A Visual Language and Environment for Specifying Design Tool Event Handling

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

Abstract

We describe a new visual language for event handling specification and its incorporation into Pounamu, a meta-tool for building diverse visual design environments. Our visual language provides end users ways to express event handling mechanisms via visual specifications.

Keywords: Visual Language, Event Handling, Meta Tool

1 Introduction

Visual design tools have many applications, including software design, engineering product design, E-learning, data visualisation, and tourism. We have used the event-driven nature of visual design tools as a vehicle to provide end users with a domain specific visual language, we call *Kaitiaki* [1], with which to specify behaviours for their diagramming tools. We have incorporated this visual language into our Pounamu meta-tool [9] to provide end users with little programming background, a mechanism to detect events and specify simple or complex actions to take.

A variety of approaches have been used to support reconfiguration of diagramming tools, including direct modification of tool code using an API [4], scripting [7], programming by demonstration [8], and Event-Condition-Action rule based languages [1, 5, 6]. The approach we describe here extends from the latter. Pounamu currently adopts the first approach through user defined Java *event handlers* which are compiled on the fly.

2 Kaitiaki

By analysing Pounamu event handlers from a range of tools key requirements identified for our *Kaitiaki* visual event handler designer were:

- A need to represent key "building blocks" of state query, data filtering and state modification (actions).
- A need to represent and query event objects/attributes; Pounamu tool state objects
- A need to represent "data" propagation between event, query, filter and action representations.
- A need to represent iteration and conditional data flow.

[1] *Kaitiaki* is the Maori word for handler, or guardian

The metaphor used by *Kaitiaki* is thus an Event-Query-Filter–Action (EQFA) model articulated as "When this event happens, I want these changes made to these things". This is loosely based on the Serendipity process modelling language [3]. An overview of the main constructs and the predefined primitives of *Kaitiaki* is shown in Table 1 with example *Kaitiaki* event handlers shown in Figure 1.

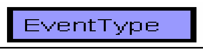
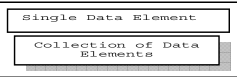
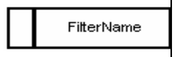
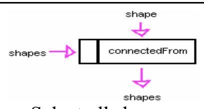
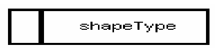
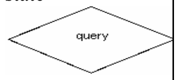
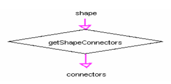
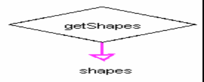
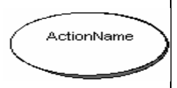
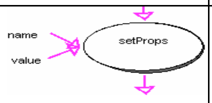
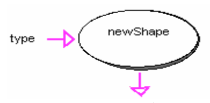
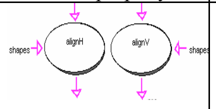
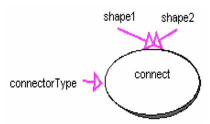
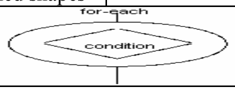
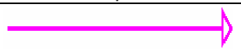
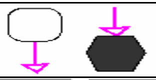
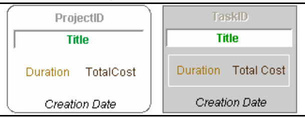
Event representation		
Abstract Pounamu state representation		
Filter 	 Select all shapes connected to a shape	 Select shapes of given or test type of single data element input
Query on a tool's state 	 Obtain all connectors connected to a shape	 Obtain all the shapes in the modeller panel
state changing action 	 Set a value to a named property	 Create a new shape
	 Horizontally/vertically align a shape with other aligned shapes	 Connect two shapes using a connector
Iteration		
Data propagation link		
Data flow ports in and out		
Concrete specification of Pounamu model elements (state)		

Table 1. Key visual constructs and building blocks

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. 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 parameter values are available (data push), but if no data flows to a construct, it is invoked on demand when all parameters to a subsequent flow element have a value (data pull).

