect-oriented systems or real-time behaviour [1], process modelling and web service orchestration tools [25], data transformation tools [11], and visualization techniques, e.g. 3D code package navigation [27]. Users need tools to navigate and collaboratively edit models defined using such DSVLs. These tools often need to generate code [7],[4] or other models such as XMI, BPEL4WS or XSLT [25],[19].

Developing DSVL tools for such domains is demanding [2]. Key challenges include specifying the desired tool meta-models and visual notations; realising tool editors; and integrating the DSVL tools with other software engineering tools. A number of tool specification techniques and associated meta-tools have been produced to make the task easier, such as MetaEdit+ [14], Pounamu [36], Escalante [21], IPSEN [15], MetaMOOSE [7], DiaGen [22] and DSL Tools [8]. Unfortunately many of these approaches suffer from insufficient expressive power to build desired visual language tools; difficulty in using the meta-tools, reliance on extending frameworks and hence programming skills; and a lack of tool integration support (data, control and presentation [31]). An additional problem is that the generated tools must compete, in terms of usability and quality, with the highly polished commercial and open source general purpose development tools and environments end users are used to, such as the Eclipse and VisualStudio IDEs [5][8]. Unfortunately building domain-specific visual language tools with Eclipse frameworks is still very challenging, time-consuming and error-prone [6], [27].

We have developed Marama[1], a set of Eclipse plug-ins that realize domain-specific visual modelling tools specified using high-level DSVL tool specifications produced from an existing meta-tool, Pounamu [36]. Marama allows users to rapidly specify or modify a desired visual language tool using Pounamu design tools and then have the tool realised as a high-quality Eclipse-based editing environment. Multiple users and multiple views are supported along with visual editing and complex behavioural specification support. Marama DSVL editors look and feel like other Eclipse graphical editors, use Eclipse code generation support, and can be integrated with and extended by other Eclipse plug-ins. Their specifications can, however, be modified on the fly using Pounamu allowing rapid trialling and deployment.

We firstly present a motivating example for our work and survey related work on meta-tools, domain-specific languages and visual language environments. We then provide an overview of Marama's approach and architecture, illustrate the development and use of Marama visual editors, and discuss key design and implementation details. We describe evaluations of the Marama toolset, discuss its strengths and limitations, and summarise possible future research directions.

2. Motivation

Visual, domain-specific languages empower software engineers by providing a set of building blocks and visual metaphors allowing them to efficiently and effectively describe models in particular problem domains. Consider a tool for specifying web service compositions.

[1] *Marama* is Maori for "moon", the generator of an Eclipse…

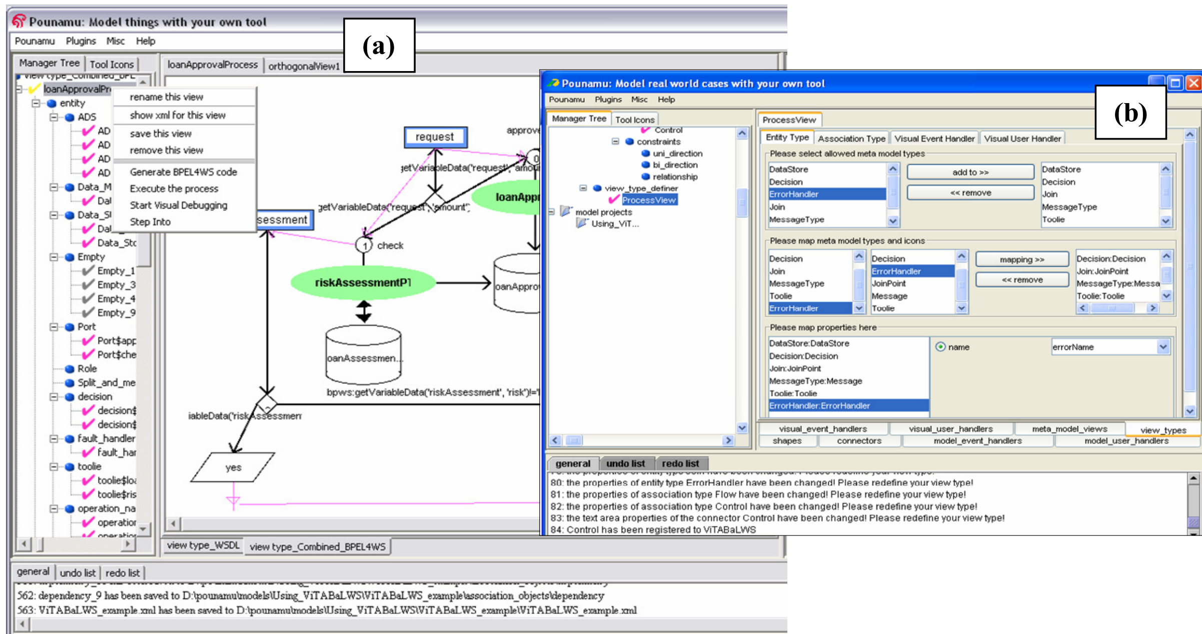

Figure 1. (a) ViTABaL-WS editing in Pounamu; (b) tool specifications in Pounamu; (c) editing in Marama.

Rather than writing process descriptions in a textual scripting language like BPEL4WS (BusinessProcess Execution Language for Web Services) [19], most users would prefer to graphically specify the web services and their composition to form a new "business process". A number of domain-specific visual language tools have been developed to do this [25],[19], the rationale being that a special-purpose tool using a special-purpose language is better than a general-purpose tool/language such as vanilla UML in a conventional CASE tool.

