Reconciling Environment Integration and Component Independence*

Kevin J. Sullivan
David Notkin
Department of Computer Science & Engineering, FR-35
University of Washington,
Seattle, WA 98195 USA

Abstract

We present an approach that eases the design and evolution of integrated environments by increasing independence among components. Our approach combines mediators, which localize relationships, and a general event mechanism, which increases the independence of components from relationships in which they participate. To clarify our notion of independence and its relationship to evolution, we analyze four designs for a simple environment. The first three show how common approaches compromise independence in various ways. The fourth design demonstrates how our approach overcomes these problems. Our event mechanism is specially designed to support integration and evolution. We discuss detailed aspects of mediators and events by presenting three environments we have built. Our approach has also given us significant insights into other related systems.

Introduction

An integrated environment is a complex piece of software. Effectively managing this complexity—especially in the face of evolving requirements—is a demanding software engineering problem. One problem—as Parnas has shown for general software systems [Parnas 72]—is that, with respect to ease of evolution, not all designs for a given integrated environment are equal. A challenge, then, is to find design approaches that isolate decisions likely to change as requirements evolve.

At the requirements level, an integrated environment is a collection of user-level software tools and a collection of automatically maintained relationships among these tools. Relationships tie tools into cohesive environments, freeing users from having to manually integrate the tools. At the design level, an integrated environment is a collection of components[1] and a collection of relationships among them.

Requirements for integrated environments commonly change in several ways. The collections of tools and the collections of relationships among them change, and individual tools and relationships change. Design techniques that ease the evolution of integrated environments must account for these classes of change.

Designs that effectively separate component from relationship concerns minimize the changes that have to be made as requirements change. To achieve this separation, relationships should be designed as independent components, distinct from the components they relate. Additionally, components should be independent of relationships in three ways: they should be able to execute without having to participate in particular relationships; their source should be defined without reference to relationships; and participation in relationships should not prevent other components from accessing them.

In practice, independence is usually compromised

*This research funded in part by AFOSR Grant 89-0282, NSF Grant CCR-8858804, Digital Equipment Corporation, and Xerox Corporation. KJS is funded by a GTE fellowship.

[1] In our loose definition, components are the design units in a given environment. Examples include simple abstract data types, objects in object-oriented systems, and perhaps even user-level programs.

in one of two ways. Either relationships are defined as separate components that hide the components they relate, or they are designed as code and data embedded in these components. The first prevents independent access to components, which makes it hard to add relationships that involve them. The second fails to isolate relationships in separate components and produces components defined to participate in specific relationships, often preventing them from executing independently.

To ease the design and evolution of integrated environments, we present a technique and supporting mechanism that achieves independence. In particular, we design our relationships as separate, first-class components, called mediators, that do not preclude independent access to the components they relate. We use a general event mechanism to allow components to participate in relationships—by invoking mediators—without having static references to them.

In contrast to designs that embed relationships in components, mediators localize relationships, which makes it easier to change existing relationships and to add new ones. By supporting indirect invocation of mediators, events allow unmodified components to participate in evolving relationships. This combination eases environment evolution by separating concerns more effectively than current approaches. An additional advantage of our approach is its lightweight character.

Alternative Design Approaches

To demonstrate the influence of independence on evolution, we analyze two representative changes in four designs of a simple integrated environment.

This environment has two tools, a set of vertices, V, and a set of edges, E. We integrate these sets by imposing a relationship that ensures that together they form a graph:

$$G \equiv E \subseteq V \times V.$$

Commands are provided to create vertices, to create edges given two vertices, and to insert and delete vertices and edges to and from V and E. On initialization, V and E are empty. When a vertex is deleted from V, edges upon which it is incident are deleted from E to maintain G. Similarly, when an edge is inserted into E, the edge's vertices are inserted into V if necessary.

The first change we consider modifies the relationship between V and E by providing a command that toggles between eager and lazy maintenance of G:

$$G' \equiv (eager \wedge G) \vee lazy.$$

When toggling to eager mode, G is established by inserting into V all vertices upon which any edge in E is incident.

The second change we consider integrates a new tool, S, to track the cardinality of V, according to the relationship:

$$C \equiv S = |V|.$$

When the contents of V change, C is maintained by incrementing or decrementing S. Similarly, when S's value is changed, vertices are arbitrarily inserted to or deleted from V.

All four designs use the same component definitions to represent vertices and edges. In contrast, each defines different versions of components `V` and `E` to represent V and E. The first and last designs define an additional component, `G`, responsible for maintaining the relationship G.

Note: In the figures, rounded boxes represent components. Components within other components are hidden. Rectangles represent methods (component entry points). Methods, such as `V_Update(e)`, intended to maintain relationships, such as G, are shaded. Black arrows represent direct calls, while gray arrows represent events. To focus on the maintenance of relationships, calls made to implement basic component operations are omitted. For example, no arrow indicates the obvious call made by `G.V_Delete(v)` to `V.delete(v)`.

Design #1: Encapsulation. The first design, shown in Figure 1, defines a component, `G`, that provides the interface through which the user manipulates V and E. `G` makes calls to `V` and `E` to insert and delete vertices and edges on the user's behalf. When such a call disrupts G, `G` invokes one of its private update methods to reestablish G. For example, after `G.V_Delete(v)` deletes `v`, it calls `E.Update(v)`, which iterates over the edges in `E` to delete any that are incident on `v`. To prevent other components from bypassing the necessary update methods, `G` hides `V` and `E`.

