

EventPorts

Anthony Lauder
Computing Laboratory
University of Kent at Canterbury
Canterbury, Kent, CT2 7NF, UK
Anthony@Lauder.u-net.com

Abstract. Explicit Invocation across collaborating components leads to tight coupling between those components. This diminishes component maintainability, flexibility, and reusability. The Implicit Invocation model de-couples components and hence reduces inter-component dependencies. Implicit Invocation, however, tends to ignore the historically determined nature of component message interests, resulting in implementations polluted with guard code to enforce components' time-ordered protocols. By combining Statecharts with Implicit Invocation, direct realization of time-ordered component protocols is achieved without code pollution, offering the potential for a cleaner, more adaptable component collaboration strategy. This paper describes the development of EventPorts, which reflect this combination and thus encapsulate a novel and promising component model. This work forms a central part of my PhD thesis, work towards which began in September 1997.

Explicit Invocation

Component-based systems [Szyperki, 1998] consist of networks of collaborating components. Components collaborate by sending action-inducing messages to one another. In a typical collaboration, a sender explicitly identifies the intended recipient of a message thus forming an explicit association between the components. Inducing action by sending messages across explicit associations is termed *Explicit Invocation*. Here the sender must be aware of the specific recipient of the message, but the sender is usually anonymous to the recipient. Thus there is a one-way tight coupling from sender to receiver (see Fig. 1). The inter-component dependencies inherent in such tight coupling constitute a major impediment to the maintainability and adaptability of collaborating components.



Figure 1 – Explicit Invocation (Tight Coupling)

Implicit Invocation

Ideally, collaboration between components would be de-coupled in both directions. That is, no component would need to be aware of any other component's identity. Such de-coupling minimizes inter-component dependencies and thus enhances component maintainability, adaptability, and re-use. De-coupling mandates withdrawal from an explicit invocation model and a move towards a model of *Implicit Invocation* [Shaw and Garlan, 1996]. Some of the important ideas underpinning implicit invocation are:

- Components implement actions that are invoked in response to messages.
- The messages a component can respond to (by invoking actions) form its message interests.
- A component's message interests could be placed in a registry.
- A component sending a message could check in the registry to determine which recipients are interested in receiving that message.

Thus sending and receiving components are de-coupled by their utilization of a mutually shared registry of message interests, from which the appropriate recipients for a given message are identified. Implicit

invocation can be thought of as a two-phase model, where message interest registration (see Fig. 2), precedes actual message transmission (see Fig. 3).

Probably the most well-know manifestation of implicit invocation is the Observer pattern [Gamma et al. 1995] wherein *observer* objects are dynamically registered with *subject* objects whose mutating state must be notified to them. Here, the registry is within each subject object itself, whose ongoing state-changes trigger the transmission of notification messages to the observer objects currently registered with it. Implicit invocation has gained considerable recent interest, and is at the heart of many published design patterns [Riehle, 1996; Schmidt, 1995; Vlissides, 1997] and a number of commercial products [OMG, 1999; SoftWired AG, 1999; Talarian, 1999; Tibco, 1999].

