Modular Event-Based Systems

Ludger Fiege[1], Gero Mühl[1], and Felix C. Gärtner[2]*

[1] Department of Computer Science
Darmstadt University of Technology
D-64283 Darmstadt, Germany {fiege,gmuehl}@gkec.tu-darmstadt.de
[2] École Polytechnique Fédérale de Lausanne
Departement de Systèmes de Communications
Laboratoire de Programmation Distribué
CH-1015 Lausanne, Switzerland fgaertner@lpdmail.epfl.ch

Abstract. Event-based systems are developed and used to integrate components in loosely coupled systems. Research and product development focused so far on efficiency issues but neglected methodological support to build such systems. In this article, the modular design and implementation of an event system is presented which supports scopes and event mappings, two new and powerful structuring methods that facilitate engineering and coordination of components in event-based systems. We give a formal specification of scopes and event-mappings within a trace-based formalism adapted from temporal logic. This is complemented by a comprehensive introduction to the event-based style, its benefits and requirements.

1 Introduction

Current developments in nearly all areas of computer science show the shift towards increasingly compound, federated information systems, which are composed out of heterogeneous, autonomous building blocks. Design, implementation and administration of these systems are not only getting more complex because of the technical problems of distributed computing [97]. In addition, conglomerates of independently created components have to be orchestrated in order to offer an integrated functionality. The scientific field of *coordination theory* investigates scenarios and techniques for managing the dependencies between a set of active components, and it is not limited to computer science alone [65].

The coordination paradigm differentiates computation from coordination [82]. It makes the interaction between components explicit in *coordination media* [29], which offer means to control and adapt a system's configuration. However, adaptability degrades if components are aware of the dependencies, even if these are managed by the coordination medium. Once information about the dependencies is stored in the components, their computation and interaction with peers is possibly based on this data, withdrawing control from the coordination medium;

* Work was performed while the author was with the Computer Science Department of Darmstadt University of Technology, Germany.

called subjective as opposed to objective, or exogenous, approaches [90]. In fact, it is not the technique of the underlying medium that determines the amount of information the programmer is apt to include into a component itself. For example, Linda tuple spaces [22] were suggested to decouple computation from coordination, but they can be used both for a request/reply-based and an event-based mode of interaction. In the first case, the caller explicitly delegates tasks to the called service, establishing a close dependency. On the other hand, in an event-based mode a component simply publishes data, not knowing which components, if any at all, receive and react to the data, separating computation and publication of data from any subsequent processing.

The event-based architectural style has become prevalent for large-scale distributed applications [25] due to the inherent loose coupling of the participants. It facilitates the clear separation of communication from computation and carries the potential for easy integration of autonomous, heterogeneous components into complex systems that are easy to evolve and scale [91,48]. Another important advantage of event-based cooperation is accuracy of information. In a system model that uses a request/reply mode and where information is assumed to be generated in various, not predetermined parts one would be forced to ask all possible sources repetitively and in an appropriate frequency, finally leading to system congestion and/or inaccurate information [43]. The use of event-based dissemination as an alternative approach is superior in these scenarios [42].

The potential of event-based coordination has been recognized both in academia and industry and there exist a considerable amount of work on event notification systems. A number of event-based middleware infrastructures were developed [26,32,95,98] and corresponding services were integrated in modern component platforms such as Corba component model (CCM) [77] and Sun's Enterprise JavaBeans (EJB) [93]. The notion of event-based systems is investigated in the context of many areas of computer science, e.g., coordination models [101,19,84], rule systems [57], software architecture [48] and engineering [38,92], database systems [86,18], and distributed systems [8,27].

The prevalence of the event-based paradigm in the design of today's distributed 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 for distributed systems, most effort is spent on implementation efficiency, completely disregarding design, engineering and administration issues. The loose coupling in event-based systems not only introduces additional degrees of freedom and more flexibility but also increases the complexity of designing and understanding these systems. Apart from some partial aspects [55,79], existing work so far dealt with unstructured applications and can be generally characterized by a 'flat design space': all published data is visible to the whole system and no explicit control of the distribution of event notifications is available.

Software engineering research early identified information hiding and abstraction [85] as basic principles that have influenced the development of structured programming, modules, classes, and components, all of which provide mechanisms to structure software systems. These ideas are an integral part of request/reply-based distributed systems, e.g., Corba [76], and are also used in coordination models [68,81] and rule systems [15]. However, these approaches either do not follow the event-based model of cooperation or do not address distributed scenarios.

In this article, the notion of *scopes* in event-based systems is proposed in order to incorporate an explicit structuring mechanism. A scope bundles a set of producers and consumers and delimits the visibility of published events. Scopes may republish internal events and forward external events to its members, and thus a scope may be viewed as a producer and consumer. It can recursively be a member of other scopes, offering a powerful structuring mechanism and a means of application modularization. A scope is used to identify the structure of an event-based application as a first-class citizen, allowing event service semantics to be tailored and adapted to an application's needs. For example, in large systems it is not sufficient to delimit the visibility of notifications because of heterogeneity issues and different administrative domains, where syntax and semantics of events differ. The scoping model allows *event mappings* to be bound to a scope, i.e., a possibility to transform events when crossing scope boundaries.

Besides introducing scopes, one of the contributions of this article is that we provide a formal specification of the semantics of scoped event systems. The specifications are given using standard approaches from distributed algorithms, i.e., the specification language is adapted from temporal logic [66] and the specification itself is divided into safety and liveness conditions [61]. The specification itself is built up in a modular way: in a first step a precise specification of a simple event system is given, which is extended to include scopes and event mappings in a second and third step. The modular approach to building event systems has many evident advantages. For example, it makes the task of building a complex event system much easier because different concerns are handled separately in an incremental fashion. Implementation sketches for each of these steps are given and they benefit from the modular design by using the implementation of the previous step, respectively. Furthermore, in conjunction with exact specifications, dealing with correctness argument gets easier.

The article is structured as follows: Sect. 2 presents a comprehensive overview of the characteristics of event-based systems and their deficiency of only supporting flat address spaces and no visibility control. In sections 3, 4, and 5 specifications of a simple event service, a scoped event service, and a scoped event service with event mappings are given, including the respective modular implementation sketches and correctness arguments, refining those presented in [40]. Sect. 6 presents a thorough overview of related work and Sect. 7 concludes the article.

2 Event-Based Systems

The notion of event-based systems is used in many areas of computer science. Unfortunately, there exist varying, ambiguous definitions of terminology, and consequently of the anticipated characteristics and usage scenarios. In this section, we concisely identify and distinguish basic cooperation models and present an overview of existing work in this area.