In this design, V, E, and G are represented by separate components. Moreover, `V` and `E` make no references to one another or to `G`. `V` and `E` are not

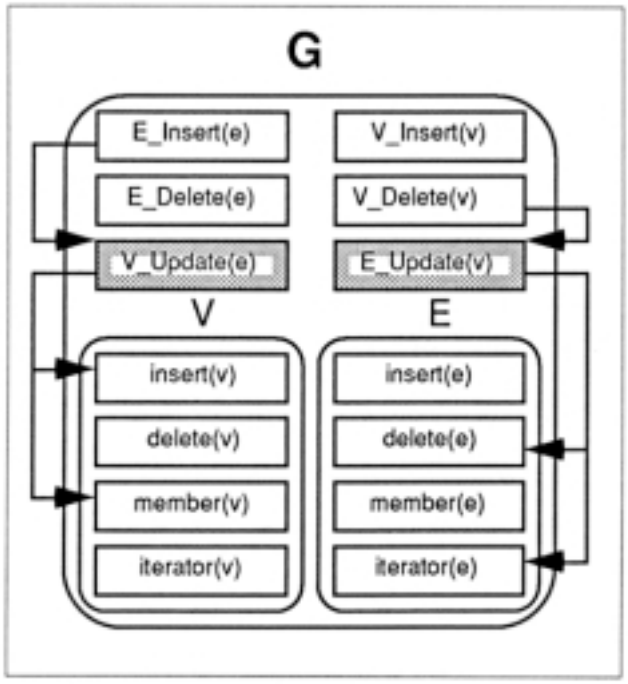

Figure 1: Integration by encapsulation.

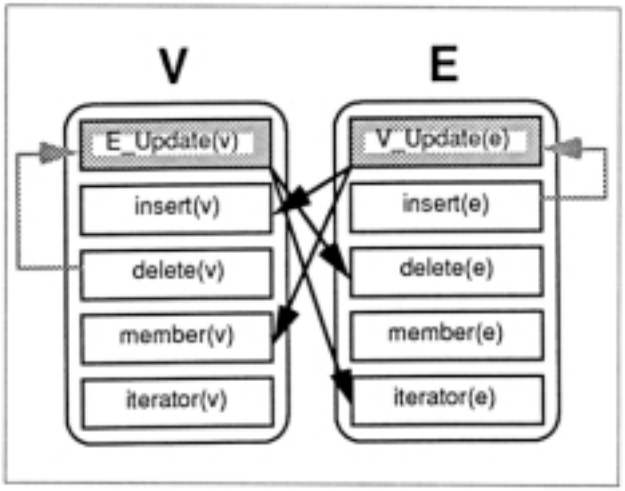

Figure 2: Integration by hardwiring.

fully independent, however, since G prohibits other components from accessing them.

Because G is represented as a separate component, the effects of changing G to G' are localized in G. Making this change requires addition of a lazy-mode bit and a method to toggle this bit and reestablish consistency. V and E need not change. However, because G hides V, integrating S is complicated. We must include component S in G and expand G's interface to provide access to S. Also, to implement C we must add update methods to G and modify G's public methods to invoke them appropriately. Further, these changes must be made without compromising the code that maintains G (or G'). The implementations of G and C are merged, reducing their independence.

Unix tools are typical of this approach. They often have rich, well-modularized, inner structures, but these are hidden from other tools. This limits the potential for integrating Unix tools without merging their implementations or making significant changes to expose representations and events.

Design #2: Hardwiring. The second design, shown in Figure 2, represents an attempt to ease the integration of S by exposing V (and E). To maintain G, V calls its update method, `E_Update(v)`, which makes appropriate calls to E. Likewise, E calls its update method, `V_Update(e)`, which calls V to maintain G.

In this design, V and E together provide the external interface to the user. G is not implemented separately, but appears as code distributed through V and E. This design allows independent access to V and E, but overall independence is diminished. V and E are programmed to maintain G by making direct calls to each other. Because of this, neither can execute in the absence of the other.

This design makes it hard to change G to G'. Because G is not localized, both V and E must be modified. The decentralization of G also makes it hard to decide whether to place the lazy-mode bit and toggling method in V, E, or both.

Also, the independent accessibility of V fails to ease the integration of S because V must be modified to maintain the new relationship, C. Changing V to handle C in addition to maintaining G further decreases the independence of V.

This design complicates evolution, making it difficult both to modify existing relationships and to add new ones.

Design #3: Events. The third design, shown in Figure 3, makes a small change to the second design to allow new components (such as S) to be integrated with V and E without changing V or E.

In this design, the update methods in V and E are swapped, allowing V and E to maintain G by making

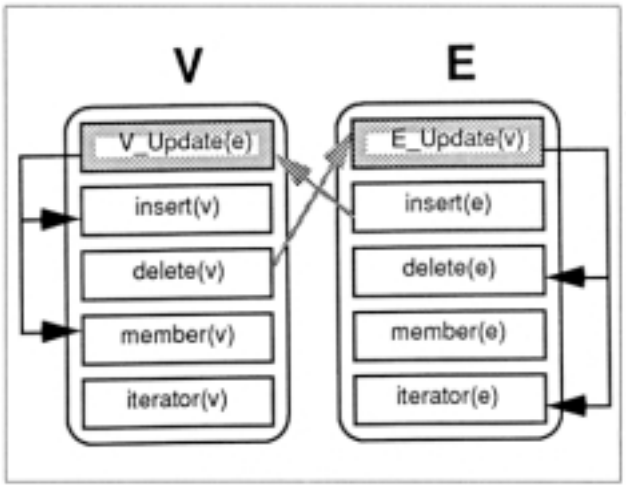

Figure 3: Integration using events.

