EventPorts: Preventing Legacy Componentware[♣]

Anthony Lauder and Stuart Kent
Computing Laboratory
University of Kent at Canterbury
Canterbury, Kent, CT2 7NF, UK
Anthony@Lauder.u-net.com
S.J.H.Kent@ukc.ac.uk

Abstract **- In our work with legacy information systems we have found two prevalent anti-patterns – tight coupling and code pollution – which, if not addressed in replacement systems, could result in today's new systems simply becoming tomorrow's new legacy system. Tight coupling results from Explicit Invocation across collaborating components. Code pollution results from implicit (rather than explicit) reflection of time-ordered collaboration protocols. These anti-patterns diminish component maintainability, flexibility, and reusability. In response, we propose a synthesis of Implicit Invocation (which reduces tight coupling) and Statecharts (which reflect collaboration protocols directly). This paper describes the development of EventPorts, which realize this synthesis and thus encapsulate a novel and promising component collaboration technology.**

I. Introduction

Computerized information systems are developed to support an organization's business processes. As business processes evolve, the information systems supporting them must evolve accordingly. Unfortunately, information systems tend to grow old "*dis*gracefully", so that it becomes increasingly difficult to modify them to reflect on-going business process change. Such systems are termed *legacy systems*. A legacy system is an information system which resists the reflection of on-going business process change [1].

We have been working with a mid-sized software development organization that has a large share of the market for particular types of commercial information system. These information systems have been developed over many years and are now perceived as legacy systems. Supporting these legacy systems is a significant drain on the resources of the organization; development staff is stretched thinly over the current maintenance burden, with few resources remaining for new development work.

Having looked into the legacy system issue on behalf of this organization, we have observed that its legacy systems tend to be characterized by a number of undesirable properties forming anti-patterns. Two of the anti-patterns that are particularly problematic are *tight coupling* between software components, and *code-pollution* within those software components. Our fear is that if replacement systems to legacy systems propagate those same anti-patterns then eventually today's replacement systems will simply become tomorrow's legacy systems. To respond to this problem, we have derived an implementation technology, termed EventPorts, which promises to alleviate the maintenance burden inherent in tight coupling and code-pollution and hence alleviate the resistance to the reflection of on-going business process change inherent in legacy systems. It is EventPorts, and the nature of the problems they resolve, which form the focus of this paper.

II. Explicit Invocation

There is an increasing trend towards the development of software systems constructed from software components [2]. A software component working in isolation usually has little value. Consequently, component-based systems tend to consist of networks of components that collaborate by sending messages to one another. These messages tend to be purposeful, in that they are intended to induce collaborative action. In a typical collaboration, one component (the sender of a message) explicitly identifies another component (the

[♣] *This work was partially funded under EPSRC grant GR/M02606, and by a generous contribution from Electronic Data Processing PLC.*

intended recipient of a message) thus forming an explicit association between them. Since the sending component has explicitly nominated the recipient component in which action is to be induced, this form of collaboration is known as *Explicit Invocation*. Fig. 1 depicts an example of explicit invocation wherein an InvoiceGenerator component is explicitly aware of and hard-codes the invocation of a printInvoice method on a InvoicePrinter component.

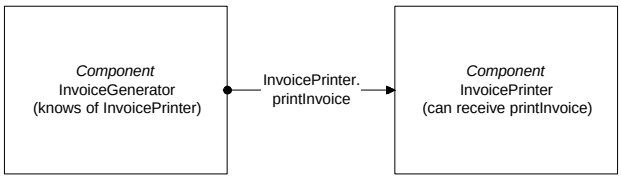

Fig. 1. Explicit Invocation

III. Tight Coupling

Explicit invocation is problematic in that it results in a tight coupling between the sender and the recipient. A major demerit of such tight coupling is that it makes components sensitive to changes in one another. For example, if the InvoicePrinter component in Fig. 1 were modified so that it no longer supported receipt of printInvoice, then the InvoiceGenerator component would need to be modified accordingly. Furthermore, explicit invocation makes a sender component sensitive to changes in the structure of the component network in which it participates. For example, if a new requirement was introduced by the business stating that all printed invoices must be logged to an audit trail, then the InvoiceGenerator would need to be modified to reflect this policy change, perhaps as shown in Fig. 2, below. An alternative is that the InvoicePrinter component could be modified to log invoice prints, but this is not the choice we have made here.