2.1 Terminology

So, what is an *event-based system*? It is a system in which the integrated components communicate by generating and receiving *event notifications*. An *event* is any transient occurrence of a happening of interest, e.g., a state change in some component. A *notification* is a data representation that reifies and describes such an occurrence. Different notifications may be created to describe the same event and which are represented in different event data models (e.g., name/value pairs [27], objects [35], or semi-structured data [72]) or use varying views on the relevant data, e.g., for security reasons. On the lowest level considered here, notification data is conveyed in *messages*, which are simply data containers transmitted through the underlying communication mechanism, a variety of which are used in existing systems. This differentiation is used to clearly separate the underlying technique from the mode of cooperation. Consequently, any messaging system can either transport notifications or requests, both of which are packaged in a message. Finally, an *event notification service*, or event system for short, applies different techniques and data and filter models to distribute notifications between the components of an event-based system.

Components act as producers and/or consumers of notification. *Producers* are components that publish notifications. The output interface of a producer is described by advertisements specifying the kinds of notifications it will publish. *Consumers* issue subscriptions that describe the notifications they want to receive, i.e., their input interface. A notification is not addressed to any specific (set of) receivers but distributed by the event service to consumers which have subscribed to that kind of notification. The expressiveness of subscriptions depends on the data model and filter language used. Essentially, three classes are distinguished. In subject-based addressing [80] the publisher assigns the path to a node of a subject tree to any published notification and subscriptions are string- or pattern-matching expressions on these subjects. Type-based approaches [12,35] uses notification type hierarchies for filtering. In content-based filtering [70] boolean expressions are evaluated on the content of notifications. The first approach is easy to implement but rather restricted, the second can be smoothly integrated in current object-oriented programming languages, and the third is the most flexible one.

2.2 Cooperation Models

The preceding definition basically refines the one given in the literature, e.g., by Carzaniga et al. [25]. It is directly implemented by publish/subscribe [80]

		Initiator	
		Consumer	Producer
Addressee	Direct	*Request/Reply*	*Callback*
	Indirect	*Anonymous Request/Reply*	*Event-Based*

Table 1. Taxonomy of cooperation models.

systems, which are, however, just an implementation technique like a Linda tuple space engine, IP multicast, or even classical remote procedure calls. All of them can be used to transport the notifications, in principle. And vice versa, a publish/subscribe system can also be used for non-directed, anonymous request/reply. The characteristics of event-based systems are not found in the API or the services used for transmitting the notifications. Instead, the components in these systems use an event-based mode of cooperation; they are built according to an event-based style, regardless of the underlying technology.

Coordination models and software architectures also identified event-based styles [20,48], but concentrated on technical descriptions of the communicative actions. In order to provide a fundamental and simple classification of component interaction, we distinguish cooperation models by the way interdependencies between components are established. A component can take two roles: consumer or provider. The former depends on data or functionality provided by the latter. The first major characteristic of a cooperation model is whether the cooperation is initiated by the consumer or by the provider. The second main distinction is whether the addressee is known or unknown.

The combination of two basic properties, initiator and addressing, leads to four cooperation models (see Tab. 1) that are independent from any implementation techniques. Any interaction between a set of components can be classified according to these models. Furthermore, the models characterize the inner structure of components, which is determined by the way they interact. From an engineering point of view, this helps to identify constraints and requirements posed by a given component on its usage scenarios and on the underlying infrastructure. Architectural mismatches are disclosed early that would otherwise have to be tackled by an integrating implementation, impeding reconfigurations and scalability sooner or later.

Such a simplistic model typically does not cover all nuances of possible interactions (like synchrony/asynchrony, or reliability), but fundamental models are nevertheless helpful and necessary to recognize basic characteristics and to infer appropriate support. We now dicuss each of these four models separately.

Request/Reply. The most widely used cooperation model is request/reply. Any kind of method call or client/server interaction where functionality is delegated belongs to this class. The initiator is the consumer that requests data or/and functionality from the provider, and it expects data to be delivered back or relies on a specific task to be done. The provider is directly addressed, its identity is known, and the caller is able to incorporate information about the callee into his own state, resulting in tight coupling of the cooperating entities.

Replies are mandatory in this model unless the system and failure model excludes errors. Otherwise, if a consumer simply does not care about the outcome of a request, the interaction belongs to the callback model of cooperation (see below).

Anonymous Request/Reply. The anonymous request/reply cooperation model also uses request/reply as basic cooperative action, but the provider that should process the request is not specified. Instead, requests are delivered transparently to an arbitrary, possibly dynamically determined set of recipients. The consumer does not know the identity of the recipient(s) a priori and one request possibly results in an unknown but fixed number of replies.

This cooperation model is apart from the event-based model the second one that is directly implemented by publish/subscribe systems [80], confusing these two models that are often mixed/intertwined with each other. Anonymity of providers adds more flexibility to the request/reply model, but the dependency on externally provided data or functionality persists.

Callback. In the callback model (which is abstracted in the well-known *observer design pattern* [45]), consumers register at a specific, known provider to be notified whenever some condition becomes true. The provider repeatedly evaluates the condition and calls the registered component back and notifies it if necessary. The provider is responsible for administering its callback list which is used to address consumers, and consumers have to register at specific providers. If multiple callback providers are of interest, a consumer must register separately for all of them. The identity of the components are known and must be managed on both sides, leading to a tight coupling with no coordination medium used in between.

On the other hand, callback processing can be customized with this knowledge so that only subsets of consumers are notified in an application-dependent way. However, it would be a component's responsibility to choose callback handlers that fit the current integrated application's functionality, which is an integration issue and should not be part of the component implementation. In any case, a sophisticated implementation of callback-handlers in terms of adaptability leads to the event-based approach, described in the next section.

Event-Based. The event-based cooperation model has characteristics inverse to the request/reply model. The initiator of communication is the provider of the data, called producer of notifications, and the notifications are not addressed to any specific set of recipients, as was described earlier. A consumer can receive events from many providers, because subscriptions are in general neither directed nor limited to a particular producer. If an event matches a subscription, it is delivered. Providers are not aware of the consumers. In contrast to the callback model, providers are relieved from the task of interpreting and administering registrations, i.e. subscriptions.

It is essential for this model that the producer sends information about its own state and that it makes no assumptions on any consumer's functionality. That means a component's implementation is 'self-focused' in that the knowledge encoded in the program and used by the programmer is limited to the component's task. It 'knows' how to react to input notifications and publishes changes to its own state, but it should never publish a notification with the intention of triggering other activity. And this distinguishes event-based components from those that are simply built using the publish/subscribe [80] paradigm. The latter may use the indirection of the publish/subscribe system in an anonymous request/reply model and explicitly rely on the existence and reply of a recipient.

All the dependencies and necessary coordination in a set of event-based components have to be handled externally, i.e., in the coordination medium under the control of an additional role. The role of an *administrator* is identified in various system models [58,93] and it is clearly separated from producers and consumers. It is the responsibility of this role to coordinate and control the integration of the loosely coupled components.

The event-based cooperation model typically turns the architecture upside down compared to a request/reply-based solution. Both models form kind of a duality and for many software problems there exist solutions with either approaches. However, the event-based style clearly separates computation from communication, offers the potential of easily evolvable systems, and sustains loose coupling. It is of great benefit for non-static systems in which unanticipated changes occur. System evolution is simplified if no cooperation mismatches have to be tackled, i.e., if the whole system follows an event-based style, extensions and modifications can be incorporated easily.