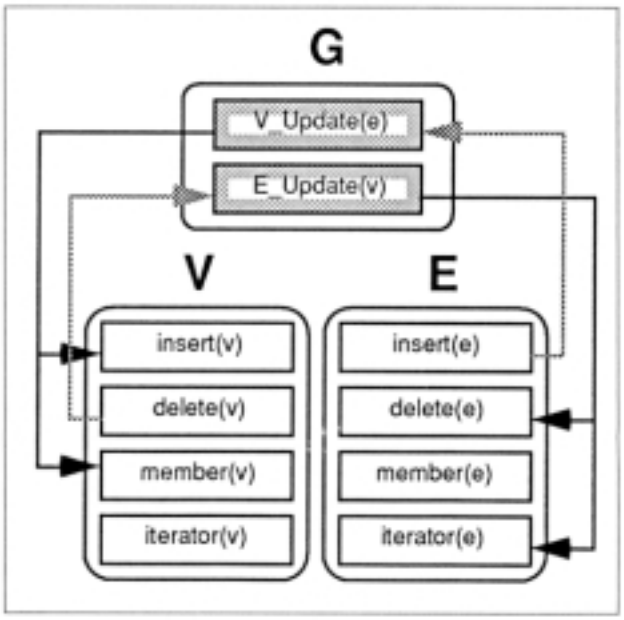

Figure 4: Integration using mediators and events.

a single call to an update method in the other. Simplifying the update protocol in this way makes it possible for an event mechanism to send update requests. Events allow V and E to indirectly and dynamically invoke update methods in other components (including each other) without having static references to them. Where V and E used to make explicit calls to update routines, they now announce an event instead.

Changing G to G' poses the same problems as in the previous design, since the code for G is still distributed through V and E.

On the other hand, this design does allow S to be integrated with V without requiring any changes to V. When a vertex is inserted into or deleted from V, V announces an event that invokes an update method in S to maintain C. When S's value is changed, S calls V's methods directly to maintain C. This approach to integrating S yields a design that is a hybrid of designs two and three. An alternative approach preserves the independence of S by modifying V to respond to update request events from S.

Neither of these alternative approaches for integrating S is perfect. The hybrid approach compromises the independence of S, hindering future evolution. The other approach modifies V, which is precisely what the third design was devised to avoid.

Many contemporary environments use this event-based style of integration. The Smalltalk-80 Model-View-Controller, for instance, uses the hybrid approach, with views responding to events but manipulating models directly [Krasner & Pope 88]. FIELD, in contrast, is entirely event-based [Reiss 90].

Design #4: Mediators and Events. The first design represented G as a separate component and defined V and E independently. However, it prevented independent access to V and E, which made it hard to integrate S. The third design exposed V and E and also enabled them to execute independently of relationships. However, G was distributed between V and E, which made it hard to change G and complicated the integration of S.

The fourth design, shown in Figure 4, combines the best aspects of the first and third designs, increasing independence and thus easing evolution. This design represents G as a separate component, allows independent access to V and E, and defines V and E independently. The update methods that maintain G have been isolated in component G. When vertices are deleted or edges inserted, corresponding events from V or E activate the appropriate update method in G. G, in turn, calls V or E to reestablish G.

As in the first design, changing G is easy. This design also makes it easy to integrate S. This can be done by introducing a new component C, with update methods to maintain the relationship, C, between S and V. For example, when a vertex is deleted from V, an event is sent to both G and C, which update E and S respectively.

In contrast with all three earlier designs, this approach makes it easy both to change G to G' and to

integrate S under C by keeping G and C (the representations of G and C) separate, and by keeping V, E and S independent.

Mediators

Mediators—such as G and C in the fourth design—are first-class components used to represent or maintain relationships among other components. Mediators can maintain state and call other components in ways that produce side-effects, which enables them to maintain many kinds of relationships efficiently. Defining mediators as ordinary components has additional advantages. They can export abstract interfaces, announce events, and be organized in arbitrary ways. This also avoids the need to define, understand, and adopt new mechanisms. Mediators thus represent a design technique, rather than a new mechanism.

Mediators depend on the components they relate, but remain independent of relationships in which they participate. For example, augmenting G in design four to announce events when vertices or edges are inserted or deleted enables the integration of G with a new tool that tracks the number of changes to G (considered as a graph component). So, whereas G knows about the components it manages (V and E), it would remain independent of the mediator that represents this new relationship. Keeping mediators independent of relationships in which they participate further eases the design and evolution of complex environments.

Events

Implementing communication among components using direct method invocation reduces component independence. Event mechanisms increase independence by enabling communication without requiring statically defined connections. For instance, in the fourth design, because V, E, and S post update requests using events, it is easy to configure the environment with any combination of G, G', or C.

Event mechanisms vary widely and depend on many factors. We have identified a simple model—based on events, methods and the relationship among them—that has helped us to analyze diverse event mechanisms.

