Reactive Integration for Traditional Programming Languages

David Garlan
Curtis Scott

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15214

December, 1992
CMU-CS-92-217

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A version of this paper to appear as "Adding Implicit Invocation to Traditional Programming Languages" in *Proceedings of the Fifteenth International Conference on Software Engineering* (May 1993).

This research was sponsored by the National Science Foundation under Grant Number CCR-9112880, by DARPA Grant MDA 972-92-J-1002, and by a grant from Siemens Corporate Research. 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 or of Siemens.

Keywords: Reactive Integration, Implicit Invocation, Event Systems, Ada, Software Architecture, Language Design

Abstract

Reactive integration based on event broadcast is an increasingly important technique for developing 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 the Ada language, and highlight the important design considerations that arise in extending such languages with facilities for reactive integration.

1 Introduction

Systems have traditionally been constructed out of modules that interact with each other by explicitly invoking procedures provided in their interfaces. However, recently there has been considerable interest in an alternative integration 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 module can announce (or broadcast) one or more events. Other modules 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 all of the procedures that have been registered for the event. Thus an event announcement "implicitly" causes the invocation of procedures in other modules.

There are numerous advantages to implicit invocation. One important benefit is that it provides strong support for reuse. Since modules need not explicitly name other modules it is possible to integrate a collection of modules simply by registering their interest in the events of the system. A second important benefit is that it eases system evolution [Sullivan 92]. New modules may be added to an existing system by registering their interest in events. Similarly, one module may be replaced another without affecting the interfaces of modules that implicitly depend on it. In contrast, in a system based on explicit invocation, whenever the identity of a module that provides some system function is changed, all other modules that import that module must also be changed.

Because of these desirable properties many systems now use implicit invocation as their primary means of composition. While applications of the technique span many application domains, these systems can be broadly grouped into two 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, Forest, Softbench and several other commercial tool integration frameworks [Reiss 90, Garlan & Ilias 91, Gerety 89].

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 to database applications [Hewitt 69]. Examples include APPL/A for Arcadia, daemons for Gandalf, relational constraints for AP5, and "when-updated" methods of some object oriented languages [Sutton, Heimbigner & Osterweil 90, Habermann 91, Krasner & Pope 88, Cohen 89]. Other specialized applications that can be viewed as exploiting the paradigm include incremental attribute reevaluation, spreadsheet updating, and some blackboard systems [Garlan *et al.* 92a].

However, despite the successes of systems in these two categories, and despite the fact that the techniques are generally applicable to any modularizable system, widespread 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.

In this paper we show how to make implicit invocation more broadly available to the software engineering community through the incorporation of implicit invocation in existing, general-purpose programming languages. The technique is quite simple: module interfaces of a procedure-oriented language are annotated to permit event declarations, announcements, and event-procedure bindings. The annotations are then preprocessed and compiled using traditional techniques. Dispatching of events is handled by a system-generated module, transparently to other modules, which can simply announce events as part of their normal code.

We begin by outlining the basic mechanism and illustrate its use in the context of the Ada programming language. While the ideas are straightforward, as we will see, attempts to add implicit invocation to standard languages raise a number of design decisions that can have a significant impact on the properties of the mechanism and on its usability. This paper highlights these design considerations so that any similar attempt to add implicit invocation to a strongly-typed, procedure-oriented programming language can benefit from this work.

2 Adding Implicit Invocation to Ada

While there are many ways to implement an implicit invocation mechanism, all are based on two fundamental concepts. The first is that in addition to defining procedures that may be invoked in the usual way, a module is permitted to announce *events*. The second is that a module may *register* to receive announced events. This is done by associating a procedure of that module 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.[1]

Thus implicit invocation supplements, rather than supplants, explicit invocation. Modules may interact either explicitly or implicitly, depending on which mechanism is most appropriate. This feature makes it possible to view implicit invocation as a natural add-on to an existing explicit invocation system, such as one provided by a standard module-oriented programming language. What is required is a way to make it possible for traditional modules to announce their own events and to register for the events of other modules.

Let us now see how this can be done in the context of the Ada language.

