Adding Implicit Invocation to Languages: Three Approaches*

David Notkin†
David Garlan‡
William G. Griswold††
Kevin Sullivan†

†Department of Computer Science & Engineering, FR-35
University of Washington
Seattle WA 98195 USA
{notkin,sullivan}@cs.washington.edu

‡School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213 USA
garlan@cs.cmu.edu

††Department of Computer Science & Engineering, 0114
University of California, San Diego
La Jolla, CA 92093-0114 USA
wgg@cs.ucsd.edu

Abstract

Implicit invocation based on event announcement is an increasingly important technique for integrating systems. However, the use of this technique has largely been confined to tool integration systems—in which tools exist as independent processes—and special-purpose languages—in which specialized forms of event broadcast are designed into the language from the start. This paper broadens the class of systems that can benefit from this approach by showing how to augment general-purpose programming languages with facilities for implicit invocation. We illustrate the approach in the context of three different languages, Ada, C++, and Common Lisp. The intent is to highlight the key design considerations that arise in extending such languages with implicit invocation.

1 Introduction

Systems have traditionally been constructed out of components, usually modules, that interact with each other by explicitly invoking procedures provided in their interfaces.

*This research was supported in part by the National Science Foundation under Grant Numbers CCR-9112880, CCR-9113367, CCR-8858804, and CCR-9211002, by DARPA Grant MDA 972-92-J-1002, by Siemens Corporate Research, and by SRA (Tokyo Japan). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government, of the Siemens Corporation, or of SRA.

There has recently been considerable interest in an alternative technique, variously referred to as implicit invocation, reactive integration, and selective broadcast. The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes the procedures that have registered interest in the event. Thus an event announcement "implicitly" causes the invocation of procedures in other components without the announcing component needing to know the name of those components.

The advantages of implicit invocation arise because of the separation of the invocation relationship from the "knows about" relationship between components. This makes it easier to add, modify, and integrate components without modifying many (if any) existing components. For example, since components need not explicitly name other components to invoke them it is possible to integrate a collection of components simply by registering their interest in events. Thus, the function of the overall system may be modified without changing any existing components: the new components are invoked based on already-existing event announcements. In contrast, in a system with only explicit invocation, invoking a new component requires that at least one existing component be modified.

Because of properties like these, many systems now use implicit invocation as a key means of composition. Although applications of the technique span many domains, these systems can be grouped into three categories.

The first category is tool integration frameworks. Systems in this category are typically configured as a collection of tools running as separate processes. Event broadcast is handled by a separate dispatcher process that communicates with the tools through communication channels provided by the host operating system (such as sockets in Unix). Examples include Field [Reiss 90], Forest [Garlan & Ilias 91], Softbench [Gerety 89], and several other commercial tool integration frameworks.

The second category is implicit invocation systems based on special-purpose languages and application frameworks. In these systems implicit invocation becomes accessible through specialized notations and run-time support. For example, many database systems now provide notations for defining active data triggers [Hewitt 69] to database applications. Examples include APPL/A [Sutton, Heimbigner & Osterweil 90], Gandalf's daemons [Habermann, Garlan & Notkin 91], AP5's relational constraints [Cohen 89], and "when-updated" methods of some object-oriented languages [Krasner & Pope 88]. Window systems like X [Scheifler and Gettys 86] and Garnet [Myers *et al.* 90] also exploit implicit invocation in a stylized manner. Other specialized applications that can be viewed as exploiting the paradigm include incremental attribute reevaluation, spreadsheet updating, and some blackboard systems [Garlan, Kaiser & Notkin 92].

Despite the successes of systems in these two categories, use of implicit invocation has been relatively limited. In particular, few applications can afford the overhead of separate processes used by tool integration frameworks, and special-purpose languages are limited by their very nature.

This motivates the third category, in which implicit invocation is incorporated into existing, general-purpose programming languages. A limited mechanism of this style is based on wrapper methods in the Common Lisp Object System (CLOS) [Steele 91]. At least two more general mechanisms in this category have been reported on in some depth, one for C++ [Sullivan & Notkin 92] and one for Ada [Garlan & Scott 93]. A third

approach has been developed for Common Lisp. In this paper, we compare and contrast the mechanisms employed by these three designs.

In all three categories, implicit invocation is intended to supplement, rather than supplant, explicit invocation. Components may interact either explicitly or implicitly, depending on which mechanism is most appropriate. This property makes it possible to view implicit invocation as a natural complement to an existing explicit invocation system, such as one provided by a standard module-oriented programming language.

This paper has two primary goals. The first is to help make implicit invocation more ubiquitous without further populating the world with special-purpose mechanisms. The second is to identify a variety of design issues that arise when embedding implicit invocation into modern programming languages: these issues are useful in clarifying various approaches to implicit invocation, as well as adding some insights into programming language design itself.

Implicit invocation mechanisms are based on two fundamental concepts. The first is that, in addition to defining procedures that may be invoked in the usual way, a component is permitted to announce *events*. The second is that a component may *register* to receive announced events. This is done by associating a procedure of that component with each event of interest. When one of those events is announced the implicit invocation mechanism is responsible for calling the procedures that have been registered with the event.

Although the basic mechanisms have substantial similarities, the details differ significantly because of the nature of the underlying languages. The paper first introduces the three mechanisms. Then we step back and consider what underlying design decisions were made. We focus on *why* different decisions were made in each language; in most cases programming language issues or "cultural style" caused particular decisions. The three languages—Ada, C++, and Common Lisp—have sufficiently different characteristics to highlight these issues. *How* the basic aspects of implicit invocation are achieved in three diverse programming languages is the central theme in this paper.