We have developed one such tool, ViTABaL-WS, for composing web services. We have developed a visual language environment for ViTABaL-WS using a meta-tool, Pounamu [19]. Figure 1 (a) shows ViTABaL-WS in use modelling the composition and orchestration of several web services using its domain-specific *tool abstraction*-based visual language. The Pounamu specification of this tool includes definitions of the tool meta-model, shapes, connectors, graphical views of the model, editing behaviour and code generation (BPEL4WS encoding of the model). Figure 1 (b) shows an example of these definitions. Key requirements for domain-specific visual language tools that we and others have identified include:

- Providing users a rich set of graphical modelling primitives and diagram editing support, producing a consistent visual "metaphor" for the domain;
- Having an underlying model shared by all diagrams (model views) with a well-defined meta-model;
- Ability to display and edit different parts of the model in different diagrams, with consistency management;
- Support for complex event handling and information import/export, with complex editing and model constraint rules, code generation and tool integration;
- Scalable persistency and collaborative work support, including asynchronous version management and synchronous diagram editing and awareness support;
- Tight presentation, data and control integration with related tools e.g. IDEs, CASE tools.

Key research questions in the area of specifying and building domain-specific visual language tools include:

- How can DSVL tools be described at a high level of abstraction avoiding the low-level coding needed in most frameworks? This includes meta-model, visual notation, multi-view and editing constraints, import/export and code generation features.
- How can we generate high-quality DSVL tools, in terms of end-user needs and satisfaction with the tool and range of tool capabilities that can be realised? Ideally the same DSVL tool specification could be used to generate different tool realisations e.g. as plug-ins for VisualStudio, Eclipse, Enterprise Architect, or providing web-based user interfaces or using different code generation technologies.
- Can we generate DSVL tools from a suitable high-level specification that can compete with hand-implemented tools, but with much easier tool modification and extension?

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 CSCW support mechanisms. Similar editors can be built with Eclipse’s Graphical Editing Framework (GEF). Such editors have the advantage of seamless integration into the (open source but commercial quality) Eclipse IDE, can directly use Eclipse’s code generation and other plug-ins, and can be readily packaged and deployed.

Unfortunately GEF is very complex and while graphical editors built with it are high quality, developing and maintaining these is challenging [6],[27]. Developers do not directly obtain model save/load, multi-view and multi-user editing support, having to code these using other (complex) Eclipse frameworks. Also, while Pounamu allows non-experts and even non-programmers to easily develop exploratory domain-specific visual language modelling tools, only expert Eclipse developers can reasonably be expected to develop GEF-based visual language editors. Ideally what we want is to realize GEF-quality editors as plug-ins to the Eclipse environment but using Pounamu-style meta-tool capabilities.

3. Related Work

Three main approaches exist for the development of the type of visual, multiple view and multi-user environment exemplified by ViTABaL-WS: the use of reusable class frameworks; visual language toolkits; and diagramming or CASE meta-tools.

General purpose graphical frameworks provide low-level yet powerful sets of reusable facilities for building diagramming tools or applications. These include MVC [13], Unidraw [30], COAST [29], HotDoc [1] and Eclipse’s GEF [12]. While powerful they typically lack abstractions specific to multi-view, visual language environments, so construction of tools is time-consuming. For example, supporting multiple views of a shared model in GEF requires significant programming effort. Special purpose frameworks for building multi-user, multi-view diagramming tools include Meta-MOOSE [7], JViews [9], and Escalante [21]. These offer reusable facilities for visual language-based environments, but still require detailed programming and a compile/edit/run cycle, limiting their ease of use for exploratory development.

Many general-purpose, rapid development user interface toolkits have been developed to reduce the edit/compile/run cycle. Many, including Tcl/Tk [32], Suite [3], and Amulet [24], are suitable for visual language-based tool development. They combine rapid application development tools and programming extensions. However, as they lack high-level abstractions for visual, multi-view environments and tool integration, more targeted toolkits have been produced to make such development easier. These include Vampire [20], DiaGen [22], VisPro [34], JComposer [9], PROGRES [28] and DSLTools [8]. Some of these use code generation from a specification model, e.g. DiaGen and JComposer. Others, e.g. PROGRES and VisPro, use formalisms such as graph grammars and graph rewriting for high-level syntactic and semantic specification of visual tools. Code generation approaches suffer from similar problems to many toolkits: an edit/compile/run cycle and difficulty in integrating third party solutions. Formalism-based visual language toolkits may limit the range of visual languages supported and are often difficult to extend in unplanned ways.

Meta-tools provide an IDE for developing other tools. These include KOGGE [4], MetaEdit+ [14], MOOT [26], GME [16], MetaEnv [1] and IPSEN [15]. Usually they aim for a degree of round-trip engineering of the target tools. Typically they provide support for their target domain environments, but are limited in their flexibility and integration with other tools [31]. These problems occur at presentation (interface) and data/control levels.

As the Eclipse environment has gained popularity a number of tools have been developed or proposed for generating Eclipse graphical editors. These include the proposed Graphical Modelling Framework (GMF), combining the EMF (model) and GEF (graphical) frameworks, the Merlin graphical editor generator, which uses EMF data models to generate a basic editor suite, and generation of Eclipse editors from graph grammar formalisms [6], supporting simple diagram notations. Unfortunately these generators provide only limited editor functionality, use limited formal expression of editor functionality, or infer editor functionality and generate simplistic graphical symbols and editors. In summary:

- Framework and UI toolkit-based approaches are very powerful and produce high-quality DSVL tools but require detailed programming and class framework knowledge;
- Most visual language tool-kits use higher-level models and formalisms e.g. graph grammars, but are limited in their expressive power and tend to produce limited visual notations and editor functionality;
- Few of these development tools support round trip engineering and live, evolutionary development. Regeneration of code can be a large problem when integrating backend code. Most have limitations with regard to integration with other tools.

