In Proceedings of SoftVis97, Adelaide, Australia, Dec 1997

Visualising Event-based Software Systems: Issues and Experiences

John C. Grundy† , John G. Hosking†† and Warwick B. Mugridge††

†Department of Computer Science University of Waikato
Private Bag 3105, Hamilton, New Zealand

††Department of Computer Science University of Auckland
Private Bag, Auckland, New Zealand

Abstract

Event-based software systems, such as componentware, tool abstraction, message passing systems, software process environments, and many data visualisation systems, are becoming ever more common. Constructing, understanding, and modifying such systems can be very difficult, however, and appropriate software visualisation support is often a great help to developers. We describe some issues in both statically and dynamically visualising a range of event-based software systems. We illustrate these issues with examples from our own experiences in developing a diverse range of event-based software visualisation notations and tools.

1. Introduction

Event-based software systems are based on software architectures that utilise the ideas of software component interdependencies, the propagation of objects representing events between components, and the appropriate response to these events by the receiving components [18]. Examples of event-based software systems include the following:

- *Componentware systems*, such as those based on Java Beans [30], Active-X [29] and OpenDoc [2], are composed of discrete, interdependent software components which exchange events. Components respond to events received by other components to appropriately redisplay data, update data structures, communicate over networks or with databases, etc.
- *Tool abstraction systems* [13] are similar to componentware in that discrete data and function-implementing software components are wired together to produce an application. Tool abstraction systems use a combination of event passing and method invocation, as well as a variety of triggering and coordination mechanisms to function.
- *Process-centred environments* [4] support models of work processes. For example, models of how software is to be developed, which are "run", or enacted, to guide or enforce how work is done. These systems are often event-driven, with events causing rules to fire or being used to enact process stages, ensuring that appropriate responses to process-related events are carried out.
- *Data and program visualisation systems* [7, 9] often utilise event-based models. On changes to visualised data, events describing these changes are used to updates to visualisations of that data.
- *Multiple view systems* [33, 28] often utilise events to keep model and view objects consistent under change. When a model object is updated, events describing this change are generated and propagated to view objects, which redisplay and/or reconcile their state to the updated model object.

Many event-based systems are complex in nature, with many interdependent components and complex event propagation and response behaviour. Their static structure can often be usefully visualised using graphical notations, to assist developers in constructing, understanding and modifying them. Annotating structural descriptions with indications of event-based behaviour, or developing separate behavioural models of a system, are also very useful. Such descriptions of static event-based system structure and behaviour are also useful in documentation.

Visualising the dynamic behaviour of a running event-based system allows developers to view abstract representations of an actual running system, rather than viewing their static structure and behavioural visualisations. These run-time visualisations help developers in understanding how actual data and events are represented and propagated, and are often very useful during debugging or understanding of such systems.

Constructing both static and dynamic event-based system visualisations can be non-trivial. Static visualisations can be assembled by reverse engineering event-based system structures, although doing this automatically is usually quite difficult. Often such static visualisations are best hand crafted by those designing and implementing the event-based software. Dynamic visualisations necessarily require both design of the visualisation software and the detection of "interesting events" from the event-based system software as it runs, used to drive the visualisation. The specification and realisation of such dynamic visualisations usually needs to take into account the kind of event-based system being visualised, its software architecture and implementation language, and the static visualisations used for design and documentation of the system.

2. Issues in Visualising Event-based Software Systems

Event-based software systems are generally comprised of a collection of interdependent software components, as shown in Figure 1 [18]. Components may exchange data and control via operation invocation, as in other software systems, but a key distinction with event-based systems is the propagation of "interesting events" between related components. A component which has undergone a noteworthy state change, or other event, typically generates some description of this event, as an object. This event object is broadcast to other components that have indicated an interest in such an event. Some event-based systems broadcast all events to all dependent components, while others broadcast selected events to components indicating an interest in the specific kind of event that has occurred [19, 35]. The structure of an event-based system may be static, or may have components dynamically added or removed at run-time. This has resulted in event-based systems proving to be a very adaptable and powerful model, suitable for use in many application domains [18, 8, 36, 7, 11, 12].

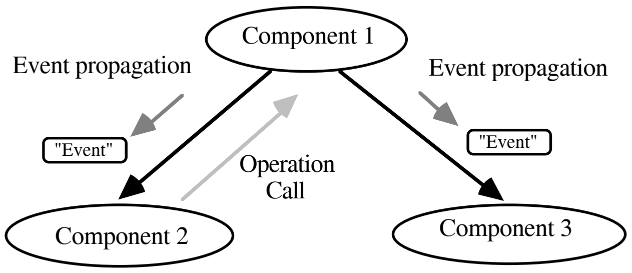


Figure 1 Basic structure of event-based software systems.

For a developer to create a static visualisation of an event-based software system, such as the simple one in Figure 1, languages are needed to express the system structure and behaviour. For example, the "notation" used in Figure 1 represents components as ovals, component interdependencies as arrowed lines, and the propagation of events as small, shaded arrow annotations. In order to visualise large, complex event-based systems, developers need appropriate notations to describe their structure and semantics, along with tools that support system design and construction with these notations, and/or reverse engineering tools which examine an implementation and produce such visualisations. Ideally, developers will have access to a range of notations, depending on the nature of the event-based system(s) they are working with. They will be able to construct multiple perspectives of the system (i.e. multiple views of parts of the system structure and behaviour). And they will be able to design, build and maintain systems using these notations and views.

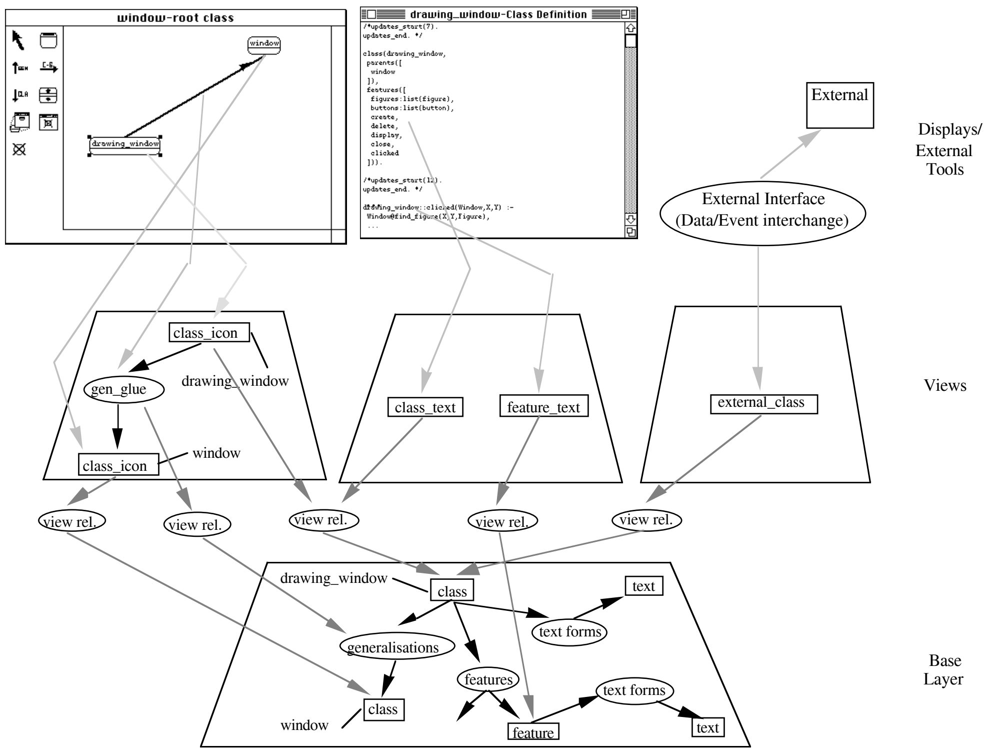
Dynamic visualisation of event-based software systems depends on both the run-time structure of the system and the detection of events as they pass between software component instances. For example, in Figure 1 a running system might have an instance of Component1 linked to zero, one or several Component2 and Component3 instances. When a Component1 instance generates an event that is propagated to instances of Component2, the developer may wish to trace such propagation, step through a series of inter-component propagations, and/or profile many such propagations to produce statistics about the run-time nature of an instance of the component-based system. Tools supporting such visualisations may output event propagation traces in a textual manner, showing relevant component and event data values. They may utilise graphical visualisations similar to the static visualisatons used in designing and building the system. Or such tools may utilise quite different visualisations; for example, timing diagrams and graphical event profiles.