2 Ada

Ada is a statically-typed, module-oriented, imperative programming language [Ada 83]. In Ada the basic unit of modularization is the package. Packages have interfaces, which define (among other things) a set of exported procedures. To provide implicit invocation for Ada, we developed a small specification language to augment package interfaces. This language allows users to define events they want the system to support, and to specify which Ada procedures (in which package specifications) should be invoked on announcing the event. This design was strongly influenced both by typing and modularization features of the language and also by the desire to reuse existing Ada compilers and tools for language processing. The following code—which would be part of a complete Ada program—illustrates the declaration of events and the binding of procedures to events.

```
for Package_1
   declare Event_1
      X: Integer; Y: Package_N.My_Type;
   declare Event_2
   when Event_3 => Method_1 B
end for Package_1
for Package_2
   declare Event_3 A,B: Integer;
   when Event_2 => Method_4
   when Event_1 => Method_2 X
end for Package_2
for Package_3
   when Event_2 => Method_3
   when Event_1 => Method_4 Y
end for Package_3
```

In the specification language, `for` clauses identify the package under discussion. The `declare` clauses specify the events that this package will announce and the parameters associated with each event (if any). Each parameter has a type: this may be any legal Ada type. For example, `Package_1` declares two events. The first event, `Event_1`, has two parameters, `X` of type `Integer` and `Y` of type `My_Type`, as defined in `Package_N`.

The `when` clauses indicate which procedures in the package are to be invoked when an event is announced, and what event parameters are to be passed to the procedure. Any of the parameters may be listed and in any order. This list indicates which parameters are to be passed to each procedure. For instance, in the above code fragment, `Package_1` declares its interest in `Event_3`. When `Event_3` is announced (by `Package_2`), `Method_1` should be invoked, passing only the second parameter, `B`.

Before compiling the Ada program the user invokes a preprocessor that translates the specifications into an Ada package called `Event_Manager`. This package provides the run-time support for handling announced events; it is compiled and linked with the rest of the system. Although not illustrated in the example, the preprocessor assumes that the event specification statements are delimited by the special comment mark "`--!`" so that they can easily be separated from normal Ada code.

The generated interface of `Event_Manager`, illustrated next, provides the list of declared events as an Ada enumerated type, along with a record with a variant part that specifies the parameters for each event. In addition, the generated specification contains the signature of the `Announce_Event` procedure, which allows components to announce events.

```
with Package_N;
package Event_Manager is
  type Event is
    (Event_1,Event_2,Event_3);
  type Argument (The_Event: Event) is
    record
      case The_Event is
        when Event_1 =>
          Event_1_X: Integer;
          Event_1_Y: Package_N.My_Type;
        when Event_2 =>
          null;
        when Event_3 =>
          Event_3_A: Integer;
   Event_3_B: Integer;
        when others =>
          null;
      end case;
    end record;
  procedure Announce_Event(The_Data: Argument);
end Event_Manager;
```

When a component wishes to announce an event, it invokes `Announce_Event`, as follows:

```
Announce_Event(Argument'(Event_1, X_Arg, Y_Arg));
```

The generated body of `Event_Manager` contains the implementation of this procedure, which, as illustrated in the next code fragment, is structured as a `case` statement with one case for every declared event:

```
with Package_1;
with Package_2;
with Package_3;
package body Event_Manager is
procedure Announce_Event(The_Data: Argument) is
  begin
    case The_Data.The_Event is
      when Event_1 =>
        Package_2.Method_2(The_Data.Event_1_X);
        Package_3.Method_4(The_Data.Event_1_Y);
      when Event_2 =>
        Package_2.Method_4;
        Package_3.Method_3;
      when Event_3 =>
        Package_1.Method_1(The_Data.Event_3_B);
      when others =>
        null;
    end case;
  end Announce_Event;
end Event_Manager;
```

3 C++

C++ [Stroustrup 87] is an object-oriented language based on extensions to C. The implicit invocation mechanism added to C++ is statically-typed, with events as objects and with dynamic registration of methods with events. The design of the mechanism was influenced by the object-oriented nature of the language, by the typing structure of the language, and by the original objective of the mechanism, which was to support behavioral entity-relation designs and implementations [Sullivan & Notkin 92].

In the C++ implicit invocation mechanism, events are associated with the interfaces of classes, treating them as equivalents of methods. That is, in addition to exporting methods, classes declare and export a set of events. For example, the interface (encoded in a .h file) for a set class is shown next; a few parts of the actual text are omitted and slightly modified to focus on the important aspects of the example.

```
class Set ...
{
  public:

    Boolean Insert(Object& v);
    Boolean Remove(Object& v);
    stream& PrintOn(ostream& s);

    // PUBLIC EVENT INTERFACE
    VoidEventObject Inserted;
    VoidEventObject Removed;
};
```

This example exports three methods and two events, `Inserted` and `Removed`. As we will show, this allows other classes to register interest in either of these events.[1]

The registration of methods with events—that is, defining the methods to be invoked when an event is announced—is done separately from the event declarations. Suppose that an instance, `this` (the "sending" object in a C++ method invocation), of a class `SetBijectionMediator`, wants to register a method of a class, `UponS1Insert`, to be invoked when an object `s1` announces an event `Inserted`. This is done—usually by the constructor of the object that wants to register with the event—by executing the statement:

```
s1.Inserted.REGISTER(SetBijectionMediator,UponS1Insert);
```

Methods make event announcements by invoking an `Announce` method associated with each event. For example, the implementation of the `Insert` method in the body of the `Set` class includes the statement:

```
Inserted.Announce(v);
```

This causes the `Inserted` event on the set instance to be announced, which in turn causes all methods registered with that event to be invoked. There is a corresponding

[1] The `VoidEventObject` type arises because events are implemented as instances of event classes. `VoidEventObject` is an event class, and the `Inserted` and `Deleted` events are instances of that class. The name of the type is chosen by convention: `Void` implies that this event has no return object, `Event` implies that it is defining an event (all event classes have this piece), and `Object` implies that the parameter list of the events of this type include a single element of type `Object`.

`UNREGISTER` macro that allows an association between a method and an event to be broken.

The details of the C++ mechanism are affected by our desire to handle typing in C++ in a reasonably flexible and uniform way. In the C++ implicit invocation mechanism this is hard because the macros, events, and `Announce` methods have to be general-purpose and because we want to allow run-time registration.

