In Proceedings of Visual Analytics in Software Engineering, Workshop at 2009 IEEE/ACM Automated Software Engineering Conference, Auckland, New Zealand, 16 Nov 2009.

# Visualising Event-based Information Models: Issues and Experiences

Karen Li, John Grundy, John Hosking, Lei Li
Departments of Computer Science and Electrical and Computer Engineering,
University of Auckland, Private Bag 92019, Auckland, New Zealand
{karen, john-g, john, l.li}@cs.auckland.ac.nz

## ABSTRACT
We describe challenges in visualising event-based system specification and execution and illustrate how we address these from our experience developing a set of notations and tools, the Marama meta-tool platform.

## 1. INTRODUCTION
The event-driven software paradigm is widely used due to its flexibility for constructing dynamic system interactions. Event-driven systems feature publish/subscribe relationships between software components and loosely-coupled system behaviours [10]. Such systems incorporate *events*, *conditions* ("filters"), and *action(s)* which may modify system state. The OMG, Microsoft and Sun all advocate event-driven systems in their architectures and technologies. Some examples of event-driven systems are:

- Workflow management systems, where process-related events cause rules to fire the enactment of process stages;
- Database systems, where events trigger relational queries to execute and maintain integrity;
- Distributed computing, where distributed user actions are events to which the applications react;
- Graphics and modelling frameworks, where event-based interaction data are captured and event handlers are used to realise model/view level constraints;
- Software tools, where events support data mappings, import/export/code generation, and tool integration/extension [26].

Despite their ubiquity specifying and understanding the execution of event-driven systems can be very difficult due to their complex behaviours. Appropriate visualisation support can help mitigate abstraction and facilitate end user specifications. Approaches for specifying event-handling include scripting, Event-Condition-Action (ECA) rules, and spreadsheets. Current approaches require users to master a programming language and API, which is unsuitable for non-programmer end users. Visual event-based authoring approaches minimise design and implementation effort and improve understandability [2, 5, 7, 9, 11].

We have identified a set of issues from our research in specifying and visualising event-based systems, particularly for non-programmer end users. These include lack of: suitable visual descriptions of event-based architectures; appropriate abstractions and separation of concerns; context-aware navigation; and runtime visualisation reusing design-level abstractions. We have used several domain-specific visual languages with different visual metaphors (Spreadsheet, Event-Query-Filter-Action (EQFA) and Tool Abstraction (TA) [8]) to support event integration specification and visualisation of event propagation. After surveying key related work in the following section we elaborate on each of our identified issues and illustrate our experience in coping with them in a variety of ways.

## 2. RELATED WORK
There has been much recent research looking at visualisation support for event-based specifications. However many approaches have focussed on visualising structures with few tackling the visualisation of event-based dynamic behaviours.

Approaches to visualise event-based behaviours include declarative rules (Rule Graph [20] and Reaction RuleML [22]), functions (Haskell [12]) and constraints (MIC [16], MetaEdit+ [25]), states (Petri Net [19], Event and Time Graph [1], UML State Diagram [21]) and flows (e.g. BPEL4WS[14], Biopera [23] and UML Activity Diagram [21]), and program-by-demonstration (PBD) (KidSim [24] and Alice [3]). Declarative semantics of rule /constraint-based techniques allow users to ignore implementation details and concentrate on high level relationships. Many such approaches use textual rule-based languages unsuitable for end users, and complex behaviour specification/visualisation is often suppressed. State-based approaches allow easy analysis of runtime changes, but sacrifice system structural details. State-based approaches convey many low level details, but for highly concurrent systems with many states raise scalability issues [15]. Flows can represent inter-state dependencies and activities based on execution sequence or conditions supporting inter-component communication, but suppress structural and behavioural details. Also, "Cobweb and Labyrinth problems appear quickly" when modelling a complex system. Users must "deal with very complex diagrams or many cross-diagram implicit relationships" [18]. PBD approaches focus on dynamic behavioural changes and visualisations; but are generally limited in specification power [6].

A hybrid visual/textual approach providing the advantages of the above approaches could more effectively specify and visualise event-based systems. We [10] have developed a toolset with such a focus but this needs refinement and improvement. Several outstanding issues exist in this domain are as yet unsolved.