4. Our Approach

We have developed Marama, a set of Eclipse plug-ins that read high-level Pounamu meta-tool specifications and

realize multi-view, multi-user graphical editors in the Eclipse IDE. Figure 2 shows the approach we use 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). We used GEF 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).

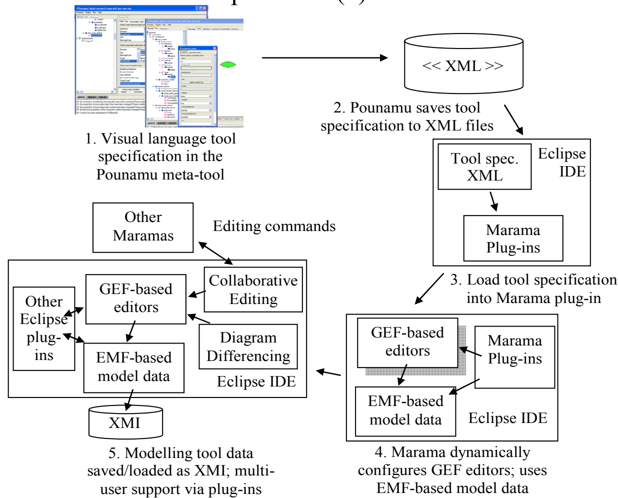

Figure 2. The Marama approach to realizing Eclipse-based visual language tools.

In addition, we have developed plug-ins to support multi-user diagramming, including diagram differencing for asynchronous work and collaborative diagram editing for synchronous work. These plug-ins use the GEF Command pattern and EMF Notification pattern standard plug-in integration mechanisms. 3rd party Eclipse plug-ins can also interact in standard ways with Marama's user interface and its diagram and model data, using standard Eclipse IDE, EMF and GEF plug-in interfaces. Marama supports dynamic update of tools by allowing a modified Pounamu tool description to be re-read, even while a tool is in use.

5. Architecture

Despite large numbers of DSVL toolkits and tools being produced over many years of research, there is no currently agreed way of specifying such tools at high levels of abstraction. A wide range of approaches have been used e.g. graph grammars (often resulting in very constrained editing functionality) [6], [35]; symbol grammars [4], [14]; object model-based generation [7], [9]; XML models [8], [26]; and custom low-level code solutions [29], [30].

Our Pounamu meta-tool represents tool specifications as a set of meta-model entities and associations; view shapes, connectors and dialogues; view types relating shapes and connectors to meta-model entities and associations; and event handlers for complex editing control and model constraint implementation. Pounamu tool specifications are hierarchically organised as a set of related XML files. Figure 3 illustrates their basic structure. This includes a tool project file specifying the configuration of a particular tool, the meta-model, shape and connector type specifications (visual icons for diagrams), view types (diagram types), and event handlers. The meta-model is a form of extended entity-relationship model. The shape and connector type specifications describe abstract GUI components that make up arbitrarily complex visual notational symbols for the tool. Event handlers are Java code, reused from a library or written by hand via a Pounamu meta-tool interface that is plugged into the running tool to implement complex editing and model behaviour, constraints, and code generation. View types are the diagram types supported by the tool: allowed shapes, connectors, and event handlers for each diagram type and mappings from them to model entities and associations.

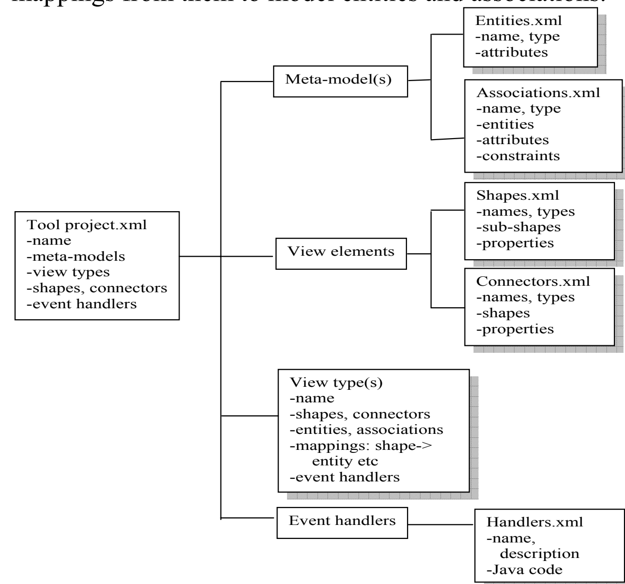

Figure 3. Structure of Pounamu tool specifications.

Figure 4 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).

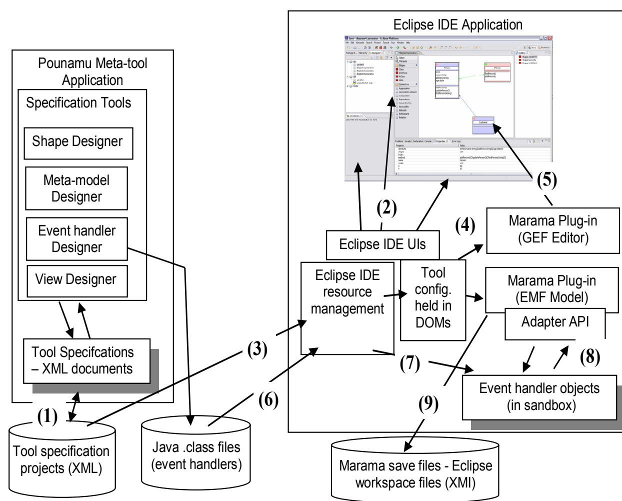

Figure 4. The architecture of Marama.