The heart of the problem is that our event registration method cannot treat pointers to non-static member functions in a uniform manner, since they share no common type. This is because in C++ every non-static member function has an implicit first parameter, `this`, which is a pointer to its own object; thus non-static member functions from objects of different classes differ in their type signatures. This interferes with our desire to support static type checking.

Our solution is to declare a function type with a first parameter of type `void*` (a generic pointer), followed by the other parameters. Each client object declares a static member function of this type, which it registers along with a pointer to itself. The static member function is called with the self pointer, which it then type-casts to call the corresponding virtual member function. Passing the object's pointer, in this case, serves a function similar to passing a closure in Lisp.

The `STATIC` macro—typically used within class declarations in header files—makes it possible to register non-static member functions. The macro takes a non-static member function and creates a corresponding static member function. Reconsidering the use of the `REGISTER` macro above, we can see that the `UponS1Insert` method had to be static. But in fact it is a non-static member function declared privately to the SetBijectionMediator class:

```
void* UponS1Insert(Object& o);
```

It is transformed into an "equivalent" static member function by the use of the `STATIC` macro:

```
STATIC(SetBijectionMediator,void*,UponS1Insert,
                            (void* target,Object& o),(o));
```

The first argument indicates the class in which the method resides; that is, which class has the method that may be invoked by the announcement of an event in another class. The second argument indicates the return type of the method; the current mechanism requires this to be `void*`. The third argument is the name of the non-static member function. The fourth argument is the formal parameter list for the static method being created by the macro. And the final argument is the actual parameter list for the non-static method being transformed.

The `STATIC` macro creates a new name for the static member function. The `REGISTER` and `UNREGISTER` macros use the same name transformation to allow the user of the macros to name the non-static member function while actually registering and unregistering with the generated static member function.

4 Common Lisp/CLOS

Common Lisp, combined with the Common Lisp Object System (CLOS), is an object-oriented dynamically-typed language with substantial run-time flexibility. The implicit

invocation mechanism that extends these is a method-based mechanism with dynamic registration. The mechanism was designed to support the construction of a meaning-preserving tool for restructuring software systems [Griswold 91][Griswold & Notkin 93].

Like the Ada and C++ mechanisms, it has support for defining events, registering methods with events, and for announcing events. Common Lisp's flexible nature strongly affected the design of the mechanism. In particular, its dynamic typing, first-class functions, and simple extensible syntax led to a straightforward design and clean syntax.

A `defevent` declaration is used to extend a standard CLOS class definition to allow an object to announce an event. For example, the declarations

```
(defclass world-element ()
  ((value :accessor value :initarg :value :initform nil)))

(defevent delete-object ((obj world-element) &optional reason))
```

create a class `world-element` with a single slot (instance variable) `value`, and then adds an event `delete-object`, which takes a value `obj` of type `world-element` and an optional argument `reason` that defaults to NIL. The first value of an event must be of the type of the class to which the event belongs.[2] The syntax of the `defevent` is similar to that of a method definition, but without a body since the actions of the event are (dynamically) defined by the registrant, not the event announcer. Subclasses of `world-element` inherit the event `delete-object`, and the inherited event may be redefined as long as the event interfaces are consistent according to the method definition rules of CLOS.

Registration for an event is performed using an `upon` expression (a macro) that creates a structure to wait for a particular event from a particular object. When the object announces the event, the body of the `upon` is executed. For example, executing the expression

```
(upon ((delete-object the-obj reason) registering-object)      ; 1
  (remhash the-obj object-table)                               ; 2
  (format t "object ~s deleted because ~s~%" the-obj reason))  ; 3
```

creates an obligation for the run-time system to execute the body (lines 2 and 3) of the `upon` when the value of `the-obj` announces the `delete-object` event. Only the first argument of the `upon`'s event pattern is used in matching an announced event; the remaining arguments of the event pattern are merely bound with the remaining values of the event announcement to allow accessing them in the body of the `upon`.

An `announce` expression, when executed, announces that an event has occurred. The expression

```
(announce (delete-object obj "no neighbors"))
```

announces the event `delete-object` with the argument, `obj` (the announcer of the event) with the optional argument provided. If the value of `obj` here and `the-obj` in the `upon` were the same, this announcement would cause the execution of the `upon`, causing the value of `the-obj` to be removed from `object-table` and an explanatory message to be printed.

[2] CLOS supports multimethod selection, which allows selecting a method to execute based on the type (or value) of any or all the arguments passed to the call, rather than just the first argument, as in C++. Consequently, the first-object convention of event announcement introduces a slight asymmetry with respect to method call.

A registration performed with `upon` can be unregistered using `downoff`, indicating the registrant (`registering-object` in following fragment):

```
(downoff the-obj 'delete-object registering-object)
```

The event mechanism is implemented in two parts. First, the `defevent` declaration extends the class definition of the announcing object with a new slot (instance variable) so it can keep a list of functions that are to be invoked when the event is announced. This extension is performed using the CLOS metaobject protocol. At the same time, a method with the same signature as the event is created for invoking the registered functions. This method is called by the `announce` expression. For example, the definition of the `delete-object` event causes a `delete-object` slot to be added to the `world-element` class, and defines a method `delete-object` that calls all the functions in the `delete-object` slot of the object announcing the `delete-object` event. Second, the `upon` expression creates an anonymous function (lambda) for the registering object and registers it with the (potentially) announcing object. The signature of the function is the signature of the event it is waiting on, and its body is the body of the `upon`. For example, the `upon` expression above creates a function that is put on the `delete-object` list of `the-obj`. The function takes two arguments named `the-obj` and `reason`, and the body contains the `remhash` and `format` expressions. Consequently, when `delete-object` is later announced by `the-obj`, the `delete-object` method is called on it and it calls the functions on the list—including the one registered by this `upon`—with the arguments of the announcement.

5 Key Design Questions

These three implementations of implicit invocation in the context of existing programming languages embody sets of design choices that are important to understand, both to see how to use an implicit invocation system, and to observe the limitations of the implementations. The design decisions can be grouped into the following six categories,

- event definition,
- event parameters,
- event bindings,
- event announcement,
- delivery policy, and
- concurrency,

which we now examine in turn.