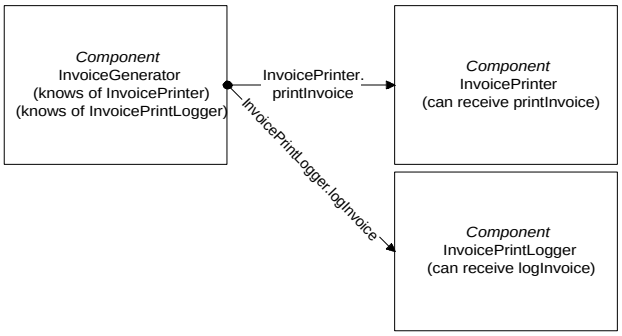

Fig. 2. Tight Coupling

Additional changes to requirements in response to developing business processes would lead to further alterations to the components in the network and the hard-coded collaborations between them. In other words, individual components are highly sensitive to evolution of their network structure and the components with which they collaborate directly. Consequently inter-component dependencies inherent in explicit invocation constitute a major impediment to the maintainability, adaptability, and reusability of software components and the systems developed from them.

IV. Implicit Invocation

Explicit invocation, then, leads to tight coupling between components. To eliminate the maintenance burden inherent in tight coupling, it would be ideal if collaborating components were completely unaware of one another's identity. Such mutual unawareness is termed component de-coupling. The reduction of inter-component dependencies resulting from component de-coupling enhances component maintainability, adaptability, and re-use.

To achieve component de-coupling we need to abandon explicit invocation and migrate towards *Implicit Invocation* [3], wherein recipient components register their message interests with a broker, via which sending components propagate messages to currently interested recipients. The broker forms an intermediary between collaborating components, hence resulting in component de-coupling.

Implicit invocation is necessarily a two-phase process. In the first phase, Interest Registration (see Fig. 3), recipient components register their message interests with a broker, specifying both a message topic and an associated action to be invoked upon receipt of that message.

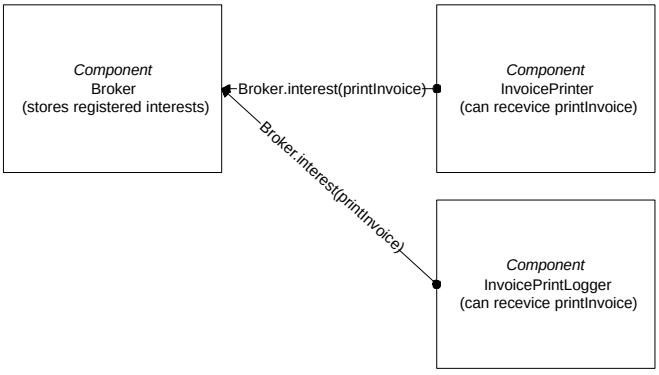

Fig. 3. Interest Registration

In the second phase or implicit invocation, termed Propagation (see Fig. 4), sending components pass messages to the broker. The broker then utilizes its knowledge of registered message interests to determine those receiving components whose currently registered interests coincide with the message just received and hence need the received message to be passed on to them (where that message will inspire the invocation of the associated action).

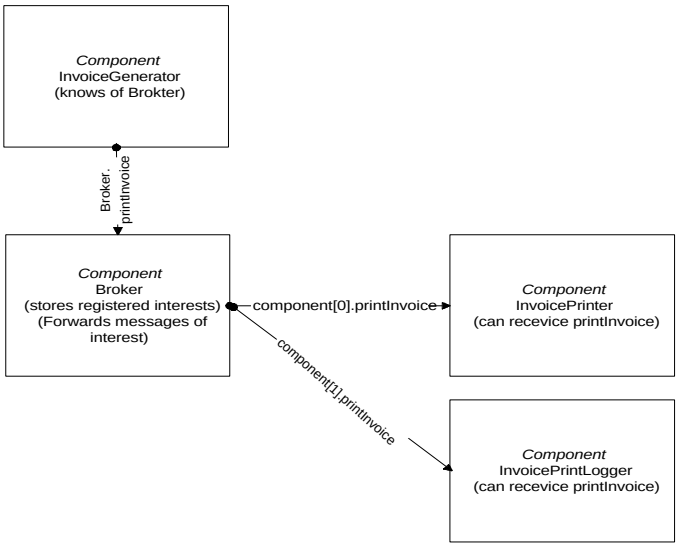

Fig. 4. Propagation

V. Historically Determined Event Interests

Experience with implicit-invocation-based patterns and products through various industrial projects, has taught the authors that the resultant de-coupling of components leads to considerable increases in flexibility, adaptability, configurability, and reusability of the resultant components and component networks. However, it is also the authors' experience that implicit invocation mechanisms still fail to resolve one major problem found in all of the legacy system we have studied.