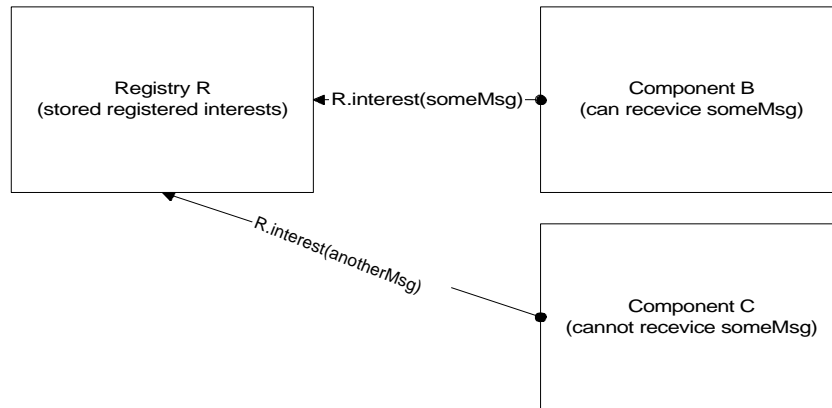


Figure 2 – Message Interest Registration

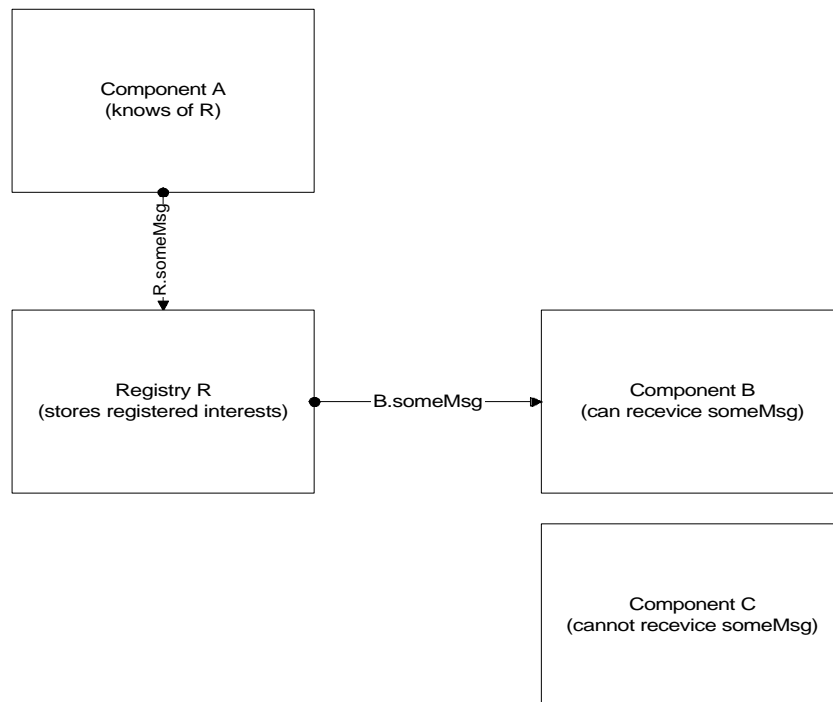


Figure 3 – Implicit Invocation

Historically Determined Event Interests

Having utilized a number of implicit-invocation-based patterns and products through various industrial projects, it has become evident to the author that the resultant de-coupling of components leads to considerable increases in flexibility, adaptability, and configurability of the resultant component networks. However, it is this author's experience – supported by conversations with colleagues past and present - that implicit invocation mechanisms suffer from one major demerit: Implicit invocation mechanisms, at least in their published manifestations, pay little attention to the reality that message interests are historically determined throughout the lifetime of a component. 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.

If the historical determination of message interests is not reflected explicitly, then a component is liable to receive messages which it is not currently appropriate for it to receive but which the component has registered interest in because those messages would be appropriate were the time right. In such cases, the actions associated with received messages must be bounded by explicit code implementing guard conditions. Those guard conditions determine whether or not a given message should have been received according to aspects of the component's current state. Such code pollutes and hence complicates the bodies of component actions and reduces component maintainability and adaptability (see Fig. 4). It also makes it difficult to understand the time-ordered protocols that a component respects, since the time ordering of protocols is implicit in the guard conditions scattered across that component's actions, rather than explicitly recorded in a single place.

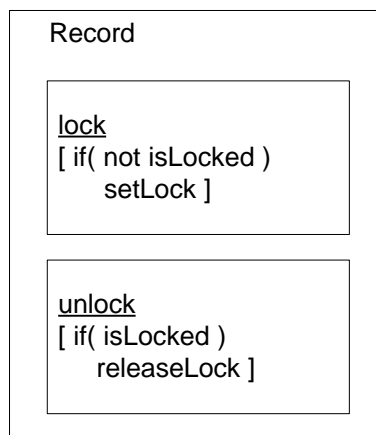


Figure 4 – Code Pollution

Statecharts

Statecharts [Harel and Politi, 1998] are a well-known notational mechanism for the description of historically determined abstract state mutations in response to historically determined event interests. A statechart comprises a set of disjunctive states each connected by a set of transitions. A transition is a quadruple of the form {event, condition, action, nextstate}. At any moment, one state in an active statechart is termed the statechart's current state. When a statechart receives an event, it takes whichever transition (if any) in its current 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 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.

