Engineering Event-Based Systems with Scopes

Ludger Fiege*, Mira Mezini, Gero Mühl*, and Alejandro P. Buchmann

Department of Computer Science
Darmstadt University of Technology, D-64283 Darmstadt
{fiege,gmuehl}@gkec.tu-darmstadt.de
{mezini,buchmann}@informatik.tu-darmstadt.de

Abstract. Event notification services enable loose coupling and they are therefore becoming an essential part of distributed systems' design. However, the development of event services follows the early stages of programming language evolution, disregarding the need for efficient mechanisms to structure event-based applications. In this paper, the well-known notion of scopes is introduced to event-based systems. We show that limiting the visibility of events is a simple yet powerful mechanism that allows to identify application structure and offers a module construct for the loosely coupled components in event-based systems. We are able to customize the semantics of scoped event notification services by binding meta-objects to the application structure that reify important aspects of notification delivery, like interface mappings and transmission policies. The scoping concept facilitates design and implementation by offering encapsulation and adaption of syntax and semantics of event-based systems.

1 Introduction

The focus of this paper is on abstractions for structuring event-based systems. The event-based architectural style has become prevalent for large-scale distributed applications [6] due to the inherent loose coupling of the participants. This loose coupling carries the potential for easy integration of autonomous, heterogeneous components into complex systems that are easy to evolve and scale. Traditional request/reply approaches, such as remote procedure calls (RPC), exhibit crucial scalability problems in data-centric environments [16]. The use of event-based dissemination as an alternative approach is superior in these scenarios [15].

The notion of event-based style used in this paper is basically the one defined in literature, e.g., [6]. In an event-based style components communicate by generating and receiving *event notifications*. An *event* is any transient occurrence of a happening of interest, i. e., a state change in some component. The affected component issues a notification that describes the event. An *event notification*

* Supported by the German National Science Foundation (DFG) as part of the PhD program "Enabling Technologies for Electronic Commerce" at Darmstadt University of Technology.

B. Magnusson (Ed.): ECOOP 2002, LNCS 2374, pp. 309–333, 2002.

service conveys the notifications between the components of an event-based system. A component in such a system can act both as producer and consumer of events.

Producers are components that publish notifications about internal events originating within that component. The output interface of a producer is described by advertisements specifying the kinds of notifications it will publish. A notification is not addressed to any specific (set of) receivers, it is rather distributed by the event service to consumers which have specified their interest in that kind of notification. *Consumers* issue subscriptions that describe the notifications they want to receive, i.e., their input interface. Hence, in the event-based cooperation model producers have no knowledge about any receivers—in particular, they do not anticipate any specific reaction on the receiver side.

A component's implementation in event-based systems is 'self-focused' in that it only publishes changes in its state and/or reacts on incoming notifications, resulting in a very loose coupling. However, it is mandatory to identify the role of an administrator[1] who assembles and orchestrates simple components. In the context of open systems, architecture references define a multitude of views of the system in addition to producers and consumers [23], including an administrator's role. It is her task to combine components in order to accomplish a common application functionality. Unfortunately, current work on event-based systems disregards this important role and do not provide any support therefor.

The potential of an event-based communication style has been recognized both in academia and industry. A number of event-based middleware infrastructures were developed [7,11,43,46] as well as the integration of corresponding services in modern component platforms based development such as CCM [35] and EJBs [44]. The prevalence of the event-based paradigm in the design of today's systems has not hindered, but rather encouraged, us in considering event-based systems from a critical point of view. The observation that we make is that while a considerable amount of work is done in the area of scalable event notification services, most effort is spent on implementation efficiency, thoroughly disregarding design, engineering and administration issues. Typical implementation techniques of publish/subscribe systems [38] concentrate on efficient notification dissemination algorithms and overlook the need for effective support of appropriate programming abstractions.

Software engineering research early identified information hiding and abstraction [39] as basic principles that have influenced the development of structured programming, modules, classes, and components, all of which provide mechanisms to structure software systems. While being an integral part of request/reply-based distributed systems, e.g., CORBA [34], comparable hierarchical structuring mechanisms are missing in event-based systems. As a result, event-based systems are generally characterized by a 'flat design space': Subscriptions select out of *all* published notifications without discriminating producers. Any further distinctions are necessarily hard-coded into the communicating compo-

[1] Currently, assemblers and administrators are not distinguished, like it is done in Sun's EJB model [44], for example.

nents, mixing application structure and component implementation. The very feature of event-based systems is thereby defeated: loose coupling.

What we are missing is the notion of a module for bundling several components into a higher-level component. Such a module construct would localize the relationships between components outside of the components themselves, playing a mediator role in the vein of Sullivan and Notkin [42]. The modules should themselves be first-class components, with their own input and output interfaces, so that they can be composed into higher-level modules much the same as objects can be composed into higher-level objects in an object-oriented system. This is to enable a hierarchical structuring of event-based systems.

In this paper, we analyze a set of engineering requirements for event-based systems and introduce the notion of scopes aimed at serving the needs. A scope bundles several components either (a) according to application structure, or (b) according to the structure of activities therein. The visibility of events is constrained in the sense that notifications are only delivered to consumers within the same scope but are a priori invisible otherwise. By being itself a component, a scope can recursively be composed into a larger scope. As a bundling unit and module, scopes represent application structure and are the appropriate location to refine and customize notification delivery semantics. In delimited parts of an application, syntax of the distributed notifications and even the semantics of delivery can be varied, while any modifications are encapsulated and do not interfere with the remaining system. Scopes provide a module construct as an abstraction for handling heterogeneity as well as for integrating security and transmission policies that deviate from the standard broadcast to all eligible consumers.

The remainder of this paper is organized as follows. In Section 2, design and engineering demands of event-based systems are discussed. Section 3 introduces the notion of scopes in event-based systems and describes their features with the help of a running example. An outline of implementation issues is given in Section 4. Related work is discussed in Section 5. The paper concludes with a summary and an outlook on future work.

2 Engineering Event-Based Systems

In this section, we discuss some requirements on engineering event-based systems and how they are supported by today's technology. Two main observations are made. The first is that event-based systems do not seem to imply other requirements for designing and engineering than those already known from engineering request/reply systems. The second observation is that while supporting abstractions are available for the latter they are missing for event-based systems. This makes them difficult to maintain, e.g., the effects of newly instantiated producers or of publishing a certain notification are not easy to determine, let alone control.

2.1 Illustrative Example

A stock trading application will be used as an illustrative example throughout the paper. This example shall not outline a perfect implementation but underline the requirements of engineering event-based applications.