A software component exports an interface detailing the methods which may be invoked on that component, or in the case of implicit invocation the interface details the messages in which the component is interested in receiving. Frequently, however, the order in which those methods may be legitimately invoked (or those messages received) is completely unconstrained. This unconstrained invocation ordering is often problematic. For example, it would be inappropriate to allow the same expense claim to be paid twice, or for a block of memory to be deallocated if it had not previously been allocated. Thus, there is an implicit appropriate order of invocation underlying a component's methods or message interests. More specifically, the messages that a component is interested in receiving at any given moment is a function of the mutations to its abstract state which have occurred in response to the messages that it has already received. Thus, the invocation ordering (the time-ordered protocol) of a component is historically determined and hence mutable.

VI. Code Pollution

In a traditional implementation of implicit invocation, then, a component registers all its message interests in advance. Thus, it is perfectly possible for that component to receive messages in an order that does not respect its implicit time-ordered invocation protocol. This could lead to disasterous consequences, and hence a component has to enforce its protocol via the protection of individual methods against out-of-sequence invocation. Consequently, the body of each individual method tends to be wrapped in protective guard code whose purpose is to ensure that any out-of-sequence invocation of that method is benign and hence can do no harm.

This guard code is secondary to the main purpose of a given method, and hence pollutes the method's essential behavior. This code pollution complicates the bodies of component methods and hence reduces component maintainability and adaptability. It also makes it difficult for a maintainer of the component to understand the time-ordered protocols that a component respects, since the time ordering of protocols is implicit in the guard conditions scattered throughout that component's methods, rather than explicitly recorded in a single place.

As an example of code pollution, imagine a component that represents an application for a loan, as shown in Fig. 5. For the sake of simplicity, only a minimal set of attributes and methods are depicted.

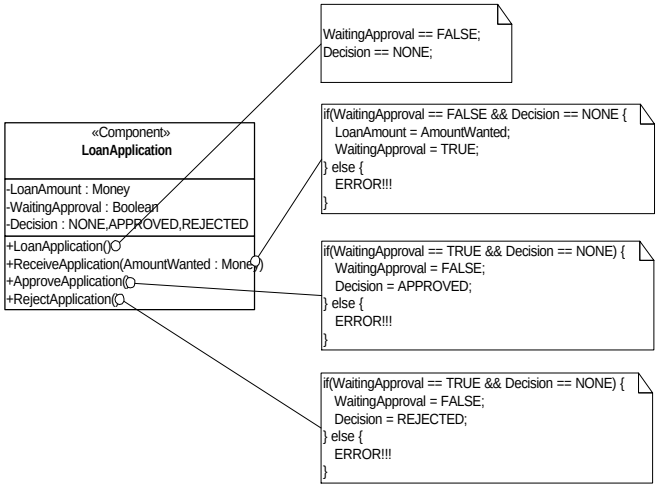

Fig. 5. Code Pollution

In Fig. 5, a LoanApplication at first represents, in effect, a blank piece of paper. When an actual loan application is received, the requested loan amount is recorded. The requested loan may then be either approved or rejected. Notice that each method is surrounded by conditional code and contains a variety of flags being set to control and indicate progress of the loan through an implicit ordering of

method invocations. The implicit protocol is that a LoanApplication must be blank when an actual loan application is received, and a loan can only be approved or rejected if that loan has been received and has not already been approved or rejected. The individual methods in the LoanApplication component, then, have become polluted with code which is not specific to the fundamental behavior of those methods, but which is essential in case methods are invoked out of order.

As the LoanApplication component becomes more sophisticated, with the addition of new data members and methods, we can expect code pollution to become more and more problematic. It is this increasing need to reflect growing intra-component dependencies across methods and attributes that leads to fragile components unable to reflect further business process change due to the fear that "if we touch it, it might break". It is this fear that leads, in large part, to systems being termed legacy systems. To quote one of the (unfortunate) maintainers of a legacy system that we have come across: "The code is in control." It is our assertion that the elimination of code pollution of this sort can help to put the developer back in control, and not the code.

VII. STATECHARTS

In response to the code pollution problem, we propose a direct realization of Statecharts [4] in a component's implementation. The Statechart notation encompasses the expression of abstract state mutations in response to the occurrence of events of interest. In other words, statecharts capture time-ordered event interests and also reflect the history of events received so far in terms of a series of mutually exclusive abstract states.

