Reconciling Environment Integration and Software Evolution

KEVIN J. SULLIVAN and DAVID NOTKIN
University of Washington, Seattle

Common software design approaches complicate both tool integration and software evolution when applied in the development of integrated environments. We illustrate this by tracing the evolution of three different designs for a simple integrated environment as representative changes are made to the requirements. We present an approach that eases integration and evolution by preserving tool independence in the face of integration. We design tool integration relationships as separate components called *mediators*, and we design tools to implicitly invoke mediators that integrate them. Mediators separate tools from each other, while implicit invocation allows tools to remain independent of mediators. To enable the use of our approach on a range of platforms, we provide a formalized model and requirements for implicit invocation mechanisms. We apply this model both to analyze existing mechanisms and in the design of a mechanism for C++.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution and Maintenance—*enhancement, extensibility;* D.2.10 [**Software Engineering**]: Design—*methodologies;* K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software development, software maintenance*

General Terms: Design

Additional Key Words and Phrases: Abstract behavior type, behavior abstraction, component independence, environment integration, event mechanism, implicit invocation, integrated environment, mediator, mediator/event design, software evolution, tool integration

1. INTRODUCTION

An integrated environment is a collection of software tools that work together, freeing the user from having to coordinate them manually. An integrated programming environment with tools for text editing, compiling, and debugging, for example, might ensure that when the debugger reaches a breakpoint the editor scrolls to the corresponding source statement. Integrating these

This research was funded in part by NSF grants CCR-8858804 and CCR-9113367, AFOSR grant 89-0282, Digital Equipment Corp., Xerox Corp., Tokyo Institute of Technology, and Osaka University. K. Sullivan was funded in part by a GTE fellowship. An earlier version of this paper appeared in the Proceedings of SIGSOFT90.
Authors' address: Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195

tools relieves the user from the tedium of locating source files and line numbers, and supports instead the primary task, finding the bug.

Integrated environments support tasks in many domains. These include word processing systems integrating editors, outlining tools, and style and spelling checkers; financial systems integrating spreadsheets, portfolio databases, and tax preparation tools; programming environments integrating compilers, editors, debuggers; and software development environments integrating tools for many phases of the software life-cycle, from planning and analysis through retirement.

Managing the complexity of integrated environments is hard. Many of the difficulties are similar to those encountered in any medium- to large-scale software system: requirements must be satisfied; implementation decisions should not be unduly constrained; components should be reused when possible; requirements evolve. But environment integration produces an additional, complicating tension. Software evolution, which is unavoidable in practice, is easiest when system components are largely independent; but integration requires fine-grained coordination of tools, and hence of system components.

Integrated tools are often designed by merging responsibility for the integration relationships into the tools themselves. Knowledge of the relationship between an editor and a debugger, for instance, might be designed into both the editor and the debugger components. This can lead to unmanageably complex tools. In the worst case, each tool interacts with every other to manage complex integration relationships in addition to its own basic functions. This approach also distributes decisions regarding integration throughout the environment: tool A has to know how it relates to B and also B to A, for instance. Furthermore, intermingled tools are statically sculpted to participate in specific relationships. The resulting complexity of individual tools, the connections among tools, and the distribution of integration concerns all complicate software evolution.

Another approach is to design integration relationships as separate components that encapsulate tools in order to effect their integration. For example, an editor and a debugger might be wrapped in a third component that interacts with the user and manages the underlying tools on the user's behalf. Although the tools themselves remain independent, encapsulated tools cannot be accessed independently. This design technique complicates evolution by making it hard to change, add, and remove tools and integration relationships. We illustrate these ideas more carefully in Section 2, through three designs for a simple specification.

These design styles arise in part because system *specifications* often merge tool and integration concerns or frame integration requirements in terms of explicit tool invocations. To ease evolution, then, we need to structure both specifications and designs to reduce the complexities that result from integration. Our goals should be to keep tool complexity proportional to function, to localize decisions regarding both tools and integration relationships, and to do this while also facilitating tool integration.

We have developed an approach that appears to reconcile environment integration and software evolution to a reasonable degree. The key to our approach is to define tool integration relationships separately from the tools themselves and to connect the tools to these integration components through an implicit invocation mechanism.

Applying this approach yields a system composed of a set of independent and visible tool components plus a set of separate, or *externalized*, integration components, which we call *mediators*. At its heart, we stress the importance of keeping tool components visible in contrast to encapsulated and independent in contrast to intertwined. New integration relationships are easier to design when tool components are visible, while tool independence eases both evolution and integration.[1]

In Section 2 we present an example that illustrates the sorts of problems encountered when traditional designs are confronted with the combination of integration and evolution. In Section 3, in the context of this example, we present our approach and show how it eases the tension between these conflicting interests. We also discuss our use of mediators and implicit invocation more fully. In particular, we present a simple, formal model that can serve as a starting point in the design of implicit invocation mechanisms. We discuss additional requirements that implicit invocation mechanisms should meet to support our approach. We also sketch the design of our mechanism for C++.

In Sections 4 and 5 we summarize our experiences with this approach. Section 4 discusses how our approach can be applied to meet requirements that integrated environments tend to generate. In Section 5 we briefly present several systems that we have built using our approach. Sections 6 and 7 are devoted to a discussion of related work. In Section 6 we apply our implicit invocation model to analyze several existing mechanisms to gain a better understanding of their key similarities and differences. In Section 7 we discuss other systems that separate integration relationships from the components they relate. We conclude, in Section 8, with a brief summary and a discussion of the strengths and limitations of our effort.

2. TRADITIONAL DESIGN APPROACHES

Traditional design approaches tend to produce integrated environments that are hard to change as requirements evolve. We illustrate this by applying three such approaches to requirements for a simple integrated environment. We then challenge each design with each of two enhancements. The basic specification and the enhancements are intended to capture key aspects of integration and evolution in a simplified framework. All three designs complicate evolution, either by encapsulating components or by merging tool and integration concerns.

[1] Independence is also closely related to tool reusability, but we do not stress this connection in this article.

2.1 Initial Specification

The two "tools" in our environment manage a set of points and a set of pairs of these points. The user can create points and, given two points, can create a pair. The user can also insert and delete points and point-pairs into and from the point and point-pair sets respectively. In addition, we specify an "integration relationship" between the point and point-pair sets: the two sets should form a graph. We thus will be justified in referring to points and point-pairs at vertices and edges, respectively.

The vertex (i.e., point) type stores a pair of coordinates and exports methods for vertex creation, for getting and setting the coordinates and for testing equality with another vertex. The edge (i.e., point-pair) type stores references to the two vertices and exports methods for edge creation, for getting these vertices and for testing equality with another edge. The set types export methods for set creation, element insertion, deletion, and membership testing, and for getting an iterator object that can return each element in turn.

To the vertex-set and edge-set tools (which we denote **VS** and **ES**) we add a third, a user interface, through which the user can create, insert, and remove vertices and edges. This structure makes it possible to violate the requirement of the integration relationship—that the sets form a graph—by inserting an edge into **ES** for which neither or only one of the vertices of the edge is in **VS** or by deleting a vertex from **VS** upon which one or more edges in **ES** is incident.

To maintain the required constraint we integrate the tools under an integration relationship **G** that ensures that the invariant

$$\mathbf{G}(\mathbf{VS}, \mathbf{ES}) \equiv \mathbf{ES} \subseteq \mathbf{VS} \times \mathbf{VS}$$

is maintained. Many different policies could be defined to maintain this invariant—one that undoes the deletion from **VS** of any vertex on which edges in **ES** are incident, for example. In this case we specify a policy that preserves the user's last action: when a vertex is deleted from **VS**, delete from **ES** all edges incident on that vertex; when an edge is inserted in **ES**, check the membership in **VS** of the edge's vertices and insert into **VS** each vertex that is not already a member.

We often specify a given integration relationship as a combination of an invariant and a policy to maintain this invariant in the face of changes. Often, these policies have to satisfy additional requirements, such as preserving rather than undoing the user's last change.

2.2 Two Representative Enhancements

We now consider two enhancements to the specification, which we discuss with respect to each forthcoming design in order to assess the robustness of these designs with respect to evolution. The first enhancement is a change to an integration relationship. We enhance **G** to support lazy, in addition to eager, maintenance of the graph relationship. The new invariant is

$$\mathbf{G}'(\mathbf{VS}, \mathbf{ES}) \equiv (\text{eager} \wedge \mathbf{G}) \vee \text{lazy}.$$

Our new policy for **G′** is to apply the policy for **G** when in eager mode, to make no compensating updates when in lazy mode, and to reestablish the invariant when toggling from lazy to eager mode by inserting into **VS** any vertex not already a member upon which any edge in **ES** is incident.

The second enhancement we consider is a change to the set of tools and integration relationships in the environment. We illustrate by integrating a new tool, which involves adding both a tool and an integration relationship. The new tool **N** stores a nonnegative integer and exports methods for incrementing, decrementing, and getting the value of this number. The new relationship is that **N** should reflect the cardinality of **VS**. We thus define an integration relationship **C** with invariant

$$\mathbf{C}(\mathbf{N}, \mathbf{VS}) \equiv \mathbf{N} = |\mathbf{VS}|.$$

Our policy for **C**, which, again, retains the effects of user changes, is to increment or decrement **N** whenever a vertex is inserted into or deleted from **VS**, and to insert a vertex into or delete one from **VS** whenever **N** is changed.

2.3 Three Traditional Designs

We now apply three traditional design approaches to the basic requirements. The resulting designs are intended to reflect characteristics that arise in practice when these design approaches are applied. All three designs include components for the vertex and edge types that are defined identically, so we do not discuss them any further. The design of the user interface differs in each case, but is always defined directly in terms of the visible components; we also omit further discussion of this component.

The components realizing the tools **VS**, **ES**, and **N** and the integration relationships **G**, **G′**, and **C**, on the other hand, vary considerably based on the design approach applied. These differences are at the heart of the matter, so we focus on these components.

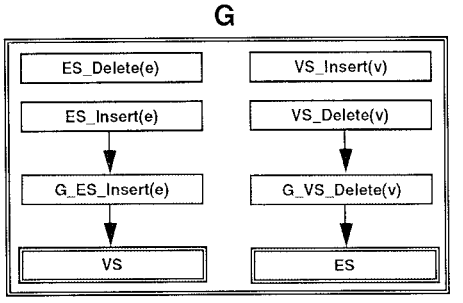
2.3.1 *Encapsulation.* The first design, shown in Figure 1,[2] defines three components, *G*, *VS*, and *ES*. The components *VS* and *ES* realize the tools **VS** and **ES**, while *G* realizes the integration relationship **G** through encapsulation of *VS* and *ES*.[3] The methods exported by *G* call *VS* and *ES* to insert and delete vertices and edges on the user's behalf. Since *G* encapsulates *VS* and *ES*, only *G*'s methods are visible to the user interface.

When one of these methods disrupts **G**, by deleting a vertex on which edges in *ES* are incident, for example, the method reestablishes the relationship by

[2] In the figures discussed in this and the following section, components are represented by double-framed boxes and the methods they export by single-framed boxes. Shaded boxes denote hidden components or methods not accessible to the user interface, while unshaded boxes denote visible components or methods. Black arrows denote explicit, and grey arrows, implicit method invocation. The tail of an arrow designates the method making the invocation, and the head, the method that is invoked. If the head points to an entire component, one or more methods in that component are invoked. For clarity, only invocations made to maintain integration relationships are represented, in contrast to those implementing basic tool functions.

[3] We use a boldface font to denote specification components such as **VS**, and an italic font for corresponding design components such as *VS*.

Figure 1



invoking a separate update method defined by G. After $G.VS\_Delete(v)$ calls $VS.Delete(v)$ to delete v from VS, for example, it calls $G.G\_VS\_Delete(v)$[4] to ensure the consistency of **G**. This method iterates over the edges of ES deleting any incident on v. G encapsulates VS and ES to prevent other components from invalidating **G** by bypassing these necessary update methods.