## 3. ISSUES
To facilitate better understanding, easier construction and modification of event-based systems, the following issues in both static and dynamic visualisation need to be addressed:

- Suitable visual description of event-based architecture the system metamodel) is needed, with the right level of abstraction and separation of concerns.
- Structural information can often be visualised using graphical notations, but behavioural attempts usually fail due to an inappropriate visual metaphor. An expressive visual language mapping closely to the event-based domain is needed.

- Event-based behaviour specifications can't be isolated from structure or cognitive dimensions [7] issues of consistency, visibility, hidden dependency, or juxtaposability will arise.
- Navigation mechanisms are needed to allow users to focus on portions of the specification, but without losing global context, and minimise diagram clutter and permit scalability.
- Dynamic visualisation of behaviour execution should reuse design-level abstractions annotated with runtime event propagation, dataflow and invocation sequence.

We have explored these issues via the Marama meta-toolset, a set of Eclipse-based plug-ins providing visual specification languages for domain-specific visual language tools. These include specification of metamodels, visual diagram elements and editors, event handling behaviour specifications, code generation and model transformation support, and design critic authoring [11]. Being highly event-based, it is a useful platform to explore issues in event-based system visualisation. Marama target end-users include non-programmers necessitating accessible metaphors and tools.

# 4. VISUAL METAPHORS

Appropriately chosen metaphors are important for mapping a specification onto a user's domain knowledge. In Marama we chose a spreadsheet metaphor to specify model-level dependencies and constraints, an ECA-based one for view-level event handlers, and a TA (Tool Abstraction) metaphor to describe event-based tool architecture and multi-view dependency and consistency. The different metaphors are integrated via a common model and unified user interface representation. Multiple specification views can be navigated from one to another.

## 4.1 Formula construction metaphor

Marama uses extended entity relationship (EER) notation for metamodel specification comprising entities, relationships, attributes, cardinalities. We extended this with declarative constraint/dependency specifications. We were attracted to formulae but wished to minimise cognitive dimensions tradeoffs (hidden dependency and visibility issues between constraint and metamodel specifications). We designed a spreadsheet-like approach to visually construct formulae to specify model level structural constraints. We chose OCL as the primary notation as OCL expressions are relatively compact; OCL has primitives for common constraint expression needs; OCL is a standardised language; and the quality of OCL implementation is increasing.

Formula construction can be done textually, via the OCL view or "visually" by direct manipulation of the metamodel view to automatically construct entity, path, and attribute references and function calls. Clicking on an attribute places an appropriate reference to it into the formula. Clicking on a relationship and then an attribute generates a path reference. Functions selected from the OCL view are inserted as function calls in the formula. A difference from the spreadsheet approach is that only certain elements are semantically sensible at each stage of editing whereas in spreadsheets, almost any cell may be referenced.

Figure 1 shows a Marama metamodel for a simple aggregate system modeller, comprising Whole and Part entities (1), both generalising to a Type entity and related by a Whole_Part relationship (2). Entities have typed attributes, such "name", "area", and "volume". The formula construction view (3) 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" of a whole is the sum of the products of its parts' "cost" and "markup" values.