Assume there is an Internet infrastructure which (also) utilizes events instead of the request/reply-oriented style of the Hypertext Transfer Protocol (HTTP) in order to facilitate the creation of the respective web-services and applications. Nearly all parts of a stock trading application are inherently event-based. The dissemination of stock quotes from the central trading floor (or its computerized equivalent) to the market participants is an accepted and plausible example of applying event notification services. The following components can be identified to constitute a stock market (see Fig. 1):

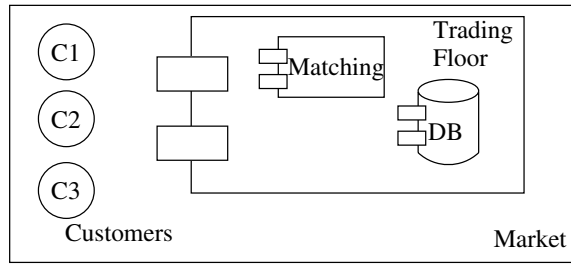

Fig. 1. An example stock trading application

- Customers monitoring quotes and issuing orders to buy or sell shares.
- A database logs the generated data to ensure consistency and persistence.
- A central matching engine implements the matching algorithm.

Database and matching engine are composed into the virtual trading floor, a component which consumes orders and publishes notifications carrying share prices of successfully executed trades.

2.2 Engineering Requirements

Four requirements posed by the engineering of event-based systems are identified: bundling of components, heterogeneity, flexible configurations, and support of activities.

Bundling Related Components

A fundamental requirement is that it should be possible to bundle individual components into higher-level syntactical and semantical units, offering higher levels of abstraction and reusability. From the syntactic point of view such a

bundle should be a collection of existing components delimiting the visibility of the events produced by them. The bundling mechanism should be orthogonal to the subscription mechanisms so that the composed interfaces need not to be changed. This is important to support drawing event deliver localities not only based on the described interests of receivers but also on other criteria, such as the organizational and geographical constraints of a company or some other application-specific semantics.

From the semantical point of view, we require component bundles to be themselves components with well-defined interfaces and own semantics. That is, the bundles should not only delimit visibility, but also publish themselves events, resulting from notifications produced within the bundle that signify an important state change of the bundle as a whole, or consume events from the outside by further propagating them to their internal locality. This opens the possibility to recursively bundle component compositions into higher-level components, and hierarchically structure the design of an event-based system.

Locality, encapsulation, and composing existing units into higher-level units are well-known concepts for mastering complexity and support evolution [39]. These concepts are used in request/reply systems, but they are as important here and should therefore be available to the engineers of event-based systems as well. To motivate the requirement, consider our running example. The virtual trading floor in the stock trading application would be a first example of a component bundle. One can imagine a 'verbose' matching engine producing detailed notifications about the progress of the matching algorithm, of which the majority is only relevant for logging purposes (e.g., to support later traceability of the system operations) and only a few are relevant for customers. Hence, it makes sense to constrain the visibility of most of the events to the DB component and to allow only a few of them to pass the boundary of the trading floor bundle.

The next reasonable structuring step would be to bundle a trading floor and a set of customers (i.e., the participants in the market described in Fig. 1) into a higher-level syntactical and semantical market component. In this way multiple trading floors would be supported without having customers that observe quotes in a system with only a single trading floor receive duplicate and inconsistent quotes if an additional exchange is instantiated. Such duplication cannot be avoided in a flat design space where all components in the system are visible to each other. The absence of market bundles would require to encode knowledge about the market structure into the subscriptions of individual components, rendering them less reusable since more sensible to the structure of the application and changes to it.

Mastering Heterogeneity

A single uniform event notification service with uniform syntax and semantics will hardly be able to cope with the requirements of all parts of large distributed systems operating in heterogenous environments. An event service that, e.g., relies on some notion of a global naming scheme is not scalable and impedes system integration. Furthermore, the semantics of notifications will likely vary

in heterogeneous environments [9]. In large distributed systems, there are inevitably different event models and representation schemes in use, ranging from hardware-dependent differences to application-dependent syntactical and semantical differences.

From the observations above, we draw the requirement that bundling of related components should not only encapsulate functionality but also delimit common syntax and semantics. This requires mechanisms to support adapting data crossing boundaries of component bundles by mapping event content and representation. To motivate the requirement consider again our running example. For efficiency reasons it would make sense in such a system to distinguish between low-volume external representations in XML versus more efficient, optimized internal representations. The matching and the database component may use such an internal representation in our example. Hence, mapping from an external XML representation to the efficient internal representation would be needed for notifications crossing the border of a virtual trading floor composite.

Flexible Configuration

Similar to the diverse requirements regarding data representation in heterogeneous environments, a static definition of notification transmission semantics is not adequate either. Application-specific needs often require that notifications are only delivered to a specific subset instead of the default broadcast to all eligible consumers. For example, an 1-of-n policy realizes load balancing features within a bundle of components in this way. In our stock application, the matching engine might be replicated to distribute processing load over multiple instances using a delivery policy that routes orders to instances dedicated to the respective share.

Furthermore, other delivery policies might be applicable and the whole event service is subject to customization: API, syntax and semantics of subscriptions, security policies, and implementation techniques of notification dissemination may vary to adapt to and fit differing needs in different parts of a complex system. For example, if the structure of the bundles are not static, some policy must control who is allowed to join. The trading floor component may be compromised if everyone is able to join and issue notifications which influence the matching engine; whereas getting prices of successfully executed trades need not to be controlled. Similarly, the implementation of the trading floor will likely use any broadcast features of a local area network while the dissemination of price information on the Internet has to use other techniques.

Supporting Sessions and Activities

The engineering of complex systems not only benefits from bundling related components according to application structure but also from identifying sessions of interdependent activities. This is especially important in event-based systems, where the identity of peers is unknown and communication is a priori stateless in the sense that consecutive notifications cannot be interrelated. By relating

notifications, components are enabled to participate in multiple, distinguishable sessions and activities made up of interrelated notifications can be modeled as well-defined structures drawing on locality just as it is possible for application structure.

An example for the first goal is a stockbroker who listens to a specific share traded on two stock markets. Obviously, notifications distributed in one market must, generally, be invisible in the other. However, our broker should be able to observe and distinguish both. In abstract terms, the issue is that it should be possible for individual components to be simultaneously engaged in multiple sessions involving components from structurally disjoint application parts. Hence, it should be possible to identify such sessions and to delimit them from each other in order to support session state. Otherwise, a component involved in multiple sessions would only be able to maintain changes of its own state, not being able to sustain dependencies on other components.

The second requirement of supporting bundling of events in activities addresses the dynamic aspects of event-based systems in a similar way as the requirement for bundling components did in the previous discussion about the slowly changing structure of an application. In general terms, activities should be structured and it should itself be a component with well-defined semantics, determining when (parts of) the 'internal' notifications are to be made visible to the outside. This will help to prevent side-effects, to build structured, hierarchical sessions, and to customize and orchestrate them. An analogy to the activity concept from the world of request/reply-based systems would be a simplified version of the notion of transactions [20].