Event-Method Relations. An event mechanism can be characterized in terms of an event-method relation, *EM*. An event-method tuple, (x.e,y.m), is in *EM* if and only if, when component x announces event e, y's method m is invoked. *EM* must minimally provide register, unregister and lookup operations. Register and unregister add and remove event-method tuples. Lookup maps an x.e to the set of y.m's to be invoked when x.e is announced.

The parameters to register (and unregister) are either explicit tuples or implicit descriptions from which the tuples are computed. In the Smalltalk-80 MVC, for example, explicit tuples are registered; when object x registers with object y, the tuple (x.change(),y.update()) is added to *EM*. Registration in the Common Lisp Object System (CLOS) is also handled explicitly; wrapper methods are associated with method invocation and return events using inheritance [Bobrow et al. 88]. In FIELD, in contrast, registration is done in terms of regular-expression/method pairs. When a component announces an event, by sending an ASCII string, an event server computes the methods to invoke by matching the event against the patterns and selecting the methods associated with the patterns that match.

An *EM* can also support other operations. ThingLab, a constraint-programming system, uses its *EM* (the constraint graph) to detect circular dependencies [Borning 81]. Some attribute grammar systems process their *EM*'s to optimize attribute evaluation. FIELD queries its *EM* (the event server) to avoid spawning additional copies of network tools. To support debugging, an *EM* could also provide queries such as "what methods will be invoked if I announce 'breakpoint set at line 12'?" In all of these cases, the *EM* is being used for purposes other than for passing update requests.

Environment Integration. Event mechanisms that support environment integration should satisfy four requirements: events should be declared explicitly; any component should be able to declare events; event names and signatures should not be system-defined; and the passing of parameters from events to methods should be specified at registration time.

Events should be declared in component interfaces because events are part of the specification of a component. Clarifying component responsibilities in this way makes it easier to design, modify and reason about components that use events. Most mechanisms, including FIELD and the Smalltalk-80 MVC,

do not meet this requirement.

To ease the integration of arbitrary components, any component should be able to declare events. Some systems limit events to certain classes of components. In APPL/A, for instance, only relation components export events [Sutton, Heimbigner & Osterweil 89]. Similarly, in AP5, only the data base announces events [Cohen 89].

Components should be able to declare events with arbitrary names and signatures to ease component design and to avoid limiting the ways in which components can be integrated. Many systems limit the events that components can announce. The Gandalf kernel, for example, defines about a dozen events used to invoke action routines [Habermann & Notkin 86]. CLOS limits events to method call and return, while LOOPS limits them to variable read and write [Stefik, Bobrow & Kahn 86]. In contrast, some other systems satisfy this requirement. AP5 events, for instance, are arbitrary relational predicates. FIELD events are arbitrary ASCII strings. The Smalltalk-80 MVC falls in the middle. Although it provides a fixed event, `change(<parameters>)`, the signature permits arbitrary objects, which can encode arbitrary events, to be passed. Events with arbitrary names and signatures also ease design of and reasoning about components that use events.

Finally, the mapping of event parameters to parameters of the associated methods should be specified at registration time. This increases the independence of components that receive events from those that send them by permitting their signatures to differ. This implies that the event mechanism must handle parameter passing when events are announced. The Smalltalk-80 MVC, for instance, requires that corresponding change and update methods have matching signatures. In FIELD, on the other hand, the patterns specify the parameters to be extracted from events.

A Lightweight Design. We have designed an event mechanism satisfying these requirements, implementing versions for use in both C++ and Common Lisp. Components, events, and connectors (which associate events with methods) are designed as object-oriented classes. Specific components, events, and connectors are instances of such classes.

An `Announce(<parameters>)` method is exported by each event class. The signature of the event is given by the signature of this method. Components declare events by declaring event objects as instance variables. A component announces an event by calling its `Announce(<parameters>)` method. Thus events are declared; any class can declare events; and arbitrary events can be declared.

A connector represents a tuple in EM. Our event-method relation, EM, is implemented by the set of existing connectors. A connector, `c`, representing the tuple (`x.e`,`y.m`) is implemented as an object that stores a pointer, `p`, to `y` and exports a dynamically-bound method, `Fire()`, defined to invoke `y.m` through `p`.

Events and connectors cooperate to implement flexible parameter passing. When a component `x` announces event `e`, the object `e` stores the event parameters in its own state. Before calling `y.m`, `c`'s `Fire()` method fetches whatever parameters are needed from `e`. To implement this protocol, a connector must store a pointer to its event and have code to fetch its parameters through an interface provided by the event.

A bootstrap event mechanism, not generally visible, is used to associate connectors with events. This bootstrap mechanism, similar to the Smalltalk-80 MVC's change/update protocol, is built into the base classes for events and connectors. This underlying event mechanism has its own, restricted event-method relation, em. When a connector, `c`, registers with an event, `e`, the tuple (`e.Signal()`,`c.Fire()`) is added to em. This completes the circuit: when the (component-level) event, `e`, is announced, it stores its parameters and then announces its own (bootstrap-level) event, `Signal()`. This activates all associated connectors by invoking their dynamically bound `Fire()` methods. In our implementations, multiple connectors associated with a given event are activated in an unspecified sequential order.

This design satisfies the requirement for flexible parameter passing.