This design is fairly clean. Separate aspects of the specification—the tools **VS** and **ES** and the relationship **G**—are represented separately; and within the design, VS and ES are independent of each other and of G: there are no invocations from VS to ES, from ES to VS, or from VS or ES to G. G does invoke VS and ES explicitly, but this is reasonable since G manages the relationship between them. However, VS and ES are encapsulated by G, and this, as we shall see below, complicates integration of new tools.

First, though, we note that designing **G** as a separate component G eases evolution by localizing the effects of changing **G** to **G′**, as shown in Figure 2. We add a mode-bit (LazyFlag), a method to switch modes, and we modify the update methods to maintain the new relationship **G′**. These changes are reasonable: they are implied by the change to the specification, and the overall change to the design is proportional to the size of the change to the specification. Notably, VS and ES do not have to change in order to change the relationship between them.

The second change, shown in Figure 3, is harder, however, because G encapsulates VS, which makes it hard to integrate N with VS. **N** can be designed as an independent component N, but then we have to migrate N into G, where VS is visible. Thus, we expand G's interface with public methods to change N and with private update methods to maintain **C** (as well as **G**). The new version of G really represents a composition of **G** and **C**, as the new name GC indicates.

[4] Our method names reflect the component defining the method and the function performed. $G.VS\_Delete(v)$ is defined in G and deletes a vertex from VS by calling $VS.Delete(v)$; $G.G\_VS\_Delete(v)$ maintains the relationship of **G** in the face of the deletion of a vertex from VS, for example. Update methods, such as $G.G\_VS\_Delete(v)$, are not strictly needed. This code could be placed directly into the methods exported from G. However, the separation makes the code and the explanation clearer by isolating the basic tool activities from the actions needed to preserve **G**.

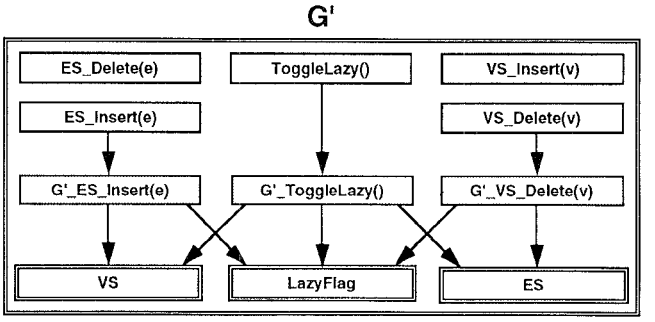


Figure 2

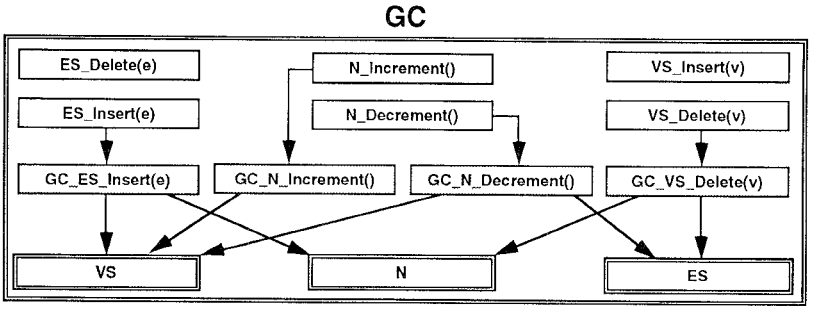


Figure 3

When faced with the integration of a new tool, the encapsulation approach thus forces a merging of separate concerns. Maintaining **C** should not require knowledge of either **ES** or of **G**, but this separation is not possible because G hides VS. When N is decremented the update method of

$$GC.GC\_N\_Decrement( )$$

must not only delete a vertex to restore **C**, but must also access ES to delete edges incident on the deleted vertex. The update methods $GC.GC\_ES\_Insert(e)$ and $GC.GC\_VS\_Delete(v)$ must be modified, too, even though **G** has not been changed at all.

While this design seemed promising at first, it degrades when a new tool is integrated. Integrating **N** substantially increases the complexity of G and damages the correspondence between specification and design by merging **G** and **C** in GC. These factors further complicate future evolution.

This approach is typical. Indeed, one can easily anticipate a programmer producing the initial design given the original specification. Unix tools exemplify this approach. They often have rich and well-modularized inner structures, but these are hidden from other tools. This decreases the potential for integrating Unix tools without merging their implementations or making other major changes to expose internal representations and activities.

2.3.2 *Hard-Wiring.* The second design, shown in Figure 4, defines two components, VS and ES, both of which export methods to the user interface.

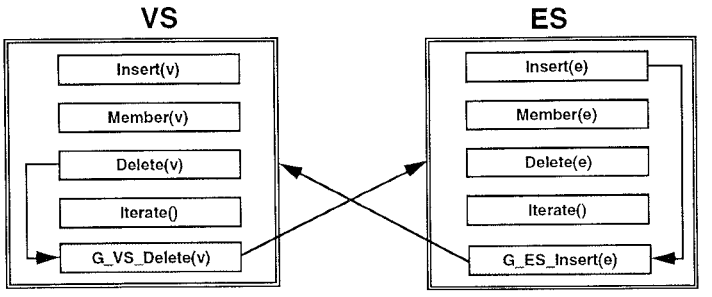


Figure 4

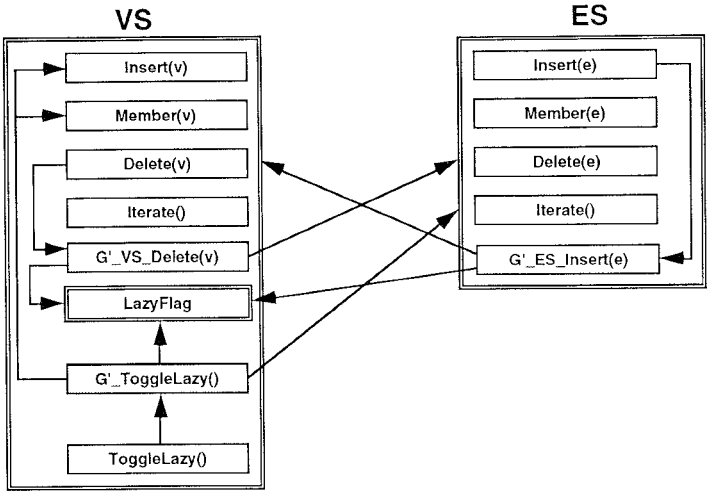


Figure 5

To maintain **G**, *VS* and *ES* invoke each other directly. For example, after *VS.Delete*(v) deletes v, it invokes *VS.G–VS–Delete*(v), which maintains **G** by making appropriate calls to the exported methods of *ES*. Likewise, when an edge is added, *ES.G–ES–Insert*(e) calls *VS* to maintain **G**.

In contrast with the previous design, this one exposes *VS* and *ES* in an attempt to ease integration of **N**. In doing so, however, it sacrifices the explicit representation of the relationship **G** in the design and creates explicit invocation dependencies between *VS* and *ES*. These structural problems complicate both enhancements.

As shown in Figure 5, adding a lazy mode to **G** is hard because **G** is not represented explicitly. Instead, both *VS* and *ES* have to be modified when the relationship between **VS** and **ES** is changed. This distribution of **G** also makes it hard to know where to place the lazy-mode bit and associated methods. In Figure 5, the change is made to *VS*, but the change to *ES* would be symmetric. Distributing the change between the components would be

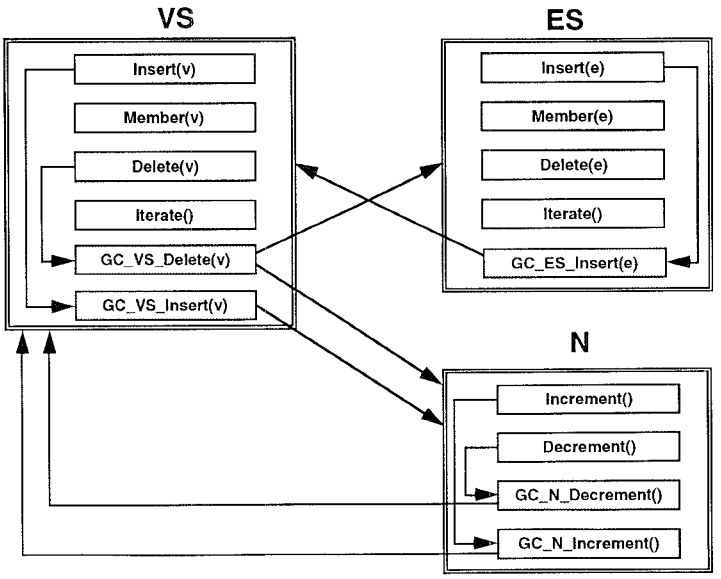


Figure 6

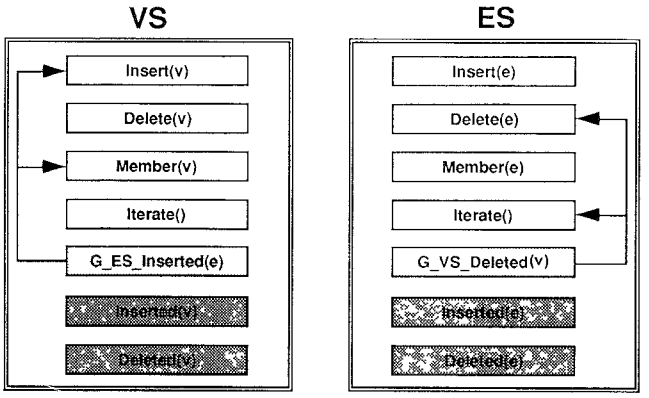
confusing. Creating another component may be the best solution, but *VS* and *ES* would still have to change and the interactions among the three components would be similar to those in our design.

Furthermore, as shown in Figure 6, this design does not ease the integration of **N**, even though *VS* is exposed. *VS* must still be modified to handle the relationship **C**. Moreover, the modified design merges aspects of **C** and **G** in *VS*, further decreasing the independence of *VS* and complicating the correspondence between specification and design.

Software designs based on this approach abound. It is hard to cite specific systems, but not hard to find advocates of this approach. Coad and Yourdon's object-oriented analysis method, for example, is based precisely on subsystem (*subject*) composition by explicit invocation [6]. Using this approach for an integrated environment would be inadvisable. It complicates evolution and produces structures that degrade rapidly as enhancements are made. It makes it hard to change and add relationships and also to replace or reuse tools, since each may be tightly intertwined with many others.

2.4 Events

The third design, a variation of the second, employs implicit invocation in place of the explicit invocations between *VS* and *ES*. Briefly, implicit invocation occurs when one component *announces* an *event* that it has defined to which other components have attached methods to be executed. The key point is that the component making an implicit invocation—the one that defines the event—need not know the names or even the number of methods invoked when it announces the event. We discuss this topic further in Sections 3 and 6.



Event	*Method*
ES.Inserted(e)	VS.G_ES_Inserted(e)
VS.Deleted(v)	ES.G_VS_Deleted(v)

Figure 7

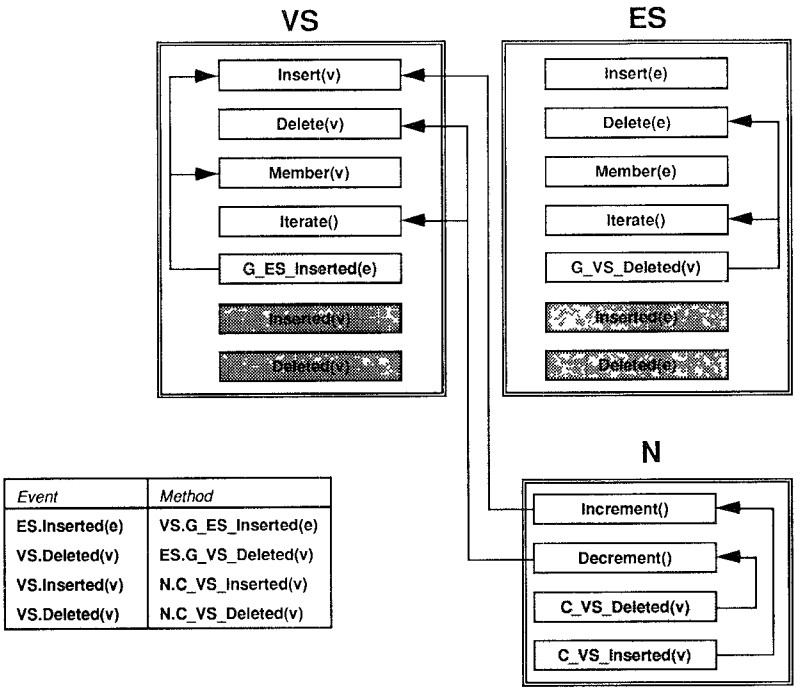
In this design, shown in Figure 7,[5] *VS* and *ES* are extended with events that they announce when insertions or deletions succeed. The update methods are similar to those in the previous design, but in this design each operates on the component that defines it rather than on the other component. To complete the design, we associate the *ES.G_VS_Deleted(v)* update method in *ES* with the *VS.Deleted(v)* event of *VS* and similarly the update method in *VS* with the edge addition event of *ES*.