2.3 Engineering Support

In request/reply-based distributed systems, like the CORBA platform [34], solutions exists for all of the outlined requirements. Components and classes according to an object-oriented programming paradigm are used for decomposition, encapsulation and bundling of components. Heterogeneity is addressed by standardized interconnection protocols (e.g., CORBA-IIOP, SOAP [4] based on XML). Bundling of activities is facilitated by transaction services [37,2] and security services are available, e.g., Kerberos [32]. Appropriate support is easily provided since the identity of each component is known.

But how can the above mentioned requirements be realized in event services? Existing services recognizes and addresses them only partially. A first approach would be to build new features on top of the existing ones. For example, one could make use of content-based filtering mechanisms [30,8] to simulate a decomposition abstraction for event-based systems in which sets of components are bundled and delimited from each other. To achieve this goal, subscriptions of individual components have to be adapted to encode additional constraints on the decomposed structure.

This approach of modifying application components has a significant drawback. It disregards the administrator's role by compiling all configuration information into the components themselves. Knowledge about the application

structure is put into the components, contradicting the idea of components being loosely coupled and self-focused. Furthermore, the structure is not explicitly enforced by the system and all components are eligible receivers if they have subscribed accordingly: 'hacked' filters may compromise security measures, and reflection, i.e., investigation and change [27], is restricted.

The following section will introduce a second approach of tackling the engineering requirements by introducing a scoping mechanism as an integral part of a design methodology and of an event service implementation.

3 Scoping in Event-Based Systems

In order to satisfy the requirements discussed in the previous section our approach introduces the concept of *scope* for decomposing event-based systems, a unifying concept to address the described requirements. A scope is *an abstraction that bundles a set of producers and consumers* and it can recursively be a member of other scopes. It offers a powerful structuring mechanism to group constituent components which belong together according to some criteria derived from the application structure and/or semantics. Vice versa, it defines locality that can be used to customize semantics in a discriminated part of the system and that provides an encapsulated module whose interaction with the remaining system can be explicitly controlled.

Formally, scoped event-based systems are modeled by a directed, acyclic graph $G = (C, E)$ (see Fig. 2) that describes the superscope/subscope relationship. The set of nodes C is comprised of simple components $\mathcal{C}$ and complex components $\mathcal{S}$, i.e., scopes. The edges E are a binary relation over C. An edge from node c_1 to c_2 in G stands for c_2 being a superscope of c_1. Next to being acyclic, the relation E must also satisfy the property that a simple component cannot be a superscope of any node in G.

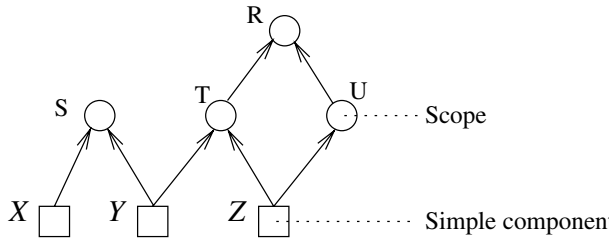

Fig. 2. A graph of components/scopes

The scope concept comes with three different flavors: *standard scopes* harnessing visibility and interfaces, *advanced scopes* that apply mappings and transmission policies, and *session scopes* which use the previous features. A more formal treatise on visibility, interfaces, and mappings is published in [14].

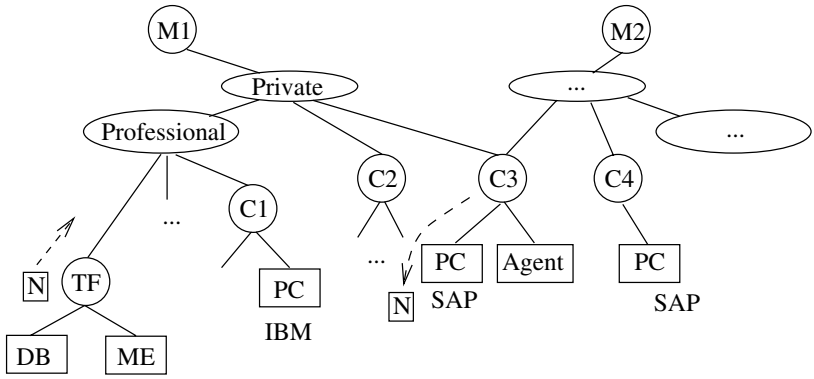

Fig. 3. The graph of the stock application

3.1 Controlling Visibility

Encapsulation is a prerequisite to system evolution [39] and the notion of visibility is widely used in software engineering as structuring technique in order to determine the impacts of changing parts of the system. The need for an equivalent notion for event-based systems was discussed in Section 2.2. The scope construct plays this role in our model in that the visibility of notifications published by a producer is confined to the consumers belonging to the same scope as the publisher.

Using the graph of scopes G given above, we define the *visibility* of components as a reflexive, symmetric relation v over C. Informally, component X is visible to Y iff X and Y have a common superscope. For a component X, let $super(X) = \{X' \,|\, (X, X') \in E\}$ denote the set of scopes that are direct superscopes of X. Formally, we recursively define

$$\begin{aligned} v(X,Y) \Leftrightarrow \quad & X = Y \\ & \vee\ v(Y,X) \\ & \vee\ v(X',Y) \text{ with } X' \in super(X) \end{aligned}$$

In the graph in Fig. 2, for example, $v(Y,U)$ holds but not $v(X,U)$.

A notification is delivered to a consumer if (a) the producer and the consumer are visible to each other, and (b) the notification matches one of the subscriptions previously issued by this consumer. Hence, the semantics of notification delivery is now not only based on the subscription mechanism but also on the visibility relation, with both dimensions being orthogonal in that they are employed independently of each other.

The visibility of notifications from different markets is restricted in a stock trade system designed with scopes (Fig. 3). The circles denote composed scopes in the figure, while rectangles represent simple components. There are two main scopes in which the simple components are organized, `M1` and `M2`, denoting two

different stock markets. Within each market customers are bundled into subscopes based on some criteria, e.g., in private and professional customers. Each customer is permanently represented by one of the scopes `C1`, `C2`, etc., which remain connected in the graph of scopes even if customers are not personally logged in. They group a customer's PCs, cellular phones, or agents running on a remote server. An example 'agent' would be a limit watcher which continuously monitors a share's price and issues a customized notification when a specific share deviates from the overall market performance. Newly and externally provided limit notifications can thereby be integrated into the application without changing existing components—one of the obvious benefits of event-based systems.

For the sake of simplicity, interest for at most one share is indicated in the figure below the rectangles representing the customers' PCs. The figure illustrates the scenario when the trading floor `TF` participates in the stock market `M1` and issues a notification concerning SAP quotes. Although both consumers `C3` and `C4` have subscribed for notifications on SAP quotes, this notification will only reach `C3`, because `C4` is not visible from the trading floor and `C1` subscribed to a different share. On the other hand, consumer `C3` listens to both markets and receives 'duplicate' SAP quotes (the implied problems are addressed in Sect. 3.6).