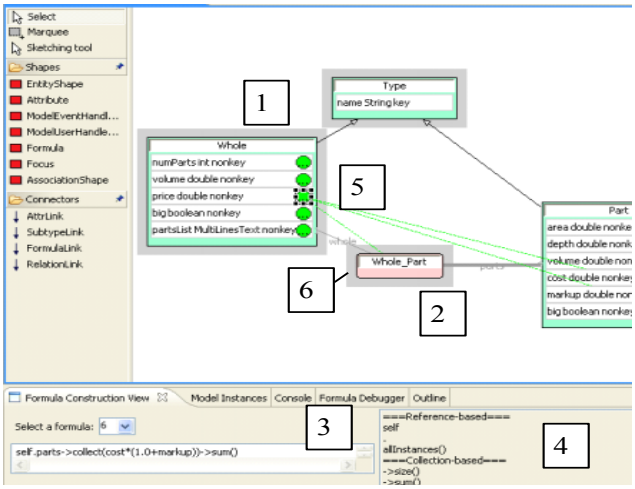


**Figure 1. Visual formula construction.**

Also shown in the visual metamodel view are circular annotations (5) on attributes where an OCL formula has been defined to calculate a value or provide an invariant. Each attribute of Whole has such a formula. Annotations are highlighted if formulae are incorrect. Dependency link annotations provide more detail about a selected formula by connecting it to other elements used in the formula. For example the formula for "price" of a Whole is selected. Dependency links show the price is dependent on the "cost" and "markup" attributes of the Parts connected to the Whole by the Whole_Part relationship. Entities and connection paths directly accessible when constructing a formula (Whole, Type, Whole_Part) have grey outline borders around them (6).

We have carefully defined interaction between the two views to enhance visibility and mitigate hidden dependency issues:

- OCL and EER editors are juxtaposed improving visibility.
- Simple annotation of the model elements indicates formulae related to them are present and semantically correct/incorrect.
- Formulae can be selected from either view so constraints can be readily navigated to/accessed.
- The dependency links permit more detailed understanding of a formula. The annotations are modified dynamically during editing for consistency. Dependencies are made visible only if a constraint is selected to minimise scalability issues and support task focus. The approach is similar to spreadsheet dependency links but applied to graphical modelling.

## 4.2 Event handler specification

Marama provides a visual "Event-Query-Filter-Action (EQFA)" notation to express event handling. Complex event handlers can be built up in parts (via sub-views) and queries, filters and actions parameterised and reused. End users select an event type of interest; add queries on the event and Marama tool state (usually diagram content or model objects that triggered the event); specify conditional or iterative filtering of the event/tool state data; and then state-changing actions to perform on target tool state objects. The approach is based on our Serendipity [9] event language.

The language design focuses on modularity and explicitly representing data propagation. We have avoided abstract control structures and used a dataflow paradigm to reduce cognitive load. Key visual constructs are events, filters, tool objects, queries on a tool's state, state changing actions plus iteration over collections of objects, and dataflow input and output ports and connectors. A single event or a set of events is the starting point. From this data flows out (event type, affected object(s), property values changed etc). Queries, filters and actions are parameterized via data propagated through inputs. Queries retrieve elements from a model repository and output data elements; filters apply pattern-matching to input data, passing matching data on as outputs; actions execute operations which may modify incoming data, display information, or generate new events.