Statecharts, then, embody precisely the reflection of historically determined message (event) interests that is lacking in prevalent implicit invocation mechanisms. That is, Statecharts make explicit the time-ordered protocols that a component respects (see Fig. 5). It is the author's thesis, then, that combining the de-coupling inherent in the implicit invocation model, with explicit respect for historically-determined event interest inherent in Statecharts results in a component collaboration strategy which enhances component

maintainability, adaptability, and re-usability. Consequently, the author has implemented a technology – termed *EventPorts* – based upon precisely these principles.

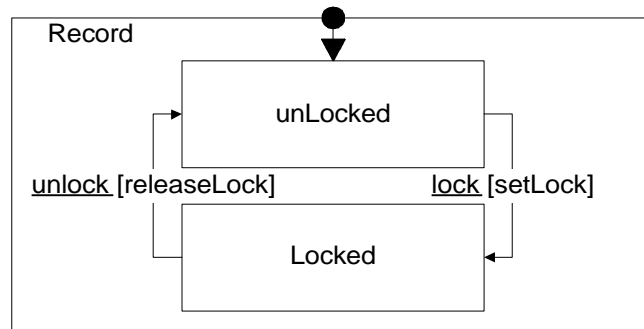


Figure 5 - Statechart

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 addition and subtraction of *EventPorts*. There are two types of *EventPort*: interfaces consist of *InPorts*, and outerfaces consist of *OutPorts*. An *InPort* exports a method, *in*, via which it receives incoming messages from other components. An *OutPort* exports a method, *out*, via which it sends outgoing messages to other components (see Fig. 6). *InPorts* and *OutPorts* are first class objects in their own right (as are *Interfaces*, *Outerfaces*, and *Components*), and hence they can be created, destroyed, passed as arguments, and so on, just like any other object.

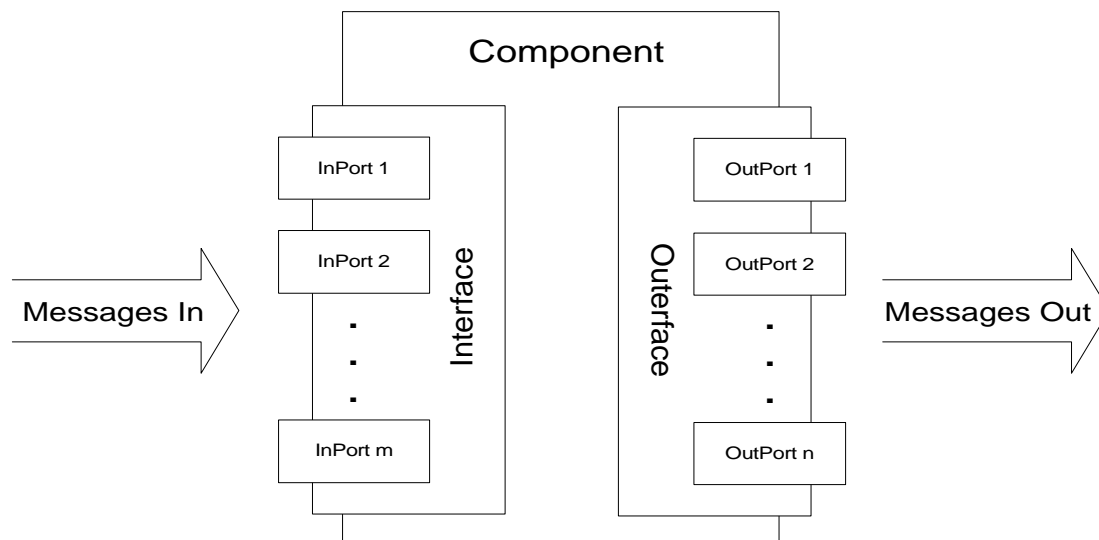


Figure 6 – Interfaces, Outerfaces, InPorts, and OutPorts

InPorts

When an *InPort* is created, it is associated with a *Statechart* (see Fig. 7). Again *Statecharts* are first class objects, and hence may be 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.