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.

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. Marama can be extended with plug-ins that enhance its capabilities using standard Eclipse mechanisms. For example we have developed diagram differencing and collaborative editing plug-ins to support asynchronous and synchronous collaboration [22]. In addition, Marama tools can use other Eclipse capabilities via their event handlers and Marama-Pounamu object wrappers. For example, we have used the EMF Java Emitter Templates (JET) toolset to implement code generation capabilities.

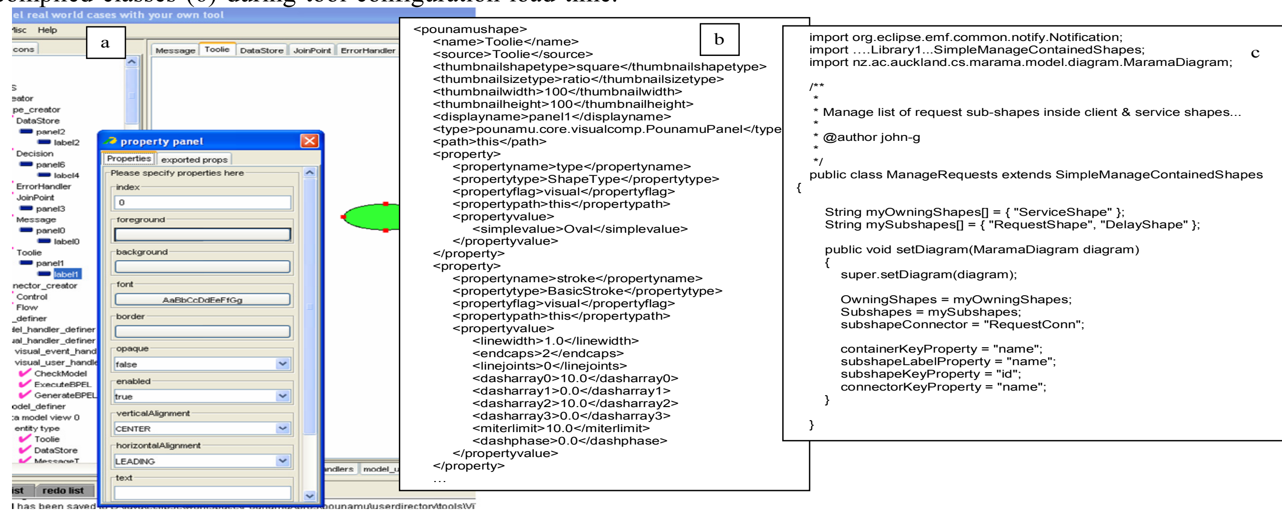

Figure 5. Part of a tool shape specification in Pounamu and part of its XML and event handler encodings.

6. Example Usage

In this section we illustrate the use of Marama to realise the ViTABaL-WS web service orchestration tool described in Section 2. One of the Pounamu meta-tool views describing a shape specification for ViTABaL-WS is shown in Figure 5 (a), along with part of the XML save file for this shape definition Figure 5 (b). Shapes have a set of properties and sub-shapes and describe the appearance of a notational symbol for use in a view type (i.e. diagram type). When this tool project is opened by Marama, the shape definition and other tool specification files are loaded and used to configure the Marama view data and GEF-based graphical editors for the tool.

Pounamu provides facilities to specify model and view meta-models (entities, associations, shapes, connectors); view types (collections and shapes and connectors and their mapping to a model); and event handlers [36]. Event handlers are Java scriptlets used to specify: editing constraints e.g. keep shape within another shape/resize shape when another is resized etc; model constraints e.g. enforce relationship arity constraint/auto-create entity on another entity creation; recalculate dependent values; and import/export e.g. code generation, import data from XMI format into tool. Many such event handlers are reusable from a library requiring little or no Java coding, an example shown in Figure 5 (c). Complex editing operations e.g. replace collection of shapes/connectors with another set are implemented as event handlers invoked by a pop-up menu or when another event occurs. Pounamu supports definition of shapes-within-shapes allowing arbitrarily complex renderings of model information in diagrams. Limited preferences are supported for generating a Marama editor from a Pounamu tool specification e.g. the editing palette items, kind of property sheet and outline views provided for generated editor. These are currently specified using an event handler to set properties on view initialisation.

A Marama user starts Eclipse and then uses the standard Eclipse IDE resource browser to create (or reopen) Marama model projects and diagrams. Figure 6 (1) shows a user has created a new ViTABaL-WS model project in Marama by selecting the Pounamu meta-tool tool project save file with the resultant meta-model viewer opened in Marama. This viewer is a GEF editor displaying the imported Pounamu meta-model entity and association types. It supports the user rearranging (but currently not defining) meta-model entity and association shapes and model instance data browsing via a property sheet viewer below. In this example, the Flow association type is selected and the property sheet shows information about the five instances of this association that are in the model.