This event mechanism design is easily implemented in any reasonable object-oriented language in a day or two. This has at least three benefits. It eliminates costly design efforts from our research. It broadens the scope of language systems—and thus domains—in which we can explore integration and independence. And, it makes it easy to adopt our approach without changing languages. We have intentionally left the semantics underspecified, providing room for users to tailor our approach to their environment. As an obvious example, our C++ implementation uses C++ typing to ensure that parameters are passed appropriately from events to methods; our Common

Lisp implementation, in contrast, does not perform static type checks.

Applications and Experiences

In this section, we present typical and useful ways of using events and mediators, as they are applied in three systems we have constructed.

CAGD. Our Computer-Aided Geometric Design (CAGD) environment allows a user to view and edit a mesh[2] from multiple view points and to apply user-defined interpolation and approximation schemes to generate curves and surfaces from that mesh. An early version of the environment supported interactive exploration of curve generation schemes in a graduate CAGD course.

Figure 5 shows a subsystem for viewing and editing vertices (edges are omitted for simplicity). Vertices of the mesh component, M, store three-dimensional points, labels, and other information. The component 3D stores a set of three-dimensional points. In this context 3D is used to represent a projection of M restricted to its three-dimensional points. Similarly, the component 2D stores a set of two-dimensional points, and, in this context, represents a projection of 3D. The component DO stores a set of display objects that correspond to and know how to display two-dimensional points on D, an Interviews-based display [Linton, Vlissides & Calder 89]. In this context, the elements of DO are associated with those of 2D.

Mediators maintain these relationships. The mediator M3D, for example, is activated when a vertex is added to M and the event VertexAdded(v) is announced. M3D then extracts the three-dimensional point associated with v and adds it to 3D. This, in turn, activates 3D2D, which computes a corresponding two-dimensional point and adds it to 2D. This activates 2DDO, which adds a new display object to DO. Finally, this activates DOD, which depicts DO on D. If multiple views are present, all are updated in the same manner.

[2] A mesh is a discretized surface from which smooth interpolants and approximations can be computed. For our purposes it is easier to think of a mesh as a graph, in which vertices are geometric points representing a discrete sample of a surface and the edges represent topological adjacencies among the points. The graph abstraction described earlier is a simplified version of the mesh component that is central to our CAGD environment.

Changes also propagate in the reverse direction. When a new display object is added to DO, both DOD and 2DDO fire. DOD updates D, while 2DDO updates 2D. This latter change propagates to 3D and then to M.

This example exhibits several typical applications of events and mediators.

Mediators support asymmetric relationship maintenance. As an example, when DO changes, DOD updates D, but if D is changed directly, the relationship will be broken. We chose this structure because D is based on an existing user interface package that does not announce events. Instead of modifying the user interface, we decided to make all changes to DO and to have D updated accordingly.

Mediators also support symmetric relationship maintenance, which allows components to operate on the most natural representations. For example, 2DDO symmetrically manages both the 2D to DO relationship and the DO to 2D relationship. This allows the mouse-input handler to operate on DO, while another component, such as the mediator 3D2D, can operate on 2D. In either case, the relationship is maintained automatically. This approach contrasts with the Smalltalk-80 MVC, which forces all modifications to be made to a single model component, such as 2D. Our approach allowed the user interface (D and DO) and the central application (2D, 3D, and M) to be independently developed and tested before integration.

In managing symmetric relationships, mediators must be careful to avoid unbounded recursions. 2DDO, for example, updates 2D when DO is updated, and vice versa. To avoid cycles, 2DDO maintains a bit that it tests and sets before performing an update. If the bit is already set, the update operation returns immediately, breaking the recursion. When there are several active graphical views of a mesh, this approach still works: when the DO component of one view is changed, the update propagates through the chain of mediators back to M, and then to all other views except the one that made the change (because its M3D, mediator, which employs the same test and set protocol, is already locked).

Mediators are often parameterized. 3D2D, for example, has viewing parameters that determine how points from 3D are projected into 2D. In addition, mediators often encapsulate policies for resolving underspecified mappings. 3D2D, for example, encapsulates the policy for updating 3D when new points are added to 2D. One policy might be to set the underconstrained coordinate to zero; another might be to

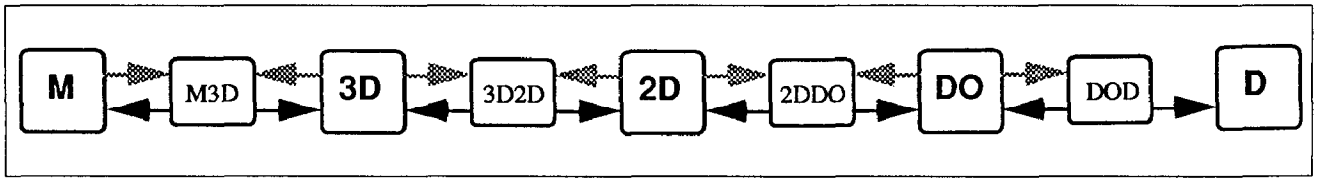

Figure 5: The organization of a CAGD 3-D view.

query the user for this coordinate. In any case, mediators encapsulate such policies, enabling them to be changed without modifying the components governed by such policies.

Mediators often use explicit relations to track associations between parts of the components they relate. For example, DOD relates two sets and also keeps a relation to record associations between the elements of these sets. These mediators insert tuples when associations are created and delete them when they are broken. When an object is added to DO, for example, DOD adds a corresponding element to 2D, and then adds a tuple to its relation. When an element is deleted from DO, the mediator queries the relation to find the element to delete, deletes it from 2D, and then deletes the tuple from the relation.