5.1 Event Definition

The first design issue concerns how events are to be defined. There are several related issues. Is the vocabulary of events extensible? If so, are events explicitly declared? If events are declared, where are they declared? There are four approaches to event extensibility and declaration.

Fixed Event Vocabulary A fixed set of events is built into the implicit invocation system: the user is not allowed to declare new events.

Static Event Declaration The user can introduce new events, but this set is fixed at compile-time.

Dynamic Event Declaration New events can be declared dynamically at run-time, and thus there is no fixed set of events.

No Event Declarations Events are not declared at all; any component can announce arbitrary events.

An example of a system with a fixed event vocabulary is Smalltalk-80 [Goldberg & Robson 83], which provides a small number of events including the `changed` event.[3] Active databases often have a fixed event vocabulary, where events are associated with primitive database operations, such as inserting, removing, or replacing an element in the database. APPL/A is an example of this approach [Sutton, Heimbigner & Osterweil 90].

At the other extreme, tool integration frameworks, such as Field and Softbench, have no explicit event declarations at all. A tool can announce an arbitrary string, although tool builders typically describe the event vocabulary of each tool as externally documented conventions.

For all three mechanisms, we rejected the first alternative as too restrictive. This is because we are embedding largely general-purpose event mechanisms in programming languages: in situations where an implicit invocation mechanism is specially designed for a given task, using a fixed event vocabulary might be a more reasonable decision. All three mechanisms also rejected the fourth alternative as too unpredictable. When it came to selecting among the other approaches, there were arguments on each side. There is no high-level reason that prohibits any of the approaches to be implemented in any of the three mechanisms.

In the Ada mechanism, static event declaration can be implemented efficiently as an Ada enumerated type, also allowing compile-time type checking of event declarations and uses. On the other hand, dynamic event declarations provide more flexibility, since they allow run-time reconfiguration. Thus, a dynamic event system may have an added benefit of reducing recompilation overhead.

In the end, predictability through static checking won out for the Ada mechanism. In particular, we felt that static interface declarations more naturally meshed with the spirit of Ada, led to more comprehensible programs, and better supported large-scale system development, which requires predictable behavior of the components.

In the C++ mechanism, we selected a combination of static and dynamic event declaration. In particular, the event classes are defined statically but instances of the event classes—that is, events themselves—can be defined dynamically. If a new set object is instantiated, for example, it includes new instances of the associated event objects.

The Common Lisp mechanism, in the spirit of run-time flexibility encouraged by the language, allows for dynamic event declaration. That is, creating `defevent` forms at run-time is a relatively straightforward activity.

[3] By convention, this "event" is announced by invoking the `changed` method on `self`. This causes the `update` method to be invoked on each dependent of the changed object. Other events could similarly be introduced by new methods that had a similar effect, but this is generally not done.

Where should the declarations of events reside? In particular, since the events represent information shared between (at least) the announcing component and the event system, it is unclear which component "owns" an event, and thus where events should be declared. There are two obvious choices:

Central Declaration of Events Events are declared at a central point and then used throughout the system.

Distributed Declaration of Events Events are declared by each module (or class), where each module declares the events it expects to announce.

The Ada implementation is neutral on this issue. Since the declarations are embedded within Ada comments, it is possible to declare events in the individual packages. However, an implementor can also place event declarations in a separate file. One drawback to this implementation, however, is that regardless of where the events are declared, they are actually compiled into a central locus of control (the `Event_Manager` package specification). When changes are made to event declarations, all files containing event declarations must be recompiled to correctly rebuild the event manager.

The C++ mechanism requires that events be associated with classes. The primary motivation was that methods and events were to be treated as equals and duals of one another; thus, events should be declared in the interfaces of classes, as are methods. (The equivalent of a centralized event declaration could be defined by convention, designating a special class that holds all the event declarations.)

Note that the Common Lisp mechanism, even with dynamic event declarations, creates a bias with respect to the placement of declarations. In particular, events—like methods in Common Lisp—can be placed anywhere, although they are most naturally placed with their class definition. So, in the same sense as the Ada mechanism, the Common Lisp mechanism is neutral with respect to where the event declarations go, but the practical tendency has been to associate them with class definitions.

5.2 Event Parameters

The next design issue is how events should handle parameters. The question here is, what forms of data passing using event announcements make them easy to use and understand? The choices we considered were:

Simple Names Events are simple names without any parameter information. Data associated with an event (if any) would have to be encoded in the name of the event or passed in global variables.

Fixed Parameter Lists All events have a name and the same fixed list of parameters.

Parameters by Event Type Each event has a fixed list of parameters, but the type and number of parameters can be different for different events.

Parameters by Announcement Whenever a component announces an event, it can specify any list of parameters. For example, the same event name could be announced with no parameters one time and with ten parameters the next.

The use of simple names is found in systems that use events as a kind of interrupt mechanism. In these systems there is typically only a small number of causes for events to be raised. Fixed parameter lists are often used in combination with a fixed set of system-defined events. For example, in an active database events might be required to pass the identity of the data that is being modified. At the other extreme, systems that use strings as events often allow arbitrary parameters: it becomes the job of the receiver to decode the string and extract parameters at run-time.

We considered the first two approaches as being unnecessarily restrictive. We also felt that letting parameters vary for each announcement could lead to undisciplined and unpredictable systems. This led to the decision that the Ada and C++ mechanisms should allow parameters to vary by event type. Allowing parameters to vary by event type over a static list of events also solves a problem of parameter passing in the Ada mechanism: with static events and static parameter lists, a record with a variant part becomes a natural way to represent parameters. The C++ mechanism handles this through the `STATIC` macros.

Technically, the Common Lisp mechanism allows parameters to vary by announcement, since source files can be loaded during execution. However, the actual use of the mechanism is more in the style of the Ada and C++ mechanisms, since an `upon` expression and an announcement that it is intended to match must agree on the parameter list.

