Categorization of Implicit Invocation Systems

ROLANDO BLANCO, PAULO ALENCAR
David R. Cheriton School of Computer Science, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Technical Report CS-2007-31

Development and maintenance of implicit invocation systems is not as well understood and supported as the development of explicit invocation systems. The situation is aggravated in systems that allow the composition of functionality developed by different organizations, potentially using different programming languages and methodologies. In this document, we categorize the various ways in which implicit invocation systems can define, bind to, announce, subscribe, and deliver events. We also categorize the architectures and topologies of implicit invocation systems. The purpose of the categorization is to increase the understanding of the type of requirements that these systems impose on software development practices. As the categories are introduced, documented representative invocation systems are discussed in order to illustrate the types of systems that fall under each given category. Our categorization applies to systems as varied as active databases, aspect-oriented applications, and distributed heterogeneous event-based systems.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Procedures, functions, and subroutines*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*; D.2.12 [**Software Engineering**]: Interoperability—*Distributed objects*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*

General Terms: Algorithms, Languanges, Design

Additional Key Words and Phrases: Implicit Invocation Systems, Reactive Systems, Event Programming

---

1. INTRODUCTION

Applications are regularly developed by composing functionality implemented by modules, classes, and/or programs. The composition of functionality can be done by procedural abstraction and implicit invocation [Dingel et al. 1998; Notkin et al. 1993]. When composing functionality by procedural abstraction, also referred to as *explicit invocation* in this document, names that identify a functional component are statically bound to the component implementing the functionality. This is the case of a function in one module invoking another function in another module, or a program in one computer using a Remote Procedure Call (RPC) to invoke functionality implemented by a different program on another computer. In contrast, when composing functionality by implicit invocation, a component *announces* an event. This event announcement triggers the invocation of a functional component. The component announcing the event may or may not be required to know the name nor location of the component triggered by the event.

Frequently, the composition of functionality by explicit invocation is synchronously done: the component invoking the functionality is blocked until the invoked func-

tionality completes its execution. Implicit invocation is often done asynchronously: the component invoking the functionality is not blocked while the requested functionality executes.

Explicit invocation of functionality produces applications that are tightly coupled, in the sense that, changes in the name or location of the components providing functionality may need to be known by the components using the functionality. Implicit invocation, on the other hand, produces applications where the coupling is reduced. This is because in implicit invocation systems, the providers and users of functionality can be decided at run time, without requiring a priori knowledge of their names nor locations. Both, explicit and implicit invocation composition modes, can be used by the same application. Explicit invocation is generally used to implement stepwise functionality. Implicit invocation is generally used when an application needs to react to changes in the environment or within the application itself.

The reduced coupling between functional components in implicit invocation systems causes a relaxation in the development and maintenance constraints imposed on the functional components in the system. Development and maintenance constraints include the requirement to use a specific programming language or development technique. The level of autonomy of the functional components is also a maintenance constraint. In explicit invocation systems, strict development and maintenance constraints force the development of homogeneous functional components, highly dependent on each other. In implicit invocation systems, on the other hand, the relaxation of the development and maintenance constraints allow the implementation of functional components that are autonomous and heterogeneous.

The assumptions developers can make about the functional components in a system are reduced when the development and maintenance constraints are relaxed, therefore, reducing the amount of knowledge developers have about the system. The situation is aggravated in systems with large number of functional components, running in different computers/devices. Such systems are expected in ubiquitous computing environments [Weiser 1993], as well as web applications integrating a large, or unpredictable, number of applications [Geihs 2001; Rosenblum and Wolf 1997].

As observed in [Mühl et al. 2006], hierarchical structuring mechanisms available when developing applications with explicit invocation do not exist for the development of applications that use implicit invocation. In the case of UML [Booch et al. 2005], the treatment of implicit invocation is limited to annotating class diagrams and using interaction diagrams to model how system components react to events. Fiege [Fiege 2005] proposes to use event visibility as a structuring abstraction in implicit invocation systems. The visibility of an event determines the components that can produce and react to the event. Fiege's proposal does not include methodologies for the identification and modeling of the structural and scope properties of an implicit invocation system.

Although it is clear what constitutes composing functionality by implicit invocation, it is not clear in how many different facets that composition can be done. In this document we first look at previous proposals for categorizing implicit invocation systems. We then propose a categorization that combines and extends previous

published categorizations. Section 3 looks at whether components share or not a program space. Section 4 categorizes systems based on how events are defined, generated, and consumed. Section 5 looks at how the event model is implemented. In section 6, based on our categorization, we discuss some of the requirements that must be met by software engineering methodologies for the development and maintenance of implicit invocation systems.

2. RELATED WORK

Notkin el al [Notkin et al. 1993] enumerate key design considerations that arise when extending traditional languages with implicit invocation. These design considerations are: (a) the vocabulary used to define events and the location of the event definition; (b) the information that is associated to events; (c) how and when are events bound to the functional components that process them; (d) whether events are announced explicitly or implicitly, and if events are explicitly announced, what procedures exist to announce the events; (e) whether events are delivered to one or all functional components interested in the event; (f) the number of threads of control that exist in the system. Although intended for single program-space applications, most of the design considerations discussed in [Notkin et al. 1993] apply to multi-program implicit invocation systems as well.

A categorization of multi-program invocation systems is done by Meier and Cahill in [Meier and Cahill 2005]. Meier and Cahill look at the *event model* and *event service* of a system as the basis for a taxonomy of distributed event-based programming systems. The event model refers to the view of the event system an application developer must have in order to develop an event-based application. In essence, the event model specifies how an application subscribes to events, and how it produces and delivers the events. The event service refers to the middleware that implements the event model.

Meier and Cahill propose to classify the event model in distributed implicit invocation systems as *peer-to-peer*, *mediator*, and *implicit*. In a peer-to-peer event model, functional components announcing and consuming events communicate directly with each other. In a mediator event model, the communication is made via one or more mediator components. In an implicit event model, functional components consuming events subscribe to a particular event type rather than to another functional component or mediator.

The event service, is categorized by Meier and Cahill according to its *organization*, *interaction model*, and *features*. The organization of the event service determines to the location of the system components. The interaction model is the communication path over which the communication between the system components takes place. The event service features can be *functional* and *non-functional*. The functional features are the event propagation model, event type, event filtering, component mobility, and composition of events. Non-functional features include quality of service (QoS), event ordering, system security, and failure mode.