In addition, a mediator must often ensure that when one part of the components it relates is changed that all associated parts are updated accordingly. For example, if the value of a point in 2D is changed (which we do not define as an operation on 2D itself) the corresponding element in 3D should be updated. At least two approaches are possible. The 3D2D mediator can handle events announced by the elements and update corresponding elements directly, or 3D2D can delegate element-wise relationships to other mediators. We use this approach—similar to that taken by Chiron to keep graphical depictions of complex objects consistent [Young, Taylor & Troup 88]—at all levels of our environment.

A mediator can implement delegation by creating and retracting element-wise mediators as associations are created and broken. A mediator could also indirectly support delegation by announcing events as associations change. In this case, another independent mediator would receive these events and create and retract element-wise mediators in response. A slightly different approach would use a similar, independent mediator that responds to events announced by the relation (component), rather than by the original mediator itself.

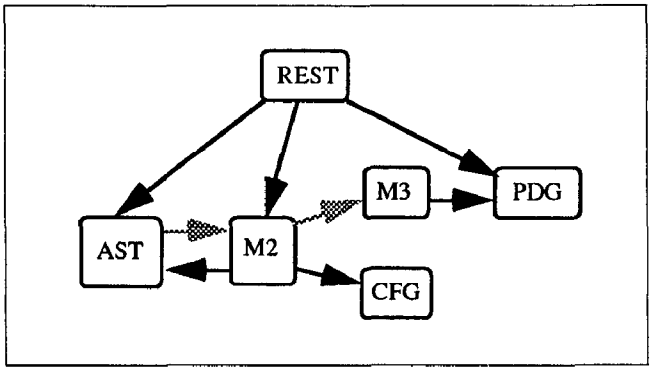

Figure 6: The organization of a program restructuring environment.

Program Restructuring. Griswold has also applied our approach in the design of his program restructuring environment [Griswold & Notkin 90]. This environment aids software engineers in restructuring a system by automatically balancing local changes with global changes to retain a system's semantics. The requirements and constraints in this domain lead to slightly different uses of mediators and events.

As shown in Figure 6, the system has four main components and two mediator components. AST is an abstract syntax tree representation of the program to be restructured, CFG is a control flow graph representation of AST, and PDG is a program dependence graph based on CFG. The mediators maintain relationships among these components in a lazy manner.

The design of this system is constrained in two ways. First, the PDG, taken from an existing system, already knows how to update itself when the CFG changes, so this function cannot easily be localized in a mediator. Second, it is expensive to rebuild the PDG. Given these constraints, the mediators in this system are responsible for *when* updates are applied as opposed to *how* updates are applied (as in the CAGD system).

The main restructuring component, REST, uses the PDG and AST to plan restructuring operations, which

are executed by making sequences of changes to the AST. M2 buffers the resulting AST change events, flushing them to the CFG only when its flush method is explicitly invoked by REST. This avoids reconstruction of the CFG on each change to the AST. M2 also announces events when either the AST or the CFG changes. M3 monitors M2's events and invalidates appropriate parts of the PDG in response. When REST needs to access a part of the PDG that has been invalidated, it first flushes any changes buffered by M2 and then rebuilds the PDG. In this design, REST manages the consistency of PDG explicitly. An alternative—which would require modification of the PDG—would be for the PDG to announce events when it is accessed, which could be used to support demand-driven semantics.

This design emphasizes the importance of having mediators be first-class components. They have state to do buffering, they actively maintain auxiliary components to support queries, and they provide explicit interfaces that the system uses to control their activities. Mediators and events, in this case provided by CLOS wrapper methods, allowed over 20,000 lines of existing Common Lisp code, comprising the CFG and PDG components, to be integrated almost without change.

Parallel Programming. We also applied our approach to a prototype parallel programming environment that partitions programs into three almost, but not quite, disjoint levels. The names and parameters of objects at lower levels are reflected at higher levels, for instance. We developed the three levels, and the tools that operate on them, independently by replicating data common to several levels in several representations. When the tools and their representations were fully functional, we integrated them by adding simple mediators to keep replicated representations consistent with one another. When a name is changed by a tool at one level, for instance, the corresponding parts of the representations at other levels are updated accordingly. This merging of little data bases into larger ones, with consistency among the parts maintained by events and mediators, is a dynamic counterpart to Garlan's static merging scheme for providing views to tools [Garlan 87].

Analysis of Related Work

Our research has helped us understand and evaluate a diverse range of systems. Since many of these systems have contributed to the use of events and separate relationships—albeit with different objectives and mechanisms—we have organized this section by system. Our work applies narrowly to integration and independence, thus we omit discussion of many other issues, such as persistence, that these systems address. We also omit mention of many important related efforts. Those we do discuss seem to cover the key issues, and to be representative of systems we have omitted.

Unix. Viewed as components, Unix programs are independent yet easy to integrate using pipes. Pipes are essentially mediators that relate Unix programs. (Alternately, programs can be viewed as mediators that relate streams.) Although this separation leads to a powerful and successful environment, there are several limitations that restrict full-fledged integration. First, the interfaces presented by Unix program components are limited to the system-defined read-stream and write-stream methods. Second, the only event is one that signals output-ready on a stream. Third, the output-ready event can be sent to at most one pipe. These restrictions combine to enforce a unidirectional flow of unstructured information, which makes it hard to build integrated, interactive environments.