This design has characteristics similar to those of the previous one. Changing **G** to **G**′ poses exactly the same problems. The representation of **G** is distributed through *VS* and *ES*, making it necessary to change both components to change the relationship between them. Implicit invocation has no fundamental affect on the difficulty of making this change.

This design does allow *N* to be integrated with *VS* without changing *VS*, as shown in Figure 8, but not without sacrificing the independence of *N*. To maintain **C** when *VS* changes, we associated update methods in *N* with events of *VS*. In contrast, when *N* changes, it invokes *VS* explicitly to add or delete a vertex. This yields a hybrid hard-wired and event-based design. *VS* remains independent, but *N* does not. An alternative that preserves the independence of *N* is to add update methods to *VS*, associated with events of *N*. But this requires changing *VS*.

Thus there is no completely satisfactory way to integrate a new tool. Furthermore, both alternatives merge tool and integration concerns, and the one that changes *VS* merges multiple relationships (**G** and **C**) in *VS*.

[5]Events are drawn as darkened boxes; the connection between events and methods to be implicitly invoked is shown in tabular form. Tracing the connections from events through the event-method association to methods is helpful.



Event	*Method*
ES.Inserted(e)	VS.G_ES_Inserted(e)
VS.Deleted(v)	ES.G_VS_Deleted(v)
VS.Inserted(v)	N.C_VS_Inserted(v)
VS.Deleted(v)	N.C_VS_Deleted(v)

Figure 8

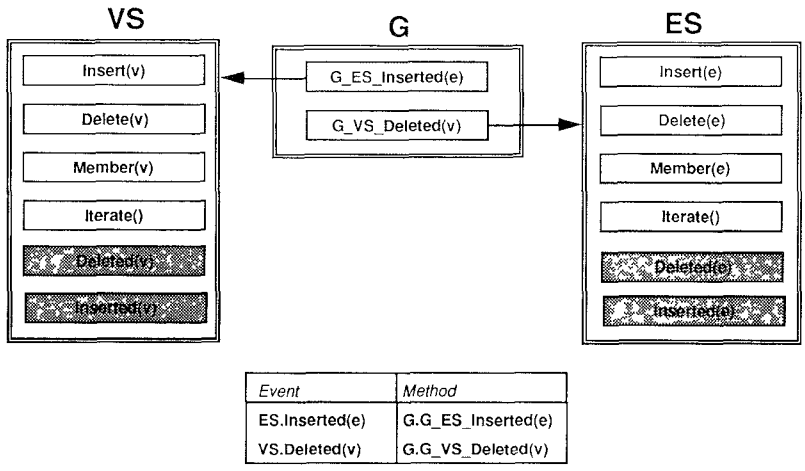
Many contemporary integration frameworks take this approach. These include FIELD [27], the Hewlett-Packard SoftBench [5], and the Smalltalk-80 Model-View-Controller (MVC) [21]. When implicitly invoked, tools update themselves to maintain the specified integration relationships. MVC, for example, takes the hybrid approach: models implicitly invoke views, but views explicitly invoke models.

3. THE MEDIATOR / EVENT APPROACH

In this section we present a design approach that appears to ease the evolution of integrated environments. We illustrate it with a design for the graph environment. Our approach is based on mediators, separate components designed to integrate independent tools, and implicit invocation. By *separate* we mean a component defined as a unit, such an abstract data type, an object, or a Unix program. By *independent* we mean a component that does not explicitly invoke any component other than itself. Following our design, we describe our approach in greater detail and discuss the role played by implicit invocation.

3.1 An Improved Design

As in the encapsulation-based design, we define three components, *VS*, *ES*, and *G*, as shown in Figure 9. *VS* and *ES* realize the tools and provide



Event	*Method*
ES.Inserted(e)	G.G_ES_Inserted(e)
VS.Deleted(v)	G.G_VS_Deleted(v)

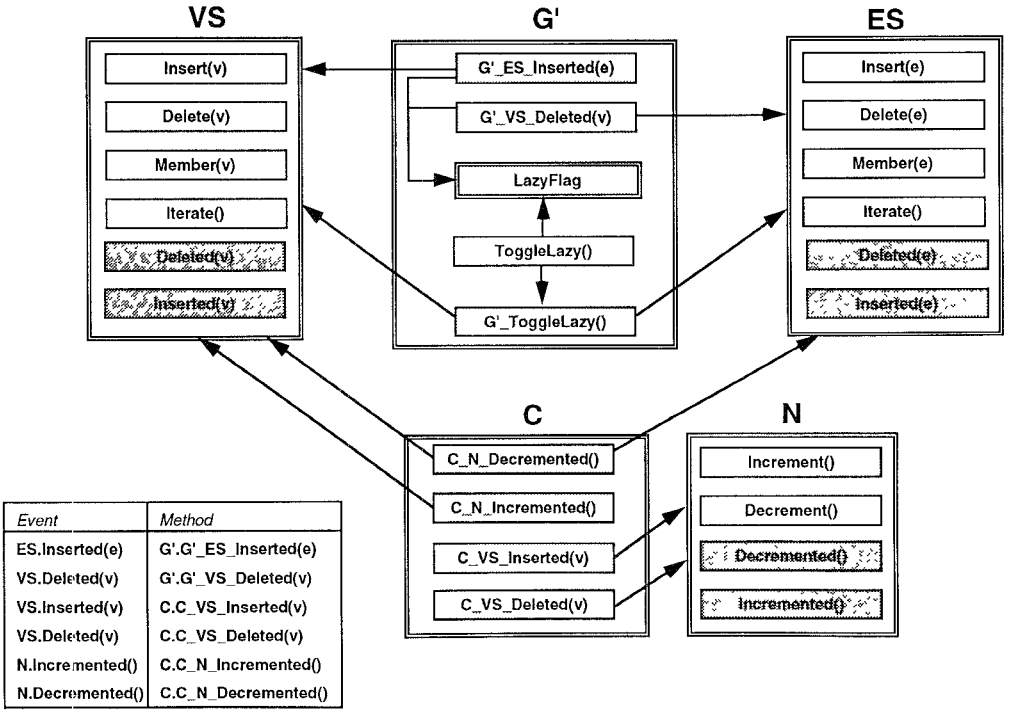
Figure 9

the methods called by the user interface; G realizes the relationship **G** and provides the update methods used to maintain the integration relationship. In contrast with the earlier design, this G does not encapsulate VS or ES. Instead, we use implicit invocation, as in the third design, to guarantee that update methods are called as necessary. Specifically, we associate the update methods of G with the insertion and deletion events of VS and ES. G in turn explicitly invokes VS and ES to reestablish **G**. This replaces the encapsulation by G of VS and ES in the first design with implicit invocation of G by VS and ES in this design.

In contrast to earlier designs, ours eases both the change in **G** and integration of **N**. Changing **G** is easy because the relationship is localized in a separate component, as in the first design. In addition, designing **N** as a component N that announces appropriate events allows us to design **C** as a mediator C that is implicitly invoked by N and VS. When a vertex is deleted from VS, for example, the event announcement invokes both G and C, which then update ES and N to restore **G** and **C**, respectively.

Figure 10 depicts our design with both enhancements simultaneously, which would have been hard under the previous approaches. In this design, VS, ES, and N are independent, thus easier to understand, reuse, and compose. More importantly, by simply adding or removing mediators the environment can be changed to provide any of the following subsets of integration relationships: { }, {**G**}, {**G**′}, {**C**}, {**G**, **C**}, and {**G**′, **C**}. Finally, the correspondence between specification and design is preserved, easing future enhancement.

In review, the first design cast **G** as a separate component and **V** and **E** as independent components. This eased evolution of **G**, but encapsulating VS and ES made it hard to integrate **N**. The second and third designs exposed VS and ES to ease integration of **N**, but intertwined these components to maintain **G**. This made it hard both to change **G** and to integrate **N**. The



Event	*Method*
ES.Inserted(e)	G'.G'_ES_Inserted(e)
VS.Deleted(v)	G'.G'_VS_Deleted(v)
VS.Inserted(v)	C.C_VS_Inserted(v)
VS.Deleted(v)	C.C_VS_Deleted(v)
N.Incremented()	C.C_N_Incremented()
N.Decremented()	C.C_N_Decremented()

Figure 10

third design used implicit invocation to preserve the independence of *VS* and *ES*. It also allowed **N** to be integrated without changing *VS*, but in other regards it had the same problems as the second because it failed to separate tool and integration concerns.

Our design combines positive aspects from each of these. We design tool integration relationships as separate components, and we keep *VS* and *ES* both independent and visible, which eases both evolution and integration.

3.2 Why Mediators and Implicit Invocation

A misunderstanding we sometimes encounter when presenting this approach is the notion that mediators represent a new programming *mechanism*. In truth, they only represent a new usage of existing mechanisms. The mediators *G* and *C*, for example, could be implemented as objects or Unix tools.

This has several advantages. Since mediators are first-class design and implementation components, they can maintain state and invoke tools as necessary to maintain complex relationships. When implemented as objects, they also respect tool encapsulation boundaries. They can export abstract interfaces, announce their own events, and be composed in arbitrary ways. For example, they can be used within designs of larger components, and can be treated as tools to be integrated with other tools. Finally, they can be used

in existing languages and on existing platforms. Mediators do not represent a mechanism, but an approach to structuring environment designs.

Taking full advantage of our mediator-based design approach is impossible, however, without an implicit invocation programming abstraction. The availability of explicit invocation alone implies dependence of an invoking component on the component invoked. The reason is that the invoking component must import the name of the method to be invoked from the component that defines it. In a mediator-based design, components would have to explicitly invoke mediators that relate them to other components, decreasing independence and, as shown in Section 2, complicating evolution.

The availability of an implicit invocation mechanism—an abstraction not supported across a wide range of languages—can be used to reverse such dependencies. Specifically, the invoking component simply exports an event; it is the component whose method is to be invoked that must import the name of the event. Since the invoking component does not import the names of the methods that it invokes, it can remain independent. This allows the designer to distinguish between situations in which a component *must* communicate and those in which it *may* communicate.

We can now present our approach in a nutshell. We design integration relationships as separate components to eliminate dependence of tools on each other. We then design tools to implicitly invoke mediators to eliminate the dependence of tools on mediators while still ensuring that update methods are invoked when necessary. The resulting combination of tool independence, tool visibility, and separate representation of relationships reconciles, to a reasonable degree, the conflict between integration and evolution.

3.3 Developing an Implicit Invocation Mechanism

Implicit invocation is not a new idea. Many mechanisms supporting this abstraction have been designed and are in use. These include, among many others, the LOOPS active values system [32], which is intended to support program debugging and simple graphical view consistency-maintenance; the change/update protocol of Smalltalk-80 used by the Model View Controller (MVC) to support complex graphical views; the FIELD message multicast mechanism, intended as a mechanism for integrating Unix-based tools; and relational triggers in AP5 [7], intended to support declarative, relational constraint-based programming.

A formal model. Despite the diversity of purposes and designs, in the abstract these mechanisms have key underlying similarities. We have developed a minimalistic formal framework to try to capture these similarities, both to better understand existing mechanisms and as a starting point for designing new mechanisms.