Most of the work in [Meier and Cahill 2005] focuses on the communication and structural properties of the implicit invocation system, whilst the work in [Notkin et al. 1993] focuses on the design options made to produce the event model for the

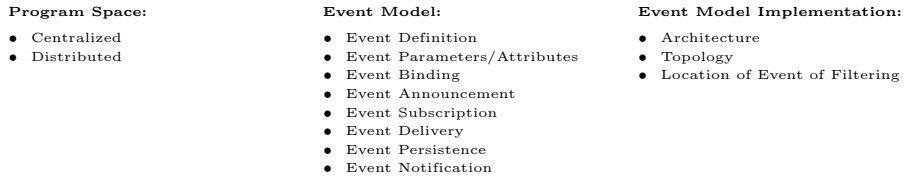

Fig. 1. Major Categorization of Implicit Invocation Systems

system. Neither work provides a classification that is precise enough to categorize the key properties of interest in most implicit invocation systems. [Notkin et al. 1993] lacks categories that would be relevant to distributed implicit invocation systems, while many of the categories in [Meier and Cahill 2005] are made from an architectural-only point of view. With the purpose of providing a general categorization of implicit invocation systems, we combine and complement [Notkin et al. 1993] and [Meier and Cahill 2005] into a extended categorization summarized in Figure 1, and discussed in the remainder of this document. The extended categorization will be used to identify the key properties that need to be considered when studying the development, verification, and evolution of implicit invocation systems.

3. PROGRAM SPACE

The program space category refers to whether or not the functional components generating and reacting to events share the same program space. *Centralized* implicit invocation occurs when the functional components are within the same program space. *Distributed* implicit invocation occurs when the components run on different program spaces, possibly on different computers.

Examples of centralized implicit invocation system include the Abstract Window Toolkit's (AWT) Java Delegation Event Model [SUN-AWT 1997], AspectJ [Kiczales et al. 2001], and the Ada extension proposed by Garlan and Scott [Garlan and Scott 1993]. In the Java Delegation Model, events are objects belonging to a `java.util.EventObject` superclass. Events are dispatched from a source object to a listener object that has registered its interest in the event. In AWT, both source and listener objects are part of the same program space.

AspectJ is a general-purpose Java implementation of aspect-oriented programming (AOP) [Kiczales et al. 1997]. In AOP, functional components called "advice" are invoked when a program reaches certain execution points, known as *pointcuts*. In AspectJ, both the aspect code and the base code triggering the aspects share the same program space.

In [Garlan and Scott 1993], interfaces for Ada packages are extended with the declaration of the events the package generates. The implicit invocation extensions are precompiled into code that executes in the same program space as the rest of the program.

Examples of distributed implicit invocation systems are the CORBA Event Service [CORBA-ES 2004], Hermes [Pietzuch 2004], Information Bus [Oki et al. 1993], the Cambridge Event Architecture (CAE) [Bacon et al. 2000], the Scalable Internet

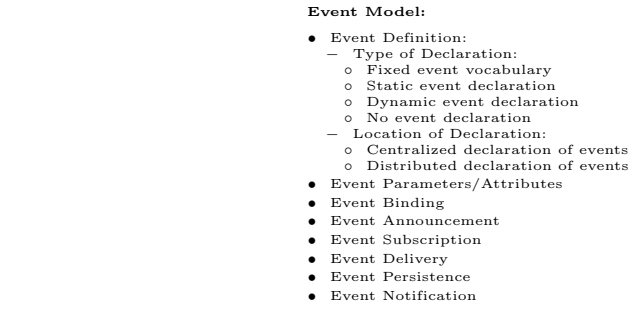

Fig. 2. Event Definition Sub-Category

Event Notification Architecture (SIENA) [Carzaniga et al. 2001], Gryphon [Banavar et al. 1999; Aguilera et al. 1999], and Elvin [Segall and Arnold 1997].

4. EVENT MODEL

The event model determines how events are defined, generated, and consumed. In [Meier and Cahill 2005], the event model is categorized structurally, based on the interactions between components in the event system. In contrast, [Notkin et al. 1993] focuses on the design options available for the definition, generation, and delivery of events. The result is a categorization of some of the essential attributes in an event model.

The categorization of the event model here presented is based on the work in [Notkin et al. 1993]. Specifically, sub-categorization of the event definition (Section 4.1), event parameters (Section 4.2), and event binding (Section 4.3) are exactly those in [Notkin et al. 1993]. The event announcement sub-categorization (Section 4.4) is partially from [Notkin et al. 1993], as well as the delivery model in the sub-categorization of the event delivery (Section 4.6). All other categorizations, although not in [Notkin et al. 1993], are here considered in order to understand the essence of the event model in an implicit invocation system.

4.1 Event Definition

4.1.1 *Fixed Event Vocabulary.* There is a *fixed event vocabulary* when the set of events that can be defined is determined by the system (Figure 2). Examples of implicit invocation systems with a fixed event vocabulary are the the signal notification system used in UNIX for inter-process communication [CSRG 1986], the Java Message Service (JMS) [SUN-JMS 2002], and AOP ([Kiczales et al. 1997]).

In UNIX based operating systems, a process can announce a signal to another process. Signals are used to notify a process that some condition has occurred [Stevens 1992]. A previously registered callback routine, referred to as a signal handler, is invoked by the process receiving the signal announcement. A signal is identified by a positive integer, and only a predetermined set of numbers represent valid signals.

Messages are generated and received by functional components in JMS. Messages must be of one of six types: `Text Message`, `Map Message`, `Bytes Message`, `Stream Message`, `Object Message`, and `Message`. No new message types can be defined.

The set of events in AOP corresponds to the points in the execution of a program where advice can be executed. As previously mentioned, advice is the name given to the functional component reacting to an AOP event.

4.1.2 *Static Event Declaration.* When new events can be defined, but the set of events must be known when the program is generated, the system has *static event declaration*. The extension to Ada proposed in [Garlan and Scott 1993] is an example of static event declaration. In [Garlan and Scott 1993], interfaces for Ada packages are extended with the declaration of the events the package generates. The name of the package methods to be invoked when specific events are generated are also part of the implicit invocation declaration. Figure 3 is an example of such event definition. In the example, the package `Accounts` declares that it generates the event `NewCustomer`. The `Accounts` package also declares that it shall be notified when the event `BankPayment` is generated. Similarly, the package `BankGateway` declares that it generates the event `BankPayment`, and that it is interested in being notified when the event `NewCustomer` is generated.

```
for Package_Accounts
   declare Event_NewCustomer
       CustomerNum:  Integer,
       CustomerName: String (1 .. 30)
   when Event_BankPayment => Method_ProcessPayment TransId
end for Package_Accounts
for Package_BankGateway
   declare Event_BankPayment
       TransId: Integer;
   when Event_NewCustomer => Method_InformBank CustomerNum
end for Package_BankGateway
```

Fig. 3. Event Definition Sub-Category

The event declaration in this Ada extension is then precompiled and Ada-only code is generated. An executable program is built by compiling and linking the generated Ada code with the rest of the application code.

4.1.3 *Dynamic Event Declaration.* When events can be defined at run time, the system has *dynamic event declaration*. An example of an implicit invocation system with dynamic event declaration is Hermes [Pietzuch 2004]. In Hermes, the functional components that react to events are programs. Programs subscribe and unsubscribe to events via event brokers. Programs generating events register event types with the event brokers. Once an event type is registered, a program can announce events of the registered event type. Event types can be registered and unregistered dynamically.