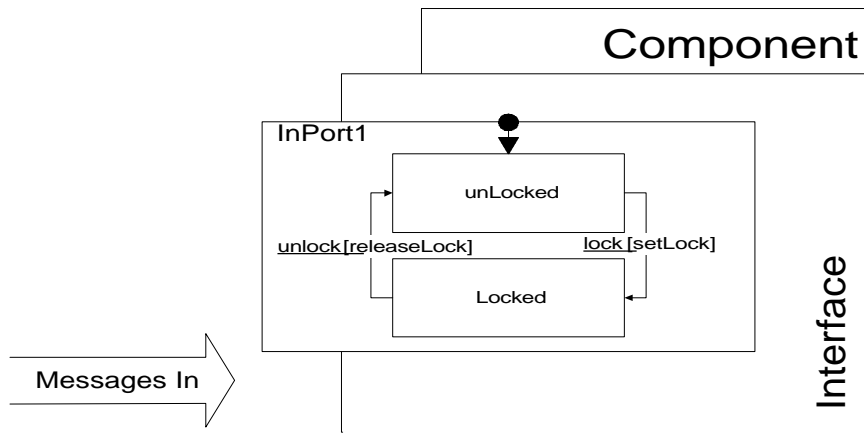


Figure 7 - InPort

Since a component may have many InPorts, and each InPort is associated with a Statechart, each component may exhibit multiple orthogonal statecharts (see Fig. 8). The sum of these statecharts represents a component's total abstract state, and the total set of current states across those statecharts determines, at any given moment, the total message interests of that component.

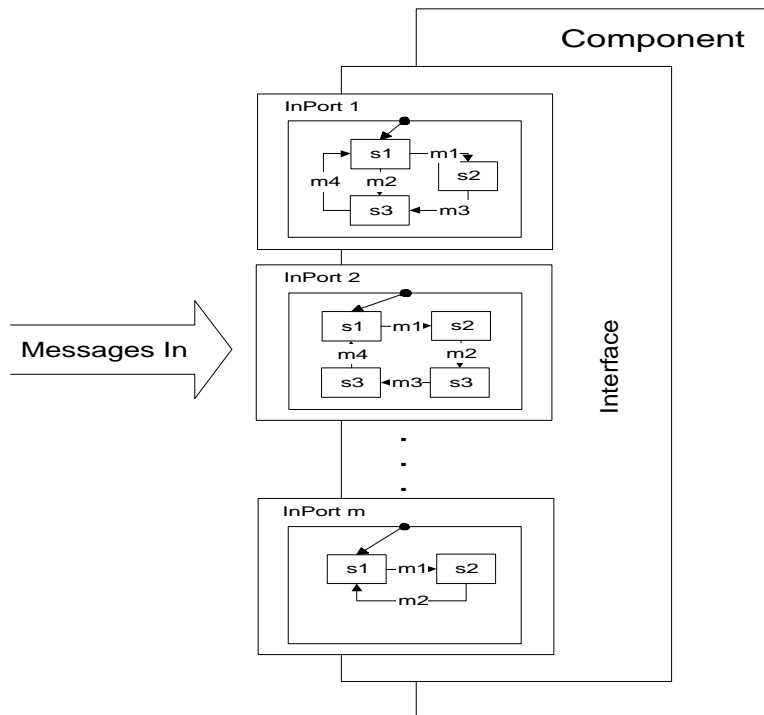


Figure 8 - Orthogonal Statecharts

OutPorts

OutPorts are somewhat simpler than InPorts. A component outputs messages via its OutPorts. The component code is completely unaware of where those messages arrive. In fact, an OutPort simply passes messages on to whichever InPorts are currently attached to it (see Fig. 9). InPorts can be attached to and detached from OutPorts dynamically. Each InPort can be attached to (registered with) 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.