We characterize implicit invocation mechanisms in terms of a set E of events, each of which can be announced; a set M of methods, each of which can be invoked; a mapping EM: $E \rightarrow 2^M$ from events to multisets of methods; and an event announcement procedure A that invokes all the methods

associated with a given event whenever that event is announced.[6] We thus characterize the underlying structure of implicit invocation mechanisms in terms of a four-tuple (E, M, EM, A). We use this model as the basis for the design of a mechanism extending C++, and in Section 6 we apply it to an analysis of several existing mechanisms.

Additional requirements. In addition to this model, we have developed a set of requirements for implicit invocation mechanisms intended to support mediator-based design. These requirements reflect the idea that implicit and explicit invocation are dual to each other and of equal importance as programming abstractions. They are also intended to lead to mechanisms that have characteristics that help in the design and maintenance of complex systems. We thus adopt usage patterns established by explicit invocation mechanisms, such as the inclusion of event declarations in component interfaces.

First, because the events that a component announces are as important to its specification as its methods, event names and signatures[7] should be declared in component interfaces. Failure to satisfy this requirement makes it harder to use components that announce events. Meyers [25] has pointed out difficulties owing to this problem, which arise in FIELD, particularly in understanding events that are announced, which obviously is important when integrating new tools.

A second, related, requirement is that event announcement be explicit in source code. This tells readers precisely when control can be passed to other components. Clarifying component behaviors in this way eases reasoning about components that use implicit invocation. It also gives the designer precise control over which events are announced and when.

Third, any component should be able to declare events. This allows the designer to use implicit invocation when it seems appropriate, independent of the kind of component involved. Limiting the components that can use implicit invocation complicates design by limiting the designer's freedom to organize dependencies: components that cannot announce events cannot be integrated in the same way as those that can.

Many systems fail to meet this requirement. These include LOOPS, in which only variables announce events; APPL/A [34], in which only *relations* announce events; and Gandalf, which limits events to abstract syntax tree nodes [16]. These systems were not primarily intended to support general programming with implicit invocation, so they cannot be faulted in this dimension, but experiences with some of them indicate that these limitations significantly constrain design.

Fourth, the designer should be able to specify new event names and signatures. The available events should not be limited to those defined by the

[6]The range of EM is a multiset because many implicit invocation mechanisms allow multiple copies of a method to be associated with an event, with each being invoked on announcement of the event.

[7]Signatures are appropriate in languages in which method signatures are also declared.

system. Failure to meet this requirement complicates design in much the same way as restricting the components that can announce events, by denying the designer full control over the use of implicit invocation. We extensively exploit the ability of components to declare arbitrary events in the systems we have built.

A system that fails to meet this requirement is the Gandalf kernel. It defines about a dozen special-purpose events that are used to invoke action routines. Another is LOOPS, which limits events to variable read and write operations. AP5, Smalltalk-80, and FIELD, on the other hand, meet this requirement. In AP5, events are specified using relational predicates. FIELD events are specified with regular expressions. Smalltalk-80 defines a few fixed events, but allows for unrestricted event parameters, which enables encoding of arbitrary events.

Design and implementation. It is straightforward to implement implicit invocation mechanisms based on our model. It should take no more than a day or two to implement one in any object-oriented language. The ease with which such mechanisms can be implemented has significant benefits. It eliminates costly development efforts, makes it easy to use our approach without changing platforms, and broadens the range of application domains in which our approach can be applied. Mechanisms for platforms other than object-oriented languages are feasible given the equivalent of procedure pointers. Our primary mechanism, used to implement the examples and systems in this article, which we now discuss, extends C++ [33].[8]

In terms of our formal model, we require that each C++ object be able to declare events, hence our E consists of the collection of events exported by all the objects in a system. To a given event we want to be able to associate nonstatic member functions defined by other objects. Thus our M is the set of nonstatic member functions exported by all objects. EM then associates multisets of these member functions with events in E. Our event announcement procedure A behaves as follows. When an event is announced, the methods associated with the event are executed in an unspecified sequential order. Event announcements should not return until all invoked methods have returned. Actual event parameters are passed to the invoked methods using standard C++ parameter passing. One limitation of the mechanism we describe here is that events do not support return values: signatures are not fully first class. Our design can be extended easily to fix this, if necessary.

A critical decision in specifying an implicit invocation mechanism is deciding how the designer specifies EM. We take the simplest route. At runtime an object can register one of its member functions with any appropriately typed visible event. We require that the signatures of events and associated

[8] These systems were actually built using a number of minor variations of the mechanism we present in this article. The program restructuring tool, described later, used a CommonLisp/CLOS [3] implementation.

member functions conform. In terms of our model, registering a member function m with event e adds the tuple (e, m) to *EM*. We also allow objects to unregister methods, which deletes tuples from *EM*. Our mechanism is based on manipulation of *EM* as a set of tuples. An alternative that we do not adopt, but which we discuss in Section 6, is to allow the programmer to manipulate *EM* indirectly—using high-level expressions that specify sets of tuples, for example. Nor does our mechanism support statically-bound event-method connections, although it would be desirable to do so.

The key to implementing this mechanism is to design events as C++ classes. An object declares an event by including an event object as an instance variable in its public interface. An event object maintains a bag of pointers to member functions, to invoke when the event is announced, and exports a method which is used to announce the event. When invoked (i.e., when the event is announced), this method simply iterates over the elements of the bag invoking the designated member functions.

A complicating factor is that in C++ nonstatic member functions cannot be treated polymorphically (except as *void**s), even if their declared parameter lists and return types appear to conform. This problem is that nonstatic member functions have an implicit first argument, *this*, with type "pointer to the class that declares the function." This makes the types of nonstatic member functions unrelated in general. Thus, our event objects do not maintain bags of pointers to nonstatic member functions.

Rather, we use the fact that *static* member functions have no implicit *this* parameter; so if their declared types appear to conform, they do conform. We therefore define event objects to store bags of pointers to static member functions, all of the same type. For each nonstatic member function to be implicitly invoked we define a corresponding static member function with a parameter list identical to the member function's, except for an additional first parameter of type *void**. A nonstatic member function is registered with an event by registering a pointer to the object in which to invoke the member function and a pointer to the corresponding static member function. When an event is announced it invokes these static member functions, passing to each the specified object pointer as the first parameter followed by the rest of the event actual parameters. The static member function casts the object pointer to a pointer of the type of the receiving object and invokes the nonstatic member function through this pointer, passing it the rest of the argument list.

Event classes themselves export three methods. One implements event announcement. The signature of this method defines the signature of the event. The other two support registration and unregistration of static member function/object-pointer pairs, as described above. We use C++ macros to declare and implement event classes with event signatures specified by the designer. This frees the designer from having to know about how events are actually announced and associated with methods. We also use C++ macros to reduce the clutter required for the auxiliary static member functions, and to make it appear to the designer that nonstatic member functions are being registered directly with events.

Extending this design to meet additional requirements is easy. A new event class could be defined to impose a partial order on the bag of methods and to invoke methods respecting this order. The announcement method could use a C++ threads package such as Presto [1] to invoke registered methods concurrently. We have not implemented either of these enhancements, but are reasonably confident that there would be no great problems in doing so.

4. MEDIATOR-BASED DESIGN STYLES

In this section we present ways in which our mediator and implicit invocation-based approach can be used to model a variety of design styles especially common to integrated environments; one example is the representation of a shared database as a collection of views. We also present idioms for solving some common problems that arise in using mediators and events; one example is avoiding unbounded circularities.

We have developed idioms for meeting requirements of these sorts while retaining key characteristics such as component independence, visibility, and encapsulation of implementation details. We present our solutions through an example extending the graph environment.

4.1 Views for Tools

Tool-specific views, or representations, of shared data ease integration by allowing tools to operate on natural interfaces, rather than on interfaces conjured up to support a range of tools.

Garlan provides views for tools by merging abstract data types (ADTs) so that each provides a different view of a shared (merged) representation [11]. In merging, separate interfaces are preserved, but multiple implementations are replaced with a single implementation that supports both the views and any necessary consistency relationships among them. A system kernel produces merged implementations automatically using *compatibility maps*, which are selected inductively based on the system's knowledge of the basic types and relationships. Different interfaces onto a merged implementation can then be used by different tools. In Garlan's approach, a system has a fixed set of compatibility maps and composition rules; follow-on work loosened this restriction by providing a programming language for defining maps and composition rules [17].

An advantage of this approach is that it allows merged implementations to be optimized, with potentially significant savings in both time and space. For example, two views containing identical elements may share a single instance of that element in a merged view. On the other hand, the approach prevents types from being integrated simultaneously in several relationships: a view interface can be bound to only one merged implementation. Nor is merging desirable in all cases: protection or reuse concerns may suggest separate implementations, for example. Finally, in practice, generating merged implementations may be costly, since the types and relationships needed for a

broad range of environments are not likely to be known by the system in advance.

Our design approach illustrates a different way of achieving a similar structure. We have presented *VS* and *ES* as tools, but they can also be seen as *views* of a shared graph. *G* then serves the same role as a compatibility map, maintaining consistency among multiple views of shared data.

Our approach reuses existing view implementations by composing them using mediators. This obviates the need for merged implementations or for a system to generate such implementations. It also allows types to participate in multiple relationships simultaneously. Consider, for example, the way in which component *VS* can participate in both the **G** and **C** relationships. In the Garlan-style approach, a single merged type representing the components *VS*, *ES*, and *N* and the relationships **G** and **C** would be produced.

Our approach, however, does not naturally allow for optimization of a set of views, because we stress the importance of keeping the underlying components independent. Thus we are driven, for example, to keep multiple instances of the identical elements consistent. In cases where the costs of this are unacceptable, a hand-crafted alternative is required in our approach. However, in the absence of a clear need for optimization, our approach is more flexible.

4.2 Views of Tools

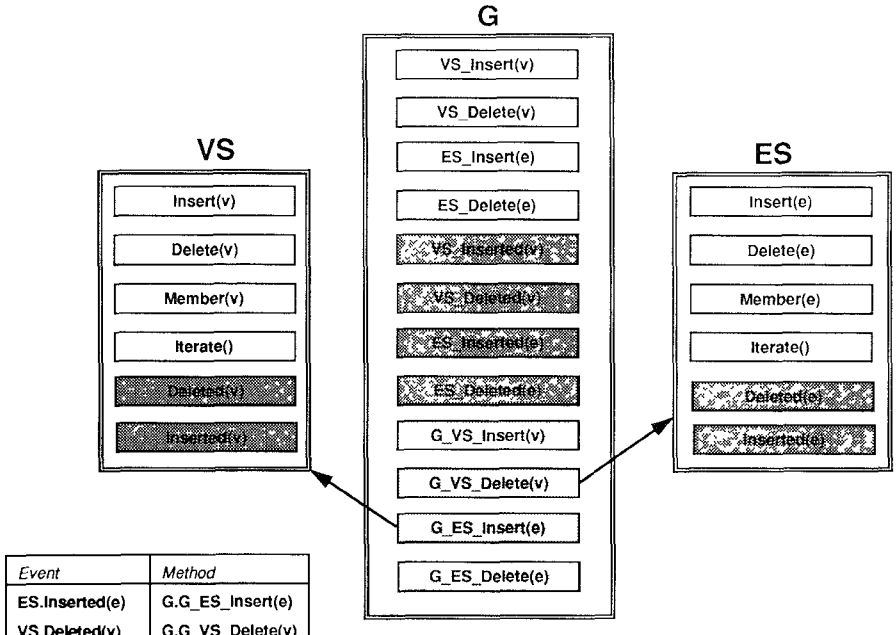
The converse problem is to provide integrated views of tools—single views representing the states, functions, and activities of multiple tools.

A common solution is to view tools as ADTs and to design integrated views as ADTs implemented in terms of these tool ADTs. This is straightforward, in theory at least, since ADT composition is well understood.

Our approach is similar, but is based on an extension of the notion of ADT. ADTs define abstractions in terms of method interfaces; our abstractions, in contrast, also include explicitly exported events. We use the term *abstract behavioral type* (ABT) to distinguish our components from ADTs. By including events in the interface, ABTs model components that can abstractly and concretely participate in integration relationships without sacrificing their independence.