4.1.4 *No Event Declaration.* In some implicit invocation systems, events are announced with *no event declaration*, for example by just announcing an arbitrary string or a list of strings. An example of such a system is the Java Event-Based Distribution Architecture (JEDI) [Cugola et al. 1998]. In JEDI, an event is an ordered set of strings. The first string corresponds to the event name. The rest of the strings are the event parameters. JEDI Functional components generating and receiving events are called "active objects". An active object announces an event by invoking the method `sendEvent`. The `sendEvent` method receives as its

argument the ordered set of strings representing the event. Active objects wishing to be notified of events, must subscribe to the events they are interested in. An event subscription specifies the name of the event and, possibly, filtering arguments on the event parameters.

A system that supports both dynamic event declarations and no event declarations is SIENA [Carzaniga et al. 2001]. SIENA can operate under what the SIENA creators call "subscription-based semantics" and "announcement-based semantics". Irrespectively of the operation mode, an event is a set of typed attributes. Objects announce events by invoking a `publish` call. Under subscription-based semantics, functional components interested in being notified, subscribe to events by invoking a `subscribe` call. A filtering expression specifying values for the event attributes is passed as parameter of the `subscribe` call. A functional component is notified of the occurrence of an event if the filtering expression specified in the subscription call matches the event notified via a `publish` call. When operating under subscription-based semantics the SIENA system requires no event declaration. In contrast, when operating under announcement-based semantics, functional components generating events, need to register the events they will be generating by invoking an `advertise` call. An `unadvertise` call is used by functional components to inform that a given event will no longer be generated. Under announcement-based semantics the SIENA system has dynamic event declaration.

4.1.5 *Location of the Event Declaration.* When events are declared, the implicit invocation system may require that all declarations be done at a given specialized location. In this case, such a system implements *centralized* declaration of events. Alternatively, an implicit invocation system may allow the event declarations to occur at multiple locations. In this former case, the system implements *distributed* declaration of events.

An example of an implicit invocation system where the events are declared in a central location is Yeast [Krishnamurthy and Rosenblum 1995]. In Yeast events are generated when object attributes change. The system provides a predefined set of objects and attributes (e.g. Object `file`, attributes `file name`, `creation time`, `modification time`, etc). Users can declare new events by defining objects and their attributes via commands that are executed on the client side. These declarations are processed and stored by the Yeast server.

An example of an implicit invocation system with distributed event declaration is Hermes [Pietzuch 2004]. In Hermes several event brokers, possibly running at different locations in the system, process the event registration and subscription/unsubscription requests from the functional components generating and processing the events.

4.2 Event Parameters

4.2.1 *No Parameters.* Implicit invocation systems may allow the association of parameters/attributes to the event (Figure 4). If parameter passing is not supported, relevant information related to the event must be deduced from the event name or be retrieved, via global variables, shared memory/database space, explicit invocation, or any other means by the component reacting to the event announce-

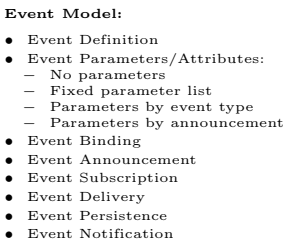

Fig. 4. Event Parameters Sub-Category

ment. The signal notification system in UNIX [CSRG 1986] is an example of an implicit invocation system where there are no parameters associated with the events. Processes using signals, must develop their own protocols for retrieving any relevant information associated to the occurrence of a signal. Moreover, the protocol used to retrieve the relevant information is separate from the signal notification system itself.

4.2.2 *Fixed Parameter List.* Other systems may associate the same *fixed parameter list* to each event. This is the case of the Java Message Service [SUN-JMS 2002]. All messages in JMS have a message header, a set of message properties, and a message body. Any application information relevant to the message must be coded in the message body by the functional component generating the message. Functional components receiving the message must know how to decode the information in the message body. JMS supports the codification of data in the message body as a string (for example representing an XML document), a list of attribute/value pairs, a serialized object, a stream of bytes, or a stream of Java primitive data types.

Another example of a system that associates the same fixed parameter list to every event is JINI [SUN-JINI 2003]. JINI allows Java objects to be notified of events occurring on other, possibly remote, objects. Every event has four associated parameters: (1) an attribute identifying the type of event; (2) a reference to the object on which the event occurred; (3) a sequence number identifying the instance of the event type; (4) a hand-back object. The hand-back object is a Java object that was originally specified by the functional component receiving the event when the component first registered its interest on the event.

Similarly to JINI, the Corba Event Service [CORBA-ES 2004] supports a "Generic Event Communication" mode where all events have a single attribute of type `any` corresponding to the event data. Functional components in the Corba Event Service must agree on how required event information, if any, is coded into the event data attribute.

4.2.3 *Parameters by Event Type.* In some systems, events are instances of a certain *event type*. In this case the set of attributes associated to the event is determined by the the type of the event. For example, Hermes [Pietzuch 2004] uses XML Schema specifications [W3C 2004] to represent event type definitions. The event type definitions are then used in Hermes to type check event subscriptions and publications. EAOP, the model and tool described in [Douence and Südholt

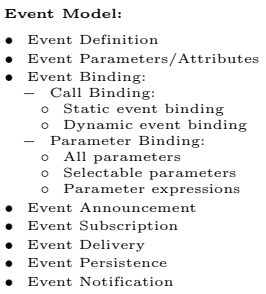

Fig. 5. Event Binding Sub-Category

2002] for event-based aspect-oriented programming, is another example of a system where the number and type of attributes associated to an event are determined by the type of the event. EAOP has a fixed event vocabulary (see 4.1) with four types of events: `method call`, `method return`, `constructor call`, and `constructor return`. Depending on the type of the event, there are a number of attributes associated to each event. For example, some of the attributes associated to events of type `method call` are the method name, the values of the arguments passed to the method, and the depth of the execution stack.

CAE, the Cambridge Event Architecture [Bacon et al. 2000], is also an implicit invocation system where the event parameters depend on the event type. An event occurrence is represented in CAE as the instance of a given event class. Event types are defined using an Interface Definition Language (IDL). Functional components interested in events of a specific class, specify a value or wildcard for each attribute of the given event class.

4.2.4 *Parameters by Announcement.* Alternatively, any number of parameters and their types may be determined at the time the event is announced. In this case, two different announcements of events of the same type may have different parameters. An example of such a system is JEDI [Cugola et al. 1998]. In JEDI, an event is an ordered set of strings. The first string is the name of the event, and the reminder strings are the event parameters. There is no guarantee that two events with the same name represent the same event type. Similarly, there is no guarantee that all announcements of events of the same type are done with same-sized ordered sets of strings.

4.3 Event Binding

A binding determines which functional components are to be invoked when an event is announced (Figure 5).

4.3.1 *Call Binding.* In *static event binding*, the functional components that react to an event are predetermined at compile time. An example of static event binding is the proposal made by Garlan and Scott in [Garlan and Scott 1993] to extend ADA with implicit invocation. As shown in the example in Figure 3, an event specification language is used to indicate, for each package, the events the package wishes to be notified about, and the methods that are to be invoked when

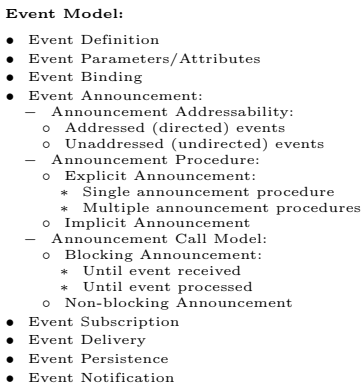

Fig. 6. Event Announcement Sub-Category

the event occurs. Hence, the functional components generating the events, and the functional components reacting to each event, are known at compile time.