The user may reopen existing Marama diagrams or create new ones. In Figure 6 a ViTABaL-WS web service composition diagram is edited in Marama. This uses a GEF editor Marama has configured using the view type specification files generated by the Pounamu meta-tool. The available shape and connector types are accessed via a palette (a); shapes and connectors can be directly manipulated in a canvas (b); properties of a selected shape or connector can be edited using the standard Eclipse property viewer (c); and tool bars and pop-up menus used to manipulate diagram content (d). A hierarchical outline view is also provided (e). Marama diagrams behave like other GEF editors using standard Eclipse drag and drop, copy/paste, printing etc. The resource view (f) shows model projects and diagrams available; these can be organised into Eclipse projects and folders.

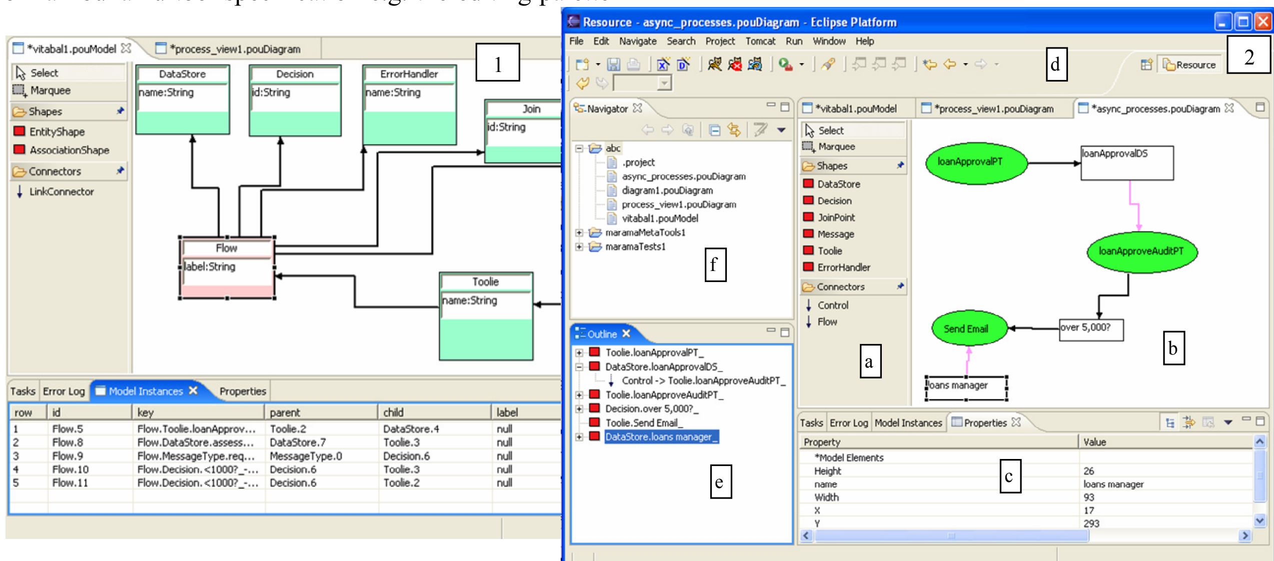

Figure 6. ViTABaL-WS views being edited in Marama.

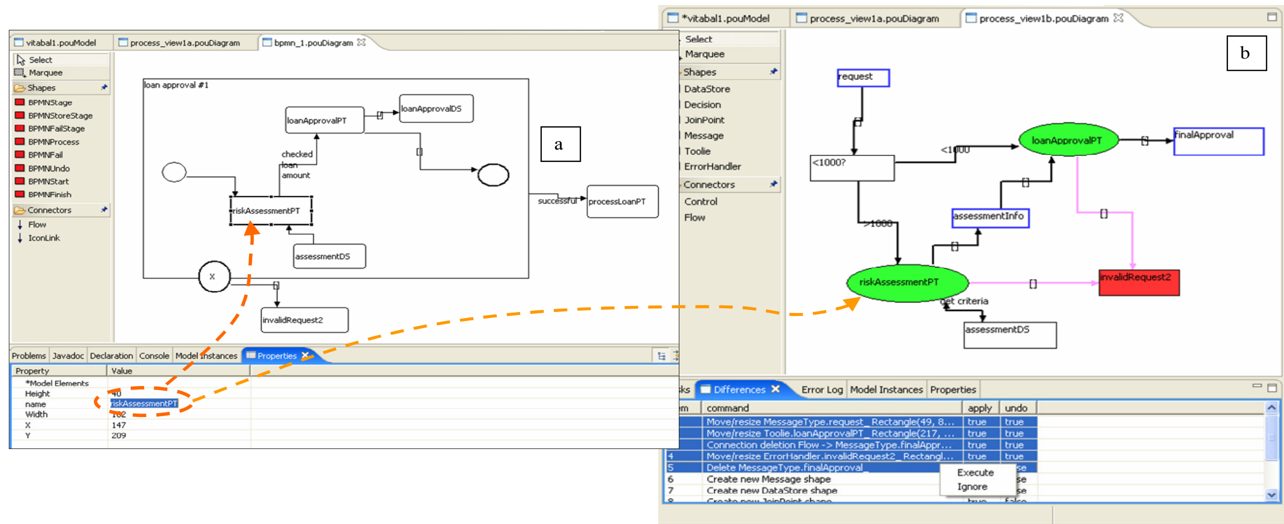

Figure 7. (a) Multi-view consistency and (b) diagram differencing for collaborative work.

Multiple views are supported and are kept consistent with one another via an EMF model. Figure 7 (b) shows a second ViTABaL-WS view with common elements to those in Figure 6. This multiple view support extends to consistency between multiple diagram types with a common EMF model. In the example in Figure 7, the user has renamed a service in a BPMN (Business Process Modelling Notation) view (a) of the web service composition, resulting in changes to the name in ViTABaL-WS composition view (b).