A statechart consists of a network of disjunctive abstract states. A statechart in execution is said to be in one (current) abstract state at a given time. Each abstract state is associated with a set of outward transitions. A transition is a quadruple of the form {event, condition, action, nextstate}. When a statechart receives an event, it takes whichever transition (if any) in its current abstract state matches that event so long as that transition's condition is satisfied. Taking a transition means invoking that transition's action and making that transition's nextstate the new current abstract state for the statechart. If no transition both matches the received event and has a satisfied condition, then no action occurs and the current state remains unchanged.

The important contribution of statecharts is that they make historically determined event interests explicit (see Fig. 6). Statecharts, then, capture time ordered protocols and thus promise to eliminate the need for polluting component methods with guard code in case of their out-of-sequence invocation. Reflecting statecharts directly in a software component, then, eliminates the necessity for code pollution, since the statechart mechanism itself ensures that a method cannot be invoked unless the protocol inherent in the statechart is respected.

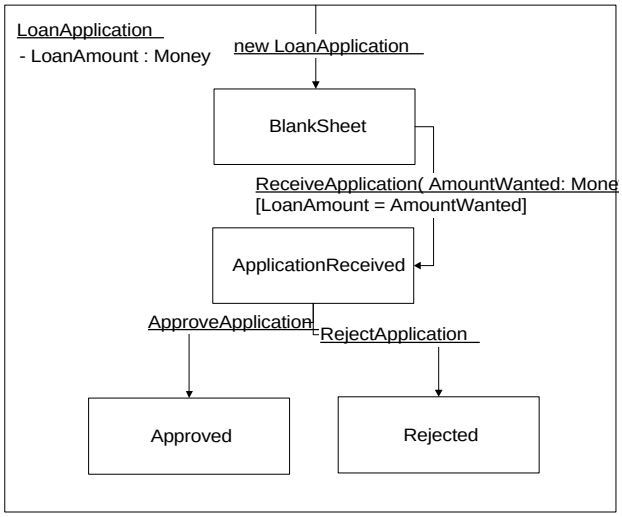

Fig. 6. Statechart

Looking at Fig. 6, we see that a LoanApplication now needs only one attribute (the LoanAmount). All other attributes from Fig. 5 have now been absorbed into the mutually-exclusive (disjoint) abstract states captured in the statechart (BlankSheet, ApplicationReceived, Approved, Rejected). When a new LoanApplication is created it immediately transitions to the BlankSheet abstract state. From there, the only acceptable next message is ReceiveApplication. Any message other than ReceiveApplication will simply be ignored (or may result in an error, if that is what we choose an implementation to do). ReceiveApplication sets the single LoanAmount attribute to the requested loan amount, and then transitions to the ApplicationReceived abstract state. From there, only two messages are of interest: ApproveApplication (which transitions to Approved) and RejectApplication (which transitions to Rejected). The statechart, then, has captured explicitly the time-ordered protocol of the LoanApplication component. This has two significant advantages over the implementation presented in Fig. 5. Firstly, the elimination of code pollution simplifies the individual methods of the component and thus eases component maintenance, and secondly the fact that the protocol is explicitly recorded in a single place help humans to comprehend and describe that protocol since it is no longer scattered across diverse guard code.

IIX. EVENTPORTS

It is the authors' thesis, then, that combining the de-coupling inherent in the implicit invocation model, with explicit respect for historically-determined event interests inherent in Statecharts results in a component collaboration strategy which enhances component maintainability, adaptability, and re-usability. It is via such an implementation technology, we propose, that we can eliminate both tight coupling and code pollution and, thus, help to prevent today's new information systems from becoming tomorrow's legacy systems.

To realize this synthesis of implicit invocation and statechart-based explicit protocol expression, the authors have implemented a technology, which we have named *EventPorts*. In the EventPorts model, a *Component* exports an *Interface* and an *Outerface*. A component receives messages from other components through its interface and sends messages to other components through its outerface. A component in execution begins its life with an empty interface and an empty outerface. Interfaces and outerfaces evolve dynamically through the run-time addition and subtraction of *EventPorts*. There are two types of EventPort: interfaces consist solely of *InPorts*, and outerfaces consist solely of *OutPorts* (see Fig. 7). An InPort exports a single method, *in*, via which it receives incoming messages from other components. An OutPort exports a single method, *out*, via which it sends outgoing messages to other components. InPorts and OutPorts are first class objects in their own right (as are Interfaces, Outerfaces, and Components), and hence they can be created, destroyed, mutated, passed as arguments, and so on, just like any other object. This run-time flexibility leads to an extremely flexible collaboration model.