2.3 Supporting the Event-Based Style

The typical application domain of event-based systems is so far information distribution that is known to exhibit crucial scalability problems if built using a request/reply [43,42] style of communication. However, the previous discussion showed that for many problems there exist two dual solutions which either are built with a request/reply or an event-based style. Structurally more complex scenarios than unidirectional dissemination of data might draw on the inherent benefits of the event-based style if an appropriate support would be available in terms of both existing systems and programming abstractions.

The event-based model offers new degrees of freedom and awarded benefits, but newly introduced indirections also always create complexity that must be handled. While appropriate support for request/reply based systems exists, research in event-based systems focused on improving the performance in obvious application domains rather than extending the domains. The engineering complexity of event-based applications was disregarded and few solutions are available for classical software engineering problems.

Information hiding and abstraction [85] have been identified as the crucial structuring principles in software engineering. These principles have determined the success of structured programming, modules, classes, and components, all of which provide mechanisms to structure software systems. Although they are an integral part of request/reply-based systems like CORBA [76] and of some coordination models like Linda [23], comparable hierarchical structuring mechanisms are missing in event-based systems. As a result, these systems are generally characterized by a 'flat design space': Subscriptions select out of *all* published notifications without distinguishing producers. Any further distinctions are necessarily hard-coded into the communicating components, mixing application structure and component implementation and thereby defeating the very feature of event-based systems: loose coupling.

The work presented in this article investigates modularity in event-based systems in order to cope with the intrinsic engineering problems. In particular, we introduce the notion of scopes, which is well-known and widely used concept in programming languages and software engineering [85]. After investigating the basic semantics of event-based systems, we present these two new structuring concepts together with a motivation of their engineering benefits. More examples and application scenarios can be found elsewhere [38].

2.4 Simple Event-Based System

A description of simple event-based systems is presented, which acts as a basis for further enhancements. In the following an intuitive, informal description is used that is formalized in Sect. 3.2 and that corresponds to the one mainly used in literature (e.g., [25]). Unfortunately, slightly varying informal semantics are published that are not formalized in most cases; only parts are specified at best, like the filter semantics in SIENA [24], or formalizations are given in the context of coordination media [100].

A simple event-based system consists of a set of producers and consumers, which are clients of an event notification service. All clients are connected to the same service and access its functionality in order to subscribe to or publish notifications according to the terminology introduced in Sect. 2.1. From a client's point of view the notification service is conceptually centralized in that clients do not distinguish different instances of the service running on different nodes in the system. The notification service in a simple event-based system is accessed as a black box data dissemination facility. It computes for every published notification the subset of all consumers with matching subscriptions and delivers notifications accordingly. Its functionality is unmodifiable in the sense that it never discriminates publishers or consumers and basically tests every notification against all subscriptions.[1]

[1] Of course, matching and routing algorithms need not look at all subscriptions individually in order to compute the set of matching subscriptions [102,73].

Possible implementation scenarios range from centralized implementations on top of databases (e.g., active databases [86] or databases with queuing enhancements [51]) to fully distributed notification routing [71].

3 A Formal Framework for Event-Based Systems

There exist a considerable amount of work on notification services, and many concrete systems have been designed and implemented (e.g., SIENA [27], JEDI [32], etc.). Unfortunately, understanding and comparing these systems is very difficult because of different and informal semantics. In fact, a lot of informal requirements can be demanded that an event system should fulfill. For example, we could demand that

(a) only notifications should be delivered to a client that match one of its active subscriptions,
(b) each notification should be delivered at most once to a client,
(c) notification should be delivered in some order with respect to their publication (e.g., in causal order, etc.), and
(d) all notifications matching one of its active subscriptions should be delivered to a client.

The following section (Section 3.1) contains a formalism that helps to specify these requirements unambiguously. In Section 3.2 we use this formalism to specify a simple event system which captures those requirements which we consider really mandatory for the basic service level of a useful event system. Finally, in Section 3.3 we present the implementation of a simple event system and relate it to the specification. In later sections of this article we will use the simple event system to construct complex scoped event systems in an incremental fashion.

3.1 Formal Background

In the literature on program verification, there exists well-developed foundations of methods to specify and validate concurrent systems. The aim of the proposed formalisms is to precisely describe the behavior of a system as a "black box", i.e., without referring to its internal (implementation) issues. Usually, the proposed formalisms model an interactive system as a state machine which moves from one state to another by means of an action. Formally this corresponds to the definition of a *labeled transition system*. In the black box view, we wish to define the behavior of such a system in terms of the states and actions it exhibits at its *visible interface*. In the literature this is termed *observation semantics* and there are many different possibilities of defining observation semantics for concurrent systems. A simple example is *trace semantics*, which amounts to defining an observation simply as a sequence of actions that are visible at the system interface. Intuitively, system evolution can be written as a sequence of transitions [17]

$$s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \ldots$$

and denotes that starting from state s_1 the system reaches state s_2 by executing action a_1 etc. Note that trace semantics can also be used to describe the behavior of concurrent systems. The global state space of a set of concurrent processes for example is defined by the cross product of the state space of the individual processes. The evolution of the system can then be viewed as a sequence of global states which occur by interleaving the individual process traces.

In this article, we use trace semantics to specify the behavior of systems by abstracting from states and reasoning only about the sequence of operations at the interface of the system. Given a set A of possible interface actions, a *trace* $\sigma = a_1, a_2, \ldots$ is a sequence of elements of A. A *specification* then is a set of such traces, namely all traces which are allowed to be generated by a system. Equivalently, a specification can be given as a predicate on traces. We will use the notation of temporal logic [87] to express such predicates. The formal language in this article is built from simple predicates (one for every action), the quantifiers $\forall$, $\exists$, the logical operators $\vee$, $\wedge$, $\Rightarrow$, $\neg$, and the "temporal" operators $\Box$ ("always"), $\Diamond$ ("eventually"), and $\circ$ ("next"). For a given action $a \in A$, the formula a is true for every trace which starts with a. The formula $\neg a$ is true for every trace which does not start with action a. The other logical operators and quantifiers are defined in the obvious analogous way.

The semantics of the temporal operators are defined as follows: Let Ψ be an arbitrary formula and $\sigma = a_1, a_2, \ldots$ be a trace. Then

– $\Diamond\Psi$ is true for trace σ iff there exists an $i > 0$ such that Ψ is true for the trace $a_i, a_{i+1}, a_{i+2}, \ldots$,
– $\Box\Psi$ is true for trace σ iff for all $i > 0$, Ψ is true for the trace $a_i, a_{i+1}, a_{i+2}, \ldots$, and
– $\circ\Psi$ is true for trace σ iff Ψ is true for $a_2, a_3, \ldots$

Note that the temporal operators have higher precedence than the logical operators.