In *dynamic event binding*, bindings between events and functional components that react to the events can be established or terminated dynamically. Most systems support dynamic event binding. Examples include Hermes [Pietzuch 2004] and SIENA [Carzaniga et al. 2001], In both Hermes and SIENA, the functional components that react to events are programs. Programs subscribe and unsubscribe to events via event brokers.

4.3.2 *Parameter Binding.* Another classification criteria related to event binding, is based on how event parameters are translated to subscriber's parameters. One option is to pass *all parameters* in the event to the functional component reacting to the event. A different option is to allow a component to *select the event parameters* of interest. Alternatively, a component may specify *parameter expressions* and receive the result of the evaluation of the expressions. All but one of the reviewed implicit invocation systems that support event parameters choose to pass all the event parameters to the components reacting to the event. The exception is the Ada extension in [Garlan and Scott 1993]. In [Garlan and Scott 1993], functional components reacting to events are package methods. The name of the event parameters to pass to the methods reacting to an event is included as part of the event declaration. Not all event parameters need to be bound to method parameters. As an illustration, consider the example in Figure 3. In the example, the package `Accounts` declares the event `NewCustomer` with parameters `CustomerNum` and `CustomerName`. The method `InformBank`, in package `BankGateway`, reacts to the `NewCustomer` event and only binds the `CustomerNum` event parameter to one of its (method) parameters. The other event parameter, `CustomerName`, is not bound to any method parameter.

4.4 Event Announcement

4.4.1 *Announcement Addressability.* Independently of whether the event binding is static or dynamic, the events can be *addressed* or *unaddressed* (Figure 6). Addressing of events, also referred to as directing of events, happens when the an-

nouncer of the event specifies the functional component that will be notified of the event. Dingel et al [Dingel et al. 1998] use this type of announcement in their model for the verification of implicit invocation systems. In [Dingel et al. 1998], functional components, called methods, send events via an announcement call where the first argument of the call is the intended recipient of the event, and the second parameter of the call is the event data itself.

The UNIX signal notification system also requires functional components to direct the events [CSRG 1986]. In UNIX, signals are events with no associated information. The functional components generating and processing the signals are called processes. A process is uniquely identified by a numeric process identifier, and a group of processes is uniquely identified by a numeric group identifier. A process invokes the system call `kill(pid, signal)` to send the signal `signal` to the process with identifier `pid`. If `0` is provided as the process identifier, the signal is sent to all processes belonging to the same group as the process generating the signal.

The requirement to address events augments the degree of coupling between event announcers and event consumers. Hence, most implicit invocation systems have unaddressed event announcement, where the component generating the event just announces the event and the system is in charge of directing the event to the components registered for the event.

4.4.2 *Announcement Procedure.* Systems can also be classified based on whether there is one or several *explicit* announcement procedures, or if the announcement of the event is *implicit*. Most systems have explicit announcement procedures that need to be used to generate an event. In some systems there is a unique "announce" or "publish" procedure or method, while in other systems there are several announcements methods. For example, Hermes [Pietzuch 2004], provides two announcement methods named `publishType` and `publishTypeAttr`. The first method announces an event that will trigger the execution of functional components that have subscribed to events of the given event type. The second method triggers the execution of functional components that have subscribed to events of certain type and with certain attribute values.

In implicit invocation systems with implicit announcement, the event is generated as a side effect of executing an instruction or procedure. AOP is an example of an implicit invocation system with implicit event announcement [Kiczales et al. 1997]. In AOP, events are generated when the execution of the program reaches certain points (method calls, control structures, assignments, etc). Another example of implicit announcement is active database systems [Cilia et al. 2003]. In active database systems, functional components, known as database triggers, are invoked as a side effect of the insertion, deletion, or updating of data in the database.

4.4.3 *Announcement Call Model.* When an event is announced, the execution of the functional component announcing the event may be blocked *until the event is received* by all functional components to be notified of the event, or *until the event is processed* by all functional components receiving the event. Alternatively, the announcement of an event may not block the execution of the component announcing the event.

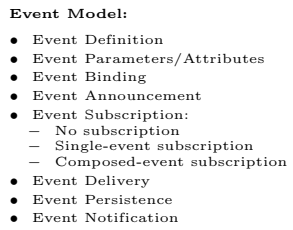

Fig. 7. Event Subscription Sub-Category

In general, centralized implicit invocation system tend to follow a blocking call model for the announcement of the event, with many of the systems blocking until the components processing the event finish their processing of the event. Most distributed implicit invocation systems, on the other hand, provide a non-blocking call model.

The model and tool for event-based aspect-oriented programming (EAOP) presented in [Douence and Südholt 2002] is an example of an implicit invocation system where the code generating events is suspended until all functional components, advise in this case, process the event. An execution monitor in EAOP tracks the events generated during the execution of a base program on which advise has been defined. When an event is generated, the execution of the base program is suspended and the execution monitor sequentially invokes every advise associated to the given event. Once each advise has executed, the monitor gives control back to the base program.

Most active databases support a blocking call model as well. Programs inserting, deleting, or updating data are blocked until the code reacting to these database operations (*database triggers*) complete their execution. An interesting feature is that, in active database, the changes made to the data in the database, by the functional components generating the events and the functional components reacting to the event , are typically part of a single database transaction. Logically, all or none of the modifications to the database are carried out – independently of whether the modification was performed by the functional component generating the event or the functional component reacting to the event.

4.5 Event Subscription

Functional components may or may not be required to register their interest to be notified when events are announced (Figure 7). If there is no requirement to register for events, also referred to as "subscribing" for events, event announcements may be broadcasted to all components in the system. Alternatively, event announcements may be registered in a shared memory space that is accessed by components wishing to inquiry if a certain event has been announced. Linda [Gelernter 1985] is an example of an implicit invocation system where functional components are not required to announce their interest for events. In Linda, implicit invocation is done via a shared memory region called the *tuple space*. Functional components, processes in the case of Linda, generate tuples that are stored in the tuple space.

Other processes monitor the tuple space and can read and, optionally, remove the tuples that have been added to the tuple space.

Most implicit invocation systems require that functional components register for the events they wish to be informed about. In systems with *single-event subscription*, there is a subscription procedure that must be invoked for each event of interest. In systems with *composed-event subscription*, a functional component can express its interest to be notified when a composition of events occurs. In this latter case, the implicit invocation system provides a method for functional components to express an event composition condition. When supporting composed event subscription, implicit invocation systems typically provide a language that allows the specification of temporal conditions on the event occurrences [Carlson and Lisper 2004; Konana et al. 2004; Liebig et al. 1999; Mansouri-Samani and Sloman 1997].