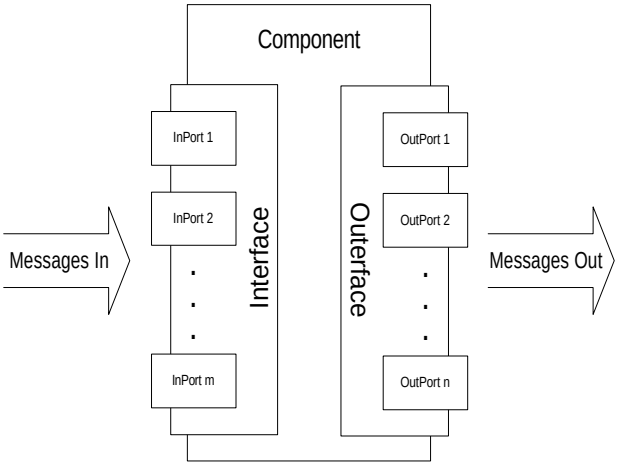

Fig. 7. Interfaces and Outerfaces

IX. INPORTS

When an InPort is created, it is associated with a *Statechart* (see Fig. 8). Again Statecharts are first class objects, and hence may be dynamically created, destroyed, mutated (e.g., adding or removing States and Transitions dynamically), passed as arguments, and so on. The *CurrentState* of a Statechart determines (via its Transitions) which messages the associated InPort is currently interested in receiving. If an InPort receives a message that is not associated with one of the Transitions of its Statechart's *CurrentState*, then that message is simply ignored. If, however, that message matches the interests of a Transition, and any associated *Condition* is satisfied, then the Transition is taken, any associated *Action* (again, a first class object) is invoked, and the target State of that Transition becomes the new CurrentState.

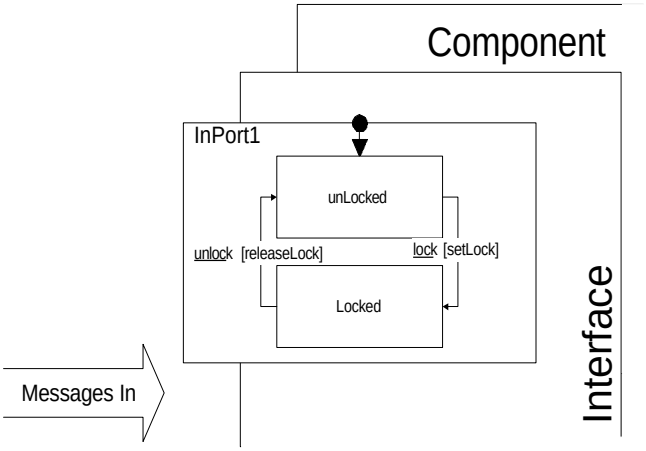

Fig. 8. InPort

Since a component may have many InPorts, and each InPort is associated with a Statechart, each component may exhibit multiple orthogonal statecharts (see Fig. 9), the sum of which represents its total abstract state, and the set of current states of which determines, at any given moment, the total message interests of that component. Each InPort may, therefore, be considered a port-hole via which one facet of a component's total abstract state is exported.

The separation of a component's total abstract state and state transitions into a set of orthogonal facets and associated statecharts leads to a clean separation of concerns and thus enhances component maintainability. In the legacy systems we have examined, the failure to observe such a separation of concerns into orthogonal facets has resulted in an unnecessary intertwining of fundamentally independent features. A major emphasis during the design of a replacement system, then, should be placed upon achieving such separation.