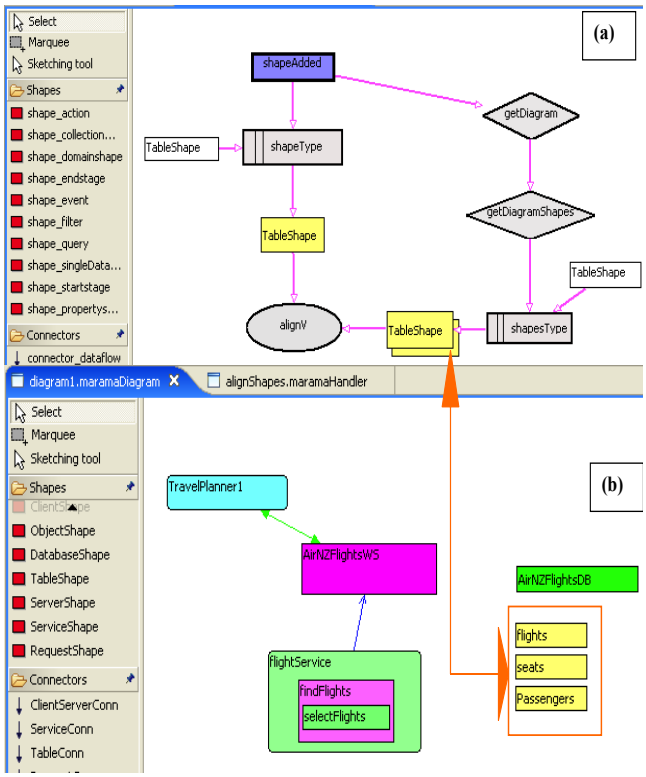


**Figure 2. Event handler specification (a) and its runtime execution effect (b).**

Queries and actions trigger when input data are available (data push). If there are no input parameters, queries and actions trigger when parameters to a subsequent flow element have values (pull). We predefined a set of primitives for these constructs providing operations useful for diagram manipulation. These involve collecting, filtering, locating or creating elements, property setting, relocating/alignment, and connection. Multiple flows are supported. Concrete end user domain icons are also incorporated into the visual specification of event handling as placeholders for Marama tool state, to annotate and mitigate the abstraction, making the language more readable. Figure 2 is an event handler aligning diagram shapes (a) and its runtime execution (b). The handler responds to a Marama "shapeAdded" event, filters on "TableShape", and aligns a new "TableShape" to existing ones queried from the diagram. Event handlers register availability in metamodels using annotations similar to formula icons.

As our target user group are inexperienced programmers, we have chosen a low-to-medium abstraction gradient based on iconic constructs and data flow between them. The abstractions support query/action composition allowing users to specify Marama data and state changing actions as discrete, linked building blocks. The abstractions require hard mental operations but are mitigated by concrete end user domain objects. We are experimenting with elision techniques to allow concrete icons and abstract event handler elements to be collapsed into a single meaningful icon. The dataflow metaphor used to compose event specification building blocks seems to map well onto users' cognitive perception of the metaphor. The current approach has reasonable visibility and juxtaposability. Information for each element of an event handler is readily accessible. The event handler specification can be juxtaposed with the modelling view that triggers its execution. However, it still has viscosity problems; the user typically has to rearrange the diagram to insert elements.

## 4.3 Event-based architecture specification

We chose to use the TA (Tool Abstraction) [8] metaphor, and have provided a view, we call the "event propagation view", to describe event-based tool architecture. This aims to mitigate multi-view dependency and consistency issue.

TA is a message propagation-centric metaphor describing connections between "toolies" (behaviour encapsulations which respond to events to carry out system functions) and "abstract data structures" (ADSs: data encapsulations which respond to events to store/retrieve/modify 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. TA supports data, control and event flow relationships in a homogeneous way, allowing a focus on architecture level abstractions and describing separated concerns include tool specific events, event generators and receivers, and responding behaviours such as event handlers.

Figure 3 shows user-defined events and their notifications among various Marama event handling toolies and structural components defining that when an "ArchitectureDiagram" instance is deleted from a model project, all mapped view data are deleted from other views, and all mapped model data are deleted from the model, so that the views and the model are still synchronised with consistent data. Three Marama structural components are involved:

- diagram – the deleted diagram of a model
- views – all the views of the model
- modelProject – the tool's model instance

A condition filters out diagrams other than "ArchitectureDiagram" instances. The "processDiagramData" toolie generates a "diagram Deleted" event to be propagated to the "deleteMappedViewData" and "deleteMappedModelData" toolies, which define the event handling responses. Further "viewUpdated" and "modelUpdated" events propagate from the respective toolies to the "views" and "modelProject" data structures indicating 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.

The primitives supporting the TA paradigm represent a range of tool abstraction components and links which provides a higher level of abstraction than the prior visual formula and event handler specifications. This aims to facilitate easier understanding of event-based architectures rather than lower-level elements in an event-based system, as supported by the previous two approaches.

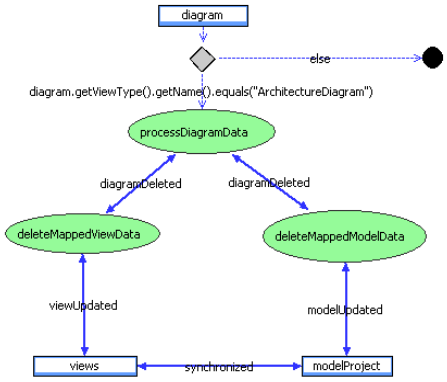


**Figure 3. Event-based architecture specification**

## 4.4 Higher level description of the metaphors

With multiple visual languages at different abstraction levels being used in Marama for event-based system construction, hard mental operations are introduced as a trade-off for specification flexibility. Users need to decide which visual language to use at a particular modelling stage. Our evaluation results [17] show that despite our emphasis on accessibility Marama presents a steep learning curve. Therefore, there is a need for a description and guidelines for these metaphors, from which users can better make choices about their specification approaches.