Figure 7 (b) also shows our diagram differencing support. In earlier work we developed a number of extensions to Pounamu to support CSCW, thin-client diagram editing, and code generation and model export facilities [36]. These used a custom plug-in extension mechanism and web service-based API for Pounamu. We have developed exemplar plug-ins for Marama to support diagram differencing and merging and collaborative editing for group work, using the same algorithms as used in [22], but using Eclipse's standard plug-in extension and integration mechanisms to extend Marama. As with the Pounamu plugins, the support is generic, working with any Marama-generated diagram type.

The diagram differencing and merging plug-in extends the pop-up menu for Marama allowing the user to compare two versions of the same diagram. In Figure 7, the user has checked-out a diagram ("process_view1a" – middle view tab at top) from a shared CVS repository, made changes to it, and then wishes to compare the diagram to the other version that has been concurrently modified by another user. The user checks-out a read-only copy of this alternate version ("process_view1b") from the repository, opens it, selects the Diagram differencing command added to the Marama editor pop-up menu and the diagram to compare to, and a list of Marama editing commands are generated (bottom). Running these commands on the diagram will convert it into the alternative version. The user may elect to run all or some of the commands, doing a full or partial merge of changes.

To support external tool integration with a Marama editor, a convenient mechanism is to use code generation via Java Emitter Templates (JET). For example, in Figure 8 (a) the user has elected to generate a BPEL4WS specification (right) from the Marama ViTABaL-WS model (left). To achieve this, an event handler is invoked when the user selects a "Generate BPEL" pop-up menu item and the event handler calls a JET-generated translator. This translator converts the Marama ViTABaL-WS business process model data into a corresponding BPEL4WS specification. Selecting "Execute BPEL" deploys this BPEL4WS code to a workflow engine and run (we used IBM's BPWS4J) as outlined in Figure 8 (b). Implementing such back end code generators is simple and straightforward using the leverage provided by the JET framework, contrasting with the need for Java or complex XSLT code to do the same in Pounamu.

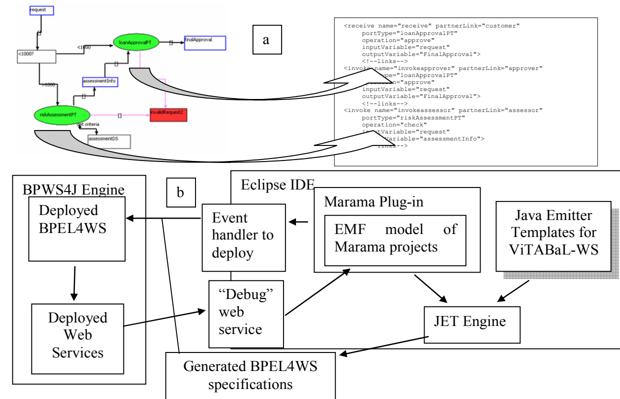

Figure 8. JET-based BPEL4WS generation.

7. Design and Implementation

To realise Marama we used a number of Eclipse frameworks to implement a dynamic interpreter for our Pounamu-generated DSVL tool specifications. Figure 9 illustrates the structure of Marama. Tool specifications are loaded from Pounamu XML files into Document Object Model (DOM) structures. A set of Marama meta-model 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 meta-model 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 event handler specifications –Java scripting code - 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. We chose to use a "sandbox" approach 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.

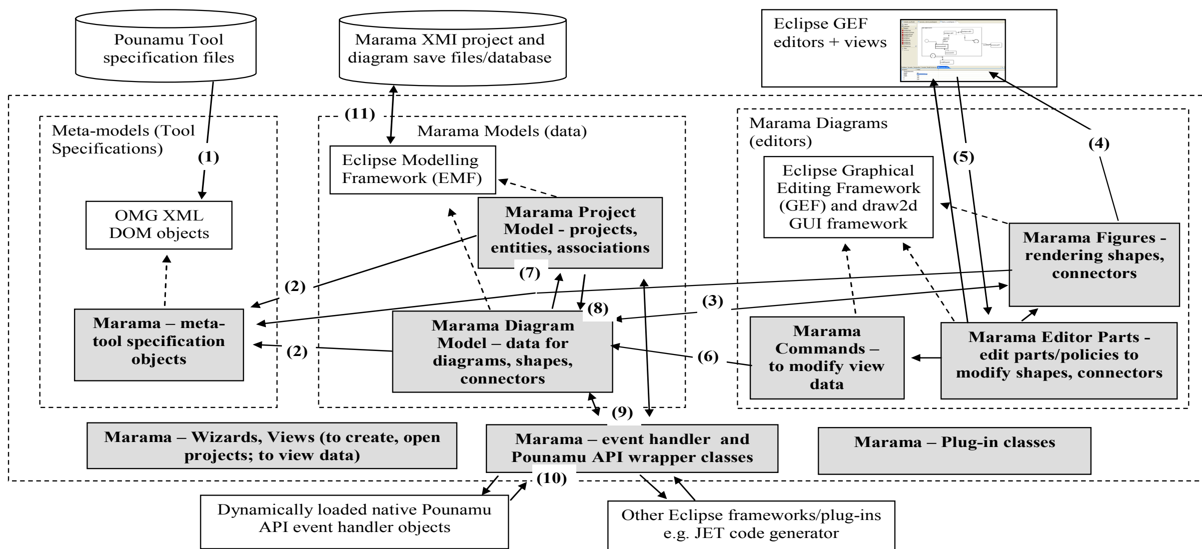

Figure 9. Implementation of Marama.