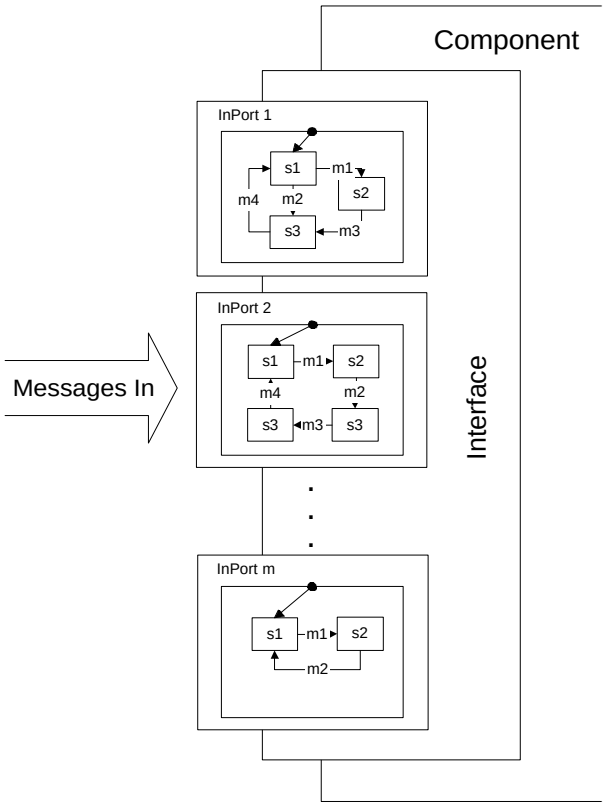

Fig. 9. Orthogonal Statecharts

X. OutPorts

OutPorts are somewhat simpler than InPorts. A component outputs messages via its OutPorts. The component code is completely unaware of where those messages are delivered. In fact, an OutPort simply passes messages on to whichever InPorts are currently attached to it (see Fig. 10).

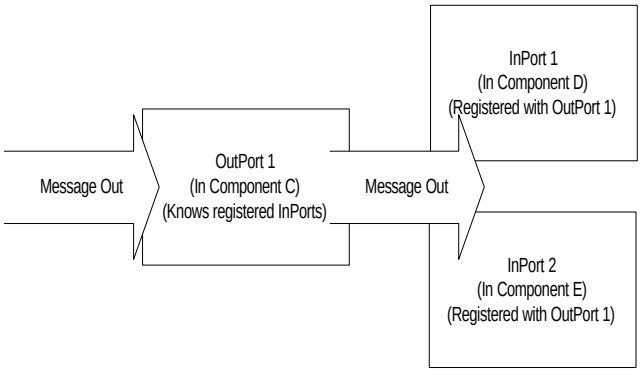

Fig. 10. OutPort

InPorts can be attached to and detached from OutPorts dynamically. Each InPort can be attached to any number of OutPorts simultaneously, and any number of InPorts can be attached to a given OutPort. An OutPort acts, in effect, as both a registry for message interests and a broker that receives and forwards messages according to those registered interests. The attachment and detachment of InPorts to and from OutPorts underlies the elimination of explicit invocation in favor of implicit invocation in the EventPorts model. The historically-determined message interests inherent in the Statechart associated with a receiving InPort ensure that only appropriate events are received (i.e., time-ordered protocols are respected) and appropriate actions invoked as a consequence.

XI. An Example

To demonstrate the use of EventPorts, we present in this section a utilization of EventPorts to realize the loan application component described above. Referring back to the Statechart in Fig. 6 may help the reader to understand the following C++ code.

Our component requires only one InPort, which we will term a LoanInPort. That LoanInPort maintains the loan amount requested in a loan application.

```
typedef int Money;
class LoanInPort : public Interface::InPort {
public:
  Money loanAmount;
  LoanInPort( Statechart * givenStatechart = 0 ) :
                    Interface::InPort( givenStatechart) {}
  virtual ~LoanInPort(){}
};
```

The Statechart in Fig. 6 covers four possible (abstract) states:

```
State blankSheet, applicationReceived, approved, rejected;
```

The initial state in which our LoanInPort's Statechart should reside is the blankSheet state:

```
Statechart loanApplicationStatechart( &blankSheet );
LoanInPort loanApplicationInPort( &loanApplicationStatechart );
```

Our (banking) component exports only a single LoanInPort, which we will label "LOANS" for future reference:

```
Component bankingComponent;
bankingComponent.interface().add("LOANS",
                                  loanApplicationInPort);
```

Our Statechart requires only a single action; when an application is received, the requested loan amount passed as an incoming message needs to be recorded in the loanAmount attribute of the Statechart's LoanInPort:

```
class SetLoanAmount : public Action {
public:
  void in( const Interface::InPort * givenInPort, Event * givenEvent )
{
    ((LoanInPort*)givenInPort)->loanAmount =
            *(int *)givenEvent->message()->message();
  }
};
SetLoanAmount setLoanAmount;
```

