In Proceedings of 2nd International Workshop on Software Architectures, ACM Press, San Francisco, Oct 1996.

Towards a Unified Event-based Software Architecture

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

Department of Computer Science[†]
University of Waikato
Private Bag 3105, Hamilton, New Zealand
jgrundy@cs.waikato.ac.nz

Department of Computer Science[††]
University of Auckland
Private Bag, Auckland, New Zealand
{john,rick}@cs.auckland.ac.nz

Abstract

Event-based software architectures are commonly used to solve a variety of problems. These architectures are complex to design and implement, however, especially with conventional, textual programming languages. We describe our recent work in developing visual languages and support environments for the design and implementation of a range of event-based software architectures. A synthesis of this work to produce a more general architecture description language and support environment is described.

1. Introduction

Complex software architectures often use event propagation between software components to maintain data consistency and achieve component coordination. Examples are: client/server architectures, where clients communicate with servers via messages; Macintosh Apple Events™, where applications exchange events [2]; FIELD [13], which integrates Unix tools via message-passing; and tool abstraction systems [3], such as spreadsheets, rule-based systems and active objects, where update events on shared data structures trigger "toolie" (processing) execution. These permit the development of flexible consistency and coordination mechanisms, and, if carefully designed, support much component reuse [3, 5, 13].

The design and implementation of such event-based software architectures using textual, general purpose programming languages is difficult [3, 5]. Each architecture component generally encapsulates some of the code to generate, propagate and handle events. Because of the distributed nature of inter-component communication, components are difficult to specify, coordinate, and visualise. Architecture Description Languages (ADLs) are needed which define the components, inter-component links, and the propagation, transformation, and interpretation of events along links. Environments to support ADLs for building such systems would also be useful. We describe a design for such an ADL and its environment.

We commence with a description of an event-based architecture (CPRGs) that motivated this work. Two visual languages are then introduced that provide elements of an ADL and which process other kinds of events (based on procedural actions). This is followed by a design that synthesizes and generalises those event-handling elements to produce EASY, a more general ADL and environment for event-based architectures.

2. CPRGs

Change Propagation and Response Graphs (CPRGs) [5], and their realisation in the MViews framework [4], provide an architecture for constructing software systems where consistency between multiple representations or views is important. The main application of CPRGs has been the development of integrated software development environments [1, 6, 7] and user interface tools [5, 11].

In CPRGs, each architectural item is represented as a *component*, which may be related to other components via *relationships* (also components). Components have *attributes* describing their state and *operations* (e.g. relationship addition, attribute modification) applied to them which change the graph's state. Operations generate *change descriptions* recording the resulting state changes. Inter-component consistency is maintained by modified components broadcasting their change descriptions to all components connected by relationships. Receiving components, including intermediate relationships, interpret the change descriptions and modify their state accordingly (possibly generating further change descriptions).

Figure 1 shows a CPRG representing the dialog box at top left. A dialog component represents the aggregate dialog box with name, position, and interface attributes. It has a parts relationship with its constituent components, and edit field components have caption components. Modification of the dialog's position attribute causes propagation of a change description (1) to each part via the parts relationship. Each part interprets the change description by changing its own position. Each edit field propagates the change on to its caption component causing it to also change position. Modification of a part's position (2) causes propagation of a change description to the dialog component. This may be interpreted in several ways: abort or resize of the dialog as a whole, if the component now lies outside the dialog box, or ignored, if there is no overlap.