Our solution to providing integrated views of a collection of tools, then, is to view tools as ABTs and to design integrated views as ABTs implemented on top of these tools. Although we have not developed either a formal theory of ABT composition or even rules of thumb about how to apply this notion, our experiences with this approach provide some evidence that ABT composition may be useful as a more general design technique.

We illustrate the approach by extending *G* to implement a graph ABT in terms of *VS* and *ES*. The main problem in making this extension is to ensure that the event announcements of *G* reflect a graph abstraction. Specifically, viewing the sequence of event announcements of the underlying vertex and edge sets is not at all the same as viewing a sequence of event announcements that are natural to the graph. For example, a vertex deletion should be



Event	*Method*
ES.Inserted(e)	G.G_ES_Insert(e)
VS.Deleted(v)	G.G_VS_Delete(v)
ES.Deleted(e)	G.G_ES_Delete(e)
VS.Inserted(v)	G.G_VS_Insert(v)

Figure 11

announced only after the incident edges are deleted, even if the deletion from the underlying vertex set *VS* actually happens first, which is possible since *VS* is visible. Otherwise, the viewer of the graph abstraction will temporarily see an inconsistent view of the graph in which there are edges containing nonexistent vertices; that is, the viewer of the graph will not be guaranteed that the integration relationship **G** always holds. It is important, however, to allow the underlying components, such as the vertex and edge sets, to continue announcing events for those components viewing the underlying components rather than the graph itself.

A new version of *G* in which the graph viewers both see a consistent sequence of event announcements, and also never see an inconsistent view of **G**, is presented in Figure 11. This graph ABT *G* exports public methods and events for vertex insertion and deletion and private update methods for maintaining the consistency of *VS* and *ES*. The public methods are implemented as direct calls to corresponding methods in *VS* and *ES*. *G.VS_Delete*(*v*), for example, simply calls *VS.Delete*(*v*). The update methods implement the solution to the event announcement problem.

The two update methods *G.G_VS_Insert*(*v*) and *G.G_ES_Delete*(*e*), are simple, announcing the events *G.VS_Inserted*(*v*) and *G.ES_Deleted*(*e*), respectively. The other update methods, *G.G_ES_Insert*(*e*) and

G.G_VS_Delete(v), maintain consistency and announce events. When *G.G_VS_Delete(v)*, for example, is invoked by deletion of a vertex from *VS*, it first deletes from *ES* edges incident on the deleted vertex and then announces the vertex deletion event of *G*. Before this event is announced, however, each deletion from *ES* implicitly invokes *G.G_ES_Delete(e)*, which announces the edge deletion event of *G* as described above. Thus, *G* does not announce the vertex deletion until after announcing deletion of all incident edges. This design implements a graph ABT *G* as a composition of the underlying set ABTs *VS* and *ES*. Components interested directly in the vertex or edge sets are unaffected.

4.3 Abstract Depictions

Chiron-1 [20] is characteristic of user interface systems that maintain consistency between internal and visual representations of components. In Chiron-1, *artists* are used to maintain consistency between an *abstract depiction* and another arbitrary data type. An abstract depiction is a representation from which a concrete (i.e., graphical) display of the type can be generated. An artist is a component that updates an abstract depiction when implicitly invoked by invocation of an operation exported by the data type, and that updates the data type when explicitly invoked by the *abstract depiction manager*, a part of the Chiron-1 system.

In our approach, we model this structure by defining appropriate new components to represent the abstract depictions and new mediators to represent the artists. In our example, for instance, to create concrete graphical views of the graph *G* we define a new component *P* (for *picture*) that exports methods and events for inserting and deleting *dots* and *segments* into and from a representation of a picture. Each dot and segment stores a graphic that can be directly displayed by a user interface system, such as InterViews [22]. Dot graphics are drawn as filled circles and segment graphics as line segments; these are the concrete depictions, or graphical realizations, of abstract depictions, as in Chiron-1.

We integrate *G* and *P* with a mediator *GP* that maintains a bijection between elements of *G* and *P*. When a vertex or edge is inserted into or deleted from *G*, a corresponding dot or segment is inserted into or deleted from *P*. Likewise, when *P* changes, *GP* updates *G* to restore consistency. This design supports both display and direct manipulation: when any tool changes *G*, *P* is updated, and when a tool such as a mouse handler changes *P*, *G* is updated. *P* thus serves as an abstract depiction and *GP* as an artist.

In contrast to Chiron-1, *P* and *GP* are undistinguished in our design, while abstract depictions and artists are fixed types used by all Chiron-1 instances. Both approaches have benefits. Chiron-1 provides extensive support for user interface development, including a rich abstract depiction language, a specialized version of Ada for implementing artists, and a mechanism implementing implicit invocation in a style similar to CLOS *wrapper methods* [3]. Our approach lacks these mechanisms but generalizes the technique, making it possible to integrate existing and new user interface systems more flexibly.

4.4 Compound Relationships

Relationships among complex representations can frequently be expressed, in part, as sets of simpler relationships among parts of the representations. To provide an example, we extend our relationships between dots in P and vertices in G by further requiring that the locations of corresponding dots and vertices remain equal as either is moved. Such a relationship would be useful for constructing intuitive user interfaces. This relationship can be expressed as a bijectivity relationship on the vertex and dot sets plus a set of equality relationships on corresponding dot-vertex pairs.

An idiom we have found useful is to use *submediators* to maintain independent parts of relationships of this sort. *GP*, for example, could deploy an instance of a new mediator *DV* (for *dot-vertex*) for each corresponding dot-vertex pair. Each would respond to events announced by associated dots and vertices to keep their locations equal. Similarly, G might deploy submediators between vertices and incident edges to keep edge locations consistent with those of their vertices.

This idiom mirrors the use of subartists in the Chiron-1 system. This works well when only a "moderate" number of submediators are needed. Although implementations of simple mediators can be kept small,[9] in an image analysis environment with millions-of-pixel images [23], a design using one submediator per pixel would probably be impractical.

4.5 Integrating Existing Tools

Another key problem for environment designers is to integrate existing tools into new environments. The possibilities are limited by interfaces exported by existing tools; these are often inadequate. While most tools export the equivalent of method interfaces to allow explicit invocation of tool functions, most do not export events by which the tools can be made to interact with other tools.

This leaves the designer with two obvious choices: to use a tool as is, limiting its usage to guarantee required invariants; or to add methods and events as needed to enable tight integration. There are at least two ways to implement the latter choice: by modifying the tool or by encapsulating it inside another component that exports a sufficient interface implemented in terms of the encapsulated tool. FIELD [27] takes the first approach; the Hewlett-Packard Encapsulator is a tool designed to facilitate the latter [10].

Our mediator approach does not dictate any of these choices, but facilitates integration once a choice is made. We have had several successes integrating existing tools without modification, for example. To illustrate, we hypothesize a user interface component D (for *display*) exporting methods to insert, delete, and move *graphic* elements, but exporting no events.

[9] In our C++ implementation a mediator to keep two integer-maintaining classes equal requires 25 bytes, 8 to reference the two integers, 16 to register with each of their *changed* events, and one to store a boolean value to limit the propagation of updates.

We integrate P and D through a mediator PD that updates D by inserting or deleting graphics into or from D upon being implicitly invoked by addition and deletion of dots and segments into or from P. We also integrate each dot or segment with D, using submediators, so that whenever a dot or segment moves, D is updated.

This is sufficient to maintain a correspondence between elements of D and P as long as no component other than PD changes D. This restriction is necessary because D does not announce events that could be used to trigger updates of P. We used this design in integrating a user interface based on InterViews with our graph ABT G.

To complete the system we defined a component H to handle a mouse input device. We handle mouse clicks by adding or deleting elements to or from P (since we cannot change D directly). We handle dragging by repeatedly moving a dot in P. Each update of a dot is reflected both to a vertex in VS and back to D by submediators. This design thus tightly integrates an "existing" user interface toolkit without modification. Griswold used this approach in integrating an existing program dependence graph component into a program restructuring tool. This system is described in the next section.

4.6 Miscellaneous

In this section, we discuss a number of mediator implementation strategies and idioms that are common and useful. We use GP as an example. The following definition shows the instance variables, update method declarations, and the implementation of one of these update methods in C++.

```
class GP {
    /* UPDATE METHODS */
    void WhenVertexDeleted(Vertex & v);
    void WhenVertexAdded(Vertex & v);
    void WhenEdgeDeleted(Edge & e),
    void WhenEdgeAdded(Edge & e);
    void WhenDotDeleted(Dot & d);
    void WhenDotAdded(Dot & d);
    void WhenSegmentDeleted(Segment & s);
    void WhenSegmentAdded(Segment & s);
    /* INSTANCE VARIABLES */
    G* graph;
    P* picture;
    Boolean VertexDotInterlock;
    Boolean EdgeSegmentInterlock;
    BinaryRelation VertexToDot;
    BinaryRelation DotToVertex;
    BinaryRelation EdgeToSegment;
    BinaryRelation SegmentToEdge;
};
void GP: :WhenSegmentAdded(Segment & s)
{
    if (ESInterlock == false)
    {
```

```
            ESInterlock = true;
            Vertex* v1 = new Vertex(s.Orig.X, s.Orig.Y);
            Vertex* v2 = new Vertex(s.Dest X, s.Dest Y);
            Edge & e = *new Edge(*v1, *v2);
            EdgeToSegment InsertPair(e, s);
            SegmentToEdge.InsertPair(s, e);
            graph → ES_Insert(e);
            ESInterlock = false;
        }
    }
```

Avoiding circularities. In general, allowing updates to either of two components participating in a relationship requires ensuring that circular propagation of updates is avoided. The mutual update dependence of G and P makes this necessary. For instance, if G changes P is updated, but this should not necessarily cause G to be updated again.

One common structure we use to overcome this is to add interlocks to the mediators. In our design of *GP*, for instance, we use two boolean-valued variables as interlocks. In many designs, one is sufficient. Here, one is used to break circularities involving edges and corresponding segments; the other is used to break cycles involving vertices and dots.

Two interlocks are needed in this design because addition of a segment to P, for example, requires addition of an edge to G, which may in turn require the addition of vertices to G, and hence of dots to P. Thus the first change to P may require the propagation of additional updates back to P.

Consider adding a segment to P, which implicitly invokes *GP.WhenSegmentAdded*(s), the implementation of which is given in the above definition of *GP*. This method sets the segment-edge interlock and then adds an edge, incident on two new vertices, to G.[10] To maintain the consistency of G, these vertices are added to the vertex set of G. The resulting vertex insertion events then implicitly invoke *GP.WhenVertexAdded*(v). Since the vertex-dot interlock is not yet set, this method sets it and then adds corresponding dots to P. Each *dot* addition implicitly invokes *GP* again, but these events are ignored since the vertex-dot interlock is now set. Finally, an edge is added to G, implicitly invoking *GP*; but the edge-segment interlock is already set, so this event is dismissed.

This design will not work in the presence of concurrency. One key problem is that updates having both local and nonlocal effects need to be serialized. It appears that a general solution compatible with our approach requires a nested transaction mechanism [26], although we have not verified this.

The design of *GP* also seems complex. However, this complexity is not a consequence of our approach, but of the requirement for a mutual update dependence between G and P. A similar interlock-based approach would also be necessary under the hardwired and event approaches discussed in Section

[10] A more useful implementation would define *GP* to look for vertices in G at locations corresponding to the endpoints of the new segment. If found, these vertices would be used; if not, one or both vertices would be instantiated, as necessary.

2. Of the approaches presented, only the encapsulation-based approach avoids this complexity: the mutual dependence is avoided by making only *GP* visible, hiding *G* and *P* entirely. This, in turn, makes integration of other tools with *G* and *P* more difficult.

Implementing *GP* seems to require a deep understanding of both *G* and *P*. It is true, and reasonable, that the implementor of *GP* must understand the specifications of *G* and *P*. However, the implementation details of the underlying types need not be known by the implementor of *GP*. That *G* is implemented as a composition of *VS* and *ES* can be ignored by clients of *G*, such as *GP*. Reasoning in terms of ABTs and ABT interactions may be more complex than in terms of ADTs, owing to the inclusion of implicit invocations in the specifications of ABTs. On the other hand, making the events explicit rather than implicit more accurately models the behavior of these components, which might make reasoning easier, not harder.