When our Statechart is in the blankSheet state, it is interested in receiving a message under the topic "RECEIVE APPLICATION". Upon receipt of this message the setLoanAmount action is invoked, before transitioning to the applicationReceived state:

```
Topic receiptTopic("RECEIVE APPLICATION");
Interest receiptInterest( &receiptTopic );
Transition toApplicationReceived(
     &receiptInterest, &applicationReceived, &setLoanAmount );
blankSheet.add( &toApplicationReceived );
```

When our Statechart is in the applicationReceived state, it is interested in receiving messages under the topics "APPROVE APPLICATION" and "REJECT APPLICATION". Upon receipt of "APPROVE APPLICATION" the Statechart transitions to the approved state. Upon receipt of "REJECT APPLICATION" the Statechart transitions to the rejected state.

```
Topic approvalTopic("APPROVE APPLICATION");
Interest approvalInterest( &approvalTopic );
Transition toApproved( &approvalInterest, &approved );
applicationReceived.add( &toApproved );

Topic rejectionTopic("REJECT APPLICATION");
Interest rejectionInterest( &rejectionTopic );
Transition toRejected( &rejectionInterest, &rejected );
applicationReceived.add( &toRejected );
```

All of the code so far has been concerned with setting up the structure of the Statechart associated with our LoanInPort withinour banking component. We now turn our attention to utilization of this component. In reality, we would probably create another component (a customer component) with an associated OutPort to which the InPort of the banking component is attached. Here, however, we demonstrate that OutPorts can be used without their containment within a component (as can OutPorts for that matter).

We create an OutPort to which the banking component's LoanInPort is attached as a listener to out-flowing messages. Remember, the Statechart associated with the LoanInPort will determine whether or not to react to those out-flowing messages:

```
Outerface::OutPort customerOutPort;
customerOutPort.addListener(
     bankingComponent.interface()["LOANS"] );
```

Through this OutPort, then, we push receipt and approval events, which are passed on to the attached LoanInPort. Since these events respect the protocol captures in our LoanInPort, they cause it to transition to the applicationReceived state and then on to the approved state:

```
customerOutPort.out(&receiptEvent);
customerOutPort.out(&approvalEvent);
```

Next, we broadcast a rejection event through the OutPort. Since the only attached InPort (the LoanInPort) is not interested in this event when in its current approved state, the event is ignored:

```
customerOutPort.out(&rejectionEvent);
```

If, however, we mutate the approved state of the LoanInPort to accommodate a new transition to the rejected state on receipt of a rejection event, then the occurrence of the rejection event will cause precisely that transition:

```
approved.add( &toRejected );

customerOutPort.out(&rejectionEvent);
```

XII. Related Work

To recap, EventPorts are built upon the idea of combining implicit invocation (to eliminate tight coupling) with statecharts (to prevent code pollution). We have not seen this combination elsewhere, and hence we consider this a particular contribution of the EventPorts model. All other utilizations of implicit invocation with which we are familiar are based on guard code (i.e. code pollution) a register/deregister discipline within the application code. Furthermore, we propose that EventPorts - irrespective of their combination with statecharts - encapsulate the most flexible choices in the implicit invocation design space when compared to other realizations of which we are aware. We can segregate the realizations of implicit invocation known to us into three partitions: design patterns [5-7], commercial products [8-10], and standards [11,12]. Below we compare and contrast EventPorts against published alternatives from the perspective of four dimensions of the implicit invocation design space.

A. Independent Brokering

In the Observer [5] and Event Notification [6] design patterns, the generator of an event also assumes the role of event-interest registry and event broker. This complicates implementation considerably. Consequently, in the Reactor [7] design pattern, and most commercial products and standards there is the notion of intermediate broker between the event generator and the event recipient. EventPorts adhere to this concept of independent broker in terms of OutPorts, which may be attached both to event-generating components and to the InPorts of other components. In addition, however, it is possible for OutPorts to be event-generating components in their own right, and thus the user of EventPorts is free to choose between the tightly integrated or independent brokering mechanisms.

B. Broker Scoping

Most design patterns do not say whether or not there may be more than one broker present. Where there is only one broker (as in commercial products such as [10]) there is no possibility of providing event scoping. That is, a multiple broker scheme allows the visibility of generated events to be limited to the scope of only those components registered with that broker. The CORBA Event Service [11] supports an event scoping mechanism in its event channel mechanism, and EventPorts follow this lead in terms of OutPort/InPort interconnection.

C. Event Types

Several design patterns permit only one event type per event generator [5,11]. The Reactor design pattern [7] is considerably closer in spirit to EventPorts, wherein event brokers are not tied to any specific event type. Most commercial products and standards also eliminate the static binding of brokers to the events they may broker.