3.2 Interfaces

So far, visibility can be mapped to an only two-level hierarchy that is induced by the top-most superscopes of the graph G. Any two components are either able to see all of their published notifications or no at all. In order to overcome this problem and to improve the structuring ability, the basic mechanism provided by scopes is refined beyond the visibility relation by assigning input and output interfaces to scopes.

Input and *output interfaces* for simple components are defined by filters that determine the set of notifications allowed to cross a component's boundaries. A filter $F \in \mathcal{F} := \{f \mid f(e) = e \vee f(e) = \varepsilon\}$ is a mapping function over the set of all possible notifications $\mathcal{N}$ plus the empty notification ε. Often, filters are defined as boolean functions returning `true` if a notification matches. In our model, we use a generalized form of filters that are allowed to pass matched events in an unchanged form. A notification n is either mapped to itself or to ε, indicating that n is matched or blocked, respectively. Allowing filters to pass matched events in an unchanged form facilitates filter composition: $(F_1 \circ F_2)(e) = F_1(F_2(e))$.

A simple consumer component describes its input interface by issuing subscriptions that contain filters. A notification passes such a set of filters if it matches at least one of them. On the other hand, a producer has to issue *advertisements* that define the set of notifications it is able to publish. Advertisements also contain filters and serve as a specification of a component's output interface.

We associate similar sets of filters with the input and output interfaces of scopes, describing the set of notifications which are allowed to cross the scope boundary. Only those notifications matching one of the scope's output filters are forwarded up into its superscopes and only those that match at least one

of its input filters are forwarded into the scope. Filters for scope interfaces are expressed in the language used for specifying subscriptions and advertisements for simple consumers and producers. With the introduction of interfaces for scopes, a notification is delivered only if producer and consumer are visible to each other, the notification is allowed to pass all interfaces along the path of visibility in the graph, and one of the receiver's subscriptions match.

Attaching interfaces to scopes allows to view scopes as ordinary producing and/or consuming components. The relationship between scopes and simple components is shown in the upper part of the UML class diagram in Fig. 4 which presents a simplified meta-model of our model.

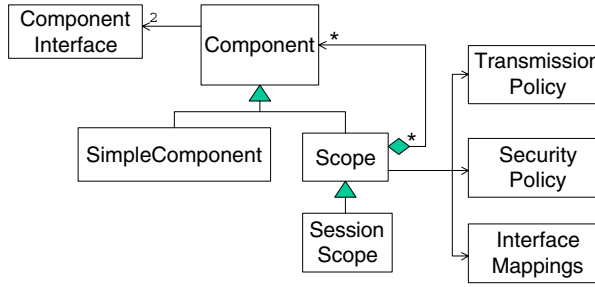

Fig. 4. The Meta-Model of the Scope Model

To illustrate how scope interfaces help in structuring event-based applications, let us consider the interfaces of the components in our running example as summarized in Table 1.

Table 1. Interfaces of the Components in the Example Application

Component	Description	Input	Output
M1, M2	The Stock Markets	–	–
Private	scope of all private customers	–	Trade
Prof	scope of all professionals	Order	Accept, Quote(delayed)
C_i	Customer representation	Accept	Order
TF	Trading Floor	Order	Accept, Quote
ME	Matching engine	Order	Accept, Quote OrderBook
DB	The logging database	Order, Quote	

Customers send out notifications of type *Order* which contain a share identification, the number to be sold or bought, and potential price limits. The trading floor `TF` listens to these orders, issues acceptance notification, and sends out *Quotes*, informing about successfully executed orders. The trading floor itself is

composed of the matching engine ME and the database DB. The matching engine maintains a list of open orders and executes the matching algorithm, while the database logs all *Orders* and *Quotes*, and it issues acceptance notifications (*Accept*). Additionally, the matching engine publishes an orderbook summary with price and volume of the 10 best bid and ask orders. The summary is only visible within the trading floor, because the interface of TF prohibit further distribution. Based on this data, additional services may be integrated into the trading floor, like market makers ensuring that there is always at least one buy and one sell order open.

3.3 Advanced Features

In addition to standard scope features discussed so far, the presented model also supports advanced scope types that customize the event service's functionality, both within the scope and with respect to other scopes. They enable dynamic adaptability of event systems by virtue of associating meta-objects [24] which reify important runtime semantics of event-based systems. Software engineering research has established the notion of meta-object protocols as a very flexible technique to implement and adapt communication between objects. With our approach similar externally provided techniques can now be applied in event-based systems, too. With scopes as first class citizens, an administrator is enabled to easily group unmodified components and tailor the composed functionality with the help of such meta-objects.

The advanced features of the scope concept are shown on the right part of the meta-model shown in Fig. 4. Currently, the following aspects of the runtime semantics are reified. Other aspects of the runtime semantics might be reified as well, resulting in other types of scopes.

- Event reception and publication, allowing the administrator of an event-based system to attach event transformers at scope interfaces. This is aimed at coping with heterogeneity in event-based systems and is described in Sect. 3.4.
- Event transmission policies, allowing the administrator to configure each scope with a strategy to be used for traversing the scope hierarchy and for delivering notifications to consumers and superscopes (see Sect. 3.5).
- Security policies attached to a scope control membership management. Scopes are a proper place to implement these policies but we do not further investigate this issue here.

3.4 Event Mappings

In large systems, it is rather unlikely that a single uniform event model is used throughout the system. Different parts will use different representations and semantics of events. Constraining the visibility of notifications is the basis for dealing with heterogeneity issues and different administrative domains. Consequently, we extend the scoped event system model to include *event mappings* that

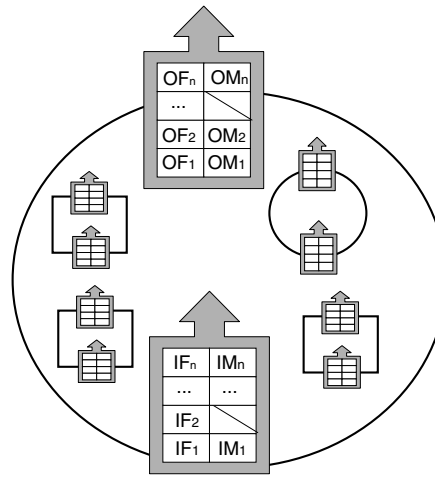

Fig. 5. Scopes with Event Mappings

transform notifications at scope boundaries. This extension clearly addresses the heterogeneity requirements stated in Section 2.2 and facilitates construction and maintenance of large systems. The structure of a scope with event mappings is schematically presented in Fig. 5.