Externalized relations. We make extensive use of separate components to track relations among independent components. *GP*, for example, uses its instance variable *DotToVertex* to track the association between dots and vertices: given a dot, it returns the associated vertex. Rumbaugh et al. discuss a number of advantages of *externalized* representations of relations, in contrast to representations based on embedded pointers [29].

From our perspective, externalized relations have two advantages. First, they support integration of independent components. Mediators themselves can be viewed as externalized associations[11] with implicitly invoked methods that define the semantics of the association. The implementation of *GP* includes a pair of pointers, for example, one to *G* and one to *P*. This is an externalized representation of the association between these components.

Second, efficiently implemented relations enable more efficient integration. For example, we can improve the performance of the vertex-deletion update method of *G* by an order of magnitude—from $O(|ES|)$ to $O(|ES(v)|)$, the number of edges incident on v—by replacing the exhaustive search of *ES* for incident edges with a lookup in a hash table keyed by vertex. The cost is $O(|ES|)$ in space, to represent the relation and its inverse, and constant time per event announced by *ES* to keep these relations up to date.

5. APPLICATIONS AND EXPERIENCES

We have developed a number of medium-scale systems[12] using our mediator-based design approach. In this section, we briefly describe these systems, noting places where our approach was most useful.

Computer-Aided Geometric Design (CAGD). McCabe developed a prototype CAGD [24] kernel that extends our graph environment. This environment supports editing of a mesh, a representation of surface (technically, a two-manifold) in an n-dimensional Euclidean space. A mesh is similar

[11] We use the terms *association* to denote a tuple and *relation* a set of tuples.
[12] The systems we describe range in size from ten to thirty thousand lines of code.

to a graph, but it also maintains topological information—the ordering of edges around vertices, in this case.

This system supports multiple graphical views of a mesh. As a vertex is dragged in one view, for example, other views animate the motion. This allows a mesh with n-dimensional geometry ($n > 2$) to be viewed as a collection of two-dimensional projections.

A mesh, M, is represented as a vertex set VS and an edge set ES. Vertices in VS have property lists storing information such as vertex geometry and labels. ES stores a topological algebra—a set of edges plus several ordering relations over this set.[13] M implements a mesh ABT as a composition of VS and ES.

Separate topological and geometrical views and an integrated mesh view are useful. They allow one tool to create a torus or a sphere, a topological property, by operating on ES; another to assign a geometric layout by operating on VS; and a third to compute a smooth surface or a projection of the mesh by operating on M.

For example, McCabe defined a mediator that projects n-dimensional meshes into two-dimensional subspaces, leaving their topology unchanged. Another mediator composes two-dimensional meshes with components that serve as abstract depictions. A third composes abstract depictions with a display based on InterViews.

McCabe used submediators to keep n and two-dimensional vertices consistent. Because the mapping from n to two dimensions discards information, the reverse mapping from two to n is ambiguous. This is an instance of the general ambiguity problem for systems supporting views. McCabe solved this with mediators that fill in the missing information by mapping two-dimensional movements to n-dimensional movements parallel to the two-dimensional plane. Mediators provide a convenient framework for implementing designer-defined policies for any given instance of the ambiguity problem. This approach increases flexibility over any system that constrains designers by fixing a policy for handling ambiguity.

Program restructuring. Griswold applied our approach in his program restructuring tool, which allows an engineer to restructure a software system while preserving its meaning, in preparation for making functional enhancements [14]. If the formal parameters in a procedure declaration are swapped, for example, the system compensates by swapping actual parameters at all call sites, unless this would change the meaning of the program.

The key to the design of this tool is the maintenance of two consistent views of the program to be restructured. A syntactic view allows the engineer to indicate local changes and provides representations that the system updates to compensate for such changes. A semantic view

[13] ES implements a *quad-edge* data structure [15]. This is not perfect for a CAGD system; it only represents manifolds without boundary. Its clean structure, however, makes it useful for our work.

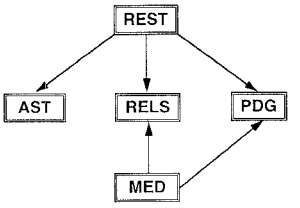


Figure 12

allows the system to determine whether a transformation preserves meaning and to find all aspects of a program that have to be changed to restore meaning in the face of a local change.

Figure 12 shows the structure of the system. *REST* is the restructuring tool; *AST* the syntactic representation based on an abstract syntax tree; *PDG* the semantic representation based on a program dependence graph; *RELS* a set of externalized relations between *AST* and *PDG* nodes; and *MED* is a mediator that maintains *RELS* while keeping *AST* and *PDG* consistent.

MED employs several interesting mediator idioms. First, it allowed over 10,000 lines of existing code, comprising *PDG*, to be integrated almost without change.[14] Second, it provides an integrated view of *AST* and *PDG*, which includes *RELS*. *REST* makes extensive use of these relations in mapping between syntactic and semantic program representation. Third, because computation of program dependence graphs from abstract syntax trees is expensive,[15] and since *PDG* consistency is required only at the beginning of a restructuring operation, *MED* caches changes made to the *AST* during restructuring. This illustrates a general issue for systems supporting views: it is often beneficial to allow *slippage* in the consistency of multiple representations. Mediators provide a convenient framework for implementing slippage and for isolating the required implemention details from the components involved.

Parallel programming. We also applied our approach in the design of a prototype environment for programming nonshared-memory parallel computers [30]. Abstractly, a program in this system is a sequential composition of *phases*, each phase is a parallel program that runs on an interconnected processor *graph*; and each processor in a graph executes a sequential *process* in any given phase. Concretely, a program is represented as a collection of objects in the C++ programming language.

The environment supports four interactive tools for manipulating programs. The phase and graph tools are graphical, depicting programs as flowcharts and interconnection graphs, respectively. The process tool, used to write simple sequential programs, is text based. The fourth tool provides a

[14]Griswold's event mechanism, which implements an earlier version of the system specified in this article, uses CLOS wrapper methods to announce events.

[15]The complexity is not known, but is thought to be roughly cubic in the size of the program.

user interface that allows instances of the other tools to be created and deleted dynamically.

The phase, graph, and process tools are related to their respective parts of a program database by mediators that keep the depictions and the database mutually consistent. Furthermore, the three parts of the database are also related to each other by mediators. If the name of a sequential process is changed, all graphs having processors that execute that process are updated accordingly; these updates then propagate to all graph tools.

This structure allowed separate designers independently to develop the parts of the database and the tools that operate on them. Once the tools and their respective parts of the databases had been developed and tested, we integrated them by adding mediators to keep related representations consistent.

6. RELATED WORK: IMPLICIT INVOCATION

Our research has helped us understand and evaluate a range of systems. Many have contributed to the nature of the implicit invocation mechanism we use. Our research in turn has produced a framework that we have found useful in analyzing existing implicit invocation mechanisms. We organize this section by system, taking a similar approach in the next section, which discusses systems related to our use of mediators. The systems we discuss cover many of the key issues in implicit invocation and mediators, and they seem to be representative of the many important systems that we omit.

In this section we analyze four implicit invocation mechanisms in terms of the model presented in Section 3. This analysis serves several purposes. It helps to validate our model by showing that it captures a key underlying similarity among of a set of seemingly diverse mechanisms. It also highlights capabilities supported by some mechanisms that are not captured by our model or implemented by our mechanism. A more recent effort uses Z [31] to explore these dimensions based on a formal description of implicit invocation systems [13].

As we discussed in Section 3, we characterize implicit invocation mechanisms in terms of a set E of events, each of which can be announced, a set M of methods, each of which can be invoked; a mapping EM: $E \rightarrow 2^M$ from events to multisets of methods; and an event announcement procedure A that invokes all the methods associated with a given event whenever that event is announced. In the analysis of related mechanisms, we focus on three issues: the manner in which the user manipulates the association EM between events and methods; the behavior of the event announcement procedure A; and restrictions, if any, imposed on E, M, and EM.

We note here (and discuss no further) that some mechanisms use their representations for purposes not strictly related to implicit invocation. An attribute grammar system may use its EM to optimize attribute evaluation and to detect cyclic dependencies among attributes.[16] FIELD queries its EM

[16] In most such systems, ES is represented explicitly only during attribute grammar processing; at run-time, associations are represented as direct links between tree nodes.

to avoid spawning redundant copies of network tools. An *EM* could also be used to support debugging by indicating which methods would be invoked by announcement of a given event.

6.1 LOOPS

LOOPS [32] extends LISP with *active values*, to support *access-oriented* programming. An active value av is an object used to annotate a variable v. Such an annotation causes each access to v to be replaced by invocation of a method exported by av. Specifically, reading v invokes $av.GetWrappedValue()$, and writing the value x into v invokes $av.PutWrappedValue(x)$. The base active value class defines default implementations for those methods that mimic variable reads and writes using the local state of active value objects. Subclass designers specialize these methods for additional behaviors.

We model this mechanism as follows. Each variable v contributes two events to E, namely $v.read()$ and $v.written(x)$. Each active value av contributes two methods to M, $av.GetWrappedValue()$ and $av.PutWrappedValue(x)$. Attaching av to v or detaching it from v adds to or deletes from EM the maplets[17]

$$v.read() \mapsto o.GetWrappedValue()$$
$$v.written(x) \mapsto o.WrappedValue(x).$$

The user thus manipulates *EM* in terms of fixed pairs of tuples specified in terms of a variable object and an active value annotation object. LOOPS also restricts both the components that announce events—to variables—and the events that can be announced—to *read* and *written*.

On the other hand, LOOPS supports a capability not captured by our model or mechanism: the programmer can specify an order on the set of methods associated with an event. This is accomplished by nesting active value annotations. When a variable is accessed, active values attached to it are activated; when these access their instance variables, nested active values may be implicitly invoked.

This suggests a possible extension of our model and mechanism: the user could be allowed to specify a partial order on the methods associated with a given event. The ordering issue arises naturally, since an event may be imported by many components simultaneously. A partial order allows for no order, a complete order, or a properly partial order.

6.2 APPL/A

APPL/A [34] is an Ada-based language that supports process programming. APPL/A's implicit invocation mechanism focuses on a relational model. APPL/A imposes restrictions similar to those of LOOPS. Only relations, instances of a special type constructor, can announce events, and the events that relations can announce are restricted to insert, delete, update, and find.

[17] By *maplet* we mean a single tuple (e, m) in the overall mapping *EM* of events to methods.

These restrictions imply that the programmer must either use explicit invocation mechanisms to connect components that are not relations, or model those components as relations. If explicit invocation is used, the potential benefits of implicit invocation, such as easier evolution, are lost. If all components are modeled as relations, then the programmer may have to forego more natural representations in situations where relations are not best.

APPL/A, in contrast to our mechanism, handles concurrency by allowing the designer to specify the synchronization of event announcement with the execution of invoked methods. APPL/A also allows a component that receives multiple events to prioritize these events. Events are queued at the component, which handles them in priority order.

6.3 Smalltalk-80

Smalltalk-80 defines an implicit invocation mechanism in which objects register themselves as dependents of other objects [21].[18] All objects in this system export three predefined events and three predefined methods. When one object x registers with another object y the predefined methods of y are associated with the corresponding events of x.

Specifically, the method *updateWith*(s, p) is associated with the *changedWith*(s, p) event, *updateRequest*(s) is associated with *changeRequest*(s), and *performUpdate*(s, p) is associated with *broadcastWith*(s, p). The event parameter s is a Smalltalk-80 *symbol*, a literal value, and p is an arbitrary object.

ChangeWith(s, p) is announced to notify registered objects, usually graphical views, that the announcing object, a *model*, has changed. Views can then update themselves to restore consistency. *ChangeRequest*(s) is announced to request permission from registered objects to make a change. The methods implicitly invoked return true or false to indicate whether permission is granted or not. These answers are conjoined and the result returned to the event announcer. This can be used to implement cooperation among components that do not know about each other. *BroadcastWith*(s, p) invokes, in each registered object, the method named by s with parameter p.