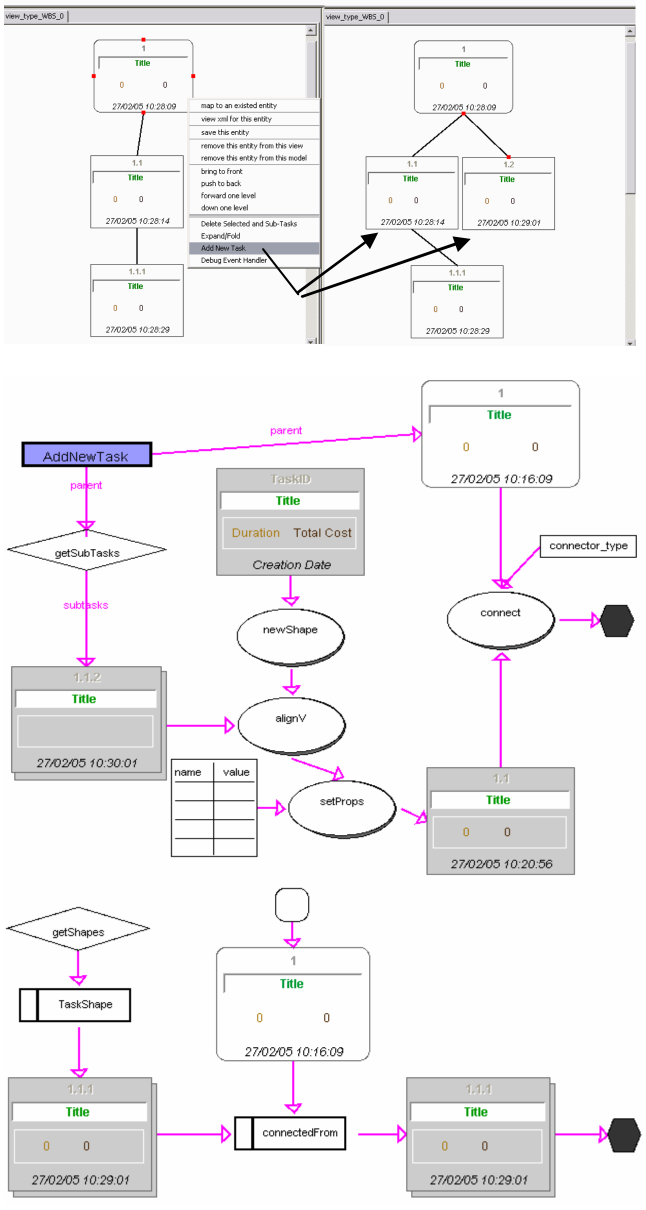
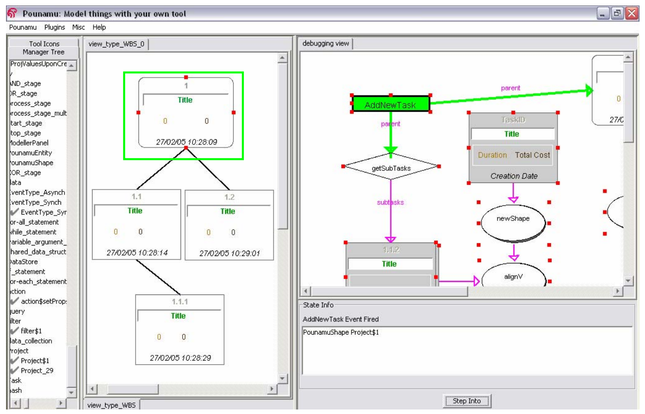


Figure 1: Work breakdown tool (top), Kaitiaki event handler (centre), and packaged query (bottom)