Event mappings convert notifications from an external to an internal representation, and vice versa. They are attached to individual edges in the graph of scopes and are applied when a notification enters or leaves a scope, i.e., it travels down against or up along an edge, respectively. The mappings are a generalization of both the visibility $v(X, Y)$ and the scope interfaces [14] in that a published notification may be visible in a different, mapped representation, or not at all if it was blocked, i.e., mapped onto the empty notification ε. The set of filters F used for subscribing is a subset of the set of event mappings $\mathcal{M} \subseteq \{m \mid m : \mathcal{N}^* \to \mathcal{N}^*\}$ usable in the system. With this definition, a uniform way of filtering and transforming notifications is achieved, and conceptually, filters and general mappings can be concatenated at scope boundaries. Figure 4 distinguishes interfaces and mappings in order to emphasize their independence: interfaces and filters have to be declared in a language that depends on the underlying transmission technique while mappings are part of the scope implementation.

The interface of a scope is strictly separated from its implementation, i.e., its constituent components. Only boundary-crossing events are considered without interfering with internal communications. This separation offers great flexibility in controlling and adapting interface access at runtime. For example, by attaching mappings to individual edges in the graph, a scope may be visible with different interfaces in different superscopes.

Returning to the stock exchange example from the previous section, quotations are typically given in a local currency which need to be transformed at the

boundary of the local scope in order to achieve comparability. As another example for the usefulness of event mappings consider XML languages like FIXML [33] that standardize financial data exchange. These languages are used to connect external partners, but they are typically too expensive for internal representations due to efficiency reasons. Also, most likely, different representations of events will be used inside the consumers, within the market, and within the trading floor, e.g., Java objects, XML financial data, and EBCDIC mainframe text fields. Event mappings are installed at the consumers and at the trading floor to map between serialized Java objects and their XML representation and between XML and EBCDIC, respectively.

Event mappings offer a link to integrate other works in the area of syntactical and semantical transformations which are applicable here [3,25] and which extend the 1:1 mappings we used for simplicity reasons here. Furthermore, event composition can be used to further enhance the idea of event mappings [26,47].

3.5 Transmission Policies

Following the arguments of Sect. 2.2, we suggest to allow refinement of delivery and dissemination semantics on a per scope basis. *Transmission policies* describe how notifications are forwarded and to which consumers. They refine the visibility definition both within a scope and with respect to its superscopes. We distinguish three different policies involved in notification transmission: delivery, traverse, and publishing policy.

Delivery policies affect deliverable notifications produced in a superscope or by some constituent subcomponent and determines which members of the scope are to receive the notification. An example is a 1-of-n policy which delivers only to one out of a group of possible receivers. The idea of meta object protocols of object-oriented programming languages is applied here [24] in order to offer the ability to order, queue, redirect, or transform incoming messages.

A *traverse policy* controls the downward path of incoming notifications in the graph of scopes. Actually, this policy allows a notification to deviate from a default path through the graph of scopes. In a top-down traverse policy eligible receivers, i.e., simple components with a matching subscription, are searched in the current scope first. A notification is forwarded to subscopes only when no-one is found. The bottom-up traverse policy starts the search in the deepest subscopes. Broadcast is the default policy which simply delivers to all components in a scope.

To make an analogy to the application of meta-object protocols in the area of object-oriented programming languages, multiple consumers of the same notification located along the inheritance/scope hierarchy can be considered to be implementing some form of generalized method overriding. Traditional programming languages like C++ and Java use only one, static policy to resolve calls to overridden methods. In a hierarchy of scopes, the traverse policies determines what kind of method lookup is used. The bottom-up policy resembles a virtual method call in C++ in that the implementation of the most derived class is used. Other policies are possible that implement other kinds of method lookups.

Policies can also be viewed in the opposite direction. A *publishing policy* controls publication into the direct superscopes. One may reject the idea of manually selecting where the data is published as contradicting with the event-based paradigm. However, this selection is part of the administrator's role and not interwoven with the application functionality in simple components. While event mappings provide the ability to support multiple interfaces, publishing policies operate on a per notification basis and might be used to delay notifications for a certain amount of time or until a condition becomes valid, for example.

To illustrate the usefulness of the advanced transmission policies, consider the categorization of the customer scopes in private and professional ones: private customers are bundled in the scope `Private` and professional traders in the scope `Professional`. Assume that the market strategy is such that *Quote* notifications should be notified to professionals first. It is because of the need to implement this application semantics that we have defined two different customer scopes and have made `Professional` an sub-scope of `Private`. By having `Professional` encapsulate `TF`, notifications from the trading floor will reach the professional scope first. In addition, the publishing policy of the professional scope is such that *Quotes* are forwarded to the `Private` superscope only after a delay of 15 minutes: a publishing policy puts all notifications in a queue and ensures the delay.

3.6 Sessions

The scoping model so far concentrated on application structure, but the discussion in Sect. 2.2 also identified the need to structure activities therein. In the following, we investigate the problems imposed by activities in event-based systems and give an outlook on how scopes can be used in solving the problem.

Dependent Notifications

The prevalent scenario of event-based applications are uni-directional flows of notifications from producers to consumers, like stock tickers and news feeds. So, it is simply about components lined up in chains. But in order to benefit from the loose coupling of event-based cooperation in other types of applications, it is necessary to support some form of stateful collaboration. Scopes offer this support to a certain extent since they build up structures of bilateral visibility. However, there is so far no explicit mechanism to identify interdependent notifications, which have a common cause and belong to the same activity.

This issue is aggravated by the fact that the graph of scopes is not a tree and a node may have multiple superscopes[2]. Consider the scopes S and T in Figure 2: notifications published in S are not visible in T, and vice versa. But an event in S which is consumed by Y may trigger an reaction in Y leading to the publication of a notification that is also visible in T. The delimitation imposed by scopes is

[2] Note that we do not address the important question of notification duplication in this paper.

diluted in this way since implications of an initially invisible event are diffused; an effect that is not always appropriate. For example, assume that Y is a security service that consumes, signs and republishes specific types of notifications so that S and T are able to publish signed notifications. Unfortunately, Y acts like a bridge and its reactions are visible in both superscopes. Obviously, it is not acceptable to publish these triggered signed data into both scopes.

One solution to this problem is to replicate the security service so that it is offered separately, but this is only feasible if the instances do not need to share a common state, i.e., they are independent. However, this solution does not work with components that cannot be instantiated but are deployed in a different administrative domain and are only accessible remotely, like web services on the Internet. Furthermore, the graph of scopes would be restricted to be a tree, resulting in a structure built from only a single point of view, even though it was corroborated that engineering of complex systems always benefits from facilitating multiple viewpoints [22].

Session Scopes

In order to support dependent notifications and solve the problem of diluting delimitation in multiple superscopes we define an extended type of scope which provides notification contexts. *Session scopes* group components and especially all of their published notifications. They are tagged scopes and the tag is appended to every notification published within. A tagged notification is processed like any other notification, but additionally, the hidden context containing the tag is maintained in every consumer by the event service. The context remains valid during a consumer's reaction to the delivery of a tagged notification. All notifications published while a valid context is available are only disseminated into that tagged session superscope from which the context originated. Forwarding into untagged scopes is not affected so that the application structure and its behavior is not influenced by the creation of session scopes.