In terms of our model, an object x contributes its three predefined events to E and its three methods to M. Registering x with, or unregistrating it from, y, respectively, adds to or deletes from EM the following three maplets:[19]

$$\begin{aligned} x.changedWith(s, p) &\mapsto y.update(s, p) \\ x.changeRequest(s) &\mapsto y.updateRequest(s) \\ x.broadcast(s, p) &\mapsto y.s(p). \end{aligned}$$

[18] While this change dependence mechanism is available for other reasons, it is used most aggressively in MVC.

[19] Actually, there are eight events with corresponding methods. The additional events are short forms of those described above. There are three for change notification, for example. One takes both a symbol and an object as parameters; one takes just a symbol; and one takes no parameters.

In this system the user manipulates *EM* indirectly by specifying, through explicit registration, an object dependence relation from which *EM* is derived. Specifically, an object dependence implies two maplets in *EM* corresponding to the *changedWith* and *changeRequest* events, plus additional maplets, one for each possible value of the parameter *s* passed to the *broadcast* event. The announcement procedure *A* invokes methods sequentially in an unspecified order.

The Smalltalk-80 mechanism suggests two extensions to our model and perhaps to our mechanism. First, specifying *EM* indirectly, in terms of another representation, can be useful. In contrast to the mechanisms discussed in this article, ours is based on direct, low-level manipulation of *EM*. In part, this is for the sake of simplicity, but it also implements a programming abstraction that emphasizes our analogy between explicit and implicit invocation. Second, the way in which *changeRequest* computes a return value suggests that returning values from an implicit invocation requires synthesis of the several values returned by the implicitly invoked methods. The Smalltalk-80 mechanism defines the semantics of this operation for the *changeRequest* event as Boolean conjunction. General support for return values would seem to require that the designer be able to specify such procedures.

A limitation of this mechanism is that only system-defined events are announced. Events defined by the designer can be encoded as parameterized versions of these events, but this solution is not completely satisfactory.[20] One reason is that designer-defined events do not appear in component interfaces, increasing the difficulty of reasoning about events. More seriously, defining new events by using a parameter conflicts with class inheritance for code reuse. The problem occurs when a subclass needs both to hide events exported by the superclass and also to export its own events. Either it can hide the system-defined events, in which case it cannot export its own, or it can leave them visible, leaving superclass events visible. The problem is that information-hiding mechanisms such as private subclasses in C++ operate on interfaces and cannot be used to hide selected parameterizations of events or methods.

6.4 FIELD

FIELD [27] defines an integration mechanism for Unix-hosted tools based on selective broadcast of ASCII messages on a network. Both explicit and implicit invocation abstractions can be implemented on top of this basic communication mechanism, although the indirect connection of implicit invocation is of most interest to this analysis.

Each FIELD tool exports a method interface by defining a set of messages to which it can respond. A message is an ASCII string that encodes both the method name and parameters. Tools can invoke other tools, either

[20] Of course, Smalltalk-80 makes it easy to define classes and methods representing new events, but in practice such modifications are not common.

explicitly or implicitly—the distinction is only in the designer's use of the mechanism—by sending messages to a distinguished tool called the *message server*, which routes invocations to their destinations.

Routing is based on a set of expressions registered by tools with the server. An expression consists of a pattern part and an action part. Patterns are written as regular expressions. When the server receives an invocation message, it finds all expressions for which the pattern matches the (ASCII) message string. Each corresponding action part specifies a method to be called in the tool that registered the expression. All such methods are then invoked. A pattern that literally names a specific tool and method supports explicit invocation, since invokers must name the tool and the method to invoke. Other patterns can be used to specify events or classes of events.

In terms of our model, a FIELD tool t contributes a set of methods $\{m_t\}$ to M. E corresponds to the infinite set of ASCII strings, since a tool can announce an event by sending an arbitrary string. EM is specified indirectly in terms of a set T of pattern-method pairs $\{(p, m)\}$. EM is then computed from T as necessary.

Specifically, each regular expression p specifies an equivalence class of events, $Events(p) = \{e \in E | matches(p, e)\}$, where *matches* is a predicate that is true if and only if event e is recognized by regular expression p. The effect of adding (p, m) to or deleting it from T is to add to or delete from EM the set of maplets

$$\{e \mapsto m | e \in Events(p)\}.$$

Thus EM, too, can be infinite, since an infinite class of events can be associated with a method by insertion of a single pair (p, m) into T. Obviously, EM is not represented as an infinite set. Rather, the announcement procedure A calculates $EM(e)$ when the event e is announced. Specifically, the multiset of methods $EM(e)$ invoked by e is

$$\{m | (p, m) \in T \wedge matches(p, e)\}.$$

FIELD thus supports a declarative event specification mechanism based on regular expressions. Our approach, in contrast, lacks the equivalent of T and a mapping from T to EM. Instead, our design is intended to support a more imperative style of programming with implicit invocation. We do not yet fully understand the tradeoffs involved in these respective choices.

FIELD also supports concurrent and asynchronous invocation. The FIELD announcement procedure A invokes methods concurrently in an unspecified order. An event announcer specifies whether the event is synchronous or asynchronous. Asynchronous events return immediately with no return value. Synchronous announcements return after all methods return, returning the string value returned by the first method to terminate.

This suggests possible extensions to our mechanism. A concurrent mechanism encapsulating the complexities of thread creation, dispatching, and synchronization may provide significant benefits at a reasonable cost.

However, we have no systematic approach to combining asynchronous invocation with mediators.

6.5 AP5

AP5 [7] extends Common Lisp with a relational database abstraction and a mechanism based on *triggers* to support declarative, constraint-based programming. An AP5 trigger $t = (p, m)$ has a predicate part p and an action part m. A predicate is written in a notation based on first-order logic, extended with relational operators, and a temporal aspect capturing the notions of *before* and *after* a proposed transaction. When a transaction is proposed, the system logically reevaluates all predicates and invokes the corresponding action parts of those that evaluate to true. To maintain a constraint, a predicate detects violations that would occur if proposed transactions were to commit; the corresponding action part then either modifies and resubmits the proposed transaction or simply aborts it. Action parts can also be used to implement arbitrary activities, such as sending mail to a project manager.

In terms of our model, E is the infinite set of proposed changes to a database. Much as in FIELD, an AP5 predicate p specifies a set of events $Events(p) = \{e \in E | satisfies(e, p)\}$, where *satisfies* is a predicate that is true if and only if the application change e would cause the database to satisfy relational predicate p. Thus, a trigger $t = (p, m)$ contributes m to M and, to EM, the set of maplets

$$\{e \mapsto m | satisfies(e, p)\}.$$

The AP5 announcement procedure A invokes the methods triggered by an event in an unspecified sequential order.

A key advantage of AP5 is its support for atomic transactions, which it uses to guarantee global consistency in the face of concurrent updates. We believe that transactions are necessary to solve the consistency problem in a general way for concurrent environments. Although we have not verified this, it appears that our imperative approach can exploit transaction-based concurrency control by implementing tools and mediators on a platform supporting nested transactions.

7. RELATED WORK: MEDIATORS

Our emphasis on the separate specification, design, and implementation of relationships among components is based, in part, on evidence produced by numerous related efforts that suggests the utility of this approach. In this section, we discuss a number of these systems. Many of these systems provide support in areas unrelated to our focus—object management and user access control, for example. Because our work applies narrowly to integration and evolution, we omit discussion of these and other important issues that these systems address.

7.1 Unix Pipes

Most Unix programs, such as ls, awk, grep, and sort are independent but easy to integrate using pipes [28]. This ease of integration derives from the separation of tool and integration concerns, combined with a flexible mechanism for recombining the two. Unix shells that allow the user easy access to pipes also encourages their use. We view Unix programs as tools, and pipes as mediators that relate them. (Viewing pipes as simple tools and Unix programs as complex mediators is also reasonable.)

Although this approach makes it easy to assemble systems by composing sequences of tools, it makes it hard to implement tightly integrated, interactive environments. The key reason is that only one-way stream composition relationships are supported. In addition, common practice in Unix limits programs to a single input (stdin) and two output (stdout and stderr) streams to which pipes can be associated. These restrictions make it hard to implement mutual dependencies among tools and make sharing of fine-grained, structured information costly, since each tool has to linearize and parse data.

7.2 Forest

Forest [12] is an extension of FIELD. As in FIELD, tools register pattern-action pairs with a message server. In Forest, however, an action part is not an invocation message to be sent to a tool, but a *policy program* to be executed. A policy program can send an invocation message to a tool, send another message to the message server, or do nothing.

Policy programs are written as sets of condition-action pairs. Conditions are Boolean expressions over variables and environment-defined functions, such as *LoadAverage*( ), which returns the system load. The conditions are evaluated in the order written. If none evaluate to true, no action is taken, otherwise the action associated with the first true expression is executed.

Policy programs can be viewed as limited mediators. Each is bound to a single event; they are not first-class (Unix tool) components; and the language in which they are specified is quite limited. On the other hand, they suggest that mediators written in an interpreted language may be a reasonable approach to integrating tools on the fly.

7.3 Contracts

Helm et al. define a language for specifying behavioral relationships based on *contracts* and *conformance declarations* [18]. A contract specifies a relationship in terms of interfaces that participating components must export; preconditions on and initialization of these components; an invariant to be maintained; and mutual invocation obligations sufficient to maintain the invariant. To encourage the identification of generally useful relationships, contracts are generic with respect to participating components. Conformance declarations specify how particular components discharge the obligations of roles specified by a contract.

This work provides additional evidence that behavioral relationships should be treated as first-class entities in software systems. We agree; but the approach we define differs significantly from that of contracts.

First, contracts specify policies in terms of explicit invocation obligations. While this does not preclude the design of behavioral relationships as separate components, it is biased towards hard-wired designs. Our approach is to state invariants and policies in a way that does not bias design. This eases adoption of a mediator-based design, but also leaves the designer free to adopt hard-wired or other approaches as circumstances dictate.

Second, contracts emphasize relationships *over* the components (classes in this system) that they relate. Helm et al. state that "the specification of a class becomes spread over a number of contracts and conformance declarations, and is not localized to one class definition" [*ibid.*, p. 178]. On the other hand, contracts are justified by the observation that in previous languages, "behavioral compositions... are spread across many class definitions" [*ibid.*, p. 169]. Our approach, in contrast, emphasizes the equal importance of relationships and the components they relate.

Third, our approach is tailored for use with existing languages. We isolate invariants in the specification, as opposed to language-level, because these declarative constructs cannot be represented easily in imperative programming frameworks.

7.4 ThingLab II and OPUS

Constraint programming is based precisely on separate specification of relationships among otherwise independent components. Constraint programming systems emphasize language support for the declarative assertion of constraints.

Many systems, including ThingLab II [8], a successor to ThingLab [4], and OPUS [19] are based on a set of *variables*—components storing values from some domain—and a set of constraints relating these values. When the value of a variable changes, a constraint satisfaction algorithm is invoked to resatisfy specified constraints by changing the values of other variables. If no satisfactory *solution* (an assignment of values to variables) can be found, an error is flagged.

Key differences among constraint programming systems follow from the constraint-satisfaction algorithms they support. OPUS uses a relatively simple incremental attribute evaluation algorithm that propagates changes in one direction only. If changing a updates b, changing b cannot update a, for instance. ThingLab II uses a more complex algorithm that allows the system to choose the direction in which changes propagate.

In more detail, a ThingLab II constraint is implemented as an object that maintains references to the variables that it constrains and that exports a set of methods to the system, one for each such variable.[21] Each method must

[21] The absence of a method for a given variable constitutes an implicit read-only annotation, indicating that the variable cannot be influenced by the presence of the constraint.

guarantee that it resatisfies the constraint by setting the value of its variable as a function of the values of the other variables.

Thus the system has a choice of which variable to change to resatisfy a constraint. Given a set of constraints and a variable to be changed, the system computes a compatible set of choices, one from each constraint object affected (directly or indirectly) by the change. ThingLab II is incremental in its computation of this set, which is called a *plan*.

This suggests an extension to our implicit invocation model. Logically, changing a variable implicitly invokes one of the methods defined in each object attached to the variable. This can be characterized as disjunctive, as opposed to conjunctive, implicit invocation: a *select* procedure chooses one method from those registered for invocation. In these systems *select* bases this decision on exactly the information used in computing a plan. In other systems this choice might be based on performance or other considerations.