D. Event Matching

All the design patterns, most of the commercial products examined, and the CORBA Event Service {OMG 1997 ID: 26} are based on the assumption of an absolute match between generated events and event interests. In practice, we have found that often an application knows only its event interests in the most general terms. One solution to this is to introduce a sophisticated Trading Service (e.g. [13] and [14]) which matches services provided against service interests where both are expressed in some constraint specification language. We consider this a rich yet heavyweight approach, and have found the lighter strategy of *polymorphic event matching* found in EventPorts, the Java Event model, and the CORBA Messaging Service [12] to be far simpler and more than adequate, at least for the applications we have dealt with. Polymorphic event matching, in its EventPorts realization, is based upon both event hierarchies and overloaded equality operators. Actually, we have even found that event hierarchies are rarely necessary in practice, and have tended towards utilizing the simplest of the event matching schemes supported by EventPorts, wherein event names are expressible as simple strings with event interests expressed as wildcarded strings against which event names are "pattern matched" (a similar approach is adopted in [10]).

XIII. Summary And Further Work

To recap upon the main themes of this paper: Tight coupling and code pollution lead to significant maintenance headaches and result in petrification of legacy systems so that they resist the reflection of on-going business process change. EventPorts combine Implicit Invocation with a direct realization of Statecharts and thus promise to alleviate this burden. Consequently, EventPorts promise to help prevent new adaptive component-based systems from becoming tomorrow's legacy systems.

Since EventPorts are a new technology, our experience with them is relatively minimal. Early results, however, have been encouraging; EventPorts, then, appear to offer a general, flexible, lightweight and promising component collaboration strategy. Theoretically, the ideas seem sound, yet only a solid history of practical application will prove their real worth. Towards this end, the authors are working with an industrial collaborator to migrate a number of legacy systems towards an EventPorts model, and the results of this work will be published in due course.

At a purely technical level, a number of new features are planned for EventPorts, including support for multi-threading, event transactions, and integration with CORBA [15]. Of wider scope, the author recognizes that developing according to this model requires some divergence from established object-oriented methodologies and notations. Consequently, the authors are currently collaborating with a number of other researchers to determine philosophical perspectives, methodological enhancements, and modeling notation features appropriate for EventPorts-based systems.

Acknowledgments

Thanks to John Corner, Bob Doncaster, and Ian Oliver for comments on early drafts of this paper. The authors also wish to thank the anonymous reviewers whose comments led to additional insights and to improvements in the paper's presentation.

One reviewer felt that the use of the term *port* in out naming scheme (EventPorts, InPorts, and OutPorts) was a poor choice, since that term has implications of low-level

networking. Consequently it was suggested that we select a naming scheme with a more application-level ring to it. The term *faces* has been suggested, but we find this term too general to convey the spirit of EventPorts. We are, therefore, interested in receiving suitable naming suggestions from interested readers.

REFERENCES

[1] M.L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*, Morgan Kaufman, 1995.

[2] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[3] M. Shaw and G. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[4] D. Harel and M. Politi. *Modeling Reaction Systems with Statecharts*, McGraw-Hill, 1998.

[5] E. Gamma, R. Helm, R. Johnson, and Vlissides.J. Observer. In: *Design Patterns: Elements of Reusable Object-Oriented Software*, eds. E. Gamma, R. Helm, R. Johnson, and Vlissides.J. Addison-Wesley, 1995.

[6] D. Riehle, The Event Notification Pattern - Integrating Implicit Invocation with Object-Orientation *Theory and Practice of Object Systems*, vol. 2, 1996.

[7] D. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Multiplexing and Event Handler Dispatching. In: *Pattern Languages of Program Design*, eds. J.O. Coplien and D. Schmidt. Addison-Wesley, 1995.

[8] SoftWired AG. *iBus*, www.softwired-inc.com, 1999.

[9] Talarian. *SmartSockets*, www.talarian.com, 1999.

[10] Tibco. *Rendezvous Information Bus*, www.tibco.com, 1999.

[11] OMG. *Corba Event Service*, www.omg.org, 1997.

[12] OMG. *Corba Messaging Service*, www.omg.org, 1998.

[13] OMG. *Corba Trading Object Service*, www.omg.org, 1997.

[14] ITU/ISO. *ODP Trading Function - Part 1: Specification*, ITU/ISO, 1997.

[15] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*, Addison-Wesley, 1999.