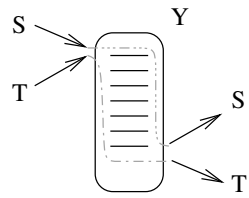

Fig. 6. Contexts of Multiple Superscopes

In our scenario, we would tag the scopes S and T, characterizing them to be *session scopes* (cf. Fig. 6). On delivery of a notification a previously registered processing function is invoked that computes the signature and publishes the result. The tag carried by the notification is maintained as hidden context during

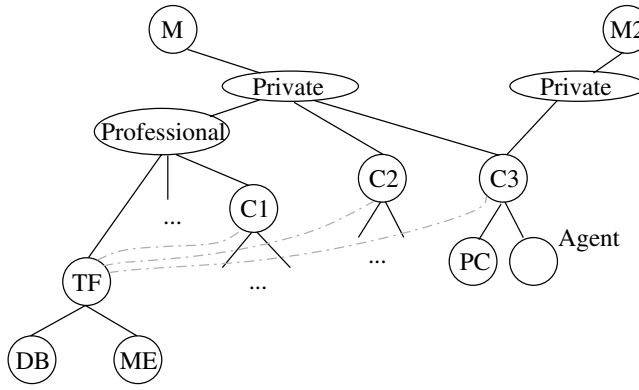

Fig. 7. The stock application with sessions

the execution of Y and any subsequent publication is transparently directed to the originating superscope. Note that the API of the event service needs not to be changed since calls to `pub(`*data*`)` are unmodified and any contexts are transparently maintained by the implementation of the API.

Session scopes are scopes that structure activities and allows components with multiple superscopes to distinguish multiple sessions. These scopes can be instantiated as first-class representatives of sessions, allowing to apply the other mentioned features of scopes and to integrate activities in the graph of scopes. One possible way of realizing sessions is pointed out, namely by using tagged scopes and transmission policies to add, strip, and enforce matching of tags. Obviously, the stated semantics of session scopes offer reasonable defaults but further investigation is necessary to explore other features.

In order to illustrate the use of session scopes, let us once again return to our running example. As already mentioned, customers send out notifications of type *Order*. However, the event-based order processing in this form is only feasible if the order data is only visible to the trading floor. For this purpose, customer scopes are tagged as session scopes and each scope also includes the trading floor, illustrated by the grey, dotted lines in Figure 7. The trading floor needs at least two interfaces, one for handling orders used for the customer scopes and one for publishing quotes into the professional market. Otherwise, the distinction between private and professional customers would be broken. The activity of putting an order is encapsulated in these session scopes so that an issued order is only forwarded to the trading floor and the resulting acceptance notification is only delivered back into the originating customer's scope.

4 Implementation Issues

Generally, scopes are not about efficiency but enable to utilize the provided constrained localities to consider efficient implementations. An implementation

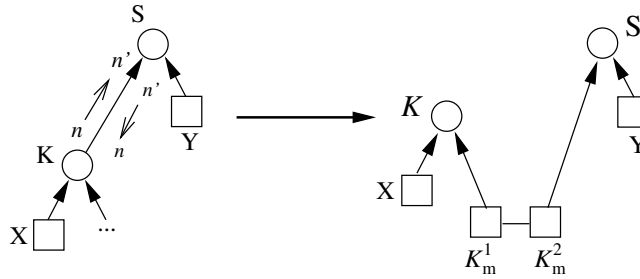

Fig. 8. Transformation of mappings into components.

of a given subgraph of scopes can draw on these locality in that it is tailored to the specific needs of the respective scopes and their constituents. For instance, a scope that groups components on a local area network will most likely use an implementation based on some broadcast mechanism, while the connections to its superscopes rely on point-to-point transmission.

The presented sketch of an implementation demonstrates feasibility in the sense that more efficient implementations may be applied in any part of the system. It is based on an event notification service that supports content-based filtering. Both are implemented in the REBECA project [13], but scopes can be based on other implementations, too. We first describe an implementation realizing scopes without interfaces and mappings. In a second step, full-featured scopes are based on top of a scoped event service.

The central idea of implementing scopes and the visibility $v(X, Y)$ is to transparently extend all subscriptions issued in the system to reflect the structure implied by the graph of scopes. This structure is orthogonal to any subscriptions issued by the components in the system and therefore the filter language must allow filter extension. For example, a filter F would be transformed to $F' = (F \wedge \mathit{Scope} = X)$, testing for a newly introduced name *Scope*. So, the scope in which a notification is published has to be appended to it. This is done transparently by an additional software layer between application code and the simple notification service, without influencing existing components or compromising loose coupling. Administration messages are sent when the graph of scopes is changed. This task is eased by using a simplification of the graph: According to the definition of the visibility $v(X, Y)$, it is sufficient to append to each notification the maximal elements, i.e., the root nodes of the graph which have no outgoing edges, that are visible to the publisher. All subscriptions are accordingly enhanced to filter on the visible maximal elements. For a detailed discussion, please refer to [14].

The second implementation realizes scopes with interfaces and mappings and relies on the previously described scoped event system; more efficient solutions with the same interface are usable, too. Figure 8 depicts the implementation idea. A scope graph with interfaces and mappings can be transformed into a graph without mappings by introducing additional components that implement

Table 2. Related Approaches

	Visibility	Activity	Flexibility	Heterogeneity
SIENA	○	–	–	○
Ready	+	–	○	+
CORBA	○	–	+	○
Information Bus	+	○	–	○
Mediators	+	–	+	–
Field	○	–	+	+
InfoBus	○	○	–	–
ActorSpace	○	–	○	–
Scopes	+	+	+	+

interface checks and event transformations. Such a component consists of two parts, one is registered in the superscope and one in the mapped scope. This is necessary in order to utilize the previously described scoped event system that delimits visibility according to the maximal elements, the roots in the graph. The depicted graph transformation creates a new root for every scope with an interface/event mapping. Although it is not the most efficient solution, it highlights the inherent problems and facilitates a modular implementation. A more efficient solution might explicitly instantiate a scope and integrate its implementation with the newly introduced components to form a 'scope manager' that transforms notifications, checks interfaces, applies delivery constraints, and controls security issues such as scope membership.

5 Related Work

In this section, we discuss related work and compare it with our model. An overview of the comparison with some of the discussed approaches is given in Table 2, with the sign '+' meaning 'supported', '–' meaning 'not supported', and '○' meaning 'not appropriately supported'.

Scoping is a well-known concept which is widely used in programming languages and software engineering [39]. It is used in blocks, functions, classes, packages, and components, but the research literature on event-based systems often lack most of the basic ideas of these structuring mechanisms. The basic concept of visibility and the related problems are of fundamental nature and they are therefore identified and addressed in many publications. However, no other approach in the area of event-based systems is based on the notion of visibility.