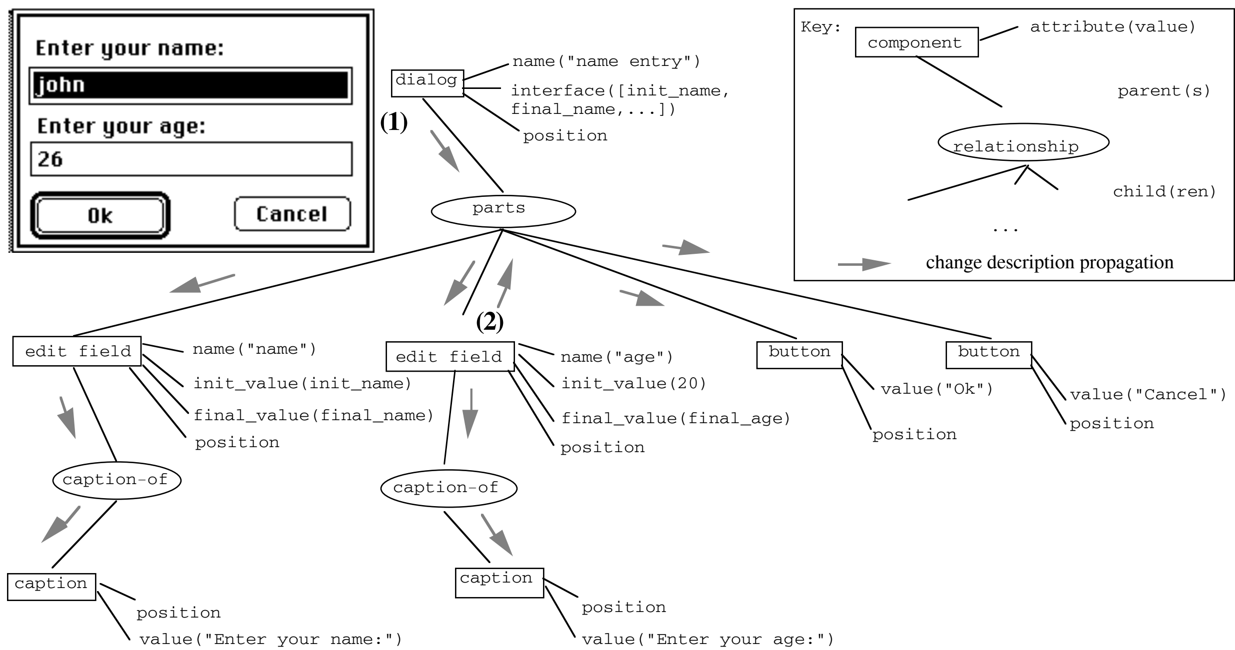

Figure 1. Event propagation in the CPRG event-based Software Architecture.

CPRGs are an example of an event-based software architecture, with state-change events propagated in the form of change descriptions via links represented by relationships. We have successfully used CPRGs to develop a wide range of software development environments and tools. However, although the architecture is an excellent implementation vehicle, we have found difficulty in adequately expressing designs. The visual architecture description language notation used in Figure 1 is useful for designing and documenting CPRG-based implementations, but is not used when implementing these systems with conventional programming languages.

Implementers must hand-code the structures of the CPRG component definitions (as object-oriented classes), and must handcode the event response mechanisms. While many CPRG components are highly reusable, especially CPRG relationships [5], much hand implementation remains for complex CPRG-based systems. Thus a major problem is that there is no explicit representation of event flows in the textual implementation language code. Event generation is combined with the coding of operations, while responses are coded into receivers. This makes the behaviour of systems difficult to understand at the design level. We have explored several approaches to more explicitly representing event flows and responses to produce an ADL for CPRGs which can be better translated into an implementation.

3. ViTABaL

ViTABaL is based on the tool-abstraction paradigm [3]. *Abstract data structures* (ADSs) are shared by a collection of co-operating *toolies*. Each toolie supplies part of the overall system function, extending or modifying the functionality of the ADSs and other toolies. Interaction is via action-based events, with toolies being able to augment or change calls to the interface of ADSs; this is in contrast to CPRGs, where the events are due to state changes. In [7] we present a design notation for tool abstraction, which elaborates on the informal one of [3], together with the ViTABaL environment for supporting the notation.

Figure 2 shows an example of the ViTABaL notation (and environment) used to design a tool abstraction version of the KWIC (Key Word In Context) system of [12]. ADSs (rectangles) are connected to toolies (ovals) by event connections. Toolies respond to action events (named calls with their arguments) specified on the event connections. A variety of event types are supported: broadcast (event(args)), where the named events are sent from the sender ADS (or toolie) to the receiver toolie; request (event(args)), like broadcast, but where the sender waits for the receiver to respond; listen_before (event(args)), where the listener is sent an event <u>before</u> the receiver responds to the event; and listen_after (event(args)), where the listener is sent the event <u>after</u> the receiver has responded to the event. The listen_before and listen_after event types are particularly powerful as they permit additional toolies to transparently "plug in" to an existing design. This simplifies design modification and better facilitates component-based software design than traditional approaches [3].