We describe our experiences in developing static and dynamic visualisations in four of our event-based software systems. MViews is a notation for describing a component-based software architecture, with CernoII providing the dynamic visualisation of the implementation language of this architecture. ViTABaL is a tool abstraction system that utilises a visual language to describe the structure and behaviour of the system, along with annotations to specify dynamic visualisations. Serendipity is a process modelling and enactment language and environment with static and dynamic visualisation of software processes. JComposer is a visual language for component-based software, which can be utilised for both static and dynamic visualisation. We compare our work to other approaches to static and dynamic program visualisation, and indicate directions for further research.

3. MViews and CernoII

Our first attempt at static program visualisation of an event-based software system was with the MViews framework. MViews is a framework for building multiple view editing tools [20]. It is based on the Change Propagation and Response Graph (CPRG) software architecture, an event- and component-based architecture for maintaining consistency between interdependent software components [19]. CPRGs, and thus MViews framework class specialisations, consist of components and inter-component relationships, with components and relationships having attributes and operations. When a component or relationship is modified by operations, change description objects representing these events are generated and propagated along the relationship links. The CPRG architecture has a graphical notation that represents components as rectangular icons, relationships between components as ovals, and event flows allow relationship links as arrow annotations [19]. Figure 2 shows an example of the MViews architecture representing a OO CASE tool structure. We have used the CPRG notation as an

Architecture Description Language to describe MViews environments during design and documentation, thus describing their environment static structure and aspects of its behaviour (event propagation between components).



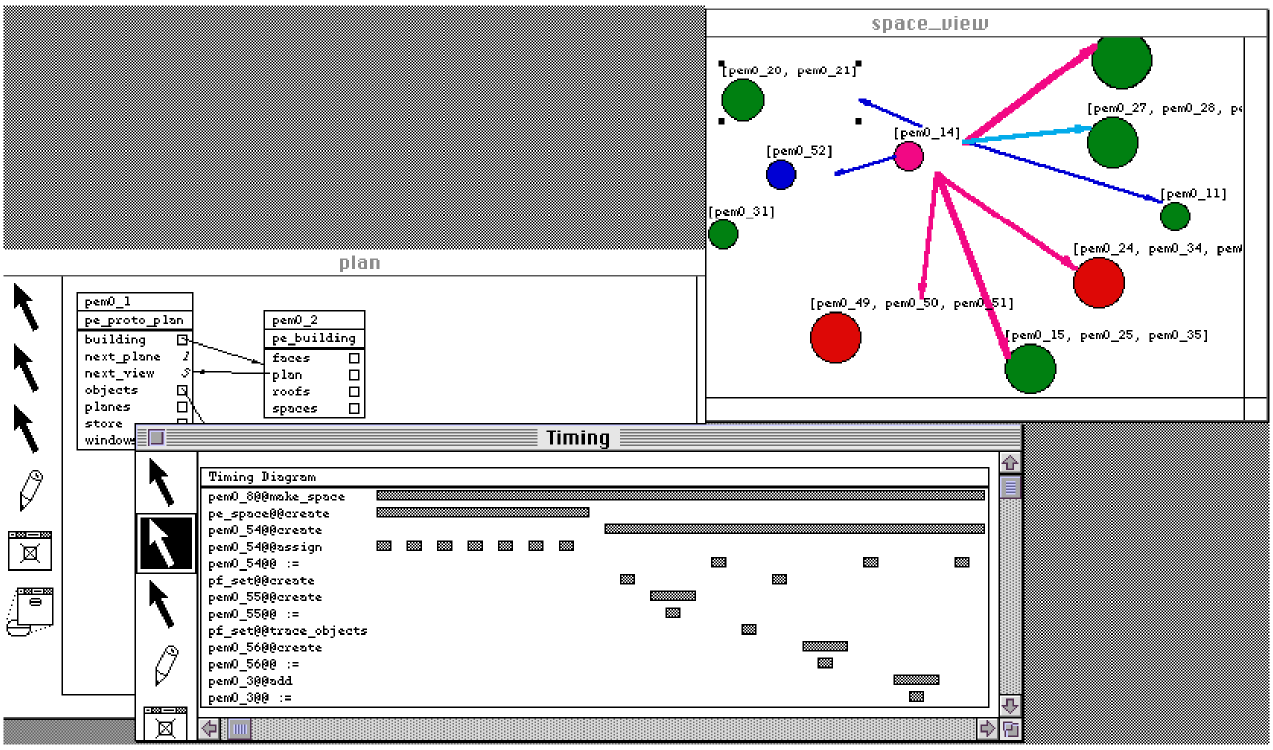
```
/*updates_start(7).
updates_end. */

class(drawing_window,
 parents([
  window
 ]),
 features([
  figures:list(figure),
  buttons:list(button),
  create,
  delete,
  display,
  close,
  clicked
 ])).

/*updates_start(12).
updates_end. */

drawing_window::clicked(Window,X,Y) :-
 Window@find_figure(X,Y,Figure),
 ...
```

Figure 2. The MViews architecture, using the CPRG notation.



```
Timing Diagram
pem0_8@@make_space
pe_space@@create
pem0_54@@create
pem0_54@@assign
pem0_54@@ :=
pf_set@@create
pem0_55@@create
pem0_55@@ :=
pf_set@@trace_objects
pem0_56@@create
pem0_56@@ :=
pem0_3@@add
pem0_3@@ :=
```

Figure 3. An example of CernoII visualising a running MViews program.