SIENA is an example of an implicit invocation system that supports composed event subscription. A filtering condition f, on the event type and event attribute values, can be specified in SIENA as part of the event subscription call. A pattern $f_1, f_2, ..., f_n$ can also be specified, where each filtering condition f_i may apply to a different event type. Such subscription indicates that the functional component running the subscription operation shall be notified if events $e_1, e_2, ..., e_n$ are generated, such that:

- e_i occurs after e_{i-1} for all $2 \leq i \leq n$
- The filtering condition f_i is true when evaluated for the event e_i, with $1 \leq i \leq n$

The language proposed in [Konana et al. 2004] for the specification of composite events allows the identification of a sequence of events that satisfy or violate timing and event attribute-value constraints. Based on real time logic (RTL), the specification of conditions of the type "the third occurrence of the event of type e_t after time t must have a value v for attribute $e_t.attr$" are possible in the proposed language. [Konana et al. 2004] also assumes the existence of data repositories in the form of relational databases. Hence, conditions on the data stored in the data repositories are also part of the event composition language.

4.6 Event Delivery

4.6.1 *Delivery Model.* Once an event is announced, the system must select the functional components that will receive the event (Figure 8). In *single delivery* of events, an event is delivered to only one of the functional components interested in the event. In *full delivery*, the event is delivered to all the functional components interested in the event. Implicit invocation systems implementing addressed events (Section 4.4), typically support single delivery of events. For example, the UNIX signal notification system ([CSRG 1986]), operates in single delivery mode when a signal is sent to a process. Full delivery operation occurs when a signal is addressed to a group of processes.

Linda [Gelernter 1985], supports both single and full delivery. Events, represented as tuples in Linda, are stored in a tuple space that is accessible to all functional components. A component reacting to an event has the option of removing the event from the tuple space. To guarantee that only one functional component

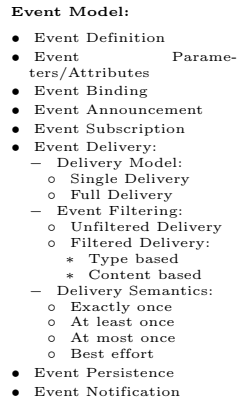

Fig. 8. Event Delivery Sub-Category

accesses the event, semaphores and other process synchronization techniques, can be modeled in Linda.

4.6.2 *Event Filtering.* Some systems allow the *filtering* of events. In systems with event filtering, an event is delivered to a functional component only if the component is interested in the event, and an expression associated with the interest of the component for the event holds. The expression can be based on the *type* of the event, or it can be based on the *content* of the event. Type based event filtering is also known as "subject" or "topic" based. Content based filtering is also known as "attribute" based filtering. As discussed in Section 4.5, some systems allow event filtering expressions to refer to more than one event [Carlson and Lisper 2004]. These composite event expressions may specify patterns of events and, in some cases, include temporal constraints [Carzaniga et al. 2001; Konana et al. 2004]. The actual event filtering may occur at central location [Konana et al. 2004], at each functional component [Oki et al. 1993], or at specialized event servers [Carzaniga et al. 2001].

4.6.3 *Delivery Semantics.* An event may be delivered *exactly once*, *at least once*, *at most once*, or in *best effort*, there are no delivery guarantees. Exactly-once and at-most-once delivery are usually more difficult to implement than the other options, since the implementation may require the use of transactional protocols. An important part of the delivery semantics, is whether or not there are order guarantees in the delivery of events. Some systems may provide order guarantees within events of a single event type. In these systems it is possible to identify, for two events of the same type, which one was generated before the other, or whether both were generated at the same time. Other systems may provide system-wide ordering guarantees. In this later case, it would be possible to identify, for any two events, even if not of the same type, which one was produced before the other.

IBM's Gryphon project [Bhola et al. 2002], implements a protocol that guarantees exactly once delivery if the functional components being notified of the events maintain their connectivity to the system. The protocol models a knowledge graph where nodes, named routing brokers, represent functional components in charge of

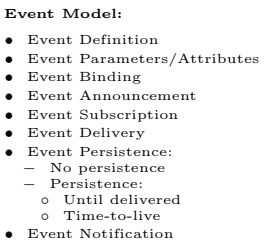

Fig. 9. Event Persistence Sub-Category

routing events. Arcs in the graph represent filtering conditions on the events. The filtering conditions are used to split the routing of events between routing brokers. The graph is dynamically adjusted in case of node/network failures. Further refinement of the protocol is presented in [Zhao et al. 2004]. In this later work, the protocol is extended to guarantee not only exactly once delivery, but ordered delivery of events matching a single subscription. Functional components named subscription brokers, receive subscription requests from other functional components in the system wishing to be notified of events. Ordered delivery is accomplished by associating, with each event generated, a vector containing information for each subscription request related to the event. Virtual timers at subscription brokers are used to identify, from a stream of events matching a subscription, the first event in the stream after which every single event is guaranteed to be delivered, in order, to the functional component that subscribed to the events. To accomplish this functionality, the protocol propagates subscription information from subscription brokers to the functional components generating the events. Event information is propagated from the functional components generating the events to the subscription brokers. In this refinement of the original protocol, routing brokers are in charge of routing, both, the subscriptions and events.

4.7 Event Persistence

When an event is delivered, the intended recipients of the event may not be available to receive the event (Figure 9). In this case, the implicit invocation system may choose to save the event and attempt the delivery at a later time, or it may choose to abort the delivery of the event to the functional component that is unavailable. When *persistence* of events is supported by the system, the event may be maintained in the system until all intended recipients receive the event. Alternatively, the event may be maintained in the system until a *time-to-live* expires. The time-to-live may be the same for all events, it may be determined by the type of event, or it may be specified when the event is announced.

Most implicit invocation systems deliver events only to functional components available at the time the event is generated. Linda [Gelernter 1985] and other systems based on data repositories are exceptions. In Linda, events represented as tuples are stored in a common area until they are explicitly removed, either by the functional component generating the event, or by any functional component reacting to the event. Similarly, in Oracle's Advanced Queuing [Oracle 2005], events in the form of messages are stored in Oracle's relational database. An expiration

Event Model:

- Event Definition
- Event Parameters/Attributes
- Event Binding
- Event Announcement
- Event Subscription
- Event Delivery
- Event Persistence
- Event Notification:
  - Synchronism:
    - Immediate notification (synchronous)
    - Deferred notification (asynchronous)
  - Provision Method:
    - Push:
      - Wait loop
      - Interrupt / Callback
    - Pull:
      - Blocking notification
      - Non-blocking notification

Fig. 10. Event Notification Sub-Category

interval can be individually associated to each generated event. When no expiration interval is associated with the event, a functional component must explicitly remove the event from the system.

As illustrated, the delivery semantics (Section 4.6) of the system highly influence the event persistence supported by the system. With the exception of best effort delivery, the system must support some kind of event persistence.

4.8 Event Notification

When an event is announced, the components receiving the event may be *immediately notified* or, in *deferred notification*, notified at a later time (Figure 10). When the system provides immediate notification of events, the events are usually *pushed* to the components interested in the the events. The receiving components may implement a *wait loop*, or their execution may be *interrupted* by calling a previously registered callback routine. The UNIX signal notification system [CSRG 1986] implements immediate notification with interrupted execution. When a UNIX process receives a signal, its execution is interrupted and a previously registered callback routine is executed. Processes can explicitly ignore notifications of certain signals. This would be equivalent to a functional component unsubscribing from certain types of events.