8. Discussion

We have developed a wide range of multi-view diagramming tools using Pounamu [36]. These have all been very easily ported to Marama by importing their Pounamu tool specifications. Existing import/export and tool integration mechanisms have been preserved by using the Marama adaptor classes to sandbox execution of the Pounamu compiled event handlers. Enhancements to the tools have been possible using Marama's EMF object interfaces and other Eclipse plug-ins. For example, the performance test bed generator's XSLT-based code generator was replaced with an improved Eclipse JET-based code generator.

From a usability perspective, the tools rate well compared to Pounamu equivalents and other comparable Eclipse-based tools. Our Marama editors are robust, have a consistent look and feel, and, based as they are on the Eclipse GEF and EMF frameworks, have good usability characteristics. In contrast to other GEF generating toolkits [6] the look and feel of Marama editors is very similar to that of a hand-implemented GEF/EMF editor. In many respects they are better as multiple views are implicitly supported and the tool specification can be changed on-the-fly via Pounamu. The diagram editors integrate naturally into the Eclipse IDE, providing excellent presentation integration. They are readily extended via the Eclipse extension point approach which provides good control integration. The Eclipse XMI save/load and JET back-ending support allow straightforward import/export capabilities to be developed providing a simple data integration mechanism. Complex data integration is possible via the EMF model.

Of more importance, the process of developing the tools is tremendously simpler than coding tools using the Eclipse frameworks directly. Reusing Pounamu to generate Marama tools means we have not had the overhead of developing a new meta-tool nor the cost of developing tools from scratch. These are obvious advantages but there are drawbacks. Pounamu and Eclipse must both be running for tool development and integration between the two is weak, only via data integration, with no presentation or control integration. There are also limitations on using EMF and GEF facilities that "built from scratch" editors would avoid. We are working to address these limitations. In particular we are developing Eclipse-based meta-tools for Marama, bootstrapping their development by defining the meta-tools in Pounamu, importing their specifications into Marama, and integrating and extending them using Eclipse integration support and customized handler code.

Stepping up a level, we see this project as a step towards a generic DSVL tool specification interchange standard. We have demonstrated that Pounamu tool specifications can be interpreted in an Eclipse based environment providing multi-platform yet consistent implementations of the same tool. With implementation of the Marama meta-tools, we can also specify tools in Marama and realize them in Pounamu. An obvious step beyond that is to leverage the Microsoft DSL Tools [8] to generate Visual Studio based realizations (including the meta-tools). We can then generalize from the 3 examples to a tool specification interchange standard. Our work with Marama/ Pounamu suggests this will have the following components:

- Icon and connector types, their attributes, and interaction mechanisms. This appears straightforward to develop a common standard for as most modern UI toolkits provide similar widgets and functionality.
- Common model element types, attributes and relationships. This also appears relatively straightforward. Most model-view frameworks support something like an Extended Entity Relationship model for the shared repository.
- Views, their icons and connector types, and the mappings from view to model elements. The mappings are somewhat more problematic here. Simple 1:1 view-model mappings are straightforward, but complex mappings may require significant hand coding in different frameworks
- Additional behavioural elements, including event handlers and constraints. These are most problematic as they are typically hand coded in most frameworks.

We are currently working on general solutions to the final two cases. This relates closely to the proposed Eclipse Graphical Modelling Framework (GMF) approach using domain models (EMF) and visual DSL models (GEF) [5]. For the behavioural specification, we are working on a generalised visual framework for specifying event based systems [17]. This integrates several event specification languages with a common intermediate model compileable to a variety of implementations (eg OCL, Java, RuleML). The view mapping problem is being solved in a similar manner, building on our prior work in mapping specification systems [10]. We are developing a meta-toolset for Marama within Eclipse, providing a single integrated toolset for both specification and generation of Marama tools. Our eventual aim is to be able to generate GMF and Visual Studio DSL tool specifications from our Marama-implemented meta-tools to leverage others' frameworks for building DSVL tools.

9. Summary

We have described an approach for generating Eclipse-based multi-view graphical editors suitable for domain specific visual language implementation. This reuses an existing meta tool, Pounamu, to specify the

underlying model and graphical editor, together with Marama, a set of Eclipse plug ins that extend the Eclipse GEF and EMF frameworks, to interpret the tool specifications and realize them as a high-quality Eclipse tool. Tools realized are well integrated within Eclipse and may use standard Eclipse extension mechanisms and backend code generation facilities for enhancement and integration with other Eclipse tools. We have used this approach to implement a wide range of Eclipse-based domain specific language tools. In each case we have been able to rapidly implement a complex Eclipse-based tool, with typically a several hundred-fold increase in productivity over coding the tool with the standard GEF and EMF frameworks. The generated tools show excellent usability and robustness.

10. REFERENCES

[1] Buchner, J., Fehnl, T., and Kuntsmann, T., HotDoc a flexible framework for spatial composition, *Proc. 1997 IEEE Symp. on Visual Languages*, IEEE, 92-99.

[2] Burnett. M., Goldberg, A., Lewis, T. (eds) *Visual Object-Oriented Programming,* Manning, Greenwich, CT, 1995.

[3] Dewan, P. and Choudhary, R. 1991. Flexible user interface coupling in collaborative systems, *Proc. of ACM CHI'91*, ACM Press, April 1991, pp. 41-49.

[4] Ebert, J, Süttenbach, R, and Uhe, I, Meta-CASE in practice: a case for KOGGE, *Proc. CAiSE'97, LNCS 1250*, 203-216.