5.3 Event Bindings

Event bindings determine which procedures in which modules will be called when an event is announced. There are two important questions to resolve. First, when are events bound to the procedures? Second, how are the parameters of the event passed to these procedures? Also central to this issue is the granularity of the bindings.

With respect to the first issue, we considered two approaches to event binding:

Static Event Bindings Events are bound to procedures statically when a program is compiled.

Dynamic Event Bindings Event bindings can be created dynamically. Components register for events at run-time when they wish to receive them, and unregister for events when they are no longer interested.

The Ada mechanism uses static event bindings, while the C++ and Common Lisp mechanisms use dynamic event bindings.

The decision to use static event bindings for Ada was largely forced by the language itself. In particular, Ada provides no convenient mechanism for keeping a "pointer" or other reference to a subprogram. It would have been possible to provide an enumerated type representing all procedures that might be bound to any event. Events could be bound to elements of this enumerated type dynamically. Procedures would then be invoked through a large case statement. However, this conflicted with the desire to have a flexible parameter passing mechanism (as described earlier), since the parameters would either have had to be fixed or encoded in the enumerated type.

The C++ and Common Lisp mechanisms allow `REGISTER` and `upon` statements, respectively, to be executed as part of the basic execution flow, thus adding event-method pairs on-the-fly. In the C++ mechanism we used this feature quite extensively.

For example, in one situation we wanted to keep two sets consistent. To do this we defined a separate component, called a "mediator," which managed consistency of the sets by listening to insert and delete events announced by each set. If an element in one of the sets is modified (rather than inserted or deleted) then modification on its associated element in the other set must be performed. To handle these modifications, whenever we created a new pair of associated elements, one in each set, we also deployed a "submediator" between the elements. The submediator constructor and destructor methods automatically register and unregister with the events of the corresponding elements (which are passed as parameters to the submediator constructor), making it easy for the associated elements to modify one another. With static event binding, this would be impossible since the elements themselves were not even in existence at compile-time.

Despite this use of dynamic event binding, the jury is still out on which approach is better. It may be that the increased flexibility of dynamic event binding is outweighed by the decrease in the predictability of the resulting system. In particular, the behavior of a system is less apparent from its static representation (that is, from the program text) if dynamic event binding is used. Moreover, dynamic event bindings may, depending on how they are realized, introduce race conditions at run-time (see the later subsection on Concurrency).

The three mechanisms represent two different binding granularities. In the Ada mechanism, bindings are between packages (modules); in the C++ and the Common Lisp mechanisms, the bindings are between objects. These decisions are consistent with the nature of the languages (for instance, Ada does not have procedure pointers but C++ and Common Lisp do).

Another design decision is how the parameters from the event are translated into the parameters for the invocation. The choices we considered were:

All Parameters An invocation of a method due to an event announcement passes exactly the same parameters (in number, type and order) as are specified for the event.

Selectable Parameters As part of the event binding, the implementor can specify which parameters of the event are passed in the invocation, and in which order.

Parameter Expressions The invocation passes the results of expressions computed over the parameters of the announced event.

The transmission of all parameters to each procedure bound to an event requires some conspiracy between the designer of the procedure to be invoked and the designer of the events. (Just as is needed between a procedure invocation and a procedure definition for explicit invocation.) We can easily imagine situations in which only some of the information in an event announcement would be useful to a component. It seemed unnecessary to require the component to accept a dummy parameter just for that reason, or, conversely, to require two events to be announced—one with and one without the unneeded data.

In the Ada mechanism, we opted to provide selectable parameters, as this provided a balance between flexibility and ease of implementation. Selectable parameters allows more freedom in matching events to procedures, thereby promoting reusability. Moreover, it is straightforward to build the argument list from the event binding declaration.

We believe that allowing non-side-affecting expressions as parameters to an event system could provide a significant and useful amount of increased flexibility. Sometimes a procedure's parameters do not match those of an event, but some of the procedure's parameters can be made constant to "customize" the procedure invocation to the context of the event. With the ability to construct expressions as part of an event binding, it becomes easier to tailor a procedure to an event without modifying either the announcer or the recipient. The implementation becomes considerably more complex, however. In particular, it is necessary to make sure that operators used in parameter expressions are in scope and have the right type.

In the C++ mechanism, we used the "all parameters" approach. Intermediary components can be interposed to get the same effect as selectable parameters. This simplifies the event mechanism itself, but may complicate the resulting systems built using the mechanism, since there may be more components.

The Common Lisp mechanism technically falls into the "all parameters" category as well. However, since a function gets created each time an `upon` is executed, it has the flexibility of "parameter expressions."

These descriptions point out the central question, which is still open: how should the potential proliferation of components and component interfaces be handled? In explicit invocation, the calling component is usually responsible for understanding all the parameters that the callee component requires; if some are not important to the caller, defaults must be provided. (Some languages, like Ada, allow defaults to be handled by language mechanisms rather than by programming conventions.) In implicit invocation, the same approach can be taken (as we did for C++), with the pressure on the receivers of events rather than on the announcers of events. The Common Lisp mechanism approaches the proliferation problem in a different way. By creating an anonymous function each time an `upon` is executed, the mechanism helps keep the name space trim.

Intermediate approaches, such as selectable parameters, show great potential. Field's mechanism includes a relatively powerful selectable parameter mechanism based on the string scanning functions common to Unix.

5.4 Event Announcement

Although announcing an event is a straightforward concept, there are several ways in which it can be incorporated.

Single Announcement Procedure Provide a single procedure for announcing any event. A closely related variant of this approach is a "language extension" in which an `announce` statement is introduced as a new kind of primitive, with a language preprocessor used to conceal the actual implementation.

Multiple Announcement Procedures Provide one procedure for announcing each event name. For example, a component might call `Announce_Changed` to announce the `Changed` event. The procedure accepts exactly the same parameters (in number, type, order, and name) as the event.

Implicit Announcement Permit events to be announced as a side effect of calling a given procedure. For example, each time procedure `Proc` is invoked, announce event `Event`.