Large systems developed in MViews, such as the OO development environment SPE [16], integrated CASE tools OOEER and NIAMER [43], and modelling environment EPE [1], utilise many different components, component inter-relationships and complex event flows. Dynamic visualisation of such MViews-based systems proved to be very useful in understanding the systems (for other users of the framework), and for debugging. To aid such visualisation, we developed some simple textual and graphical program visualisation components, using MViews itself [15]. These provided simple traces of events flowing in part of an MViews system, and simple graphical renderings of these event traces.

These visualisation components proved too limited in general. Hence the CernoII program visualisation system was developed, to not only visualise MViews-based environments but also to visualise any program developed using Snart, the OO Prolog implementation language of the MViews framework [16]. CernoII is an MViews-implemented tool, which allows developers to visualise their running Snart programs using a simple, visual representation of objects and references, and to create multiple visualisations of different parts of their programs. It also allows developers to create complex visualisations, such as timing diagrams and event trace profiles. A textual language is used to specify the appearance of visualisation elements. Reusable "abstractors" are used to generate abstractions of underlying program data and execution structures. These allow tailored visualisations of interesting program events or abstractions of sequences of events to be readily developed. Figure 3 shows three CernoII visualisations. View 1 is a timing diagram, similar to an instantiation of a UML sequence diagram. View 2, showing component referencing structures together with event flows, is similar to a UML collaboration diagram. View 3 is a map metaphor view [15] which uses size and colour to provide a visual indication of event propagation frequency between and within clusters of components. An important lesson we learned from CernoII was the need to allow simple specification of new types of visualisation, particularly those abstracting from specific program events to sequences of event flows.

4. ViTABaL

MViews uses a simple type of event-based model, in which a component generating an event-representing object propagates this object to all connected components to inform them of the event. A more sophisticated kind of event-based system is “tool abstraction”, where components are distinguished by data-structure-implementing and function-implementing “toolies” [13]. Events are propagated between selected, linked components, with a variety of event responses supported. ViTABaL is a Visual Tool Abstraction Language, implemented using MViews but which generates Snart code to implement tool-abstraction-based systems [17]. ViTABaL provides a novel visual language for describing the static structure of such systems. An example, using Parnas’ KWIC (Key Word In Context) example, is shown in Figure 4. Toolies implementing event-driven functionality are shown as ovals, while a shared pool of ADT toolies are shown as rectangles. Toolies are connected by event dependencies, and annotations indicate different kinds of events and event parameters. For example, the “insert(-line)” event sent from the input functionality toolie to the line_buffer ADT toolie indicates a insert event sent to the line_buffer is triggered by the input toolie and passed a line to insert into the buffer. ViTABaL supports multiple views of toolies, including orthogonal and hierachical visual views, textual event response implementation views, and supports event propagation annotations such as broadcast, request, listen before and listen after.

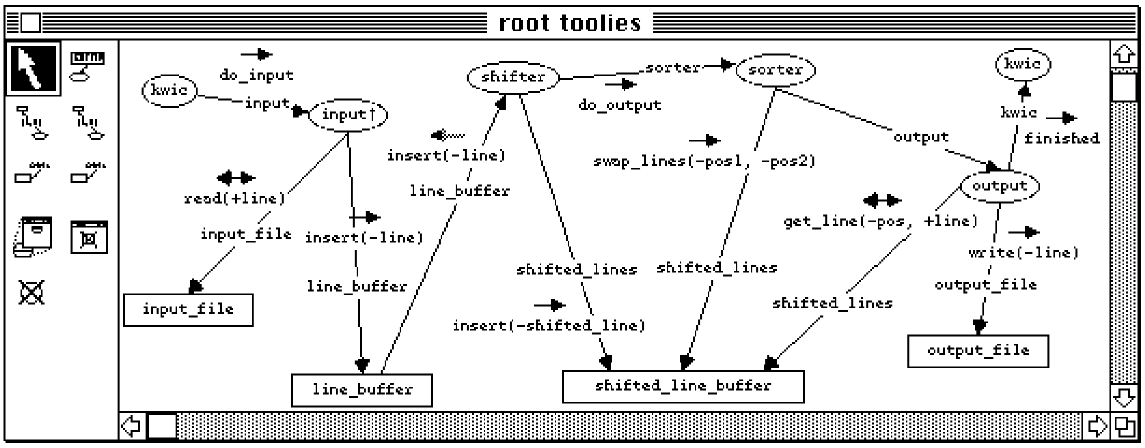
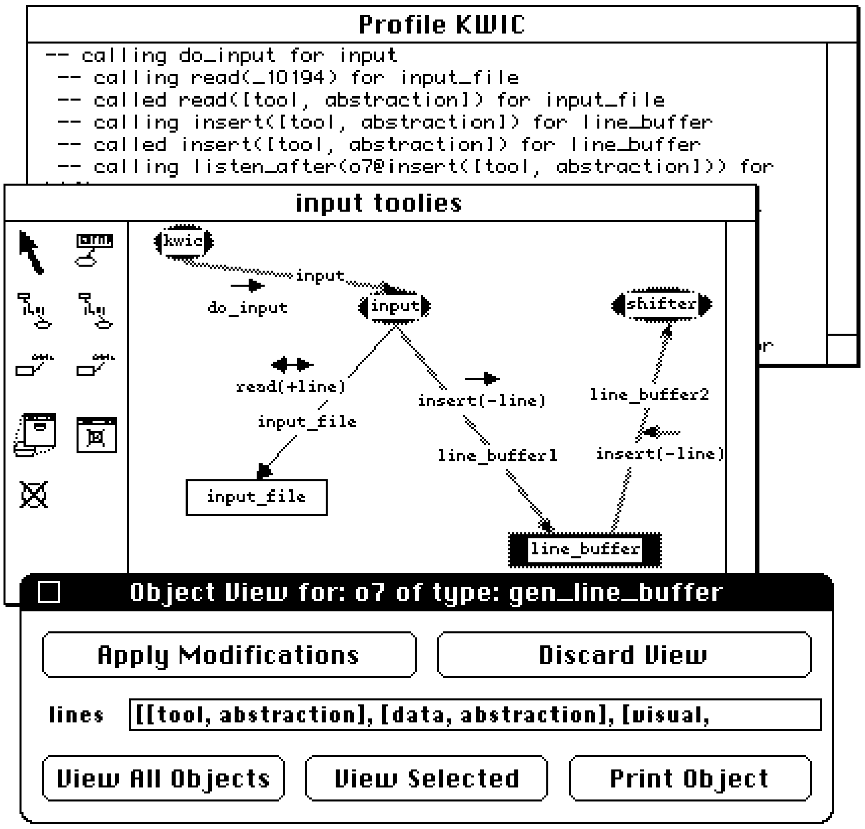


Figure 4. An example of a static ViTABaL program structure.