Additional annotations are supported in ViTABal for indicating that toolies execute concurrently [7]. Concurrent toolie execution requires synchronisation at various points to ensure toolies operate in a sensible order (when required) and that data fetches/updates from ADTs are atomic. Synchronisation is achieved by having ADTs which are accessed by concurrent toolies enforcing locking and sequencing of toolie requests. More sophisticated synchronisation is achieved by designers using coordinator toolies which process requests from other, concurrently executing toolies. We have had ViTABaL programs running on several networked Macintosh computers with concurrent, interacting toolies running on different machines.

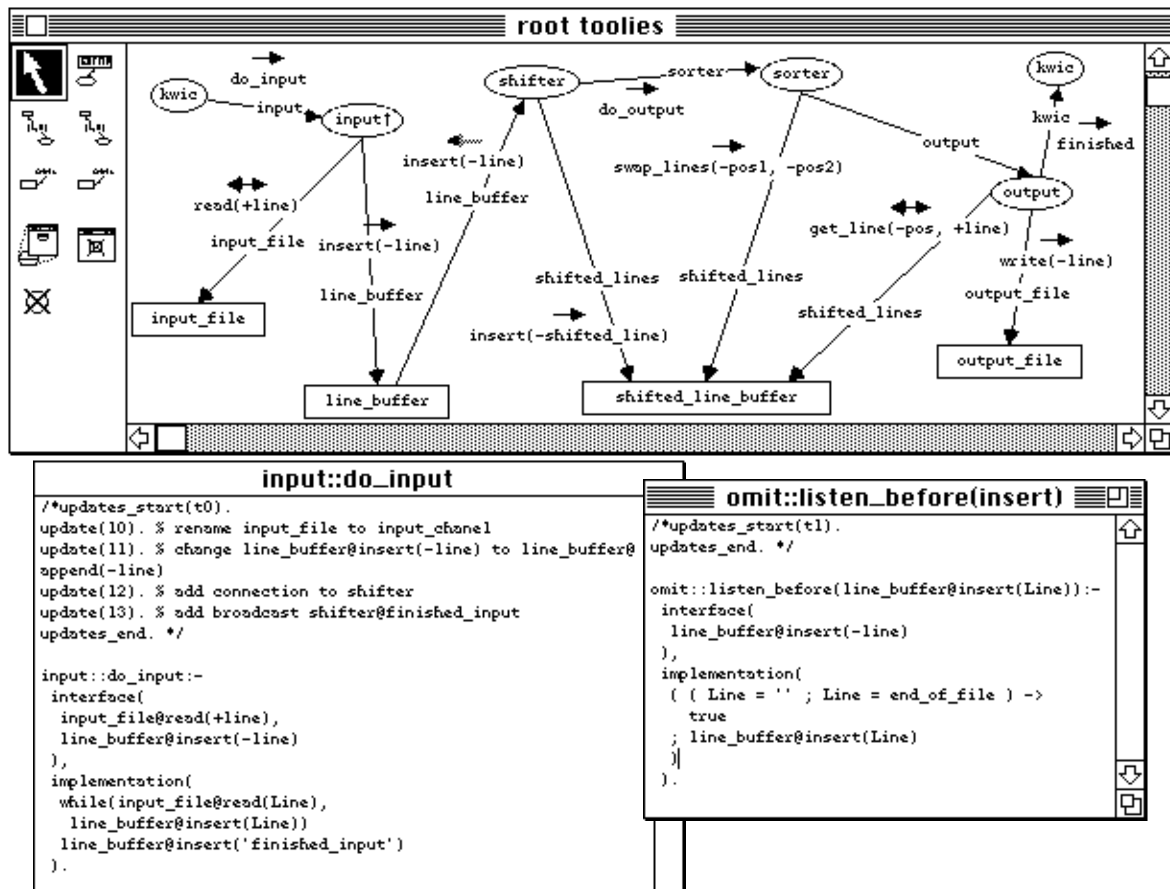

Figure 2. Specifying event propagation and handling in ViTABaL.

A great advantage of ViTaBAL over most other Tool Abstraction systems is that when a program is moved from multiple machines back to one machine, or vice-versa, the designer need change no ViTABaL designs. The ViTABaL compiler generates the required low-level message passing and synchronisation code.