2.1 Overview of the Implementation

In Ada the basic unit of modularization is the package [Ada83]. Packages have interfaces, which define (among other things) a set of exported procedures. We developed a small specification language to augment package interfaces. This language allows users to identify

[1] When multiple procedures have registered for the same event, the order of invocation is typically not specified. Thus users of implicit invocation must write their applications in a way that correctness does not depend the existence or ordering of event registrations.

events they want the system to support, and to specify which Ada procedures (in which package specifications) should be invoked on announcing the event. Figure 1 illustrates its use.

```
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
```

Figure 1: Event Specification Language Example

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` 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 Figure 1, `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 interface and body for a package called `Event_Manager`. (Although not illustrated in the figures of this paper, 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` is illustrated in Figure 2. It 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.

The generated body of `Event_Manager` contains the implementation of `Announce_Event`.

```
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;
```

Figure 2: Generated Specification for `Event_Manager`

As illustrated in Figure 3, the procedure is structured as a `case` statement, with one case for every declared event. When a component wishes to announce an event, it invokes `Announce_Event`, as illustrated in Figure 4.

2.2 Key Design Questions

This simple implementation provides many characteristics of more complex implicit invocation systems. However, it embodies a set of design choices whose consequences are important to understand, both to see how to use an implicit invocation system, and to observe the limitations of the implementation. The design decisions can be grouped into the following six categories:

1. Event definition
2. Event structure
3. Event bindings
4. Event announcement
5. Concurrency

```
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;
```

Figure 3: Generated Body for `Event_Manager`

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

Figure 4: Event Announcement

6. Delivery policy

We now examine each of these in turn.

2.2.1 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?

We considered three 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 be 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, which provides a single `changed` event.[2] 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. At the other extreme, tool integration frameworks, such as Field and Softbench [Reiss 90, Gerety 89], 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.

All four approaches can be implemented in Ada. In the first and second cases, events are naturally represented as enumerated types. In the third and fourth cases events are often represented as strings.

We rejected the first alternative as too restrictive. When it came to selecting among the other approaches, there were arguments on each side. Static event declaration has an efficient implementation basis as an Ada enumerated type, and allows 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. Moreover, since dynamic event systems do not use recompilation to maintain consistency between announcements and event bindings, a dynamic event system could be used to reduce recompilation overhead. A similar case can be made for non-existent event declarations.

In the end, predictability through static checking won out. 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 systems development, which require predicatable behavior of the components.

Once we had decided on using static events, we were faced with the question of where the declaration of events should 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" the event, and thus where events should be declared. There were 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, where each module declares the events it expects to announce.

[2] 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.

Our 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.

2.2.2 Event Structure

The next design issue is how events should be structured. We wanted a model of events that would make it easy to use them in system construction and easy to understand the interactions between components. The choices we considered were:

Simple Names Events are simple names without any parameter information.

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 require as a parameter 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 quickly settled on allowing parameters to vary by event type. 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. Allowing parameters to vary by event type over a static list of events also solves a problem of parameter passing: with static events and static parameter lists, a record with a variant part becomes a natural way to represent parameters.

2.2.3 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?

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 *deregister* for events when they are no longer interested.

The decision to use static event bindings was largely forced on us by Ada. 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.

However, even if Ada would have supported dynamic event binding, it is not clear that it would be the right alternative. As with dynamic event declarations, dynamic event bindings decrease the predictability of a system. In particular, the behavior of a system is not apparent from its declaration. Moreover, dynamic event bindings can introduce race conditions at run time. This is because a newly registered binding may or may not catch an existing announced event, depending on the timing of the event and dynamic registration.

Having decided on static event bindings, we were faced with the question of how the parameters from the event would be translated into the parameters for the invocation. The choices we considered were:

All Parameters The invocation 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. We could easily imagine situations in which only some of the information in an event announcement would be useful to a component, and 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.

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.

Although we did not implement it this way, 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.

2.2.4 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 that would announce any event. Pass it a record with a variant part containing the event type and arguments.

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

Language Extension Provide an `announce` statement as a new kind of primitive to Ada and use a language preprocessor to conceal the actual Ada implementation.

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`.