Dynamic visualisations of ViTABaL programs utilise visual static structure views by highlighting “active” toolies and event propagation links. We have also provided textual event tracing views that show event data, and toolie inspection dialogues which allow the attribute values of toolies to be visualised. An example of a running ViTABaL program being visualised is shown in Figure 5. These views, like CernoII visualisations, are updated as a program runs; i.e. as events are propagated and data changes. Due to ViTABaL’s support of concurrent toolie execution, such visualisations are very helpful in understanding toolie interdependencies and operation; these become quite complex even for simple system with a limited number of toolies and toolie interconnections. We intend to construct "concurrent timing diagram" for ViTABaL programs, using CernoII.

ViTABaL shows the benefit of reusing design level abstractions in a concrete instantiated form for visualisation purposes, providing a ready mapping of concepts from design to implementation. Similar reasoning was behind the development of early CernoII visualisations, which aimed to instantiate analysis and design level object and interaction diagrams for visualisation purposes.



```
-- calling do_input for input
 -- calling read(_10194) for input_file
 -- called read([tool, abstraction]) for input_file
 -- calling insert([tool, abstraction]) for line_buffer
 -- called insert([tool, abstraction]) for line_buffer
 -- calling listen_after(o7@insert([tool, abstraction])) for
```

lines [[tool, abstraction], [data, abstraction], [visual,

Figure 5. A running ViTABaL program being visualised.

5. Serendipity

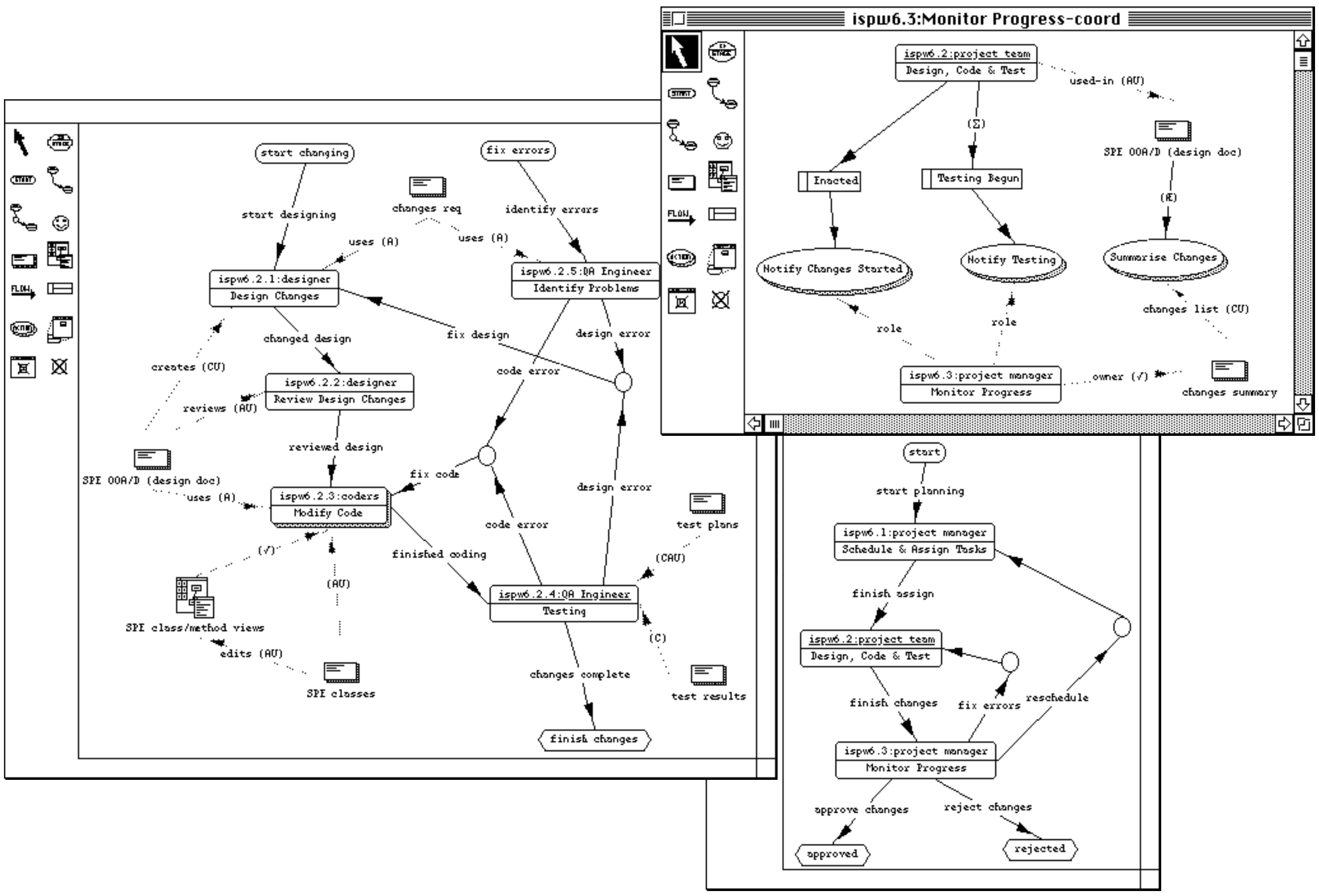
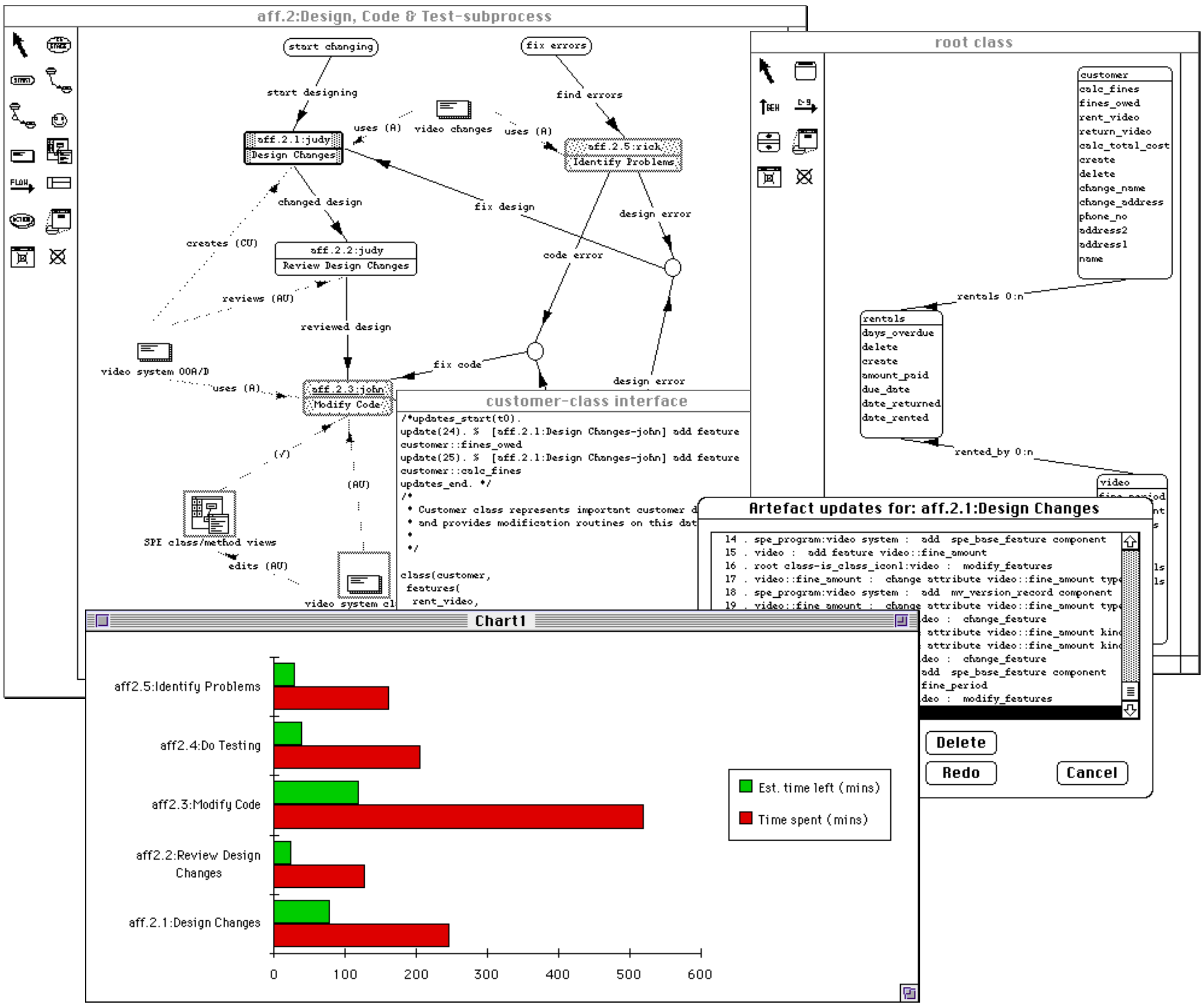