When the event notification is deferred, the functional component may *pull* the system for information about any events of interest that may have occurred since any previous pull call. The pull operation may be *blocking* or *non-blocking*. An example of a system where events are pulled by functional components reacting to the events is the Corba Event Service Notification CESN [CORBA-ES 2004]. In CESN, a blocking `pull` call is used by a functional component to retrieve an event generated by an event producer. If no events have been generated, the execution of the functional component is suspended until an event is available. An alternate `try_pull` call in CESN may be used when the functional component reacting to the event does not wish to be blocked in the absence of events. The Java Message Service JMS [SUN-JMS 2002] also supports blocking event polling. Blocking is supported via a `receive` method call. A mode of event pushing is also supported in JMS via an extra functional component called a message listener. Upon arrival of an event, the message listener invokes a previously registered call-back method.

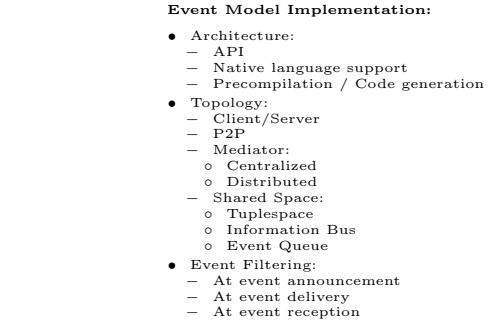

Fig. 11. Event Model Implementation Category

5. EVENT MODEL IMPLEMENTATION

In the classification here presented, the event model implementation is characterized by the architecture of the event system, the system topology, and the location of the event filtering. Of these subcategories, the topology and location of event filtering are inspired by the work in [Meier and Cahill 2005].

5.1 Architecture

5.1.1 *API.* The event model may be implemented as an *API* that must be linked to, or loaded by, the components in the system (Figure 11). Examples include Java's Abstract Window Toolkit (AWT) Delegation Model [SUN-AWT 1997], .Net's Delegation Event Model [DOTNET 2005], Java's Message Service [SUN-JMS 2002], and JINI's Distributed Events Specification [SUN-JINI 2003].

When several implementations of the API exist for different programming languages, functional components developed in one language can react to events generated by components developed in another language. Hermes [Pietzuch 2004] is an example of such a system. In Hermes, functional components generating and reacting to events link to language dependent API libraries. These functional components are called "clients" in Hermes. Brokers in charge of routing events and subscriptions implement language independent functionality that is accessible via the client APIs. The actual API calls and exchanged data are represented in Hermes using XML Schema specifications [W3C 2004]. The language-dependent APIs implement a binding layer between the data/functionality represented in XML and data/functionality available in a given language.

5.1.2 *Native Language Support.* Some languages provide *native language support* to implicit invocation. In this case, functional components developed in one language may or may not be able to interact with components developed in a different language. FLO [Ducasse 1997], an object-oriented functional language, is an example of a programming language with native support for implicit invocation. FLO supports three object oriented entities: objects, classes, and connectors. Connectors are special objects that connect other "participant" objects. The main function of a connector is to manage the message passing between participant objects. The specification of connectors include information on how each message

affects participant objects. In particular, what functionality of the participant object must be invoked when a message is delivered to the object. Moreover, if guard conditions are not met by the messages, connectors have the ability to block messages from reaching the participant objects. Connectors themselves can participate in connections managed by other connectors.

5.1.3 *Precompilation / Code Generation.* Yet another option is to extend programming languages. In this latter case, functional components are developed by combining instructions in a base language with instructions specific to the implicit invocation extension. The source code containing both types of instructions is then processed by a *precompiler or code generator* that generates source code containing instructions in the base language only. The extension to Ada proposed in [Garlan and Scott 1993] is an example of such a system (see Section 4.1). Another example is AspectJ [Kiczales et al. 2001], where application code containing Java and AspectJ directives is compiled into a Java-only program. In the CORBA Event Service [CORBA-ES 2004], an specialized Interface Definition Language (IDL) is used for the definition of the events and the bindings between events and functional components. The declarations made with the IDL are then are compiled into stubs that are linked to programs written in traditional programming languages.

It is possible to have implicit invocation systems where the above techniques are combined. For example, a system may provide a definition language for the declaration of events, and an API for the announcement and consumption of events. In such a system, the declaration of events may be used to generate code that is linked to the functional components invoking the API.

5.2 Topology

5.2.1 *Client/Server.* The topology of an implicit invocation system indicates the types of interactions between the functional components in the system. In a *client/server* system, events generated by client functional components are communicated to a single server component that reacts to the events. Yeast [Krishnamurthy and Rosenblum 1995] is an example of such a system. In Yeast, a server receives notification of attribute changes in non-temporal objects running on client functional components. When the Yeast server is notified about the change on a non-temporal object, it decides if the change should trigger an action. This decision is based on event specifications submitted to the server by client functional components. Actions associated to event specifications are a sequence of commands that are executed on the computer where the Yeast server runs. Hence, the Yeast server is, effectively, the functional component determining if an event has been generated, and if so, it is also the functional component reacting to the event.

The E-Brokerage architecture defined in [Konana et al. 2004] is also an example of a client/server topology. In this architecture, events are streamed into an event server. The event server establishes if conditions on the event stream are satisfied. As previously discussed in Section 4.5, the conditions specify timing and event attribute-value constraints. Actions associated to a given condition are executed when the condition is satisfied by the event stream.

5.2.2 *Peer-to-Peer.* In *peer-to-peer* (P2P) systems, a component announcing an event interacts directly with the components that react to the event. Similarly, a functional component wishing to receive events must directly inform the component generating the events of its interest. An implicit invocation system with P2P topology may or may not be implemented on top of a P2P routing system. Similarly, an implicit invocation system implemented on top of a P2P routing system may or may not have a P2P topology. Hermes [Pietzuch 2004], for example, is implemented on top of Pastry [Rowstron and Druschel 2001]. Although Pastry is a P2P routing system, as discussed later in this section, Hermes itself does not implement a P2P topology. This is because, in Hermes, functional components generating events do not interact directly with the functional components reacting to the events.

5.2.3 *Mediator.* In *mediator* systems, components announcing and processing events communicate indirectly via specialized components called mediators. The mediators process event subscriptions and event announcements, and filter and dispatch events. Hermes [Pietzuch 2004] is an example of such a system. In Hermes components called *event brokers* process subscription requests and disseminate events to interested components. A component wishing to announce events registers with an event broker close-by. The event registration includes a description of the type of the event and its attributes. The event broker then advertises the event by providing the event registration when requested by event consumers. Event consumers register their interest on particular events with their closest event broker. Once the event is registered, the component may announce events of the registered type to the event broker. The event broken then disseminates the announced event to the functional components that have registered their interest in the event.