APPL/A. APPL/A is a programming system that supports the specification of relationships among objects using separate, active relations. APPL/A relations announce events upon acceptance and completion of the insert, delete, update, and find operations. These events are forwarded to triggers, general-purpose constructs that update other components including relations.

APPL/A's event mechanism is limited—only relations announce events, and they only announce predefined events—restricting the integration of general components. For instance, triggers, which act as mediators, can only be associated with relations.

ThingLab. ThingLab is a constraint-based programming system providing constraints, objects that (like mediators) maintain relationships among other independent objects. ThingLab components provide methods to get and set their values, and an-

nounce a predefined change event when their values are changed. A ThingLab constraint provides a method to determine when the relationship it defines is violated. It also provides one or more methods any one of which is capable of computing and assigning new values to resatisfy the constraint. The basic restriction in ThingLab is that constraints are functional: they operate solely in terms object values and are prohibited from producing side-effects.

Because of this limitation, using constraints to relate large values is inherently inefficient. For example, maintaining a one-to-one correspondence between two sets takes time linear in the sizes of the sets whenever an element is inserted or deleted. A simple mediator, in contrast, can keep such a one-to-one correspondence in constant time if the sets announce events about individual insertions and deletions.

Our approach is lightweight compared to ThingLab. A small constraint, for example, requires hundreds of bytes of storage; a small mediator takes tens. ThingLab also requires significant infrastructure, such as a constraint-graph representation, an incremental satisfier, and a constraint compiler for constraints that are specified declaratively.

ThingLab is richer than our lightweight method in several dimensions. Constraints, for example, provide a set of equally valid methods to reestablish consistency. The ThingLab *EM* lookup operation therefore returns not a set of methods, but a set of sets of methods (one method-set per constraint). One method is chosen from each set for execution. This introduces non-determinism not present in our approach.

LOOPS. LOOPS active variables support an event mechanism in which only variables announce events, and the only events they announce are `read()` and `written()`. Though such mechanisms provide convenient support for some activities, such as debugging, they are too low-level to serve as a basis for designing integrated environments.

AP5. AP5 is an active, in-core, relational data base extension to Common Lisp. AP5 users can register triggers with the data base. A trigger consists of a condition, written in first-order logic with temporal extensions, and a body, written in Lisp. Triggers can guarantee data base integrity by modifying or rejecting data base transactions, and can invoke non-data base activities in response to transactions. An AP5 trigger condition defines a data base event to be announced, while a trigger body represents the code executed when an event is announced. AP5 events are announced after transactions are submitted but before they commit, allowing transactions to be modified or aborted.

In contrast with our approach, the only component in AP5 that announces events is the data base. Integration of arbitrary objects, then, requires that they be stored in and modified through the relational data base. On the other hand, trigger bodies, like mediators, are general pieces of code, which permits flexible responses to events.

AP5 provides a rich integration mechanism. However, its richness is balanced by the cost of implementing the required trigger compiler, transaction manager, query optimizer, etc. In comparison, our basic approach requires minimal infrastructure.

MVC. The Smalltalk-80 MVC is a general event mechanism. It permits any component to announce events, and any event to be announced (by parameterization of its `changed(<object-array>)` event).

Though not inherent in MVC, the MVC style does not encourage the use of separate components to represent relationships. Rather, update code is merged into the components to be related (as in our third design above). Also, MVC is usually used asymmetrically. Specifically, models update views using events, but views update models explicitly. In addition, MVC events are not declared and their parameters have to be decoded by receivers, obscuring both the events and also the behavior of the components that use them.

FIELD and Forest. FIELD integrates Unix tools that have been retrofitted with events and methods. FIELD overcomes the problems with pipes by allowing Unix tools to expose internal data and to be manipulated through a method interface. Like MVC, the FIELD model encourages designs analogous to that in the third design, where intercomponent relationships are merged into components.

Garlan and Ilias's Forest extensions to FIELD overcome this problem to some extent by associating "policy programs", rather than methods, with the event patterns registered by tools [Garlan & Ilias 90]. When the event manager receives a messsage, the associated policy programs are executed (instead of the tool itself). The policy programs can use global state to decide whether to invoke tools, to send further messages to the event server, or to take other actions.

This approach increases the separation of relationship and component concerns, by localizing policy programs. In contrast to mediators, however, policy programs are not first-class components: they cannot register patterns, the policy definition language is limited, and they do not have their own state. Also in contrast to our approach, this design merges policy-program evaluation with the event delivery mechanism.

FIELD's event model has two powerful features. First, event registration is done implicitly, in terms of patterns and methods, rather than explicitly, in terms of event-method pairs. The actual relation—the set of methods activated in response to an event—is computed by pattern-matching when events are announced. This increases independence by making it possible to register for events without knowing who might send them. Second, in contrast to the distributed representation that we share with MVC, the FIELD event-method relation is centralized in the event manager, making it easier to operate on the relation as a whole. Although FIELD does not take significant advantage of this, Forest does.

Compatibility Maps. Compatibility maps are an approach to defining multiple views for tools in an environment [Garlan 87, Habermann et al. 88]. A collection of types is statically merged to define a relationship that is kept consistent when tools operate on the original types. While merging leaves the underlying types unchanged, bundling a type into one relationship prevents it from being simultaneously bundled into other relationships. This reduces the flexibility with which types can be integrated with other types. On the other hand, static merging allows methods and representations in the merge type to be optimized, which is hard or impossible in approaches like ours.