Table 2. Interface operations of a simple event system

$sub(X, F)$	Client X subscribes to filter F
$unsub(X, F)$	Client X unsubscribes to filter F
$notify(X, n)$	Client X is notified about n
$pub(X, n)$	Client X publishes n

To better understand temporal formulas, we now give a few examples using the interface operations of a simple event system which are listed in Table 2. In order to capture which client is affected by an operation, the operations include a reference to the respective client. For example, $sub(X, F)$ means that client X subscribes to filter F.

Intuitively, $\Diamond\Psi$ means that Ψ will hold eventually, i.e., there exists a point in the trace at which Ψ holds. For example,

$$\Diamond \mathit{notify}(X, n)$$

specifies all traces in which client X eventually is notified about n. On the other hand, $\Box\Psi$ means that Ψ always holds, i.e., for all "future" points in the trace Ψ holds. For example,

$$\Box\neg \mathit{unsub}(X, F)$$

specifies all traces in which X never unsubscribes to F. Finally, $\bigcirc\Psi$ means that Ψ holds in the next step, i.e., for the trace starting with a_2. For example,

$$\Box\left[\mathit{notify}(Y, n) \;\Rightarrow\; \bigcirc\Box\neg \mathit{notify}(Y, n)\right]$$

specifies all traces in which if Y is notified about n then Y is never notified about n again. This formalizes requirement (b) from the beginning of this section. Note that we assume free variables to be implicitly universally quantified.

Given an arbitrary labeled transition system with a set of initial states, we say that the system *satisfies* a given temporal formula iff every observable behavior of the system is a trace specified by the temporal formula. This means that the set of traces generated by the system must be a subset of the set of traces specified by the formula. This notion of satisfaction means that the system *implements* the specification and is sometimes called *refinement* [1] or *process preorder* [14]. Of course, in order to correctly implement a specification, a system usually has to execute internal (unobservable) actions different from the interface actions. To model this, some formalisms define an internal action τ and allow for any finite number of internal actions in between two interface actions. This is sometimes called *weak equivalence* [14] or *stuttering equivalence* [62,2]. Inference rules and other proof techniques can then be used to formally derive the satisfaction relation.

In this article, we will be very precise when defining the specification of event systems, but we will be rather informal when presenting the algorithms which implement the specification. We give the former in the defined temporal notation and the latter as textual description. The proofs that the algorithms implement the specification will also follow the standard mathematical textbook style. We feel that a fully formal derivation of the correctness of the given algorithms would have obscured the main contributions of the article, which lie more in the design and specification areas than in verification. However, we hope that the algorithms and proofs are detailed enough to be transformed into fully formal proofs with medium effort.

3.2 Specification of a Simple Event System

Interface Operations. Formally, a *simple event system* is viewed as a black box with an interface (see Figure 1). The possible interface operations are listed in Table 2. All these operations are instantaneous and take parameters from

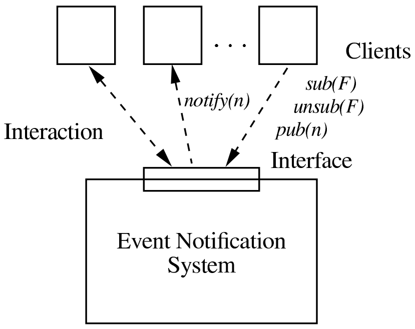

Fig. 1. Black box view of an event system.

different domains: the set of all clients $\mathcal{C}$, the set of all notifications $\mathcal{N}$, and the set of all filters $\mathcal{F}$. Formally, a filter $F \in \mathcal{F}$ is a mapping from $\mathcal{N}$ to the boolean values *true* and *false*. A notification n *matches a filter* F if $F(n)$ evaluates to *true*. Moreover, the set of all notifications that match F is denoted by $N(F)$. Additionally, two further assumptions are made: Firstly, it is assumed that notifications are unique, i.e., each notification can only be published once. Secondly, every filter is associated with a unique identifier in order to enable the event system to identify a specific subscription.

Specification Variables. In the formalization a set of *specification variables* is assumed to be present. Specification variables are fictitious devices which keep track of the internal history of the system within a specification and simplify the temporal formulas. Two sets of specification variables are assumed at the interface for every client $X \in \mathcal{C}$:

1. a set S_X of *active subscriptions* (i.e., filters which X has subscribed and not unsubscribed to yet), and
2. a set P_X of *published notifications* (i.e., the subset of $\mathcal{N}$ containing all notifications X has previously published).

These sets are not part of the state of the system. It is assumed that they are initially empty and that they are updated faithfully (e.g., by an external observer) according to the operations that occur at the interface of the system. For example, whenever X subscribes to F, F is added to S_X, and whenever X unsubscribes to F, F is removed from S_X. Hence, multiple (un)subscriptions to the same filter are idempotent. This also implies that if a client X subscribes to a filter F multiple times and then unsubscribes to this filter once then F is no longer in S_X afterwards.

The behavior of the event system is specified by giving a set of temporal formulas like the examples introduced in Section 3.1. Of course, it is also possible to refer to the specification variables. For example,

$$\Box\big[\mathit{notify}(Y, n) \;\Rightarrow\; [\exists F \in S_Y .\, n \in N(F)]\big]$$

specifies all traces in which the fact that Y is notified about n implies that *at this point in time* there exists a subscription F in S_Y that matches n. This formalizes requirement (a) from the beginning of this section. It is important to keep in mind that the temporal operators determine the place in the trace to which the imposed conditions are applied. As a last example,

$$\Box\big[\mathit{notify}(Y,n) \;\Rightarrow\; [\exists X.\, n \in P_X]\big]$$

requires that the fact that Y is notified about n implies that there is a client X for that n is in P_X *at this point in time*. Subsequently, this implies that n has been published by X before.

Simple Event Systems. In the following, a specification of simple event systems is presented that relies on the trace-based semantics introduced above.[2]

Definition 1. (simple event system) *A* simple event system *is a system that exhibits only traces satisfying the following requirements:*

– *(Safety)*

$$\Box\Big[\mathit{notify}(Y,n) \;\Rightarrow\; \big[\bigcirc\Box\neg\mathit{notify}(Y,n)\big] \\ \wedge\; \big[\exists X.\, n \in P_X\big] \\ \wedge\; \big[\exists F \in S_Y.\, n \in N(F)\big]\Big]$$

– *(Liveness)*

$$\Box\Big[\mathit{sub}(Y,F) \;\Rightarrow\; \big[\Diamond\Box\big(\mathit{pub}(X,n) \wedge n \in N(F) \;\Rightarrow\; \Diamond\mathit{notify}(Y,n)\big)\big] \\ \vee\; \big[\Diamond\mathit{unsub}(Y,F)\big]\Big]$$

The specification consists of a safety and a liveness condition [61]. A *safety condition* demands that "something irremediably bad" will never happen, while a *liveness condition* requires that "something good" will eventually happen.[3] It is known that all properties on traces can be expressed as the intersection of safety and liveness conditions [6,50,49]. Here, the safety condition states that a notification should never be delivered to a consumer more than once, that a delivered notification must have been published by a client in the past, and that a notification should only be delivered to a client if it matches one of the client's active subscriptions. The liveness condition is probably most complicated to understand. It describes precisely under which conditions a notification must be delivered. The condition can be rephrased as follows: if a client Y subscribes to F, then there exists a future point in time where the publishing of a notification

[2] The specification is extended in [71] to include advertisements and to be self-stabilizing in the case of failures.