Figure 6. An example of Serendipity environment process models.

Process-centred environments allow users to describe work processes, for example the process by which software is, or should be, developed [4]. Such environments typically utilise a form of event-based system to run, or "enact" such process models, with process stages triggered by events sent to them by other stages or external sources. These environments are useful for planning and coordinating multiple developers working on software projects. We have developed the Serendipity process modelling and enactment environment, which utilises two kinds of event propagation and response to drive process model enactment and to act on a variety of stimuli [23, 24].

Figure 6 shows examples of static visualisations of Serendipity process models for the ISWP6 Software Process example [26, 21]. The bottom right diagram is the top-level model of this process, with the left hand diagram an exploded description of the "Design, Code and Test" process stage. Stages (ovals with an ID, role and name) are connected by "enactment event" flows. When a stage completes, a "finished" event flows into the connected stage(s) and enacts them appropriate. Additional modelling constructs include artefacts, tools and roles, and "usage" connections describing how different stages make use of these. The top, right diagram shows how enactment and artefact modification events can be process by filters (rectangular icons) and actions (shaded ovals). Filters detect interesting events and pass them onto other filters or actions, which then carry out some operation(s) based on the events they receive. This allows developers to specify work coordination strategies, to automate various event-triggered tasks, and to integrate third-party tools into their process models.

Figure 7 shows an enacted Serendipity process model (left-hand side, top). Enacted stages are highlighted by shading and colouring, indicating who has enacted stages and which enactment links have caused a stage to become enacted. Dialogues showing enactment histories for stages, work done (i.e. artefact modifications) for stages, and modifications for artefacts are also supported. The bottom diagram in Figure 7 shows an Excel graph indicating time spent on process stages and estimated time remaining. This was based data on time spent between enactment of stages (recorded by Serendipity) and estimated time remaining for each stage (recorded by the project leader). A Serendipity action generated a delimited text file for import into Excel, where a simple graphing macro was used to generate the visualisation. These different forms of dynamic visualisations of an enacted software process assist developers in using the process, coordinating their work, tracking and analysing their work, and ultimately in improving their software processes.



```
/*updates_start(t0).
update(24). %  [aff.2.1:Design Changes-john] add feature
customer::fines_owed
update(25). %  [aff.2.1:Design Changes-john] add feature
customer::calc_fines
updates_end. */
/*
 * Customer class represents important customer d
 * and provides modification routines on this dat
 *
 */

class(customer,
 features(
  rent_video,
```

```
14 . spe_program:video system :  add  spe_base_feature component
15 . video :  add feature video::fine_amount
16 . root class-is_class_icon1:video :  modify_features
17 . video::fine_amount :  change attribute video::fine_amount type
18 . spe_program:video system :  add  mv_version_record component
19 . video::fine_amount :  change attribute video::fine_amount type
```

Figure 7. An enacted Serendipity process model.

The Serendipity enactment visualisations use the same approach as CernoII and ViTABaL in concretely instantiating, then annotating, design level diagrams. The timing graphs, however, also show the need to be able to specify views representing higher level abstractions, as was the case in CernoII. In Serendipity, however, such visualisations have been hand crafted. The lack of a CernoII-like ability to specify such visualisations has thus limited Serendipity's capabilities.

6. JViews and JComposer

Due to various limitations with the implementations of MViews, ViTABaL and Serendipity, we have recently developed the JViews framework, a successor to MViews and CPRGs. This is a component-based framework implemented using the Java Beans componentware API [22]. Component-based software systems (often referred to as "componentware") have become popular, in large part because of their event-based nature, which allows reusable components to be readily "plugged-and-played" with other components. Component-based software architectures include Java Beans [30], Active X [29], and Open Doc[2].

JComposer is a tool to design and generate JViews-based systems. It utilises a modified form of the CPRG static visualisation notation to allow developers to design JViews-based software systems. Figure 8 shows an example of JComposer being used to design a multiple-view EER modelling tool. Components (square icons) are linked by relationship links and relationship components (oval icons). Components have attributes and operations, and are interdependent. Events generated by one components are propagated to other components. The behaviour of components is either specified textually (coding in Java) or by attaching visualisations of reusable, Serendipity-style filters and actions to components via relationships. Filters and actions can themselves be defined in several ways: via parameterised filters and actions specified via dialogues, hierarchical definitions composed of simpler filters and actions connected together, or via Java code. The visual notation allows designers to describe the structure of JViews-based systems at a high level of abstraction, and allows new systems to be composed by reuse of existing components.