Mediator systems can be further categorized as *centralized* or *distributed*. A centralized mediator system has one or multiple mediators all running on the same server. In a distributed mediator system, the mediators are running on more than one computer. Moreover, mediators can be *hierarchically* organized. If hierarchically organized, the hierarchical structure is used to propagate event registrations and announcements.

Jedi [Cugola et al. 1998] is an example of an implicit invocation system with a centralized mediator. In Jedi, events are delivered via a centralized event dispatcher. Functional components generating events, communicate the events to the event dispatcher. Upon communication of an event, the event dispatcher decides what functional components are interested in the event, and delivers the event to them. Systems with distributed mediators include Siena [Carzaniga et al. 2001], Gryphon [Banavar et al. 1999; Aguilera et al. 1999] and Hermes [Pietzuch 2004].

The system described in [Tam et al. 2003], is an example of an implicit invocation system with a hierarchical mediator architecture. In this system, event notifications are multicasted via a tree-like structure of functional components. For a given tree, the functional component at the root manages subscriptions and notifications of events of a certain event type. Functional components at the leafs are interested in receiving notifications when events of the type managed by the root of the tree are produced.

5.2.4 *Shared Space.* A fourth type of topology occurs when the interaction between components happens via a *shared space*. Shared space interaction may take the form of a *tuple space*, and *information bus*, or an *event queue*. A tuple space is a collection of tuples that can be accessed by the components of the system [Gelernter 1985]. Tuples can be added, read, and removed from the collection. Reading of tuples is done by providing a template. Tuples matching the template are returned as the result of the read operation. In an information bus [Oki et al. 1993], components are linked via a logical communication bus. Events are announced to the bus and broadcasted to all components.

In an event queue, events are appended to a shared queue when the events are announced. There may be a unique queue, or a different queue for each event type. Events may be addressed (Section 4.4) in which case, queue managers are in charge of delivering the events that are queued. If the events are unaddressed, components interested in the events must retrieve the queued events. Examples of implicit invocation systems implementing event queues are IBM's MQSeries [Gilman and Schreiber 1996], and Java's Message Service (JMS) when operating in point-to-point mode [SUN-JMS 2002].

5.3 Event Filtering

When an implicit invocation system supports event filtering (Section 4.6), the filtering may happen when the event is announced, at a mediator component when the event is being delivered, or locally at each component registered for the given event type. In implicit invocation systems where events are produced in high numbers or in a constant stream fashion, the performance and location of the event filtering functionality is critical for the overall performance of the system. Recent database research on data streams focuses on the filtering functionality for event streams [Babu and Widom 2001; Plale and Schwan 2003].

Yeast [Krishnamurthy and Rosenblum 1995] is an example of an implicit invocation system where filtering occurs when the event is announced. As discussed in Section 5.2, a Yeast server component, decides, based on an event specification that may include filtering conditions, if an event has occurred or not.

Some implicit invocation systems where filtering occurs at mediators include [Tam et al. 2003] and Gryphon [Aguilera et al. 1999]. In [Tam et al. 2003], a distributed hash table implementation is used to store indexing information on event attributes. The indexing information itself may be stored on several nodes in the system. Event filtering occurs by applying indexing information to verify filtering conditions. The nodes in the system are the functional components generating and reacting to events. Hence, the work to filter events is distributed among all the functional components in the system.

In Gryphon, a functional component generating an event communicates the event to an assigned event broker. The event broker is then in charge of doing the event filtering and, based on the results of the filtering, decide which functional components need to be notified of the occurrence of the event.

In the Information Bus architecture, described in [Oki et al. 1993], each component registered for a given event type, locally performs event filtering. The Information Bus broadcasts events to all functional components attached to a logical

shared area. Functionality local to each component implements the filtering and decides whether or not to notify the local functional component that reacts to the events.

6. REQUIREMENTS OF IMPLICIT INVOCATION SYSTEMS

Since there are few systems that compose functionality by implicit invocation only, most implicit invocation systems use some form of explicit invocation as well. Hence, methodologies and abstractions developed for explicit invocation systems are regularly used to model and develop implicit invocation systems. The problem with this approach, is that the implicit invocation aspect of the application or system is not abstracted, modeled and structured as well as the rest of the application.

Moreover, the functionality being composed in an implicit invocation system may be independently developed and maintained, potentially using different languages and methodologies. This reduced coupling causes developers to only have partial knowledge of the functionality of the whole system.

Ideally, abstractions and methodologies for implicit invocation should be applicable to the diverse event models and event model implementations discussed in this document. They should also address the reduced coupling and heterogeneity common in some implicit invocation systems. At this time it is not clear what abstractions should be used to structure implicit invocation systems.

7. CONCLUSIONS

Implicit invocation systems have been categorized based on the way events are declared, announced, and how functional components subscribe to events. The actual semantics used in the delivery of the events has been also considered in the categorization, as well as the implementation options for the implicit invocation functionality. Figure 12 summarizes the categorization. The goal of the categorization is the identification and understanding of the essential characteristics of implicit invocation systems.

REFERENCES

Aguilera, M. K., Strom, R. E., Sturman, D. C., Astley, M., and Chandra, T. D. 1999. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC)*. ACM Press, 53–61.

Babu, S. and Widom, J. 2001. Continuous queries over data streams. *SIGMOD Rec. 30,* 3, 109–120.

Bacon, J., Moody, K., Bates, J., Hayton, R., Ma, C., McNeil, A., Seidel, O., and Spiteri, M. 2000. Generic support for distributed applications. *Computer 33,* 3, 68–76.

Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R., , and Sturman, D. 1999. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society, Washington, DC, USA, 262.

Bhola, S., Strom, R. E., Bagchi, S., Zhao, Y., and Auerbach, J. S. 2002. Exactly-once delivery in a content-based publish-subscribe system. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*. IEEE Computer Society, Washington, DC, USA, 7–16.

Booch, G., Rumbaugh, J., and Jacobson, I. 2005. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional.

Program Space:

- Centralized
- Distributed

Event Model:

- Event Definition:
  - Type of Declaration:
    - Fixed event vocabulary
    - Static event declaration
    - Dynamic event declaration
    - No event declaration
  - Location of Declaration:
    - Centralized declaration of events
    - Distributed declaration of events
- Event Parameters/Attributes:
  - No parameters
  - Fixed parameter list
  - Parameters by event type
  - Parameters by announcement
- Event Binding:
  - Call Binding:
    - Static event binding
    - Dynamic event binding
  - Parameter Binding:
    - All parameters
    - Selectable parameters
    - Parameter expressions
- Event Announcement:
  - Announcement Addressability:
    - Addressed (directed) events
    - Unaddressed (undirected) events
  - Announcement Procedure:
    - Explicit Announcement:
      - Single announcement procedure
      - Multiple announcement procedures
    - Implicit Announcement
  - Announcement Call Model:
    - Blocking Announcement:
      - Until event received
      - Until event processed
    - Non-blocking Announcement
- Event Subscription:
  - No subscription
  - Single-event subscription
  - Composed-event subscription
- Event Delivery:
  - Delivery Model:
    - Single Delivery
    - Full Delivery
  - Event Filtering:
    - Unfiltered Delivery
    - Filtered Delivery:
      - Type based
      - Content based
  - Delivery Semantics:
    - Exactly once
    - At least once
    - At most once
    - Best effort
- Event Persistence:
  - No persistence
  - Persistence:
    - Until delivered
    - Time-to-live
- Event Notification:
  - Synchronism:
    - Immediate notification (synchronous)
    - Deferred notification (asynchronous)
  - Provision Method:
    - Push:
      - Wait loop
      - Interrupt / Callback
    - Pull:
      - Blocking notification
      - Non-blocking notification