The Ada and Common Lisp mechanisms selected the "single announcement" approach for a number of reasons. First, in comparison to the multiple procedure approach, it is transparent: all event announcements look similar. Second, our users were fairly proficient with the language at hand, and we wanted to stay as close to "pure" Ada and Common Lisp as possible. This discouraged us from modifying the language. We also wanted to avoid the extra complexity of a preprocessor that would have to process the full Ada or Common Lisp language (and not just specially delimited annotations).

In the Common Lisp mechanism, event announcements are implemented as simple method calls. The announce syntax was added because it is confusing to read Common Lisp code with no knowledge of what is a method call and what is an event announcement.

In the Ada mechanism, we realized that instead of requiring the user to construct an `Event_Manager.Argument` record as a local variable and pass the variable to the procedure, the user could simply pass a record aggregate containing the desired information. This brought the syntax close enough to an `announce` statement to satisfy our desire for promoting events as first-class, without requiring any modification to Ada syntax.

In the C++ mechanism, multiple announcement procedures are used. Specifically, one announcement procedure is defined for each event class. The "similar look" is still achieved by defining special template macros to define these event classes. Thus, this mechanism makes event announcements look different from, but not too different from, method invocations. In C++, the language extension approach was even less attractive, since adding another preprocessor stage would complicate language processing and debugging.

The fourth approach, implicit announcement, has been used as a triggering mechanism for databases [Dayal, Hsu & Ladin 90], for some programming environments [Habermann, Garlan & Notkin 91], and for some language extensions (in particular CLOS's wrapper methods). It is attractive because it permits events to be announced without changing the module that is causing the announcement to happen. Although we could have additionally supported this form of announcement, we chose not to, largely because it would require a preprocessor to transform procedures so that they announce the relevant events.

5.5 Delivery Policy

In most implicit invocation mechanisms, when an event is announced all procedures bound to it are invoked. However, in some mechanisms this is not guaranteed. The design options include:

Full Delivery An announced event causes invocation of all procedures bound to it.

Single Delivery An event is handled by only one of a set of event handlers. For example, this allows such events as "File Ready for Printing" to be announced, with the first free print server receiving the event.

Parameter-Based Selection This approach uses the event announcement's parameters to decide whether a specific invocation should be performed. This is similar to the pattern matching features of Field in that a single event can cause differing sets of subprograms to be invoked depending upon exactly what data is transferred with the event.

State-based Policy Some systems (notably Forest [Garlan & Ilias 91]), associate a "policy" with each event binding. Given an event of interest, the policy determines the actual effect of it. In particular, the policy can choose to ignore the event, generate new events, or call an appropriate procedure. Policies can provide much of the power of a dynamic system without incurring the complexities of a dynamic system.

The single delivery model did not match our general interest in supporting implicit invocation as opposed to indirect explicit invocation,[4] and so was not used in any of the mechanisms. In fact, all three mechanisms use variations of the full delivery model. The only "twist" is that in the C++ and in the Common Lisp mechanisms, which both permit dynamic registration, a given method can be associated with a given event multiple times: that is, the event-method "relation" isn't really a relation because it can have duplicate entries. The implication is that when an event is announced, all *instances* of the methods registered with the event are invoked in these two mechanisms.

It is still unclear what are the benefits of each delivery mechanism. We are beginning to believe that there is a tension between the delivery policy and the independence of the methods that are registered in the same event. In particular, if the methods are independent (that is, whether they are executed in a particular order, or are even executed sequentially is immaterial to the resulting system state), then a full delivery policy seems to lead to the most straightforward analysis by the users. In the face of increasing dependence among the registered methods, it may be that alternative policies or variations on policies are more appropriate. Further, if there is extensive dependence among the registered methods, it is likely that using implicit invocation is inappropriate, and that explicit invocation is more appropriate.

A central question yet to be settled in its entirety is where the desired delivery policies can and should be specified. In language-based approaches, such as the ones described here, placing them in the mechanism itself is important. But more complicated policies may be more effectively placed in the receiving components or in intermediaries situated between the announcing and receiving components.

5.6 Concurrency

The only one of our three languages that has built-in support for concurrency is Ada. (Most Common Lisp implementations have a concurrency mechanism, but there is apparently no standard as yet.) Thus, for Ada's mechanism, we had to consider further whether to or how to handle concurrency. In the Ada design we considered three options.

Package A component is a package, and an invocation is a call on a procedure in the package interface.

Packaged Task A component is a statically-defined task with an interface in a package specification, and an invocation is a call on an entry in the task interface.

Free Task A component is a dynamic task (created, for example, by a task manager). An invocation is a call on an entry in the task interface. However, rather than providing an enclosing package, the task is built inside the `Event_Manager` package.

[4] Indirect explicit invocation occurs when passing procedures as parameters or when calling virtual methods in object-oriented languages. In these cases, the specific procedure body that is to be invoked is not known by the invoking procedure, but the fact that such a procedure exists *is* known.

The first choice leads to a non-concurrent system: events are executed using a single thread of control. The second and third choices would permit concurrent handling of events. Although we do not forbid tasks inside of packages, our implementation adopts the first approach. That is, events are used at the module level, not the task level, and thus the programmer—not the mechanism—is responsible for handling any interactions between the event mechanism and tasks.

Our decision was based primarily on the fact that, given the current understanding of systems built with implicit invocation, it is much easier to develop correct systems using a single thread of control. The central problem is that when you add concurrency, events may return before the actions they initiate are completed, which requires more complex reasoning about the system.

In addition, there are a set of problems in implementing concurrent implicit invocation mechanisms that we could avoid addressing due to this decision. For example, if we had adopted a concurrent approach, it would have either been necessary to require all recipients of an event to be re-entrant, or for the `Event_Manager` task to provide its own internal synchronizing task to ensure that invocations occurred only one at a time. Should a receiving task have attempted to announce another event while in its rendezvous, this could cause a deadlock. Another question might be whether each event spawns a task and returns immediately, or whether they run to completion before returning the thread.