[5] Eclipse.org, www.eclipse.org

[6] Ehrig, K., Ermel, C. Hänsgen, S. and Taentzer, G. Generation of Visual Editors as Eclipse Plug–Ins, *Proc. 2005 ACM/IEEE Automated Software Engineering*.

[7] Ferguson R, Parrington N, Dunne P, Archibald J, Thompson J, MetaMOOSE-an object-oriented framework for the construction of CASE tools, *CoSET'99*, LA, May 1999.

[8] Greenfield, J., *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, http://msdn.microsoft.com/vstudio/DSLTools/ 2004

[9] Grundy, J.C., Mugridge, W.B. and Hosking, J.G. Constructing component-based software engineering environments: issues and experiences, *Information and Software Technology*, Vol. 42, No. 2, pp. 117-128.

[10] Grundy, J.C., Mugridge, W.B. and Hosking, J.G. Visual specification of multi-view visual environments, *Proc IEEE VL'98*, Halifax, Nova Scotia, Sept 1998, pp. 236-243.

[11] Grundy, J.C, Hosking, J.G., Amor, R., Mugridge, W.B., Li, M. Domain-specific visual languages for specifying and generating data mapping systems, *JVLC* 15, 2004, 243-263.

[12] Hudson, R. *Create an Eclipse-based application using the Graphical Editing Framework*, www-128.ibm.com/ developerworks/opensource/library/os-gef/

[13] Krasner, G.E. and Pope, S.T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *JOOP*, vol. 1, no. 3, pp. 26-49, Aug. 1988.

[14] Kelly, S., Lyytinen, K., and Rossi, M., Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, *Proc. of CAiSE'96*, LNCS 1080, 1996.

[15] Klein, P. and Schürr, A. Constructing SDEs with the IPSEN Meta Environment, *Proc. SEE'97*,1997, pp. 2-10.

[16] Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., Karsai G.: Composing Domain-Specific Design Environments, *Computer*, 44-51, Nov, 2001.

[17] Liu, N. Visual Languages for Event Integration Specification, *ICSE'06 Doctoral Symposium*, May 2006.

[18] Liu, N., Hosking, J.G. and Grundy, J.C. A Visual Language and Environment for Specifying Design Tool Event Handling, *Proc. VL/HCC'2005*, Dallas, Sept 2005.

[19] Liu, N., Grundy, J.C. and Hosking, J.G. A Visual Language and Environment for Composing Web Services, *Proc. 2005 IEEE/ACM ASE*, Long Beach CA, Nov 7-11 2005.

[20] McIntyre, D.W., Design and implementation with Vampire, *Visual Object-Oriented Programming*. Manning Publications, Greenwich, CT, USA, 1995, Ch 7, 129-160.

[21] McWhirter, J.D. and Nutt, G.J. Escalante: An Environment for the Rapid Construction of Visual Language Applications, *Proc. VL '94*, pp. 15-22, Oct. 1994.

[22] Mehra, A., Grundy, J.C., Hosking J.G., A generic approach to supporting diagram differencing and merging for collaborative design, *2005 IEEE/ACM ASE*.

[23] Minas, M. and Viehstaedt, G. DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams, *Proc. VL '95,* 203-210 Sept. 1995.

[24] Myers, B.A., The Amulet Environment: New Models for Effective User Interface Software Development*, IEEE TSE*, vol. 23, no. 6, 347-365, June 1997.

[25] Pautasso, C. and Alonso, G. Visual Composition of Web Services, *Proc. 2003 IEEE HCC*, 2003, pp. 92-99.

[26] Phillips C, Adams S, Page D, Mehandjiska D, The Design of the Client User Interface for a Meta Object-Oriented CASE Tool, *Proc TOOLS 1998,* Melbourne, p156-167.

[27] Rayside, D., Litoiu, M., Storey, M.A.D., Best, C., Lintern, R. Visualizing Flow Diagrams in WebSphere Studio Using SHriMP Views, *Information Systems Frontiers* 5 (2), 2003.

[28] Rekers, J Schuerr, A Defining and parsing visual languages with layered graph grammars, *JVLC*, 8, 27-55, 1997.

[29] Shuckman, C., Kirchner, L., Schummer, J. and Haake, J.M. 1996. Designing object-oriented synchronous groupware with COAST, *Proc ACM CSCW*, Nov. 1996, pp. 21-29.

[30] Vlissides, J.M. and Linton, M., Unidraw: A framework for building domain-specific graphical editors, *Proc. UIST*'89, ACM Press, pp. 158-167.

[31] Wasserman, A.I., Tool Integration in Software Engineering Environments, *Proc. SEE'89*, IEEE, pp. 137-149.

[32] Welch, B. and Jones, K. *Practical Programming in Tcl and Tk,* 4th Edition, Prentice-Hall, 2003.

[33] Younas, M.a.I., R. Developing Collaborative Editing Applications using Web Services. *Proc. 5th Int. Workshop on Collaborative Editing, Helsinki*, Finland, Sept 15, 2003.

[34] Zhang, D.-Q. and Zhang, K. VisPro: A Visual Language Generation Toolset*, Proc. VL'98*, pp. 195-202, Sept. 1998.

[35] Zhang, K. Zhang, D-Q. and Cao, J. Design, Construction, and Application of a Generic Visual Language Generation Environment", *IEEE TSE*, 27,4, April 2001, 289-307.

[36] Zhu, N., Grundy, J.C. and Hosking, J.G., Pounamu: a meta-tool for multi-view visual language environment construction, *Proc. VL/HCC 2004*, 254-256.