[3] For a formal definition of safety and liveness refer to Broy and Olderog [17].

n matching F will lead to a delivery of n to Y. This can only be circumvented by Y unsubscribing to F. The liveness condition can be regarded as a precise formulation of requirement (d) from the beginning of this section. Note that we do not impose any delivery order on notifications (like causal order) and so we do not need to formalize requirement (c).

As examples consider the following traces where F is a filter and n_i are notifications matching F while n' is a notification not matching F:

$$\begin{aligned}
\sigma_1 &= \mathit{sub}(Y,F), \mathit{pub}(X,n_1), \mathit{notify}(Y,n') \\
\sigma_2 &= \mathit{pub}(X,n_1), \mathit{sub}(Y,F), \mathit{unsub}(Y,F), \mathit{notify}(Y,n_1) \\
\sigma_3 &= \mathit{sub}(Y,F), \mathit{pub}(X,n_1), \mathit{pub}(X,n_2), \mathit{pub}(X,n_3), \ldots
\end{aligned}$$

Traces σ_1 and σ_2 violate the safety requirement because a notification is delivered to Y that does not match an active subscription. In trace σ_3 client Y subscribes to F and client X starts to publish a continuous sequence of notifications matching F. Since there is no *notify* in σ_3 it perfectly satisfies safety. However, it violates the liveness requirement (to satisfy liveness, there must be a point in the trace following the subscription where either Y unsubscribes to F or Y begins to receive notifications).

Intuitively, the liveness requirement states that any *finite* processing delay of a subscription is acceptable. By abstracting away from real time a concise and unambiguous characterization of what types of actions must be produced by the system under which conditions is obtained. For example, if a client has subscribed to a filter F and later unsubscribes to it, the system does not have to notify the client about *any* notifications which match F and are published in the meantime. It may nevertheless do so, but only as long as the other requirement of Definition 1 is met. On the other hand, delivery of a notification is only necessary if the client continuously remains subscribed to F. Because the system cannot tell the future, it must still make a good effort to prepare delivery even though the client may later unsubscribe to F.

One might argue that defining a trace as a total order is unrealistic in a distributed system because it is not possible or desirable to enforce total ordering of operations. Indeed, it is possible to give specifications which are not (efficiently) implementable because of the lack of a notion of global time in distributed systems. For example, Siena [24,27] demands that a notification is only delivered to a client if the client had a matching subscription at the time the notification was published. However, the specification of Def. 1 is implementable because it imposes ordering relations only on operations which intentionally should be causally related in any sensible implementation. For example, being notified about n does not make sense without n having been published previously.

A system that satisfies only the safety condition is trivial to implement. Any system that never invokes a *notify* operation satisfies the imposed conditions. Similarly, it is easy to implement a system which satisfies only the liveness condition. Any system that delivers every published to all clients fulfills this condition. Hence, the challenge is to implement a system that satisfies *both* requirements.

3.3 Implementation

We now show how to implement the specification of a simple event system from Section 3.2. For brevity, we have chosen to present a very simple implementation. Approaches to improve the scalability of the implementation can be found elsewhere [70,73,74,27,24]. Since we later build extensions (like scoped event systems) by employing instances of simple event systems, it is possible to exchange the implementation with a more efficient one as long as the interface specification is maintained.

We base all our implementations in this article on a system model where a set of asynchronous processes communicate over message passing channels. The channels are assumed to be reliable, i.e., no messages are lost or altered and no spurious messages are delivered, and incoming data is served in a fair manner. The communication topology of processes is assumed to be acyclic and connected (see Figure 2). In practice, acyclic connected topologies can be established manually or through spanning tree construction algorithms.

In the context of an event system, we call a process an *event broker*. To invoke the interface operations of the event system, every client invokes a form of local library function causing messages to be inserted into the system. This means that the client process can be considered to be an event broker (see Figure 2). For every client C we call this event broker the *local event broker* of C.

Data Structures. Every local event broker holds two data structures:

1. a table S of active subscriptions, and
2. a table D of previously delivered events.

Both are initially empty.

Algorithm. If a client invokes $\mathit{sub}(X, F)$, the local event broker of X adds F to S. Conversely, if $\mathit{unsub}(X, F)$ is invoked, F is removed from S. Events are processed within the system by a technique called *flooding*. An invocation of $\mathit{pub}(X, n)$ causes sending a message containing n to the neighbor of the local event broker in the network. If any (non-local) event broker receives such a message, it forwards it to all neighbors except the one the message was received from. A local event broker (say of client Y) receiving such a message checks if there exists a filter F in S_Y such that n matches F. If so, it checks whether n is already present in D_Y. If one of these checks fails, it discards n. Otherwise n is added to D_Y and delivered to the client via a call to $\mathit{notify}(Y, n)$.

3.4 Correctness

We must show that the algorithm from the previous section satisfies the requirements given in Definition 1, i.e., the safety and liveness condition. In the proof, the implementation variable S essentially plays the role of the specification variable S_Y.

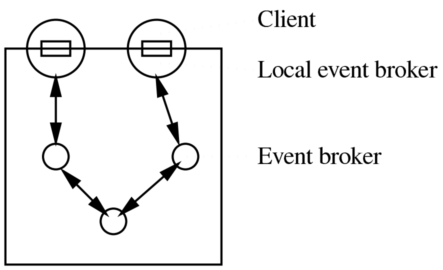

Fig. 2. A possible implementation view of a simple event system.

Proof of Safety. Assume that the implementation invokes *notify*(Y, n) at client Y. We have to show that this implies that the three conjuncts of the implication in the safety property hold.

From the algorithm and the use of D follows that n will not be delivered again. This proves the first conjunct.

Also from the algorithm and the use of S, invocation of *notify*(Y, n) implies that there exists an F in S_Y such that $n \in N(F)$. This proves the third conjunct.

It remains to be shown that n was previously published by some client: Invocation of *notify*(Y, n) implies the receipt of a message containing n at the local event broker of Y. Because of the reliable channel assumption, this message must have been sent by some neighbor. Because of the forwarding algorithm of the event brokers, the acyclic topology, and the reliable channel assumption there must exist a local event broker of come client X from which n originated. From the algorithm, this implies that X previously published n. This proves the second conjunct and concludes the proof.

Proof of Liveness. Assume a client invokes *sub*(Y, F) and never unsubscribes to F. We must show that there is a time after which every event which is published and matches F is eventually delivered to Y. In this case this is rather easy to show since subscriptions become active immediately: Let n be an event matching F and consider a client X which invokes *pub*(X, n) immediately after Y subscribed to F. From the algorithm follows that invocation of *pub*(X, n) leads to the sending of a message containing n to all neighbors of the local event broker. From the forwarding algorithm of the event brokers, the acyclic topology, and the reliable channel assumption follows that the message is eventually received at every local event broker, including that of Y. Since Y has not unsubscribed to F and n matches F the algorithm invokes *notify*(Y, n), concluding the proof.