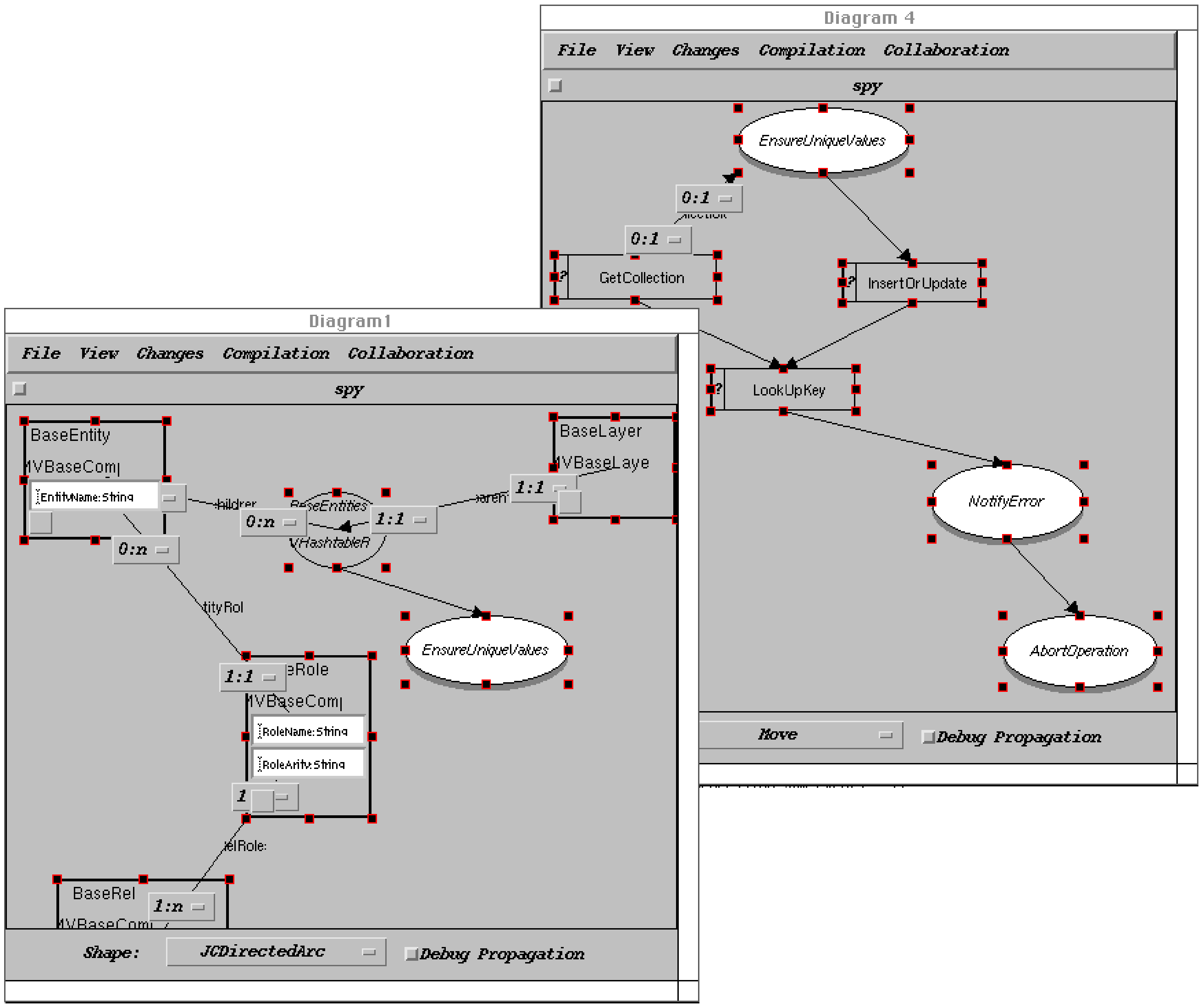
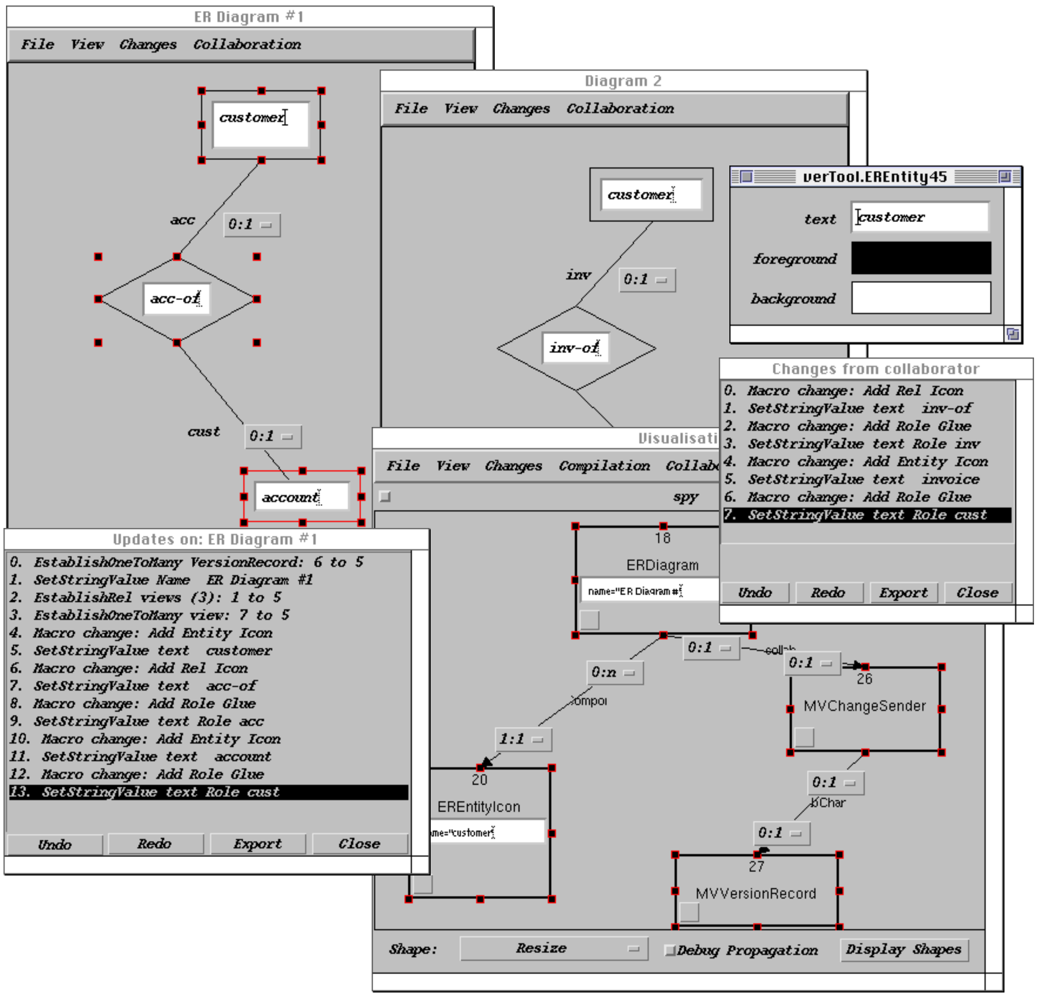


Figure 8. A JViews system being defined with JComposer.

A slightly modified form of the JComposer notation is used to allow users of JViews-based environments to visualise running JViews components, and to modify them. End-users can tailor their environments by adding additional components, filters and/or actions to visualisations using the same editing techniques and notation as in JComposer itself. Filters and actions may be added by end-users, for example, to notify them of changes made by other users, to automate simple tasks such as backing up data or specifying defaults for newly created components, and to invoke third-party tools from within JViews-based systems. Figure 9 shows an example of a running JViews-based environment, with the bottom-right view showing visualised, running JViews components. Other visualisation techniques support by JViews environments include group awareness (the highlighted entity icon in the top left diagram indicates another user is editing it), changes broadcast from other users (in the right hand diagram) and modification histories for components (left hand dialogue).