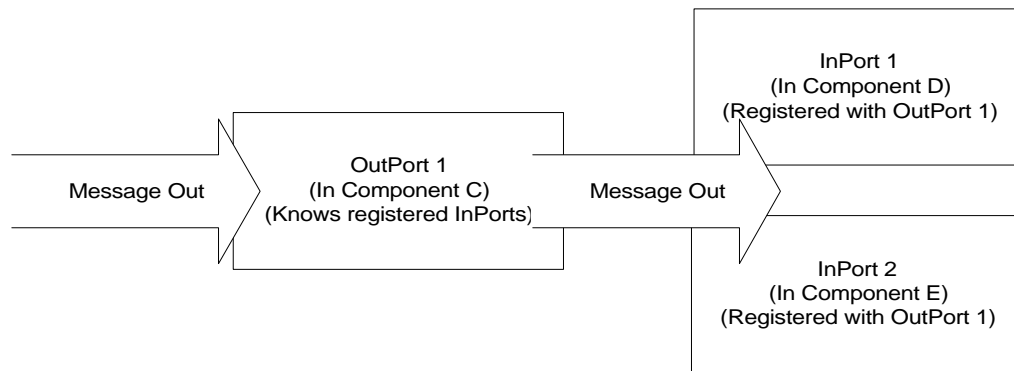


Figure 9 - Outport

Summary and Further Work

EventPorts are built on the following premises:

- Components collaborate by communicating action-inducing messages
- An implicit-invocation model enhances component adaptability and maintainability
- Statecharts can add respect for time-ordered protocols to an implicit-invocation model
- EventPorts allow the dynamic construction of flexible component interfaces and outerfaces
- Dynamic attachment of InPorts to OutPorts leads to flexible, configurable, component collaboration

A C++ implementation of the EventPort model is up and running. Each concept in the model (Component, Interface, Outerface, InPort, OutPort, Message, etc) is provided as a base class in a library. Applications either use these base classes directly, or provide derived classes with appropriate differential behavior. Greater detail on the implementation of EventPorts will be the subject of an up-coming paper.

Early experience with EventPorts has been encouraging; EventPorts appear to offer a general, flexible, and promising component collaboration strategy. However, EventPorts are a new technology, and hence their utility has yet to be proven outside of an academic setting. Theoretically, the ideas seem sound, yet only practical application of EventPorts will prove their real worth. Consequently, the author is currently working with an industrial collaborator to migrate a number of legacy information systems (primarily in the accountancy and sales and distribution domains) 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 (possibly) integration with CORBA [Henning and Vinoski, 1999]. Of wider scope, the author recognizes that developing according to this model requires some divergence from established object-oriented methodologies and notations. Consequently, the author is currently

collaborating with a number of other researchers to determine philosophical perspectives, methodological enhancements, and modeling notation features appropriate for EventPort-based systems.

Acknowledgements

Thanks to Stuart Kent for his encouragement and supervision. Thanks to John Corner, Ian Oliver, and the anonymous reviewers for insightful comments on earlier drafts of this paper.

This work was partially funded under EPSRC grant G16365U, and by a generous contribution from Electronic Data Processing PLC.

References

- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.
- Harel, D. and Politi, M. (1998) *Modeling Reaction Systems with Statecharts*, McGraw-Hill.
- Henning, M. and Vinoski, S. (1999) *Advanced CORBA Programming with C++*, Addison-Wesley.
- OMG (1999) *Corba Event Service*, www.omg.org.
- Riehle, D. (1996) The Event Notification Pattern - Integrating Implicit Invocation with Object-Orientation. *Theory and Practice of Object Systems* 2, 1
- Schmidt, D. (1995) Reactor: An Object Behavioral Pattern for Concurrent Event Multiplexing and Event Handler Dispatching. In: Coplien, J.O. and Schmidt, D., (Eds.) *Pattern Languages of Program Design*, Addison-Wesley.
- Shaw, M. and Garlan, G. (1996) *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.
- SoftWired AG (1999) *iBus*, www.softwired-inc.com.
- Szyperski, C. (1998) *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley.
- Talarian (1999) *SmartSockets*, www.talarian.com.
- Tibco (1999) *Rendezvous Information Bus*, www.tibco.com.
- Vlissides, J. (1997) Multicast. *C++ Report* September SIGS.