4 Event-Based Systems with Scopes

We extend the specification of the simple event system and introduce the notion of *scopes*. For presentation purposes, we restrict our attention to *static scopes* that are not reconfigured once the first event has been published. This restriction is softened in Section 4.3.

4.1 Motivation: Coping with Engineering Complexities

So far, the presented simple event system merely provides the functionality to distribute notifications, but still fails to offer any support for coping with the complexities of designing and engineering distributed systems. A number of engineering requirements can be identified [38], all of which are similar to those needed in request/reply-based systems. Encapsulation and information hiding are principal engineering techniques and are investigated in the following; heterogeneity aspects are postponed to Sect. 5.

Bundling. A fundamental requirement is the ability to bundle and encapsulate individual components into 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 that delimits and constrains the visibility of events produced and consumed by them, essentially defining the possible interaction history of the bundle as a whole. Without such a mechanism, the event system is flat: every producer is able to communicate with any other consumer, and it is impossible to explicitly describe any interdependencies or identify and control any side-effects in the system. The effects of introducing additional producers can only be determined if all clients of the notification service are analyzed; an approach that makes analyzing rule based systems difficult, if not intractable [10,11].

From the semantical point of view, we require component bundles to be available for further bundling, that is, they are themselves complex components with well-defined interfaces and semantics, acting as publisher of those internal notifications that match the bundle's output interface and as consumer of outside notifications that match the input interface. This opens the possibility to recursively bundle component compositions into higher-level components, and on the other hand, to hierarchically decompose structure and complexity of the design of an event-based system.

The bundling mechanism should be orthogonal to the subscription mechanisms so that the composed interfaces need not to be changed. This is necessary in order to exploit 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. Otherwise, loose coupling, reconfigurability, and reusability would be severely restricted.

Flexible Configuration. The loose coupling in event-based systems opens up more degrees of freedom than are available in a request/reply-based systems, extending the design space of service implementations in terms of synchronization, reliability, efficiency, etc. Since these properties are decisive for the functionality of the composed system, they must best be fit to application requirements. But a single uniform event notification service will hardly be able to meet all diverging requirements in an evolving distributed system, and if it is tailored to a specific (part of an) application, it will decreasingly fit other demands.

Instead of pressing applications onto an inappropriate notification service, we advocate to decompose and tailor the service to meet the requirements directly. Bundling of producers and consumers as described in the previous paragraph not only allows us to decompose and structure event-based systems, but also offers the possibility to use such a decomposition to adapt the event notification service locally.

We consider two kinds of customization: first, choosing different techniques to implement the notification distribution semantics presented so far, and second, adapting the semantics itself. The given specification of a simple event-based system is minimal and deliberately excludes further aspects like reliability and ordering, which are nevertheless crucial to determine for system development. The bundling of components according to some application structure is the optimal place to determine these characteristics. An appropriate implementation technique for conveying notifications can be chosen for this bundle. In this way, different parts of the system will benefit from tailoring notification service implementation to their specific needs.

Second, the described default semantics of notification distribution, in which a notification is delivered to each consumer with a matching subscription, is adequate for communicating data between otherwise independent applications; that is more specifically, the consumers are independent and are not differentiated; news and stock quote dissemination are examples hereof. In other scenarios, however, where producers and consumers are composed to achieve a common integrated functionality, consumers are not independent, and notification delivery cannot be considered with respect to individual consumers only. It is the system engineer's task to control delivery to a group of related consumers and to program and adapt their behavior, harnessing the degrees of freedom introduced by the loose coupling. The event-based style is not induced by the notification service but is a characteristic of the components. Therefore, they shall continue to adhere to the event-based style, whereas the semantics of notification delivery may change if applications require that notifications are only delivered to a specific subset instead of the default broadcast to all eligible consumers. For example, within a bundle of equal components a 1-of-n delivery can realize load balancing features.

The notion of scopes in event-based systems is introduced in the next section in order to bundle producers and consumers and to localize communication.

4.2 Specification

A scope bundles a set of producers and consumers in order to utilize locality, to hide "internal" configurations, or to delimit administrative domains. The visibility of published events is restricted by the scopes and their composition.

To deal with scopes, we need an additional specification variable G which keeps track of the *current scopes* in the system. Formally, $G = (C, E)$ is a di-

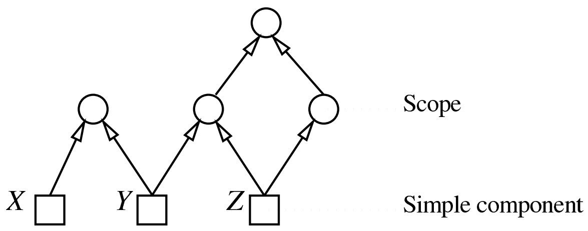

Fig. 3. A graph of components

rected acyclic graph that signifies the superscope/subscope relationship between components and scopes (see Figure 3). We extend the notion of a component to be either a simple component from $\mathcal{C}$ or a scope from a set $\mathcal{S}$ of all possible scopes and define the set $\mathcal{K}$ of *complex components* to be $\mathcal{S} \cup \mathcal{C}$. The nodes C of G are a subset of $\mathcal{K}$ and the edges E are a binary relation over $\mathcal{K}$. 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. As noted above, we assume here that scopes are static, i.e., the scope graph does not change once the first event is published.

Using G, we define the visibility of components as a reflexive, symmetric relation v over $\mathcal{K}$. Informally, component X is visible to Y iff X and Y "share" a common superscope. For a component X, let $\mathit{super}(X)$ denote the set of components that are superscopes of X. Formally, we recursively define

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

In the graph in Figure 3 for example, $v(X,Y)$ holds but not $v(X,Z)$.

Now we extend the specification of a simple event system to also deal with scopes.

Definition 2 (scoped event system). *A* scoped event system *is a system that exhibits only traces satisfying the following requirements:*

– *(Safety)*

$$\begin{aligned} \Box\Big[\mathit{notify}(Y,n) \;\Rightarrow\; & \big[\bigcirc\Box\neg\mathit{notify}(Y,n)\big] \\ & \wedge \big[\exists X.\, n \in P_X \;\wedge\; v(X,Y)\big] \\ & \wedge \big[\exists F \in S_Y.\, n \in N(F)\big]\Big] \end{aligned}$$

– *(Liveness)*

$$\Box\Big[sub(Y,F) \Rightarrow \\ \Big(\Diamond\big[\Box v(X,Y) \;\Rightarrow\; \Box\big(pub(X,n) \wedge n \in N(F) \;\Rightarrow\; \Diamond notify(Y,n)\big)\big]\Big) \\ \vee \Big(\Diamond unsub(Y,F)\Big)\Big]$$

The specification would describe the same set of system if used with the transitive closure of v, but we retain the structure information in order to to reuse the specification for further refinement and underline the importance of knowing the application structure (cf. Sect. 4.4).