As a result, we have generalised the three metaphoric languages to a canonical event handling model to enable integration, reuse and framework evolution. We aim to develop a higher level visual notation based on this model to use as a description language for event-based specifications, to facilitate better understanding of our predefined vocabularies, and to allow users to describe their own extensions (e.g. to more easily define new event metaphors). This description must not only include a high level behaviour model, but also critics and layout mechanisms, to provide guidelines for specification and verification, and automation to ease the burden of use. We also plan to provide software process support, mainly aiming at deriving design-level components from users' requirements authoring but also to guide the design process.

We will employ program-by-demonstration techniques to allow users to play pre-recorded macros to learn the event-based visual languages and their modelling procedures, and to specify their own domain systems following demonstrated examples or patterns. In addition to the current procedure of generating DSVL environments from meta-level structural and behavioural specifications, we wish to also allow the users to demonstrate the intent of their DSVL tools and automatically generate the specifications (both structural and behavioural) reversely from that, with further refinement allowed via round-trip engineering.

# 5. CONTEXTUAL NAVIGATION

We have designed several "contextual highlight" techniques including show/hide, collapse/expand, zoomable and fisheye views, and partition and sub-view generation for managing size and complexity of both structural and behavioural specifications. The formula dependency links described earlier, are an example of show/hide, with dependency links only visible for a currently selected formula. As another example users can selectively display a series of semantically connected constructs as in EML's process overlays [18] (Figure 4), where multiple processes are defined in the same diagram but one is selected to be shown at a time.

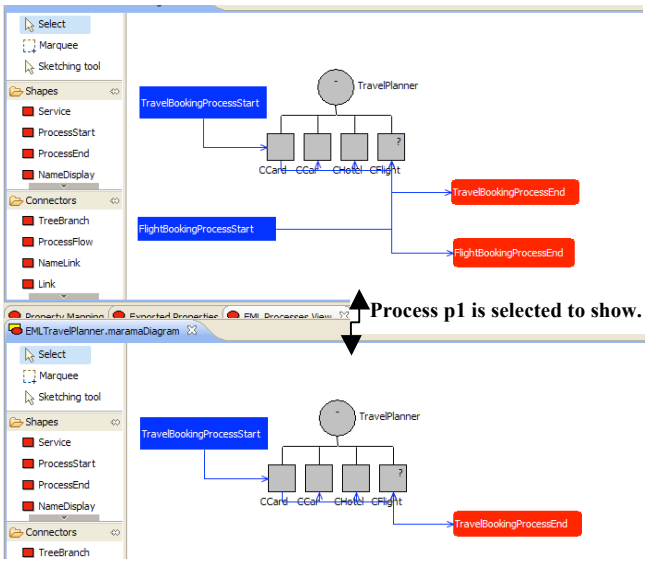


**Figure 4. Process overlays in EML**

Collapse/expand was originally introduced in EML's [18] trees, where a tree node can be collapsed/expanded with adjusted visualisation (e.g. re-rendering of overlays on the collapsed tree). We plan to apply this to the Marama metamodel, extended/ adapted with features of our Visual Wiki work [13]. Our aim is to provide better visualisation support when a Marama metamodel becomes large. Figure 5 shows the design. Two diagrams are used: the left for navigation and the right for detailed display. At left, entities have a labelled node notation and relationships are elided and replaced by links between entities. Companion nodes to an entity include shapes used to display the entity, views containing a representation of it, and formulae and handlers that apply to it. Nodes further from the centre scale down in size, but expand when navigated to becoming the new central node in the diagram. The selection of a node, which can be an entity, shape, view, formula or handler, triggers its corresponding detailed specification to be displayed in a synchronised view (right).

To manage scale, zooming functions are provided with a Marama diagram (Figure 6 a). The "zoom in/out" functions zoom in/out the entire diagram by predefined scaling factors, with the "Radar" zoom view (Figure 6 b) indicating visible items inside the screen boundary and those outside of the boundary. "Zoom fit" provides a best fit view and "selection zoom" allows user to select an area of the diagram and zoom to that.