APPL/A is a system that uses an Ada-based event mechanism along with concurrency. Event announcement methods are queued at the recipient. We do not fully understand the consequences of this decision with respect to the use of such an event mechanism.

6 Evaluation

Each of these implicit invocation mechanisms have been fully implemented and have been used in varying degrees.[5]

6.1 Ada

The Ada mechanism was initially developed for use in a masters-level software engineering course [Garlan *et al.* 92]. The students had an average of five years of industrial experience. Most were familiar with Ada. This early use of the system has resulted in both praise and criticism.

On the positive side, users of the system have had virtually no conceptual problems transferring their abstract understanding of implicit invocation to the use of our implementation. The declarative nature of events apparently fit well with their abstract model. In addition, experienced Ada users found little difficulty adapting their programs to an implicit invocation style. Our attempts to remain close to Ada syntax certainly contributed to this.

On the negative side, there appeared to be two limitations. The first was the common problem of debugging preprocessed source code. Since compiler errors are produced with respect to the preprocessed source, users have to translate between the output of the preprocessor and their initial source input. However, this problem was mitigated by

[5] Each of the implementations is available by sending electronic mail to the authors.

the relative orthogonality of the language extensions since the event-oriented extensions are largely isolated from normal code. The second was the absence of dynamic event declaration and binding. Although Ada programmers are used to strongly typed, static system designs, our users were also aware that other implicit invocation systems are more dynamic. (For example, some of them had used Softbench.)

To these drawbacks we would add our own concern with the lack of concurrency supported by our design. As indicated earlier, we believe that it should be possible to exploit the tasking model of Ada, and see this as an opportunity for future work.

6.2 C++

The C++ mechanism was initially developed to support our work in tool integration, with a specific interest in reducing the cost of evolution for integrated environments. We have used the mechanism (and its predecessor) extensively, for efforts in constructing parallel programming environments, computer-aided geometric design environments, etc.

On the whole, programmers seem relatively happy using the mechanism. There are two high-level issues that arise. The first is that the particular style of use that we encourage is non-trivial to learn. We have anecdotal feedback from outside our group, however, that once the approach is learned people tend to produce "cleaner" designs. The second issue is that the typing structure takes some time to learn and understand. This is not a fundamental problem, but it does imply that further work is needed in understanding the relationship between implicit invocation and static typing.

The C++ mechanism has not been used in concurrent applications. Thus we have been able to avoid a collection of issues that will, at some point, be critical. The same is true for issues related to distribution.

6.3 Common Lisp

The Common Lisp mechanism has been used more narrowly than the other two mechanisms. In particular, it has been used almost exclusively to aid in the design and implementation of a program restructuring tool. (One other use has been to implement a facility for tracing events.)

In the restructuring tool, the event mechanism integrates three independently developed subsystems—the core restructuring component, the underlying CFG/PDG representations [Larus 89], and the window system [Rowe *et al.* 91]. To integrate their behavior, intermediary "mediator" components are created and receive events from one subsystem and translate them to direct invocations on another [Sullivan & Notkin 92]. For example, when a program restructuring transformation completes, an announcement of the change is made and all registered restructuring-window mediators are notified of the change, which in turn will update windows containing text or graphics of the program. The separation of these components aids evolution of the system and maintains independence.

Based on this experience, we have noted that programmers who have never used events before usually have some trouble understanding how an `upon` declaration interacts with event announcements. However, we perceive this as part of the natural learning curve associated with implicit invocation, not with this particular mechanism. Another problem is that programmers do not immediately understand that registered objects that are

otherwise unreferenceable will continue to receive event announcements unless a `downoff` is executed or the announcing objects themselves become unreferenceable. This problem is a consequence of the fact that Common Lisp supports automatic garbage collection, but event registration references prevent garbage collection of otherwise dead objects. Although this problem can be solved, it is non-trivial. An event tracing mechanism built into the event mechanism has successfully countered these problems by assisting the programmers in understanding the links between components and the consequent behavior.

7 Related work

A large number of systems have adopted implicit invocation as an integration mechanism. As discussed earlier, most of these tend to fall into the categories of process-oriented tool invocation mechanisms and special-purpose languages. Here we have attempted to broaden the base of applicability by showing how to provide similar functions for standard programming languages.

This work is strongly motivated by others' research, which has demonstrated that implicit invocation is an important new integration mechanism. In particular, Field showed how implicit invocation could be applied to tool integration. Follow-on systems, such as Softbench, have elaborated this style of broadcast-based integration in commercial environments.

Another use of implicit invocation is in the context of some object-oriented systems. One of these is the change propagation mechanism used to support the Model/View/Controller paradigm in Smalltalk-80 [Krasner & Pope 88]. In this system, any object can register as a dependent of another object. When an object "announces" a *changed* event, an *update* method is called on each of the dependent objects. While this use of implicit invocation is limited by the fixed nature of the mechanism (i.e. the events and methods are wired into the Smalltalk environment design), the approach creates problems in conjunction with class inheritance.

We see two significant disadvantages to that approach. First, it forces the event announcer to be aware of the mechanism by which events are being handled. For example, change announcement is actually done by the procedure call "self changed". An alternative would be to perform the announcement on some external entity, as in "dispatcher announce ...": both suffer from the same problem that the announcer must think of the announcement as a procedure call on a specific entity. But a second, and more serious problem, is that one might prefer to think of the events that are to be announced by an object as being part of its interface. Just as procedures determine the functionality of a module (or class) in traditional systems, so, too, are events an integral part of that module's functionality.

CLOS has its own language features for implicit invocation. Specifically, it has a wrapper method mechanism that can be used as an event mechanism. However, it is not as flexible as the mechanism introduced here, which is what led us to introduce the new Common Lisp mechanism. In CLOS, one or more wrapper methods can be defined to augment the behavior of a normal method by executing before, after, or "around" it. When a method is executed, some of its wrapper methods may also be executed, which can be interpreted as the wrapper methods "receiving" the announcement of the call of the main method. Whether one is executed is based on how the types of the arguments

match the type signature of the wrapper method, which can deviate from that of the main method by being a sub- or super-class type. However, the selection mechanism can be overridden.