ViTABaL toolies may be implemented within exploded ViTABaL diagrams, allowing hierarchical toolie event responses to be coded. Bottom-level toolie event responses are coded textually (a variety of Prolog is currently used), as shown in the textual views in Figure 2. These responses can be quite small, or can be implemented by large programs, using any conventional program structuring technique (i.e. not tool abstraction-based). ViTABaL generates object-oriented code which very efficiently implements tool abstraction designs, and supports visualisation of executing systems through design diagram animation [7]. ViTABaL code is very efficient as the code generator is able to determine whether listen before/after processing and synchronisation or inter-machine message passing is required. If these high-overhead activities are not used by some toolies, then very efficient method-calling code is generated. This is unlike most other Tool Abstraction systems, which always assume interdepencies may exist and thus always encode high-overhead active object or trigger mechanisms into code.

ViTABaL thus provides an ADL for the event-based tool abstraction paradigm, together with facilities for efficiently realising that design. Weaknesses of ViTABaL include the use of event connections as the sole mechanism for system structuring, which is not as rich a structural notation as CPRG relationships. Unlike CPRGs, ViTABaL cannot handle state change events directly; they must be handled via action events, leading to redundancy when several actions can lead to a common state change. Event responses are coded textually, and design level support for this is desirable.

4. Serendipity

Serendipity is a process-centred environment that handles both state change and simple action events, albeit in a less-general framework than our other approaches. Serendipity provides graphical and textual views for process modelling, enactment and improvement [8, 9, 10]. In Figure 3, 'm1:model1-process' shows a Serendipity model for part of a software process. Process *stages* are enacted by completion events from preceding stages. Serendipity also provides graphical views for specifying event handling for process models via filters and actions, such as 'm1.3:done testing' in Figure 3. Filters (rectangles) apply pattern-matching to incoming events, propagating matching events to other filter/actions. Actions (ovals) respond to events by executing operations which generate new events, display information, or modify data. The 'm1.3:done testing' event handler specifies that when process stage 'm1.3:check changes' is made current (i.e. testing begun, detected by filter 'Made Current') or the stage finishes (testing stopped, detected by filter 'finished testing'), action 'Notify Role' notifies the 'coders' for stage 'm1.2:implement changes'.

Events may be caused by stage enactments (as shown above), artefact modifications, or tool applications. The filter/action language also supports: filters and actions which receive multiple events and do not terminate after handling a single event; event handling submodels; and other filtering annotations [8]. An API interface allows filters and actions to be defined in Prolog and called when an event is received, making the language extensible. This also allows filter/actions to be implemented as hierarchically using our visual language, reusing library filter/actions ("tenplates"), or using the API interface, which may be a small piece of Prolog code or even an interface to a large existing system (a database, client-server connection, etc.). Serendipity filter/actions can run concurrently in a similar manner to ViTABaL toolies, and use a similar approach to coordination. Either actions are built which handle inter-action coordination or concurrent access to data is moderated by artefact interfaces. Serendipity is implemented using MViews, utilising CPRG structures and event propagation.

The filter action event handling component of Serendipity thus extends the event-based model of CPRGs. Filters provide a visual design level notation for specifying event responses, whereas ViTABaL visually describes only the toolie event connections, and not the event responses. Serendipity has the same advantages as ViTABaL over related textual approaches (such as rules, action routines or active object triggers) in that event propagation and handling is explicitly represented in a high-level, graphical manner. However, it is currently strongly tied to the Serendipity process modelling environment, and thus cannot be used to design general event-based systems.

5. EASY: A Unified Event-Based Architecture

To unify and generalise our work on CPRGs, ViTABaL and Serendipity, we have designed EASY (Event Abstraction SYstem). EASY permits CPRGs state-change events and ViTABaL action events to be handled in a unified manner, while incorporating the event filtering and response abilities of Serendipity. This aims to maintain the advantage of ViTABal and Serendipity of visual representation of event propagation and response mechanisms, while improving on their handling of the structural aspects of event-based software architectures, which is done well in CPRGs.

Figure 4 shows an example of an EASY instance modelling a dialog similar to that of Figure 1, but with multiple edit fields. CPRG components and relationships form the "backbone", specifying data components and interconnectivity, such as dialog, fields and ok_button. ViTABaL toolies, such as follow_drag and abort_input, use CPRG structures as their shared ADS pool, and respond to the events affecting these components. Toolies specify events they're interested in, as in ViTABaL, and listened-to components generate event representations as change descriptions, as in CPRGs, before or after the component state changes, as required by the listening toolies. The change descriptions (event representations) are propagated to the listening toolies which match them to the event patterns they respond to, and the appropriate response code is invoked. Serendipity filters and actions are used to specify toolie responses, and support complex event propagation via pattern-matching and actioning via library actions, definition of submodels, or the API interface. Usage connections indicate toolie parameters (e.g. field_value) and toolie usage (e.g. used_by).