Fisheye view or "distortion based display" functionality is also supported. The benefit is that a local context is presented against a global context, thus allowing the user's point of interest to be focused on without losing the extensive surrounding information. Figure 6 c shows a fisheye view of an EML [18] tree structure. The mouse pointer is the default Focal Point. The degree of interest (DOI) of the certain part of the tree structure is based on the Distance of Focus. A shorter distance will lead to higher value of DOI, thus, the shape will be represented in a bigger size. The longer distance brings lower value of DOI, which leads to the smaller size of the shapes. As the mouse moves, the DOI value

and shape size of the tree nodes is recalculated dynamically.

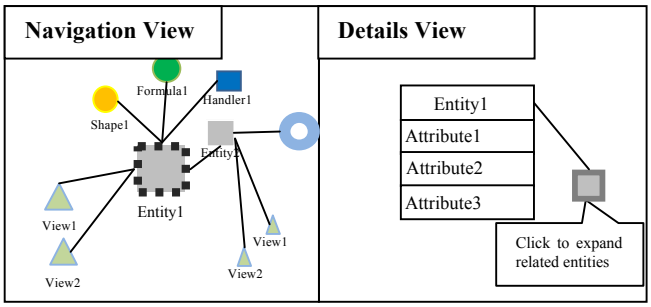


**Figure 5. Semantic navigation of tool specifications**

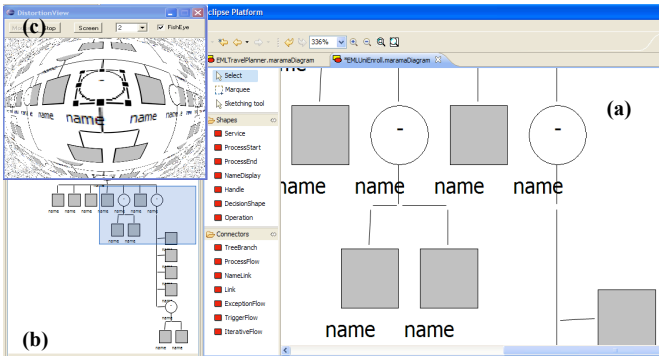


**Figure 6. Zoomable and fisheye views in Marama**

Though various contextual navigations are supported in the base diagram, users are also able to partition by element selection with regeneration and display in a sub-view. Marama supports cohesive consistency between multiple views, and the generated sub-view can again function as a base view for further partitions.

We are experimenting with automatic layout techniques which will be useful to improve the user's ability to show/hide, collapse/expand, or juxtapose parts of a specification, and thus to manage size, complexity and visibility more effectively. These are based on end user specifications at a high level, with the focus on indicating which visual components are to be affected and how.

# 6. VISUAL DEBUGGING

A consequence of introducing new visual languages to specify and generate event handlers in Marama is the need to support incremental development and debugging using these languages. Event propagation can become very complex so tool support for tracing and visualising event propagations and their effects is needed. Such visualisations need to incorporate both the static dependency structure and dynamic event handling behaviour. Event-based system executions are highly time related, and many phenomena may occur in a very short time making real-time visualisations ineffective [4]. Step-by-step visualisation that is interactively controlled by the user is thus required.

Marama's visualisation of dynamic event handling behaviour uses a model-view-controller approach which reuses event handler specification views by dynamically annotating modelling elements with colours and state information in response to events. A central repository stores runtime information which can be retrieved and manipulated by controller code for presentation in views. A specialised debugging and inspection tool (visual debugger) allows execution state of event-based systems to be queried, visualised and dynamically modified. It provides a common user interface connecting the model-level constraint and view-level event handling specifications with an underlying debug model.