Event Model Implementation:

- Architecture:
  - API
  - Native language support
  - Precompilation / Code generation
- Topology:
  - Client/Server
  - P2P
  - Mediator:
    - Centralized
    - Distributed
  - Shared Space:
    - Tuplespace
    - Information Bus
    - Event Queue
- Event Filtering:
  - At event announcement
  - At event delivery
  - At event reception

Fig. 12. Categorization of Implicit Invocation Systems

Carlson, J. and Lisper, B. 2004. An event detection algebra for reactive systems. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*. ACM Press, New York, NY, USA, 147–154.

Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. 2001. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst. 19,* 3, 332–383.

Cilia, M., Haupt, M., Mezini, M., and Buchmann, A. 2003. The convergence of aop and active databases: towards reactive middleware. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*. Springer-Verlag New York, Inc., New York, NY, USA, 169–188.

CORBA-ES. 2004. Corba event service, version 1.2. `http://www.omg.org/technology/documents/formal/event_service.htm`. Object Management Group.

CSRG. 1986. *Unix Programmer's Reference Manual*, 4.3 BSD ed. Computer Systems Research Group, University of California, Berkeley.

Cugola, G., Nitto, E. D., and Fuggetta, A. 1998. Exploiting an event-based infrastructure to develop complex distributed systems. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*. IEEE Computer Society, Washington, DC, USA, 261–270.

Dingel, J., Garlan, D., Jha, S., and Notkin, D. 1998. Reasoning about implicit invocation. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM Press, New York, NY, USA, 209–221.

DOTNET. 2005. .NET framework developer's guide. `http://msdn2.microsoft.com/en-us/library/`. Microsoft Corporation.

Douence, R. and Südholt, M. 2002. A model and a tool for event-based aspect-oriented programming (EAOP). Tech. Rep. 02/11/INFO, Ecole des Mines de Nantes.

Ducasse, S. 1997. Message passing abstractions as elementary bricks for design pattern implementation. In ECOOP '97 Workshop Reader, European Conference on Object-Oriented Programming, J. Bosch and S. Mitchell, Eds. *Lecture Notes in Computer Science 1357*, 96–99.

Fiege, L. 2005. Visibility in Event-Based systems. Ph.D. thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany.

Garlan, D. and Scott, C. 1993. Adding implicit invocation to traditional programming languages. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, USA, 447–455.

Geihs, K. 2001. Middleware challenges ahead. *Computer 34,* 6, 24–31.

Gelernter, D. 1985. Generative communication in linda. *ACM Trans. Program. Lang. Syst. 7,* 1, 80–112.

Gilman, L. and Schreiber, R. 1996. *Distributed Computing with IBM MQSeries*. John Wiley & Sons, Inc., New York, NY, USA.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. 2001. Getting started with aspectj. *Communications of the ACM 44,* 10, 59–65.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag.

Konana, P., Liu, G., Lee, C.-G., and Woo, H. 2004. Specifying timing constraints and composite events: An application in the design of electronic brokerages. *IEEE Transactions on Software Engineering 30,* 12, 841–858. Member-Aloysius K. Mok.

Krishnamurthy, B. and Rosenblum, D. S. 1995. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering 21,* 10, 845–857.

Liebig, C., Cila, M., and Buchmann, A. 1999. Event Composition in Time-dependent Distributed Systems. In *Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems (CoopIS 99)*. IEEE Computer Society, 70–78.

Mansouri-Samani, M. and Sloman, M. 1997. Gem: A generalised event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal* 2 (June), 96 – 108.

Meier, R. and Cahill, V. 2005. Taxonomy of distributed event-based programming systems. *The Computer Journal 48,* 5, 602–626.

Mühl, G., Fiege, L., and Pietzuch, P. R. 2006. *Distributed Event-Based Systems*. Springer-Verlag.

Notkin, D., Garlan, D., Griswold, W. G., and Sullivan, K. J. 1993. Adding implicit invocation to languages: Three approaches. In *Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software*. Springer-Verlag, London, UK, 489–510.

Oki, B., Pfluegl, M., Siegel, A., and Skeen, D. 1993. The information bus: an architecture for extensible distributed systems. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*. ACM Press, New York, NY, USA, 58–68.

Oracle. 2005. Oracle® streams advanced queuing user's guide and reference 10g release 2 (10.2), part number b14257-01. `http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14257/toc.htm`. Oracle Corporation.

Pietzuch, P. R. 2004. Hermes: A scalable event-based middleware. Ph.D. thesis, University of Cambridge, Queens' College.

Plale, B. and Schwan, K. 2003. Dynamic querying of streaming data with the dquob system. *IEEE Trans. Parallel Distrib. Syst. 14,* 4, 422–432.

Rosenblum, D. S. and Wolf, A. L. 1997. A design framework for internet-scale event observation and notification. In *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*. Springer-Verlag New York, Inc., New York, NY, USA, 344–360.

ROWSTRON, A. I. T. AND DRUSCHEL, P. 2001. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. Springer-Verlag, London, UK, 329–350.

SEGALL, B. AND ARNOLD, D. 1997. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the Australia Unix Users Group Conference AUUG97*. Brisbane, Australia, 243–255.

STEVENS, W. R. 1992. *Advanced programming in the UNIX environment*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

SUN-AWT. 1997. Java awt: Delegation event model. `http://java.sun.com/j2se/1.3/docs/guide/awt/designspec/events.html`. Sun Microsystems, Inc.

SUN-JINI. 2003. Jini's distributed events specification, version 1.0. `http://java.sun.com/products/jini/2.1/doc/specs/html/event-spec.html`. Sun Microsystems, Inc.

SUN-JMS. 2002. Java message service (jms) specification, version 1.1. `http://java.sun.com/products/jms/docs.html`. Sun Microsystems, Inc.

TAM, D., AZIMI, R., AND JACOBSEN, H.-A. 2003. Building content-based publish/subscribe systems with distributed hash tables. In *Proceedings of the 1st International Workshop on Databases, Information Systems, and P2P Computing (DBISP2P)*. Lecture Notes in Computer Science, vol. 2944. Springer-Verlag.

W3C. 2004. W3C xml schema. `http://www.w3.org/XML/Schema`. World Wide Web Consortium.

WEISER, M. 1993. Some computer science issues in ubiquitous computing. *Communications of the ACM 36,* 7, 75–84.

ZHAO, Y., STURMAN, D., AND BHOLA, S. 2004. Subscription propagation in highly-available publish/subscribe middleware. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, USA, 274–293.