In contrast to our approach, constraint systems guarantee global properties based on an analysis of their underlying constraint representations. Both ThingLab II and OPUS detect and prohibit cyclic dependencies, and ThingLab II ensures that no two constraint methods writing to the same variable are included in the same plan.

On the other hand, these systems are not suitable as integration mechanisms for complex environments. First, they limit the components that can be related, the relationships that can be specified, and the policies that can be used to maintain relationships. ThingLab II prohibits methods that change more than one variable, for example, precluding policies that update both a set and a relation. OPUS prohibits mutual dependencies. Both limit constraints to variables, as opposed to arbitrary abstract types.

Second, constraint systems based on variables are incompatible with encapsulation, and hence with data and behavioral type abstraction: objects other than variables can be constrained only by constraining variables in their respective implementations. This is true of OPUS, ThingLab II, and Kaleidoscope [9], an object-oriented language that tightly integrates the ThingLab mechanism. This is a serious shortcoming with respect to our objective, to help manage complexity in the face of integration. Type abstraction and encapsulation are key tools for managing complexity; and integration mechanisms compatible with these techniques are needed.

Our approach supports the integration of arbitrary types, specification of arbitrary relationships, and definition of relationship-maintenance policies that respect type abstractions and encapsulation boundaries.

8. CONCLUSION

8.1 Limitations

Our approach has proven useful in a number systems that we have developed. Several problems have to be solved, however, for it to be applied in the development of larger integrated environments.

First, a key problem is to extend it for design of concurrent and distributed environments. Some systems use atomic transactions to maintain global

consistency in the face of concurrency. It appears that our approach can be combined with nested transactions for this purpose; it is not clear, though, that this is a reasonable solution. We have yet to find the best way to balance the benefits of our lightweight approach with the need for global consistency. The experience with FIELD suggests that practical environments can be built without a general solution to this problem. Whether a mediator-based approach will prove useful in such an environment is not known. To enable research in this dimension we have designed and are implementing a *remote* implicit invocation mechanism to complement a remote procedure call [2] mechanism.

Second, reasoning about systems constructed using implicit invocation mechanisms may not be easy as the scale of systems increases. Our experience with medium-scale systems, such as the CAGD system discussed earlier, gives us some confidence—but no conclusive evidence—that scale will not be a serious problem. Unconstrained implicit invocation would, in all likelihood, present a reasoning problem (just as unconstrained explicit invocation would). However, designing with mediators is quite constrained, which may account for the absence of problems so far. Additionally, mediators use explicit invocation to maintain relationships among components, so considering our approach as a reasoned balance of implicit and explicit invocation is more accurate than considering it as implicit invocation alone.

8.2 Synthesis

Our design approach. One way to manage the complexity of integrated environments is to employ programming constructs with restricted semantics in order to automatically ensure that basic properties, such as termination and global consistency, hold. The change propagation mechanisms used by ThingLab II and OPUS represent such mechanisms.

These mechanisms are appropriate for problems suited to their particular strengths, but the restrictions they impose on components, relationships, and relationship-maintenance policies make them unsuitable as general mechanisms intended to ease development and evolution of integrated environments. The constructs they support may not be appropriate for implementing a given design, and the semantic restrictions they impose may exact significant tolls in decreased efficiency.

Overcoming these problems may require semantically less restricted constructs, such as imperative programming with implicit and explicit invocation. On the other hand, undisciplined use of an imperative style can make it impossible to reason about a resulting system. Our approach is intended to exploit the familiarity and flexibility of these imperative constructs, while helping the designer to control complexity by providing design composition techniques that encourage software structures robust with respect to changing requirements.

Implicit invocation. Our approach includes implicit invocation as a first-class abstraction, dual to explicit invocation. We presented a model

of implicit invocation in terms of which we characterized a number mechanisms. Each of these placed a layer between the programmer and the event-method association, *EM*. The Smalltalk-80 *EM*, for instance, is specified indirectly in terms of object dependencies. Reducing implicit invocation to a minimum by stripping away this layer, as in our model and in our mechanisms, reveals more clearly the duality between explicit and implicit invocation.

We also showed how object-oriented languages can be extended easily to support our implicit invocation abstraction. Although we have only limited experience with this model, its utility in characterizing a diverse range of mechanisms provides additional evidence that it, and hence our mechanism, is reasonable.

Design tradeoffs. Our approach encourages the decomposition of requirements as well as designs and implementations along lines separating components from the integration relationships among them. Specifications that fail in this regard may bias designers toward designs that complicate environment integration and evolution. On the other hand, specifications organized in this manner leave the designer free to choose among a range of designs. We identified four idealized approaches that encourage designs with distinct tradeoffs among ease of integration, evolution, and, perhaps, efficiency.

Given a specification in this form, which approach should a designer choose? In the absence of other concerns, a software system should be designed for change. Our analysis suggests that, of the canonical approaches we discuss, our mediator approach yields the greatest ease of evolution in integrated environments. An interesting characteristic of this approach is that it preserves the specification structure separating components and relationships in both design and implementation. In the absence of this correspondence, which is disrupted by the other approaches we discuss, it is unlikely that the overall cost of an evolutionary change will be proportional to its apparent size in the specification. Although we cannot yet define *apparent size* or *proportional* with precision, this property is clearly desirable: the client specifies changes in terms of the specification and bases expectations about costs on it. Designs that amplify actual costs for reasons not apparent to the client are likely to prove unacceptable as requirements evolve.

ACKNOWLEDGMENTS

Tony DeRose, Bill Griswold, and Larry Snyder provided the domains for the environments discussed herein. Alex Klaiber, Bob Mitchell, and Sitaram Raju implemented the tools in our parallel programming environment. Tom McCabe's experience with his CAGD system provided valuable feedback that helped significantly in the development of our approach. Yoshi Yamane and Derrick Weathersby contributed to the ideas of ABT composition and disjunctive implicit invocation, respectively. Robert Wahbe pointed out the utility of macros in simplifying the use of a C++ implicit invocation mechanism. In addition to many of these people, Gail Alverson and Michael Hanson gave us

useful comments on an earlier version of this paper. Finally, the comments and suggestions made by the anonymous referees were cogent, valuable, and appreciated.

REFERENCES

1. BERSHAD, B. N., LAZOWSKA, E. D., LEVY, H. M. PRESTO: A system for object-oriented parallel programming. *Softw. Pract. Exper. 18*, 8 (1988), 713–732.
2. BIRRELL, A. D., AND NELSON, B. J. Implementing remote procedure call. *ACM Trans. Comput. Syst. 2*, 1, (Feb. 1984), 39–59.
3. BOBROW, D. G. ET AL. Common Lisp object system specification X3JI3. Doc. 88-002R. *ACM SIGPLAN Not. 23* (Sept. 1988).
4. BORNING, A. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Trans. Program. Lang. Sys. 3*, 4 (Oct. 1981), 353–387.
5. CAGAN, M. R. The HP SoftBench environment: An architecture for a new generation of software tools. *Hewlett-Packard J. 41*, 3 (June 1990), 36–47.
6. COAD, P. AND YOURDON, E. *Object-Oriented Analysis*. Yourdon Press, Englewood Cliffs, N J., 1991.
7. COHEN, D. Compiling complex transition database triggers. In *Proceedings of the 1989 ACM SIGMOD* (Portland, Ore., 1989) pp. 225–234.
8. FREEMAN-BENSON, B., MALONEY, J., BORNING, A. An incremental constraint solver. *Commun. ACM 33*, 1 (Jan. 1990), 54–63.
9. FREEMAN-BENSON, B. N. Kaleidoscope: Mixing objects, constraints, and imperative programming. In *Proceedings of OOPSLA / ECOOP 90*, (Ottawa, 1990), 77–88.
10. FROMME, B. D. HP encapsulator: Bridging the generation gap. *Hewlett-Packard J. 41*, 3 (June 1990), 59–68.
11. GARLAN, D. Views for tools in integrated environments. Ph.D. dissertation, Carnegie-Mellon Univ., 1987.
12. GARLAN, D., AND ILIAS, E. Low-cost, adaptable tool integration policies for integrated environments. In *Proceedings of SIGSOFT90: Fourth Symposium on Software Development Environments* (Irvine, Calf., 1990), 1–10.
13. GARLAN, D., AND NOTKIN, D. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91, Formal Software Development Methods. Lecture Notes in Computer Science 551*, Springer Verlag, New York, 1991.
14. GRISWOLD, W. G., AND NOTKIN, D. Program restructuring as an aid to software maintenance. Tech. Rep. 90-08-04, Dept. of Computer Science and Engineering, Univ. of Washington, Aug. 1991.
15. GUIBAS, L., AND STOLFI, J. Primitives for the manipulation of three-dimensional subdivisions. *ACM Trans. Graph. 4*, 2 (April 1985), 74–123.
16. HABERMANN, A. N., AND NOTKIN, D. Gandalf software development environments. *IEEE Trans. Softw. Eng. SE-12*, 12 (Dec. 1986), 1117–1127.
17. HABERMANN, A. N., KRUEGER, C., PIERCE, B., STAUDT, B., AND WENN, J. Programming with views. Tech. Rep. CMU-CS-87-177, Carnegie-Mellon Univer. Jan. 1988.
18. HELM, R., HOLLAND, I. M., AND GANGOPADHYAY, D. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA / ECOOP 90.* 1990, 169–180.
19. HUDSON, S. E., AND MOHAMED, S. P. Interactive specification of flexible user interface displays. *ACM Trans. Inf. Syst. 8*, 3 (1990), 269–288.
20. KELLER, R. K., CAMERON, M., TAYLOR, R. N., AND TROUP, D. B. User interface development and software environments: The Chiron-1 system. In *Proceedings of the 13th International Conference on Software Engineering* (Austin, Tex., May 1991), 208–218.
21. KRASNER, G. E., AND POPE. S. T. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program. 1*, 3 (Aug./Spet. 1988), 26–49.
22. LINTON, M. A., VLISSIDES, J. M., AND CALDER, P. R. Composing user interfaces with InterViews. *Computer 22* 2 (Feb. 1989), 8–22.

23. MacDonald, J. A., and Stuetzle, W. Painting multiple views of complex objects. In *Proceedings of OOPSLA / ECOOP.* 1990, 245–257.
24. McCabe, T. Programming with mediators. Developing a graphical mesh environment. M.S. thesis, Univ. of Washington, 1991.
25. Meyers, S. Difficulties in integrating multiview development systems. *IEEE Softw.* (Jan. 1991), 49–57.
26. Moss, J. Nested transactions: An introduction. In *Concurrency Control and Reliability in Distributed Systems.* B. Bhargava, Ed., Van Nostrand Reinhold, New York, 1987.
27. Reiss, S. P. Connecting tools using message passing in the field environment. *IEEE Softw.* 7, 4 (July 1990), 57–66.
28. Ritchie, D. M., and Thompson, K. The Unix time-sharing system. *Bell Syst. Tech J. 57*, 6, part 2 (July-Aug. 1987), 1905–1930.
29. Rumbaugh, J. Relations as semantic constructs in an object-oriented language. *Proceedngs of OOPSLA.* 1987, 466–481.
30. Snyder, L. The XYZ abstraction levels of poker-like languages. In *Proceedings of the Second Workshop on Parallel Compilers and Algorithms* (Urbana, Ill., 1989).
31. Spivey, J. M. *The Z Notation: A Reference Manual,* Prentice-Hall Internatinal, Englewood Cliffs, N.J., 1989.
32. Stefik, M. J., Bobrow, D. G , and Kahn, K. M. Integrating access-oriented programming into a multiparadigm environment. *IEEE Softw.* (Jan. 1986), 10–18.
33. Stroustrup, B. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 1976.
34. Sutton, S., Heimbigner, D., and Osterweil, L. Language constructs for managing change in process-centered environments. In *Proceedings of SIGSOFT90· Fourth Symposium on Software Development Environments.* (Irvine, Calif., 1990), 206–217.

Received May 1991; revised December 1991; accepted February 1992