Updates on: ER Diagram #1

```
0.  EstablishOneToMany VersionRecord: 6 to 5
1.  SetStringValue Name  ER Diagram #1
2.  EstablishRel views (3): 1 to 5
3.  EstablishOneToMany view: 7 to 5
4.  Macro change: Add Entity Icon
5.  SetStringValue text  customer
6.  Macro change: Add Rel Icon
7.  SetStringValue text  acc-of
8.  Macro change: Add Role Glue
9.  SetStringValue text Role acc
10.  Macro change: Add Entity Icon
11.  SetStringValue text  account
12.  Macro change: Add Role Glue
13.  SetStringValue text Role cust
```

Changes from collaborator

```
0.  Macro change: Add Rel Icon
1.  SetStringValue text  inv-of
2.  Macro change: Add Role Glue
3.  SetStringValue text Role inv
4.  Macro change: Add Entity Icon
5.  SetStringValue text  invoice
6.  Macro change: Add Role Glue
7.  SetStringValue text Role cust
```

Figure 9. A running JViews system being visualised and extended.

7. Related Work

Various approaches to static program visualisation have been developed to aid developers during implementation and maintenance of programs. These approaches generally focus on the structure of modules making up logical components of the system. Examples include the Eiffel short and flat utilities [27] for viewing abstractions of OO programs, programming environment browsers, such as Semantic C++ [38], program filtering tools, such as multi-dimensional browsers [41] and star diagrams [14]. These approaches to supporting static software visualisation can only be used to visualise the basic structure of event-based software. They tend to focus on data and procedural flow of control and do not express the flow of events and triggering of actions via events. Some Architecture Description Languages (ADLs) and reverse engineering tools [44] provide additional abstractions which capture some event-based aspects of software architectures. Some modelling notations and supporting CASE tools also provide abstractions for capturing information about event-based systems, such as UML event diagrams [32]. Such modelling languages and ADLs tend to be at a higher level than our approaches to visualising event-based systems, and do not always provide detailed documentation of event protocols.

Dynamic program visualisation systems provide support for specifying, running and analysing executing programs. Examples include BALSA-II [7], Tarringim [31], and Zeus [8]. These systems try to provide high-level abstract analyses of programs, rather than the lower-level visualisations that we have concentrated on to support debugging and architecture understanding. Other approaches which use these lower-level visualisation techniques include 3D call graph analysis and highlighting [37, 39] and Self's visual debugger [9]. These approaches do not focus on the event-based nature of systems, however, being more general in nature. Our work, including ViTABaL, Serendipity and JComposer focus not only on program structuring, dynamic animation of structures and component inspection, but also on the passing of events between components at run-time.

Visual languages often provide a combination of static program structure visualisation using a visual language to structure and code a program, and animation of this visual language to dynamically visualise a running program. Examples of this approach include Prograph [10], Fabrik [25] and Garden [34]. Visual languages have also been used to design and visualise parallel and distributed systems, for example PEDS [46] and Meander [45]. These approaches are similar to ViTABaL's use of an ADL for structure and then animation of this structure to visualise a running program.

Process-centred environments (PCEs) tend to be event-focused in nature, with events either being used to control process enactment or used to trigger rules which constraint process enactment. Examples of event-driven PCEs include Regatta [40], Teamware FLOW [42], ADELE-TEMPO [5] and ProcessWEAVER [12]. Rule-based environments include SPADE [3] and Oz [6]. Because many of these environments use graphical languages to only visualise structure and not event behaviour, they can be very difficult to understand and use. Serendipity improves on both the static and dynamic visualisation of process models by indications of event flows between stages, the use of graphical filters and actions to explicitly program event-handling, and the use of various highlighting techniques to visualise enacted models.

8. Summary

We have described some of the issues involved in static and dynamic visualisation of event-based software systems. This includes the need for suitable, often visual, descriptions of the architecture of such systems, and for dynamic visualisations of parts of running event-based systems. We have described our experiences in providing static and dynamic visualisations for a range of such systems. These include notations and simple dynamic visualisations of CPRG event-based systems, visual languages supporting both static and dynamic visualisation of tool-abstraction based systems, static and dynamic visualisations of software process models, and static and dynamic visualisations of componentware systems. Much of our work has focused on the need to describe new abstractions for static and dynamic program visualisation, and to allow the reuse of existing abstractions.

We are working on improving the static and dynamic visualisation languages for JViews-based systems. We plan to use more expressive visual annotations along with the component, relationship, filter and action abstractions currently used. We are also using the dynamic visualisation views to support sophisticated visual querying of running systems. Finally, we are building reverse-engineering tools to examine third-party Java Beans in order to incorporate these into JComposer specifications, using our visual notations.

References

[1] Amor, R., Augenbroe, G., Hosking, J.G., Rombouts, W., and Grundy, J.C., "Directions in modelling environments," *Automation in Construction*, no. 4, 173-187, 1995.

[2] Apple,*OpenDoc Users Manual*, Apple Computers Inc., 1995.

[3] Bandinelli, S., Fuggetta, A., and Ghezzi, C., "Process model evolution in the SPADE environment," *IEEE Transactions on Software Engineering*, vol. 19, no. 12, 1128-1144, December 1993.

[4] Bandinelli, S., DiNitto, E., and Fuggetta, A., "Supporting cooperation in the SPADE-1 environment," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, .

[5] Belkhatir, N., Estublier, J., and Melo, W.L., *The Adele/Tempo Experience*, Software Process Modelling & Technology. Research Studies Press, 1994.

[6] Ben-Shaul, I.Z. and Kaiser, G.E., "A Paradigm for Decentralized Process Modeling and its Realization in the Oz Environment," in *Sixteenth International Conference on Software Engineering,* May 1994, pp. 179-188.

[7] Brown, M.H., "Exploring algorithms using BALSA-II," *COMPUTER*, vol. 21, no. 5, May, 14-36 1988.

[8] Brown, M.H., " Zeus: A System for Algorithm Animation and Multi-View Editing," in *Proceedings of the 1991 IEEE Symposium on Visual Languages,* IEEE Computer Society Press, 1991, pp. 4-9.

[9] Chang, B.W., Ungar, D., and Smith, R.B., "Getting close to objects," in *Visual Object-Oriented Programming*, Burnett, M., Goldberg, A., and Lewis, T., Eds. Manning/Prentice-Hall, 1995, chap. 9, pp. 185-198.

[10] Cox, P.T., Giles, F.R., and Pietrzykowski, T., "Prograph: a step towards liberating programming from textual conditioning, , IEEE Computer Society Press," in *Proceedings of the 1989 IEEE Workshop on Visual Languages,* 1989, pp. 150-156.