Conclusion

One way to manage the complexity of integrated environments is to restrict the semantics of programming constructs to automatically ensure that basic properties, such as termination, hold in a given environment. In practice, however, these restrictions—such as those defined by attribute grammars—do not necessarily ease the development and evolution of environments. One reason is that the constructs may not be appropriate for describing a given design. Another reason is that the restricted semantics may significantly decrease the efficiency of the resulting environment.

Overcoming these problems may require the use of semantically unrestricted programming constructs. Undisciplined use of such constructs, however, makes it impossible to reason about the resulting environment. Our approach is to take advantage of the familiarity and power of such constructs and to control complexity by providing design techniques that encourage environment designs that are robust with respect to changing requirements.

To ease the evolution of integrated environments, it is important to localize relationships and to keep components independent of relationships in which they participate. Mediators localize relationships, while our event mechanism permits components to invoke mediators without being statically bound to them. Our experience with several environments indicates that our combination of events and mediators achieves a level of independence not easily reached using previous design approaches. This increased level of independence in turn gave us the leverage we needed to build and evolve relatively complex environments.

To make our approach practical in even more domains we need to solve several lower-level technical problems. For instance, we need to design an asynchronous event mechanism for multi-threaded language systems. We also need to handle multiple address spaces and distribution. We anticipate that the lightweight character of our approach will allow us to more easily explore these and related problems.

Of course, open questions remain. One, some systems use atomic transactions to keep global consistency. We have not yet decided how to balance the benefits of independence in our lightweight approach with the need to guarantee global consistency. Two, controlling concurrency in systems designed as independent components related by mediators is difficult. Hiding components, as in the first design of the simple graph environment, simplifies concurrency control. Granting independent access to components, as we suggest, complicates matters substantially. Three, Meyer suggests that communication between components should be reflected in the source code [Meyer 88]. Events are designed precisely to avoid this. Whether systems that are designed using events are sufficiently understandable remains to be seen.

Acknowledgments Alex Klaiber, Bill Griswold, Calvin Lin, and Bob Mitchell contributed to the initial design effort. Tony DeRose, Bill Griswold, and Larry Snyder provided the domains for the environments we discussed here. Alex Klaiber, Sitaram Raju, and Tom McCabe constructed many of the tools for these environments. In addition to many of these people, Gail Harrison and Michael Hanson gave us useful comments on drafts of this paper.

References

[Bobrow et al. 88] D.G. Bobrow et al. Common Lisp Object System Specification X3JI3 Document 88-002R. *ACM SIGPLAN Notices 23.* (September 1988).

[Borning 81] A. Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems 3,*4 (October 1981), pp. 353–87.

[Cohen 89] D. Cohen. Compiling Complex Transition Database Triggers. *Proceedings of the 1989 ACM SIGMOD,* Portland, OR (1989), pp. 225–34.

[Garlan 87] D. Garlan. *Views for Tools in Integrated Environments.* Ph.D. Thesis, Carnegie-Mellon University (1987).

[Garlan & Ilias 90] D. Garlan and E. Ilias. Low-cost, Adaptable Tool Integration Policies for Integrated Environments. *Proceedings of SIGSOFT90: Fourth Symposium on Software Development Environments.* Irvine CA (1990).

[Griswold & Notkin 90] W.G. Griswold and D. Notkin. Program Restructuring to Aid Software Maintenance. University of Washington, Department of Computer Science and Engineering Technical Report 90-08-05 (September 1990).

[Habermann & Notkin 86] A. N. Habermann and D. Notkin. Gandalf Software Development Environments. *IEEE Transactions on Software Engineering SE-12,*12 (December 1986), pp. 1117–1127.

[Habermann et al. 88] A.N. Habermann, C. Krueger, B. Pierce, B. Staudt, and J. Wenn. Programming with Views. Technical Report CMU-CS-87-177, Carnegie-Mellon University (January 1988).

[Sutton, Heimbigner & Osterweil 89] S. Sutton, D. Heimbigner, and L. Osterweil. APPL/A: A Prototype Language for Software Process Programming. University of Colorado Technical Report CU-CS-448-89, University of Colorado, Boulder (1989).

[Krasner & Pope 88] G.E. Krasner and S.T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming 1,*3 (August/September 1988), pp. 26–49.

[Linton, Vlissides & Calder 89] M.A. Linton, J.M Vlissides, and P.R. Calder. Composing User Interfaces with InterViews. *Computer 22,*2 (February 1989), pp. 8–22.

[Meyer 88] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Cambridge (UK), 1988.

[Parnas 72] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM 5,*12 (December 1972), pp. 1053–58.

[Reiss 90] S. P. Reiss. Connecting Tools using Message Passing in the Field Environment. *IEEE Software 7,*4 (July 1990), pp. 57–66.

[Stefik, Bobrow & Kahn 86] M.J. Stefik, D.G. Bobrow, and K.M. Kahn. Integrating Access-Oriented Programming into a Multiparadigm Environment. *IEEE Software* (January 1986), pp. 10–18.

[Young, Taylor & Troup 88] M. Young, R.N. Taylor, and D.B. Troup. Software Environment Architectures and User Interface Facilities. *IEEE Transactions on Software Engineering 14* 6 (June 1988), pp. 697–708.