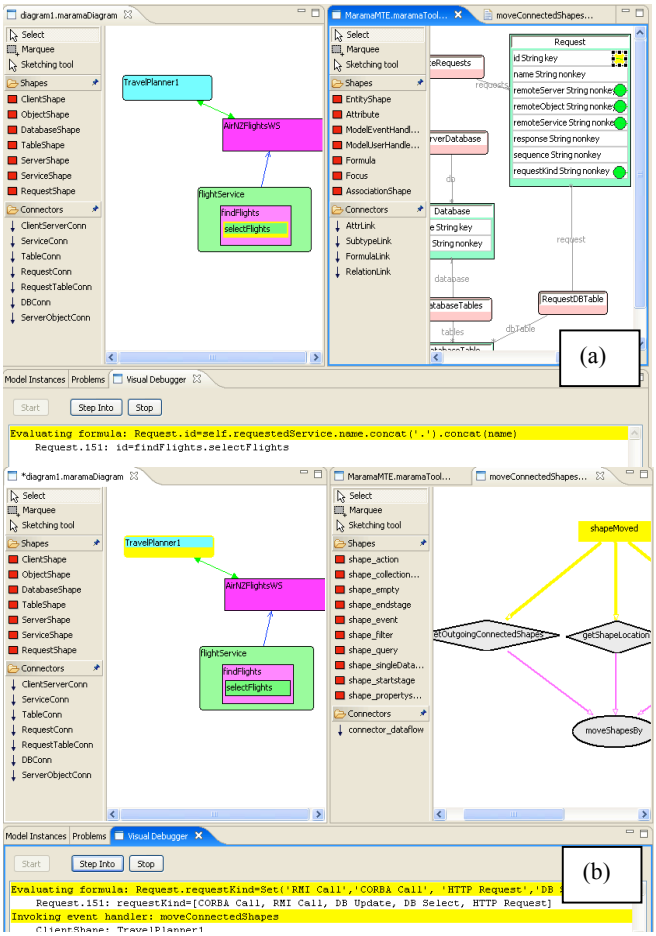


**Figure 7. Visual debugging formulae (a) followed by a view event handler (b).**

Figure 7 shows the visualisation of runtime interpreted formulae (a) and an event handler (b) on a Marama model. The metamodel view and the event handler specification views are respectively juxtaposed with the runtime Marama model view for parallel visualisation of dependency evaluation or event handling in the running model instance's context. A traditional "debug and step into" metaphor is used for step-by-step visualisation. Affected runtime model elements are annotated (yellow background) to indicate application of the formula/handler (left), while the formula/handler node and dependency links are annotated similarly in the specifications showing invocation status (right). Detailed information is presented in textual form (bottom). Run-time monitoring of Marama for performance analysis could also potentially be supported via the visual debugging sub-system.

We are currently working on representing visual debugging at a higher abstraction level, to better enable users to query both the static model and dynamic execution state. A visual query language will provide users with a means to specify query intent and generate results. Sensible display of queried results in a diagrammatic form using layout mechanisms is also being

addressed. More advanced query support is being planned to query of multiple end user tools for reusable specifications. From that, a semantic knowledge base with structured metamodel, model and transformation information is needed so that reasoning and pattern mining can be effectively performed.

## 7. SUMMARY

We have described general issues involved in visualising event-based information models, including abstraction and visual metaphor, hidden dependency, consistency and step-by-step visualisation. We have addressed these from our own experience in developing a set of notations and tools, from which we have generalised a canonical representation to enable the specification and visualisation of general purpose event-based systems. The generalised framework includes the following components and provides reuse via both inheritance and composition:

- Structural components, e.g. entity, relationship, attribute, role, cardinalities, event, model, view
- Behavioural components, e.g. query, filter, action, formula, and various event notification schemes such as broadcast, subscribe-notify, listen-before and listen-after.
- Layout, e.g. for shapes: containment, on border, enclosure, horizontal/vertical alignment, show/hide, and collapse/expand; for connectors: straight/curved/angled routing and show/hide; and overall: horizontal/vertical tree, top-down/left-right process start/end, zooming/fisheye and view juxtaposition.
- Runtime, e.g. focus/highlight

Our future work directions include a higher level description of our visual event handling metaphors, automatic layout support and query-based runtime visualisation.

## 8. REFERENCES


[1] Berndtsson, B., J. Mellin, and U. Hogberg, *Visualization of the Composite Event Detection Process*, in *the 1999 International Workshop on User Interfaces to Data Intensive Systems*. 1999, IEEE CS Press. p. 118-127.

[2] Burnett, M., et al., *Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm.* Journal of Functional Programming, 2001. **11**(2): p. 155-206.