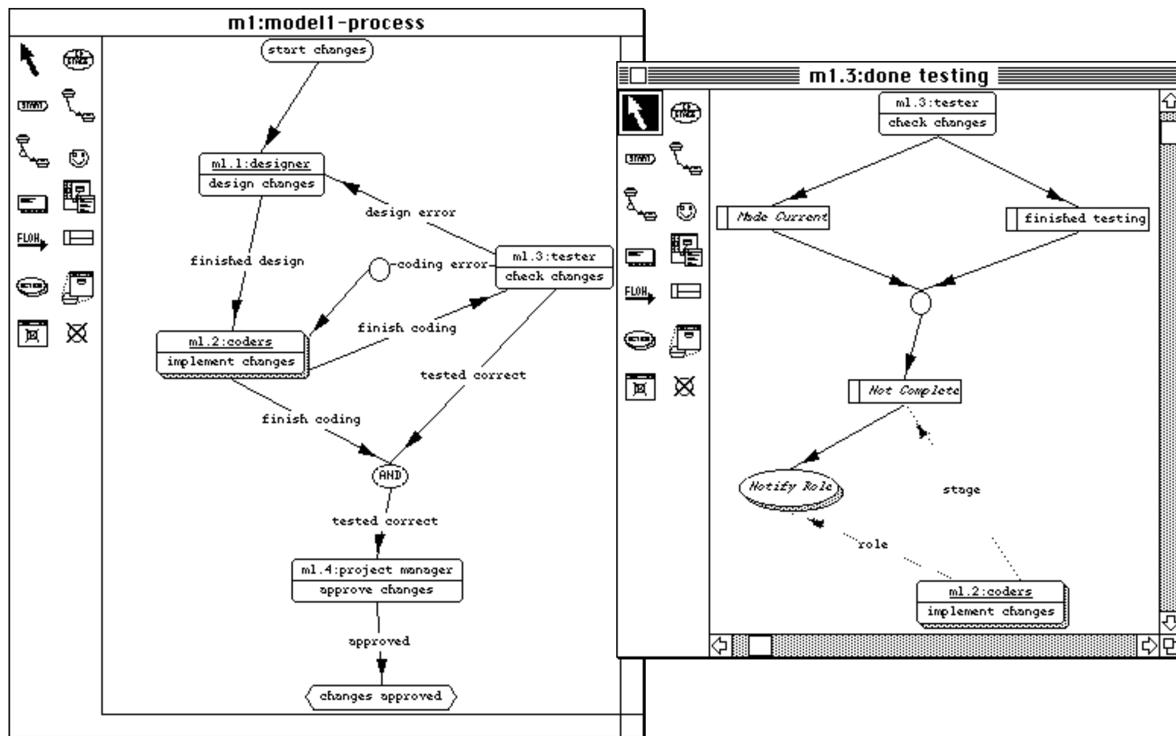

Figure 3. Specifying event filtering and actioning in Serendipity.