We elaborate on how Definition 2 differs from Definition 1. The safety requirement contains an additional conjunct $v(X,Y)$. This means that in addition to the previous conditions, the publisher and the subscriber must also be visible to each other when a notification is delivered. The liveness requirement has an additional precondition that can be understood in the following way: If component Y subscribes to F, then there is a future point in the trace such that if X remains visible to Y, every publishing of a matching event will lead to the delivery of the corresponding notification.

Note that Definition 2 is a generalization of Definition 1. A simple event system can be viewed as a system in which all components belong to the same "global" scope. This implies a "global visibility", i.e., $v(X,Y)$ holds for all pairs of components (X,Y) and can be replaced by the logical value *true* in the formulas of Definition 2, resulting in Definition 1.

4.3 Dynamic Scopes

In Definition 2 we have assumed a static scope hierarchy. The case of dynamic scopes, however, is not so different from the static case. As in other open systems that support reconfiguration at runtime, we assume the role of a manager who is responsible for arranging scopes and components. The individual components do not necessarily need to know about their scope membership; according to the event-based paradigm, they concentrate on the tasks they have to accomplish. To the manager, four additional operations are offered: $cscope(S)$ and $dscope(S)$ to create and destroy a scope S, $jscope(X,S)$ and $lscope(X,S)$ are used to join X to scope S or leave it, respectively. A system with static scopes can then be simulated by having the manager set up the scope hierarchy with the appropriate operations before clients start to publish and subscribe.

However, for the dynamic case, a problem arises when trying to implement Definition 2: A notification n may only be delivered to Y if the publisher X of n is visible to Y. But because X may "spontaneously" leave the scope before delivery, Y must double check that X is still visible at this point to ensure safety. In effect, this results in a type of synchronization similar to that of a global transaction: scope joins and scope leaves must be reliably acknowledged by all other local brokers before the action is performed. Obviously, this type of dynamic scope semantics are unfavorable since they incur a high synchronization overhead. However, scope reconfigurations may be so infrequent in practice that this is tolerable for medium size systems. At least these semantics have the advantage that the safety part of Definition 2 can be used in an unmodified form. We know of no other sensible (possibly weakened) form of semantics for dynamic scopes that avoids this problem.

Note that the liveness part of Definition 2 is perfectly compliant to dynamic scopes. As an example, consider the following trace

$$\sigma_4 = \mathit{sub}(Y, F), \mathit{jscope}(X, s), \mathit{jscope}(Y, s), \mathit{pub}(X, n_1), \mathit{lscope}(Y, s), \ldots, \\ \mathit{jscope}(Y, s), \mathit{pub}(X, n_i), \mathit{lscope}(Y, s), \ldots$$

In σ_4 components X and Y start off in the same scope and X publishes an "infinite" sequence of events n_i. However, since Y leaves the scope again after every publish operation, there is no point in time from which on X and Y remain in the same scope. Therefore, a notification is not required and σ_4 satisfies the liveness requirement.

4.4 Engineering with Scopes

The scoping concept leverages the engineering of event-based systems [38] by providing support for system design and for implementation, corresponding to the bundling and configuration aspects outlined in Sect.4.1.

System design is supported by offering a decomposition mechanism that makes engineering techniques long known in other areas of computer science, namely information hiding and encapsulation, available to event-based systems. Scopes decompose in that they constrain the view on the system and reduce the number of components and notifications to consider. On the other hand, by assigning input and output interfaces to the group of composed entities, scopes are a composition mechanism, too, which facilitates creating new, more complex event-based components in the flavor of component frameworks of Szyperski [96]: they encode the interactions between components and can themselves act as components in higher-level frameworks. Scope composition is external to the composed entities, using an objective [90], or exogenous, approach in which shaping of interaction is separated from and generally invisible to the computation in the entities.

Lets consider auction platforms and stock trading engines as an example. They use continuous quotation, balancing bids and asks, until the best price for a trade is determined. The 'inner' communication needed during this phase is of interest only to a small set of market participants. The last quotation, however, is taken as the final price and this information must also be available to the outside. An implementation using scopes is straightforward: the market is defined by a scope, whose interface lets notifications of completed trades pass while all other internal quotations are blocked so that only participants belonging to the same market/scope receive them.

The scope graph arranges simple and complex components (scopes) in a directed acyclic graph, which can be seen as a composition of multiple tree-like views onto the system. Decomposition in a tree-like manner is a common approach applied to describe computer networks, inheritance hierarchies, and organizational and administrative hierarchies. It was one of the first addressing schemes used in event notification services [80], which is also the basis for today's Java Messaging Service [95]. But a tree only allows for decomposition from a single point of view; a major limitation for evolving systems [56]. The scope graph accommodates different, interconnected points of view and promises to have the expressiveness necessary to model complex distributed systems.

A methodology describing how to design graphs of scopes is desirable here but beyond the scope of this paper. It is not clear whether the plethora of approaches available in object-oriented and component-oriented software design is applicable here, too (e,g, [11]).

The second contribution of having scopes is that a number of implementation-specific aspects related to visibility, which are not necessarily new to event-based systems, can now be incorporated in a distributed event-based system in a uniform way. This makes it easier to determine and tackle interrelationships and dependencies between (sub-)component coordination and underlying communication techniques on the one hand and the application functionality as implemented in the participating components on the other. For example, the underlying communication technique has major influence on the integrated application's functionality in terms of notification ordering, reliability, or atomicity. As we demanded in Sect. 4.1, scopes offer the appropriate place to bind a technique to a specific part of an application, thus enabling the application to benefit from multicast networks or group communication protocols to increase distribution efficiency or atomicity and reliability guarantees.

From an engineering point of view, scopes reify application structure and offers the ability to program the composition of lower-level entities. In order to adapt the behavior of the composed entities, we have to deviate from the default event distribution semantics of delivering a notification to all matching consumers. An essential extension that is bound to a scope are *transmission policies*. These policies control the flow of notifications in the scope graph and are used to adapt and customize the semantics of publishing notifications. Delivery policies control the downward flow in the graph. For example, in a specific scope a 1-of-n can be used to deliver notifications only to one consumer out of the set of eligible consumers, effectively establishing load balancing here. Conversely, publishing policies control upward notification forwarding. They are used in addition to interfaces, because they delay and discard notifications in a context-dependent way; they need not to be stateless as interfaces are.

Furthermore, security policies can be bound to the application structure to control scope membership and to determine the quality of service parameters necessary for the underlying communication (e.g., authentication and encryption as necessary).

4.5 Implementation

We present a possible implementation of the previous specification. The implementation uses a simple event system as specified in Sect. 3.2 as a basic transport mechanism. This modular approach underlines the system's structure and shows the possibility of implementing the specification. But again, it does not concentrate on efficiency issues, and any available notification service satisfying the simple event system specification can be used instead.