We decided on the "single announcement" approach for a number of reasons. First, in comparison to the multiple procedure approach, it is simple: all event announcements look similar. Second, with respect to the third option, our users were fairly proficient with Ada, and we wanted to stay as close to "pure" Ada 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 language (and not just specially delimited annotations). Finally, 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.

The fourth approach, implicit announcement, has been used as a triggering mechanism for databases [Dayal 90] and some programming environments [Habermann 91]. 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 have required the preprocessor to transform procedures so that they announce the relevant events. As noted above, we wanted to avoid having to process the full Ada language itself. However, this would be a reasonable extension in a future version of the system.

2.2.5 Concurrency

Thus far our enumeration of design decisions has left open the question of exactly what a component is. In our 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 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 task. 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.

The first choice leads to a non-concurrent event system: events are executed using a single thread of control. The second and third choices would permit concurrent handling of events. While we do not forbid tasks inside of packages, our implementation adopts the first approach.

Our decision was based primarily on the fact that, given the current understanding of event systems, it is much easier to develop correct systems using a single thread of control. 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.

2.2.6 Delivery Policy

In most event systems, when an event is announced all procedures bound to it are invoked. However, in some event systems this is not guaranteed. While delivery policy was not a major question in our development, there is enough variation in the way this is done in other systems to explore the design options. The ones we considered are:

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. This delivery policy provides a form of "indirect invocation", as opposed to "implicit invocation".

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 [Reiss 90] 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 interest in supporting implicit invocation, and so was quickly discarded. Although we considered the parameter-based policy model, we eventually decided on the full delivery model, since it allowed the most straightforward analysis by our users. In a future implementation we would certainly consider adding policies as an additional form of flexibility.

3 Evaluation

The system described in previous sections was initially developed for use in a masters-level software engineering course [Garlan *et al.* 92B]. 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 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. While 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 [Gerety 89].)

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.

4 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 [Reiss 90], showed how implicit invocation could be applied to tool integration. More recently Sullivan and Notkin [Sullivan 92] have shown how implicit invocation can be used to ease system evolution without compromising properties of integration. In their work, integration relationships are encapsulated in separate entities, called mediators, which depend on implicit invocation to decouple the maintenance of system invariants from the components that modify and store system state. They also outline techniques for adding implicit invocation to a C++, providing another (quite different) choice of implementation.

In a similar spirit is the use of implicit invocation in the setting 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 raises the issue of using inheritance to handle implicit invocation.

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 would really like 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.

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. For example, Anna augments Ada with specifications [Luckham 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.

5 Conclusion

The contributions of this work are twofold. First, we have shown by example how to add implicit invocation to a statically typed, module-oriented programming language, such as Ada. While some of the design decisions were constrained by the properties of Ada itself, many of those constraints are similar to those found in other programming languages (for example, strong typing). 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.

Acknowledgments

The implementation of implicit invocation in Ada was carried out in the context of the course “Architectures for Software Systems” in Spring, 1992. We gratefully acknowledge the advice and help of those involved in the design of that course: Mary Shaw, Chris Okasaki, and Roy Swonger. We would also like to thank the students and colleagues who have used the system and given us feedback on its effectiveness. David Notkin and William Griswold provided insightful comments on earlier drafts. Finally, we thank David Notkin, Kevin Sullivan, and Robert Allen for their collaborative efforts in developing a scientific basis for using implicit invocation.

References

[Ada83] Reference Manual for the Ada Programming Language. United States Department of Defense. (January 1983).

[Balzer 86] R.M. Balzer. Living in the Next Generation Operating System. *Proceedings of the Fourth World Computer Conference.* (September, 1986).

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

[Dayal 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 *et al.* 92a] David Garlan, Gail E. Kaiser, and David Notkin. Using Tool Abstraction to Compose Systems. *IEEE Computer.* (June, 1992).

[Garlan *et al.* 92B] David Garlan, Mary Shaw, Chris Okasaki, Curtis Scott, and Roy Swonger. Experience with a Course on Architectures for Software Systems. *Proceedings of the SEI Conference on Software Engineering Education.* (October 1992).

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

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

[Gerety 89] Colin 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).

[Habermann 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] Carl 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.

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

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

[Sullivan 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).