Carzaniga et al. [8] describe the SIENA event notification service, which is a popular example of a service utilizing content-based filtering. A thorough presentation of filtering semantics and design choices is given, focusing on network bandwidth efficiency. As for all other content-based filtering approaches, the filters may be used to realize visibility constrains, but these issues are not explicitly addressed. Similar to other works on event services, the flat namespace of notification attributes inhibit scalability because globally unique names are assumed.

The Ready event notification service additionally offers event zones, partitioning components based on logical, administrative, or geographical boundaries [21] and delimiting the visibility of events. But a component belongs to exactly one zone so that there is no multi-level hierarchy, and the system is structured only based on one specific point of view, prohibiting composition and mixing of aspects [22]. It is only mentioned that concepts from group communication [40] may be applicable, offering the flexibility of changing notification delivery semantics. Boundary routers are able to connect event zones and apply transformations on crossing notifications. This work presents some scoping aspects, but they do not offer one basic concept that integrates the different aspects of visibility.

The event channels of the CORBA notification service [36] offer a structuring mechanism in that notifications are only visible within the channel in which they were published. Channels can be connected to compose the reachable components, facilitating visibility and composition. However, producers must explicitly publish notifications in a specific channel, moving information about application structure into the components and limiting dynamic system evolution.

In subject-based addressing schemes for notification delivery, a tree of subjects is used to partition and select notifications; the Information Bus [38] and the Java Message Service [43] are prominent examples of this addressing scheme and even a lot of commercial products are available, from Tibco [46] and others. But the simplicity of the model results in severe disadvantages. Similar to selecting event channels, producers have to select the appropriate subjects, and the predefined tree of subjects constrain the view onto the system, impeding composition in heterogeneous environments [22]. Nevertheless, the simplicity of the concept led to wide acceptance and a multitude of implementations, e.g., in Tibco Rendezvous the basic characteristics are extended to support additional features such as bridges connecting multiple busses, integration of transactional activities, and security considerations.

Sullivan and Notkin introduce mediators [42] in order to offer a design approach which explicitly instantiates and expresses integration relationships. An implicit invocation abstraction is used to bundle components and mediators, and, with its own interface, to compose new components. A similar approach regarding visibility is used as in our scoping model, but no default semantics is outlined so that they 'only' suggest a framework that facilitates design without identifying features that are attached to visibility: transmission policies, activities, security, etc.

The Field environment [41] is an early work on tool integration and it is built around a centralized server that distribute messages. Messages sent to the server were selectively re-broadcasted to receivers that registered patterns matching the message. The original approach realized content-based filtering in a flat space of notifications. With the Field Policy Tool, it was later possible to extend the semantics by introducing a mapping of any sent message to a set of message-receiver pairs. While this opened up Field to include any delivery semantics, it is a mechanism which is very hard to control because it is based on rule and trigger

evaluation. An additional extension allowed to limit the visibility of messages to a set of receivers, but did not support composition and interfaces.

The InfoBus [10] is a small Java API which allows JavaBeans or cooperating applets on a Web page to communicate data to one another. Multiple instances of InfoBus might be manually connected with bridges, providing a limited means of structuring without any inherent interfaces or composition support. It is merely a mechanism to distribute change notifications and requests for data items. Matching of messages is done by names, i.e., string matching. Besides being limited to one virtual machine, it is a tool for connecting components not for composing new ones.

Research on coordination models is dominated by Linda-like systems [18], although it was criticized that race conditions are possible in Linda and its variants, resulting from the inherent concurrency of the model [1]. In comparison to Linda, event-based systems offer a more loose coupling of components, facilitating distributed deployment of independent components. A general difference between our approach and Linda-like systems is that we have identified the administrator role and the need for externally provided configuration mechanisms that do not change instantiated components. The need to specify names or identities of tuplespaces is a major characteristic of many works on multiple tuplespaces [19]. In this way, many of the considered ideas are relevant for event-based systems but the suggested solutions are not directly adoptable. There exist some work on Linda systems which establish structures on the components. Agha and Callsen propose actorSpaces to limit distribution of messages [1]. The basic drawback of their approach is that, even though previously unknown objects are intended to cooperate, senders have to specify destination addresses. The sketched implementation is rather limited. In [29], Merrick and Wood introduce scopes to limit the visibility of tuples in Linda, but again, senders have to specify destination scopes. Furthermore, nesting of scopes is restricted to two levels. LIME [31] realizes an transparent access to multiple tuplespaces, although the approach is limited to a three-level hierarchy bound to the physical layout of the system. It is focused on the intended application domain of mobile agents and do not offer a general solution.

Cardelli and Gordon propose a process calculus for mobile ambients [5]. It is used to describe the management of a tree of ambients whose intended purpose of grouping computation resembles our graph of scopes. The calculus might be used to model scope graph dynamics, but communication across ambients is only indirectly supported and destination identities must be known.

Ported objects [28] are objects which communicate by processing messages which arrive at ports. A port is a connector in a data stream which is not directed by the object. A compound ported object encapsulates a number of ported objects and hides the data flow inside. This resembles the idea of grouping producers and consumers in scopes, while all the other features are lacking.

Evans and Dickman defined 'zones' in order to support partial system evolution [12]. The meta object protocol [24] shows the relationship between OO programming languages and scoping in event-based systems. Controlling and

modifying method calls is similar to the handling of notifications in transmission policies and event mappings presented in this paper.

Garlan and Scott presented delivery policies for implicit invocation systems [17]. Four different delivery policies are distinguished: full (broadcast) and single delivery (1-of-n, that is 'indirect invocation'), parameter-based selection (filter), and a state-based policy. The policies resembles our definition but does not include the other transmission policies.

6 Conclusion

Former work on event-based systems has concentrated on efficiency issues, neglecting to support the engineering of complex systems. We have applied the notion of scopes as a fundamental structuring mechanism for event-based systems. Following classical developments in software engineering, the scoping concept is based on the notion of visibility of components and notifications. A set of design requirements for engineering event-based systems is investigated showing that, similar to approaches in traditional request/reply-based systems, visibility is the main underlying principle here. Although many other earlier contributions tackled some of the involved problems the scoping concept offers a unified approach for event-based systems based on visibility.

From an engineering point of view, scopes offer a module construct for event-based systems, being an abstraction and encapsulation unit at the same time. As an abstraction unit, a scope provides the rest of the world with common higher-level input and output interfaces to the bundled subcomponents. As an encapsulation unit, a scope constrains the visibility of the notifications published by the grouped components. It hides the details of the composition implementation, such as the underlying data transmission mechanisms, the interface mappings that map between internal and external representations of notifications, security policies, transmission policies controlling the way notifications are forwarded, etc. The structure built thereby is orthogonal to the components' implementation, separating concerns of implementation and interaction. As defined in our model, scopes have the flavor of component frameworks in the sense of Szyperski [45]: they encode the interactions between components and can themselves act as components in higher-level frameworks. The ability to model, to integrate, and to realize dynamic sessions with this concept shows the flexibility of the presented scoping concept.