Figure 1 (top) shows a Pounamu implemented work breakdown tool. Users create a new task by a right-click on an existing task; the new task is made a child of the existing task and a link drawn between the old and new tasks. The new subtask and its siblings are then aligned. The Kaitiaki event handler specification for this task is shown in the centre. This event handler responds to a user defined menu trigger event called *AddNewTask*. The *getSubTasks* query (a packaged query, also defined using Kaitiaki, and shown at the bottom) locates existing subtasks of the currently selected project/task shape (*parent* input); the *newShape* action creates the new task shape; the *alignV* action does a vertical alignment of the new task shape with the other existing subtasks; the *setProps* primitive then sets default properties for the newly created task shape; and the *connect* primitive connects the new task shape with its parent shape using a specified connector type, now the event handler leads to a final stage, i.e. the end of the event handler specification.

The packaged *getSubTasks* query, shown at the bottom of Figure 1, 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 (data pull) is invoked. The *TaskShape* filter selects all shapes that are of the *TaskShape* 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 a set of subtasks of the parent task. This query is invoked in the event handler in the centre, but can be reused by other event handlers. Actions and filters can similarly be specified and reused.



AddNewTask Event Fired

PounamuShape Project$1

Figure 2: Visualizing execution of a visual event handler

We have also developed a visual debug viewer which dynamically annotates an event handler specification view during execution as shown in

Figure 2. This includes the visualization of EQFA element invocation (by flashing the corresponding graph

node) and visualization of data propagation (by highlighting the dataflow path). The traditional "debug and step into" metaphor is used and step-by-step visualization controlled by menu command.

3 Evaluation

We have carried out a Cognitive Dimensions [2] evaluation of our visual event specification language and prototype environment to gauge its effectiveness. Key issues seen include *Kaitiaki*'s Abstraction Gradient, Closeness of Mapping, Error Proneness, Progressive Evaluation support, Viscosity, requirement for Hard mental operations and use of Secondary Notation.

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, Pounamu 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 generated and visual debugging 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.

4 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.

We are exploring a programming by example extension to allow users to make several changes to an existing modelling tool view and generate actions and data flow connections between actions in an event specification view. These would then be tailored and abstracted via the addition of queries and filters to make a generic event handler. The dataflow metaphor used to compose a specification results in interesting potential concurrency issues if parallel flows are defined. We are examining extra synchronisation constructs to manage this. In addition, automatic layout of a event handler specifications may be useful to improve a users ability to show/hide/ collapse parts of a specification to manage size and complexity.

References

1. Costagliola, G., Deufemia, V., Ferrucci, F., Gravino, C.: The Use of the GXL Approach for Supporting Visual Language Specification and Interchanging. *Proc HCC'02*, Arlington, Virginia, 2002, pp131-138.
2. Green, T. R. G., Burnett, M. M., A Ko, J., Rothermel, K. J., Cook, C. R., and Schonfeld, J., Using the Cognitive Walkthrough to Improve the Design of a Visual Programming Experiment, *Proc VL2000*, 172-179
3. Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. An architecture for decentralised process modelling and enactment, *IEEE Internet Computing*, 2:5, 53-62, 1998.
4. Kelly, S., Lyytinen, K., and Rossi, M., Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, in *Proceedings of CAiSE'96*, LNCS 1080, Springer-Verlag, Crete, Greece, May 1996.
5. Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., Karsai G.: Composing Domain-Specific Design Environments, *Computer*, 44-51, Nov, 2001.
6. Lewicki, D. and Fisher, G. VisiTile - A Visual Language Development Toolkit, Proc VL'96, Boulder, pp. 114-121.
7. Myers, B.A., The Amulet Environment: New Models for Effective User Interface Software Development, *IEEE TSE*, vol. 23, no. 6, 347-365, June 1997.
8. Smith, D.C., Cypher, A. and Spohrer, J. KidSim: programming agents without a programming language, Communications of the ACM, vol. 37, no. 7, pp. 54 – 67.
9. Zhu, N., Grundy, J.C. and Hosking, J.G., Pounamu: a meta-tool for multi-view visual language environment construction, *Proc VL/HCC'04*, pp. 254-256.