CLOS's mechanism can be characterized as module-based, with implicit declaration of events when methods are defined, and with static event bindings (i.e., declaration of an wrapper method). The delivery policy is based on a match of the wrapper method's type signature with that of the call's arguments (roughly). Event structure is synonymous with method structure, which could be interpreted as "parameters by event type." The parameter passing mode is a combination of passing all parameters and computing parameter expressions.

A primary focus of this paper is better understanding of the design space associated with implicit invocation mechanisms. In that regard it is related to work in formalizing implicit invocation models [Garlan & Notkin 91]. Such efforts are complementary: a formal model makes clear what are the fundamental abstractions necessary to understand implicit invocation, while our concrete application relates these abstractions to the constraints imposed by the real world.

Finally, this work is related to other uses of language extension as a means for enhancing the expressiveness of existing programming languages. One example of language extension is Anna, which augments Ada with specifications [Luckham and won Henke 85]. The primary difference between that kind of work and ours is that we are attempting to change the fundamental mechanisms of interaction in module-oriented languages. That is to say, events are not just additional annotations to permit some tool to perform additional checks, but become an essential part of the computational model for the modules that use them.

8 Conclusion

The contributions of this work are twofold. First, we have shown by example how to add implicit invocation to three quite different programming languages. Some of the design decisions were constrained by the specifics of the languages themselves; but most are more general, based on constraints that are similar to those found in other programming languages (for example, static typing). Therefore, many of our lessons could directly apply to the definition of implicit invocation mechanisms in other programming languages. Second, we have elaborated the design space for this approach and shown how the decisions in this space are affected by the constraints of the programming language that is being enhanced. Ultimately this is the most important thing, since it serves as a checklist for those attempting to apply these techniques to other languages.

References

[Ada 83] Reference Manual for the Ada Programming Language. United States Department of Defense (January 1983).

[Cohen 89] D. Cohen. Compiling Complex Transition Database Triggers. *Proceedings of the 1989 ACM SIGMOD* (1989).

[Dayal, Hsu & Ladin 90] U. Dayal, M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proceedings of the 1990 ACM SIGMOD* (June 1990).

[Garlan & Ilias 91] D. Garlan and E. Ilias. Low-cost, Adaptable Tool Integration Policies for Integrated Environments. *Proceedings of SIGSOFT '90: Fourth Symposium on Software Development Environments.* Irvine, CA (December 1990).

[Garlan, Kaiser & Notkin 92] D. Garlan, G.E. Kaiser, and D. Notkin. Using Tool Abstraction to Compose Systems. *IEEE Computer* (June 1992).

[Garlan & Notkin 91] D. Garlan and D. Notkin. Formalizing Design Spaces: Implicit Invocation Mechanisms. *Proceedings of VDM'91: Formal Software Development Methods.* Springer-Verlag, LNCS 551 (October, 1991).

[Garlan & Scott 93] D. Garlan and C. Scott. Adding Implicit Invocation to Traditional Programming Languages *Proceedings of the 15th International Conference on Software Engineering.* IEEE Computer Society Press, pp. 447–455 (May 1993).

[Garlan *et al.* 92] D. Garlan, M. Shaw, C. Okasaki, C. Scott, and R. Swonger. Experience with a Course on Architectures for Software Systems. *Proceedings of the SEI Conference on Software Engineering Education* (October 1992).

[Gerety 89] C. Gerety. HP SoftBench: A New Generation of Software Development Tools. Technical Report SESD-89-25, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado (November 1989).

[Goldberg & Robson 83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley (1983).

[Griswold 91] W.G. Griswold. *Program Restructuring as an Aid to Software Maintenance.* Department of Computer Science & Engineering, University of Washington (1991).

[Griswold & Notkin 93] W.G. Griswold and D. Notkin. Automated Assistance for Program Restructuring. To appear, *ACM Transactions on Software Engineering and Methodology* (July 1993).

[Habermann, Garlan & Notkin 91] A.N. Habermann, D. Garlan and D. Notkin. Generation of Integrated Task-Specific Software Environments. In *CMU Computer Science: A 25th Commemorative.* ACM Press (1990).

[Hewitt 69] C. Hewitt. PLANNER: A Language for Proving Theorems in Robots. *Proceedings of the First International Joint Conference in Artificial Intelligence.*, Washington DC (1969).

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

[Larus 89] J.R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* UC Berkeley Computer Science (May 1989).

[Luckham and won Henke 85] D. Luckham and F.W. von Henke. An Overview of Anna, a Specification Language for Ada. *IEEE Software* (March 1985).

[Myers *et al.* 90] B.A. Myers, D.A. Giuse, R.B. Dannenberg, B. Vander Zanden, D.S. Kosbie. E. Pervin, A. Mickish, and P. Marchal. Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer 23*,11, pp. 71–85 (November 1990).

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

[Rowe *et al.* 91] L.A. Rowe, J.A. Konstan, B.C. Smith, S. Seitz, and C. Li. The PICASSO Application Framework. *Proceedings of the 14th ACM Symposium on User Interface Software and Technology* (1991).

[Scheifler and Gettys 86] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics 5*,2, pp. 79–109 (April 1986).

[Steele 91] G.L. Steele. *COMMON LISP, the Language,* 2nd edition. Digital Press, Burlington MA (1991).

[Stroustrup 87] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA (1987).

[Sullivan & Notkin 92] K.J. Sullivan and D. Notkin. Reconciling Environment Integration and Software Evolution. *ACM Transactions on Software Engineering and Methodology 1*,3 (July 1992).

[Sutton, Heimbigner & Osterweil 90] S.M. Sutton, Jr., D. Heimbigner, & L.J. Osterweil. Language Constructs for Managing Change in Process-Centered Environments. *Proceedings of ACM SIGSOFT90: Fourth Symposium on Software Development Environments*, pp. 206–217 (December 1990).