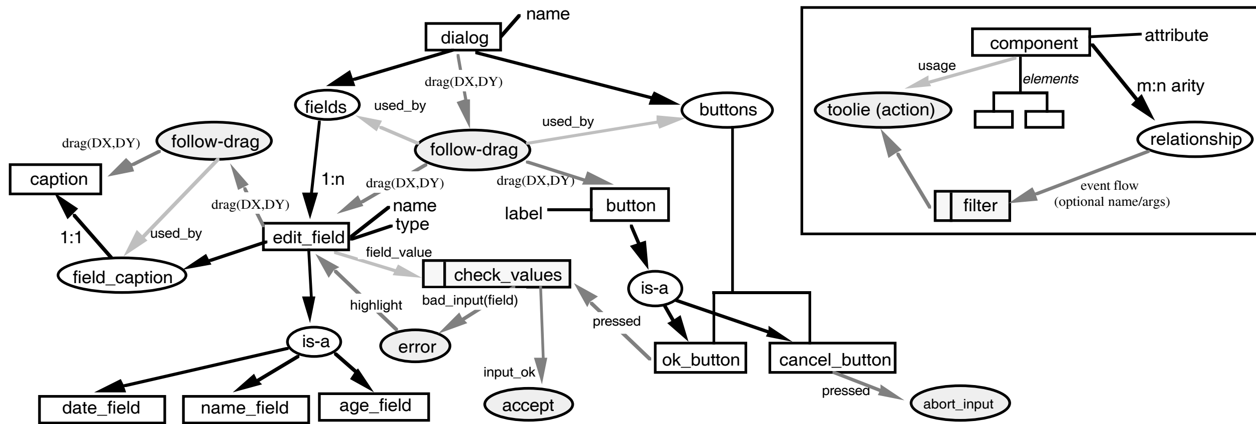

Figure 4. Merging CPRG organisation, ViTABaL event propagation and Serendipity event filtering/action.

We are currently implementing an environment to support EASY. Multiple views will be provided as in ViTABaL and Serendipity, allowing system structure to be modularised vertically, via multiple, partially overlapping views, or horizontally, using hierarchical views. Serendipity's API is preserved to allow developers to extend the functionality of the language by defining filters and actions (toolie responses) using the environment's implementation language. A ViTABaL-like visualisation system will allow developers to examine the execution of architectures using EASY views. Serendipity filter/actions are all currently interpreted, but these will be compiled where there are not dynamic structures to achieve better performance, as is currently done with ViTABaL event response code.

6. Summary

Our work has focused on providing visual ADLs and supporting environments for the design and construction of complex event-based software architectures. CPRGs provide an ADL for building such architectures using components, relationships and versatile state-change event propagation and response. ViTABaL provides a visual language and environment for the design and construction of tool-abstraction action-event-based architectures. Serendipity provides an extensible filter/action language for propagating and responding to events. A synergy of these languages and environments in EASY will provide wider-ranging support for event-based architecture design and construction. We are currently building an environment for EASY; this will support concurrent toolie execution and synchronisation enhancements. Integrating this environment with user interface specification and construction tools, including MViewsDP [5] and Skin [11], will allow user interface events to be processed via EASY abstractions.

References

[1] Amor, R., Augenbroe, G., Hosking, J.G., Rombouts, W., and Grundy, J.C. Directions in modelling environments. *Automation in Construction* , 4 (1995), 173-187.

[2] Apple Computer Inc. *Inside Macintosh: Volumn IV,* Addison-Wesley (1991).

[3] Garlan, D., Kaiser, G.E., and Notkin, D. Using Tool Abstraction to Compose Systems. *COMPUTER 25*, 6, 30-38.

[4] Grundy, J.C. and Hosking, J.G. A framework for building visual programming environments. In *Proceedings of the 1993 IEEE Symposium on Visual Languages,* IEEE CS Press, Bergen, Norway, September 1993, pp. 220-224.

[5] Grundy, J.C., Hosking, J.G., and Mugridge, W.B. Supporting flexible consistency management via discrete change description propagation. to appear in *Software - Practice & Experience*.

[6] Grundy, J.C., Hosking, J.G., Fenwick, S., and Mugridge, W.B. Connecting the pieces. Chapter 11 in *Visual Object-Oriented Programming,* Burnett, M., Goldberg, A., Lewis, T. Eds, Manning, (1995).

[7] 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.

[8] Grundy, J.C., Hosking, J.G. "Serendipity: integrated environment support for process modelling, enactment and improvement," Working Paper, Department of Computer Science, University of Waikato, 1996.

[9] Grundy, J.C. and Hosking, J.G. Visual Language Support for Planning and Coordination in Cooperative Work Systems. In *Proceedings of the 1996 IEEE Symposium on Visual Languages,* IEEE CS Press, Boulder, 1996.

[10] Grundy, J.C. and Venable, J.R., "Towards an environment supporting integrated Method Engineering," In Proceedings of Method Engineering '96, Atlanta, August 26-28, 1996.

[11] Hosking, J.G., Fenwick, S., Mugridge, W.B., and Grundy, J.C. Cover yourself with Skin. In *Proceedings of OZCHI'95,* Wollongong, Australia, Nov 28-30 1995, pp. 101-106.

[12] Parnas, D.L. On the Criteria To Be Used in Decomposing Systems into Modules. *CACM 15*, 12, 1053-1058.

[13] Reiss, S.P. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software 7*, 7, 57-66.