[3] Conway, M., et al., *Alice: Lessons Learned from Building a 3D System for Novices*, in *the SIGCHI conference on Human factors in computing systems*. 2000. p. 486-493.

[4] Coupaye, T., C.L. Roncancio, and C. Bruley, *A Visualization Service for Event-Based Systems*, in *15emes Journees Bases de Donnees Avancees, BDA*. 1999.

[5] Cox, P.T., et al., *Experiences with Visual Programming in a Specific Domain - Visual Language Challenge '96*, in *the 1997 IEEE Symposium on Visual Languages*. 1997.

[6] Cypher, A., *Watch What I Do: Programming by Demonstration*. 1993: The MIT Press.

[7] Green, T.R.G. and M. Petre, *Usability analysis of visual programming environments: a 'cognitive dimensions' framework.* JVLC, 1996. **7**: p. 131-174.

[8] Grundy, J.C. and J.G. Hosking, *ViTABaL: A Visual Language Supporting Design by Tool Abstraction*, in *the 1995 IEEE Symposium on Visual Languages*. 1995, IEEE CS Press: Darmsdart, Germany. p. 53-60.

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

[10] Grundy, J.C., J.G. Hosking, and W.B. Mugridge, *Visualising Event-based Software Systems: Issues and Experiences*, in *SoftVis97*. 1997: Adelaide, Australia.

[11] Grundy, J.C., et al., *Generating Domain-Specific Visual Language Editors from High-level Tool Specifications*, in *the 21st IEEE/ACM International Conference on Automated Software Engineering*. 2006: Tokyo, Japan. p. 25-36.

[12] Haskell. [cited 2007]; Available from: http://www.haskell.org/

[13] Hirsch, C., J. Hosking, and J. Grundy, *Interactive Visualization Tools for Exploring the Semantic Graph of Large Knowledge Spaces*, in *Workshop on Visual Interfaces to the Social and the Semantic Web (VISSW2009), IUI2009*. 2009: Sanibel Island, Florida, USA.

[14] IBM. *Specification: Business Process Execution Language for Web Services Version 1.1*. [cited 2003]; Available from: http://www.ibm.com/developerworks/library/ws-bpel/.

[15] Kraemer, F.A. and P. Herrmann, *Transforming Collaborative Service Specifications into Efficiently Executable State Machines*, in *GT-VMT 2007*. 2007.

[16] Ledeczi, A., et al., *Composing Domain-Specific Design Environments.* Computer, 2001: p. 44-51.

[17] Li, K.N.L., *Visual languages for event integration specification* in *Computer Science*. 2007, University of Auckland: Auckland.

[18] Li, L., J.C. Grundy, and J.G. Hosking, *EML: A Tree Overlay-based Visual Language for Business Process Modelling*, in *ICEIS*. 2007: Portugal.

[19] Li, X., W.B. Mugridge, and J.G. Hosking, *A Petri Net-based Visual Language for Specifying GUIs*, in *the 1997 IEEE Symposium on Visual Languages*. 1997: Isle of Capri, Italy.

[20] Matskin, M. and D. Montesi, *Visual Rule Language for Active Database Modelling.* Information Modelling and Knowledge Bases IX, 1998: p. 160-175.

[21] OMG. *UML Superstructure*. [cited 2009]; Available from: http://www.omg.org/spec/UML/2.2/Superstructure/PDF.

[22] Paschke, A., *ECA-LP / ECA-RuleML: A Homogeneous Event-Condition-Action Logic Programming Language*, in *RuleML'06*. 2006: Athens, Georgia, USA.

[23] Pautasso, C. and G. Alonso, *Visual Composition of Web Services*, in *IEEE HCC'03*. 2003: Auckland, New Zealand.

[24] Smith, D.C., A. Cypher, and J. Spohrer, *KidSim: programming agents without a programming language.* Communications of the ACM, 1995. **37**(7): p. 54 - 67.

[25] Tolvanen, J., *OOPSLA demonstrations chair's welcome: MetaEdit+: integrated modeling and metamodeling environment for domain-specific languages, Companion to the 21st ACM*. 2006.

[26] Zhu, N., et al., *Pounamu: a meta-tool for exploratory domain-specific visual language tool development.* Journal of Systems and Software, 2007. **80** (8).