The main ideas presented in this paper have been implemented in a prototype of an event notification service as part of the REBECA project. We are currently evaluating the prototype with the help of the stock trading example used throughout this article and two other example applications dealing with Internet auctions and self-actualizing Web pages. This allows us to investigate the design of event-based systems and of the necessary infrastructure not only in theory but also in practice.

Future work will include a more detailed discussion of session scopes and their relation to traditional transactions, and engineering tools that allow to build and administer scoped event-based systems via a graphical user interface.

References

1. G. Agha and C. J. Callsen. ActorSpace: an open distributed programming paradigm. *ACM SIGPLAN Notices*, 28(7):23–32, July 1993.
2. P. A. Bernstein. Transaction processing monitors. *Communications of the ACM*, 33(11):75–86, Nov. 1990.
3. C. Bornhövd and A. Buchmann. A prototype for metadata-based integration of internet sources. In *11th International Conference on Advanced Information Systems Engineering (CAiSE'99)*, volume 1626 of *LNCS*, Heidelberg, Germany, June 1999. Springer-Verlag.
4. D. Box et al. Simple object access protocol (SOAP) 1.1. Technical report, W3C, 2000. http://www.w3.org/TR/SOAP/.
5. L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, Berlin, Germany, 1998.
6. A. Carzaniga, E. Di Nitto, D. S. Rosenblum, and A. L. Wolf. Issues in supporting event-based architectural styles. In *ISAW '98: Proceedings of the Third International Workshop on Software Architecture*, pages 17–20, 1998.
7. A. Carzaniga, D. Rosenblum, and A. Wolf. Design of a scalable event notification service: Interface and architecture. Technical Report CU-CS-863-98, Department of Computer Science, Univ. of Colorado at Boulder, USA, 1998.
8. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.
9. M. Cilia, C. Bornhövd, and A. P. Buchmann. Moving active functionality from centralized to open distributed heterogeneous environments. In *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS '01)*, volume 2172 of *LNCS*. Springer, 2001.
10. M. Colan. *InfoBus 1.2 Specification*. Lotus.
11. G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 2001.
12. H. Evans and P. Dickman. DRASTIC: A run-time architecture for evolving, distributed, persistent systems. In M. Akşit and S. Matsuoka, editors, *European Conference for Object-Oriented Programming (ECOOP '97)*, volume 1241 of *LNCS*, pages 243–275. Springer-Verlag, 1997.
13. L. Fiege and G. Mühl. Rebeca Event-Based Electronic Commerce Architecture, 2000. http://www.gkec.informatik.tu-darmstadt.de/rebeca.
14. L. Fiege, G. Mühl, and F. C. Gärtner. A modular approach to build structured event-based systems. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)*, pages 385–392, Madrid, Spain, 2002. ACM Press.
15. M. J. Franklin and S. B. Zdonik. A framework for scalable dissemination-based systems. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '97)*, Atlanta, Georgia, USA, Oct. 5–9, 1997.
16. M. J. Franklin and S. B. Zdonik. "Data In Your Face": Push Technology in Perspective. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 516–519. ACM Press, 1998.

17. D. Garlan and C. Scott. Adding implicit invocation to traditional programming languages. In *Proceedings of the 15th Intl. Conference on Software Engineering (ICSE '93)*, pages 447–455. IEEE Computer Society Press / ACM Press, 1993.
18. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.
19. D. Gelernter. Multiple tuple spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89: Parallel Architectures and Languages Europe*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27, 1989.
20. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
21. R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the READY event notification service. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop*, Austin, Texas, USA, May 1999.
22. W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In A. Paepcke, editor, *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, pages 411–428, 1993.
23. ISO/IEC. Reference model of open distributed processing. Draft Standard, May 1995.
24. G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge, MA, USA, 1991.
25. O. Lassila and R. R. Swick. Resource description framework (RDF) model and syntax specification. W3C Recommendation, Feb. 1999. http://www.w3.org/TR/REC-rdf-syntax.
26. C. Liebig, M. Cilia, and A. Buchmann. Event Composition in Time-dependent Distributed Systems. In *Proceedings of the 4th Intl. Conference on Cooperative Information Systems (CoopIS '99)*, Sept. 1999.
27. P. Maes. Concepts and experiments in computational reflection. In N. Meyrowitz, editor, *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 147–155, Orlando, FL, USA, Oct. 1987. ACM Press.
28. J. McAffer. Meta-level programming with CodA. In W. Olthoff, editor, *European Conference for Object-Oriented Programming (ECOOP '95)*, volume 952 of *LNCS*, Aarhus, Denmark, 1995. Springer-Verlag.
29. I. Merrick and A. Wood. Coordination with scopes. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2000)*, pages 210–217, Como, Italy, Mar. 2000.
30. G. Mühl. Generic constraints for content-based publish/subscribe systems. In *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS '01)*, volume 2172 of *LNCS*, pages 211–225. Springer, 2001.
31. A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In F. Golshani, P. Dasgupta, and W. Zhao, editors, *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 524–533, May 2001.
32. B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, Sept. 1994.
33. Oasis. *FIXML - A Markup Language for the Financial Information eXchange (FIX) protocol*, July 2001. http://www.oasis-open.org/cover/fixml.html.

34. Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Version 2.3. Object Management Group, Framingham, MA, USA, 1998.
35. Object Management Group. *CORBA Components*. OMG, Framingham, MA, USA, 1999. orbos/99-07-01.
36. Object Management Group. Corba notification service. OMG Document telecom/99-07-01, 1999.
37. Object Management Group. Corba transaction service v1.1. OMG Document formal/00-06-28, 2000.
38. B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus—an architecture for extensible distributed systems. In B. Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 58–68, New York, NY, USA, Dec. 1993. ACM Press.
39. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.
40. D. Powell. Group communication. *Communications of the ACM*, 39(4):50–53, Apr. 1996.
41. S. P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.
42. K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions of Software Engineering and Methodology*, 1(3):229–269, July 1992.
43. Sun. Java message service specification 1.0.2, 1999.
44. Sun Microsystems, Inc. Enterprise javabeans specification, version 2.0. Proposed Final Draft, 2000. http://java.sun.com/products/ejb/index.html.
45. C. Szyperski. *Components Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
46. TIBCO, Inc. TIB/Rendezvous. White Paper, 1996. http://www.rv.tibco.com/.
47. S. Yang and S. Chakravarthy. Formal Semantics of Composite Events for Distributed Environments. In *Proceedings of the 15th International Conference on Data Engineering (ICDE '99)*, pages 400–407. IEEE Computer Society Press, 1999.