[11] Dannenberg, R.B., " A Structure for Efficient Update, Incremental Redisplay and Undo in Graphical Editors," *Software-Practice and Experience*, vol. 20, no. 2, 109-132, February 1990.

[12] Fernström, C., "ProcessWEAVER: Adding process support to UNIX," in *2nd International Conference on the Software Process: Continuous Software Process Improvement,* IEEE CS Press, Berlin, Germany, February 1993, pp. 12-26.

[13] Garlan, D., Kaiser, G.E., and Notkin, D., " Using Tool Abstraction to Compose Systems," *COMPUTER*, vol. 25, no. 6, 30-38, June 1992.

[14] Griswald, W., Chen, M.I., Bowdidge, R., and Morgenthaler, J.D., "Tool Support for Planning and Restructuring of Data Abstractions in Large Systems," in *Proceedings of the Fourth ACM Symposium on the Foundations of Software Engineering,* ACM Press, San Francisco, 1996, pp. 33-45.

[15] Grundy, J.C., "Multiple textual and graphical views for Interactive Software Development Environments," Ph.D. thesis, University of Auckland, Department of Computer Science, June 1993.

[16] Grundy, J.C., Hosking, J.G., Fenwick, S., and Mugridge, W.B., Connecting the pieces, Chapter 11 in *Visual Object-Oriented Programming*. Manning/Prentice-Hall, 1995.

[17] Grundy, J.C. and Hosking, J.G., "ViTABaL: A Visual Language Supporting Design By Tool Abstraction," in *Proceedings of the 1995 IEEE Symposium on Visual Languages,* IEEE CS Press, Darmsdart, Germany, September 1995, pp. 53-60.

[18] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Towards a Unified Event-based Software Architecture," in *Joint Proceedings of the SIGSOFT'96 Workshops,* Vidal, L., Finkelstein, A., Spanoudakis, G., and Wolf, A.L., ACM Press, October 14-15 1996, pp. 121-125.

[19] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Supporting flexible consistency management via discrete change description propagation," *Software - Practice & Experience*, vol. 26, no. 9, 1053-1083, September 1996.

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

[21] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Low-level and high-level CSCW in the Serendipity process modelling environment," in *Proceedings of OZCHI'96,* IEEE CS Press, Hamilton, New Zealand, Nov 24-27 1996.

[22] Grundy, J.C., Mugridge, W.B., and Hosking, J.G., "A Java-based toolkit for the construction of multi-view editing systems," in *Proceedings of the Second Component Users Conference,* Munich, Germany, July 14-18 1997.

[23] Grundy, J.C. and Hosking, J.G., "Serendipity: integrated environment support for process modelling, enactment and work coordination," *Automated Software Engineering*, vol. 5, no. 1, .

[24] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Support for end user specification of workflows, work coordination and tool integration," to appear in the *Journal of End User Computing*.

[25] Hosking, J.G., Visualisation of Object Oriented Program Execution, Proceedings 1996 IEEE Symposium on Visual Languages, Boulder, IEEE CS Press, Sept 1996, pp. 190-1.

[25] Ingalls, D., Wallace, S., Chow, Y.Y., Ludolph, F., and Doyle, K., "Fabrik: A Visual Programming Environment," in *Proceedings of OOPSLA '88,* ACM Press, 1988, pp. 176-189.

[26] Kellner, M.I., Feiler, P.H., Finkelstein, A., Katayama, T., Osterweil, L.J., Penedo, M.H., and Rombach, H.D., "Software Process Modelling Example Problem," in *Proceedings of the 6th International Software Process Workshop,* (Ed), T.K., IEEE CS Press, Hokkaido, Japan, 28-31 October 1990.

[27] Meyer, B., *Object Oriented Software Construction.* Prentice-Hall, 1988.

[28] Meyers, S., "Difficulties in Integrating Multiview Editing Environments," *IEEE Software*, vol. 8, no. 1, 49-57, January 1991.

[29] Microsoft, C., "ActiveX," http://www.microsoft.com/com/, 1997.

[30] Sun Microsystems, "Java Beans 1.0 Specification," http://www.javasoft.com/javabeans/,1996.

[31] Noble, J., Groves, L., and Biddle, R., "ObjectOriented Program Visualisation in Tarringim," *Australian Computer Journal*, vol. 27, no. 4.

[32] Rational, C., *UML Document Set Version 1.1*, Rational Corporation, http://www.rational.com/uml/references/, Version 1.1, 1997.

[33] Reiss, S.P., "PECAN: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering*, vol. 11, no. 3, 276-285, 1985.

[34] Reiss, S.P., " Working in the GARDEN Environment for Conceptual Programming," *IEEE Software*, vol. 4, no. 11, 16-26, November 1987.

[35] Reiss, S.P., "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, vol. 7, no. 7, 57-66, July 1990.

[36] Reiss, S.P., " Interacting with the Field environment," *Software practice and Experience*, vol. 20, no. S1, S1/89-S1/115, June 1990.

[37] Reiss, S., "A Framework for Abstract 3D Visualisartion," in *Proceedings of the 1993 IEEE Symposium on Visual Languages,* IEEE CS Press, Bergen, Norway, 1993, pp. 108-115.

[38] Semantec, I., *Semantec C++ Reference Manual*, Semantec Corporation, 1993.

[39] Stasko, J.T. and Wehrli, J.F., "Three-dimensional Computation Visualisation," in *Proceedings of the 1993 IEEE Symposium on Visual Languages,* IEEE CS Press, Bergen, Norway, 1993, pp. 100-107.

[40] Swenson, K.D., Maxwell, R.J., Matsumoto, T., Saghari, B., and Irwin, K., "A Business Process Environment Supporting Collaborative Planning," *Journal of Collaborative Computing*, vol. 1, no. 1.

[41] Taivalsaari, A., "Multidimensional Browsing," in *Proceedings of 8th Conference on Software Engineering Environments,* IEEE CS Press, Cottbus, Germany, 1997.

[42] TeamWARE, I., *TeamWARE Flow*, (http://www.teamware.us.com/products/flow/), 1996.

[43] Venable, J.R. and Grundy, J.C., "Integrating and Supporting Entity Relationship and Object Role Models," in *Proceedings of the 14th Object-Oriented and Entity Relationship Modelling Conferece (OO-ER'95),* Lecture Notes in Computer Science, Springer-Verlag, Gold Coast, Australia, 1995.

[44] Proceedings of the Joint SIGSOFT96 Workshops, Vidal, L., Finkelstein, A., Spanoudakis, G., and Wolf, A.L., ACM Press, ACM, San Francisco, 1996.

[45] Wirtz, G., "A Visual Approach for Developing, Understanding and Analyzing Parallel Programs," in *Proceedings of the 1993 IEEE Symposium on Visual Languages,* IEEE CS Press, 1993, pp. 261-266.

[46] Zhang, D.Q. and Zhang, K., "A Visual Programming Environment for Distributed Systems," in *Proceedings of the 1995 IEEE Symposium on Visual Languages,* IEEE CS Press, Darmsdadt, Germany, 1995.