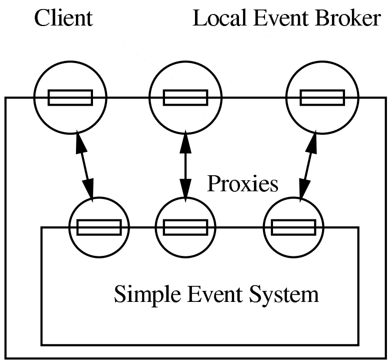

Fig. 4. A possible implementation of a scoped event system.

The architecture of the implementation is sketched in Figure 4. The interface operations of the scoped event system are local library calls which are mapped to appropriate messages of the underlying simple event system. Again, we call that part of the client process that handles these calls the *local event broker* of the client. Conceptually, for every client an additional process at the interface of the simple event system is generated, which we call the client's *proxy*. Practically, the proxy will be part of the local event broker. Note that the clients' proxies are the only components accessing the underlying simple service; no complex components are instantiated in this implementation scenario.

Although we do not deal with dynamic scopes here, the presented algorithm can easily be extended to include dynamic scopes as of Section 4.3. This restriction resembles an object-oriented programming approach where new subclasses and new methods are readily added, but modifying the inheritance hierarchy is complicated (and forbidden here). To simplify the implementation, we restrict the changes which can be made to the graph $G = (C, E)$ of scopes: Only components with no incoming edges may join or leave scopes. This restriction implies that individual brokers do not need to store G completely, as we now explain.

The scope hierarchy expressed by edges E describes a transitive partial order $\leq$ on C, where $X \leq X' \Leftrightarrow (X, X') \in E$. The maximal elements of C have no outgoing edges, i.e., they have no superscopes. These elements are termed *visibility roots* because the recursive definition of $v(X, Y)$ is terminated by common superscopes. The maximal elements that are visible from a component are used to determine visibility of events.

Data Structures. For every client X, its proxy Prox_X holds a list V_X of its visibility roots. In a system with static scopes, V_X is initialized to the set of its visibility roots in the given scope graph. With dynamic scopes where changes are limited to the addition of new leaves—nodes with no incoming edges—V_X is set at the time of addition. In both cases, it remains constant and is not changed until the whole systems stops or X is deleted, respectively.

Algorithm. If a client invokes $\mathit{pub}(X, n)$, a message (pub, X, n) is sent to the client's proxy. At the interface of the simple event system, the proxy then invokes $\mathit{pub}(\mathit{Prox}_X, (n, R))$, where R is set to the constant value V_X.

Calls to $\mathit{sub}(X, F)$ and $\mathit{unsub}(X, F)$ are sent in a similar way to Prox_X. Using F, the proxy derives a filter $\tilde{F}$ that matches all notifications $\tilde{n} = (n, R)$ for which n matches F, and subsequently calls $\mathit{sub}(\mathit{Prox}_X, \tilde{F})$.

Whenever the simple event system notifies the proxy of Y about a notification $\tilde{n} = (n, R)$, the proxy checks whether $V_Y \cap R \neq \emptyset$. If the test succeeds, a message is sent to the local broker of Y to invoke $\mathit{notify}(Y, n)$. Otherwise the notification is discarded.

4.6 Correctness of the Scoped Event System

In order to show that Definition 2 is satisfied, the presented implementation must obey the visibility $v(X, Y)$ of the safety condition and the additional precondition $\Box v(X, Y)$ of the liveness condition. The remaining part is satisfied by using the simple event system which satisfies Definition 1.

We first prove a basic lemma.

Lemma 1. *For every pair of clients X and Y and for the set of visibility roots V_X and V_Y held at the proxies holds:*

$$v(X, Y) \Leftrightarrow V_X \cap V_Y \neq \emptyset$$

Proof: We need to show two implications. The first implication ($\Rightarrow$) is proved by induction over the "visibility" path from X to Y. The second implication ($\Leftarrow$) is shown as follows: If $V_X \cap V_Y \neq \emptyset$, there exists a maximal element Z of $\leq$ such that $X \leq Z$ and $Y \leq Z$. By the definition of $\leq$ this implies $v(X, Y)$. $\Box$

Proof of Safety. Assume that the implementation invokes $\mathit{notify}(Y, n)$ at client Y. We have to show that this implies that the three conjuncts of the implication in the safety property of Def. 2 hold.

The first conjunct follows directly from the safety property of the simple event system.

To prove the second and the third conjunct, assume that the local broker issues $\mathit{notify}(Y, n)$ at client Y. This means that (a) the proxy of Y has previously received a notification $\tilde{n} = (n, R)$ and that (b) the test $V_Y \cap R \neq \emptyset$ succeeded.

From (a) and the safety property of the simple event system follows that $\tilde{n}$ was previously published by some proxy Prox_X. From Lemma 1 and (b) follows that $v(X, Y)$ holds. This proves the second conjunct.

From (a) and the safety property of the simple event system follows that $\tilde{n}$ matches some transformed filter $\tilde{F}$ of Prox_Y. This together with the algorithm proves the third conjunct. This concludes the proof of the safety property.

Proof of Liveness. Assume a client Y invokes $\mathit{sub}(Y, F)$ and never unsubscribes to F. From the algorithm we have that an "equivalent" subscription $\tilde{F}$ is issued into the simple event system. Since we are dealing with static scopes, the values of V_X and V_Y do not change. From Lemma 1 this implies that $v(X, Y)$ is always true for all clients X and Y for which $V_X \cap V_Y \neq \emptyset$.

From the liveness property of the simple event system and the algorithm follows that there is a point in time after which every published event of a notification $\tilde{n} = (n, R)$ which matches $\tilde{F}$ is delivered to every client proxy. So assume that after this point in time some client X publishes a notification n matching F. From the algorithm we have that $\tilde{n} = (n, V_X)$ is published within the simple event system. Its liveness property gives us that $\tilde{n}$ is eventually delivered at the client proxy of Y. From the algorithm and because $v(X, Y)$ holds, the test $V_X \cap V_Y \neq \emptyset$ will succeed and Y will eventually be notified of n.

5 Scoped Event-Based Systems with Event Mappings

The previous specifications assumed the underlying event data and filter model, which describes notification and filter expression syntax, to be uniform in the whole event system. However, in distributed systems divergent application needs might be met best with multiple underlying data models, without impeding cooperation thereby. We provide a specification and an implementation for a scoped event system with event mappings, which transform notification representation when passing through the scope graph. The mappings are required to be static in the same sense as the scopes are: Changes are limited to components whose published events have already been notified to all visible peers.

5.1 Motivation: Coping with Heterogeneity

The discussion on the flexible configuration of notification services in Sect. 4.1 can be carried on: a single uniform event notification service with uniform syntax and semantics is not able to cope with the requirements of distributed systems operating in heterogeneous environments. For example, various event systems published in literature claim to be scalable, and on the other hand, assume a global naming scheme for notification representation [32,27]. All clients of such a service are forced to agree on the same data model used for all notifications. This impedes system integration and limits scalability. With an appropriate support, one part of an application can exchange binary encoded notification while still being able to communicate via serialized Java objects or XML encoded notifications with other parts of the system. Efficiency considerations lead to the distinction between low-volume external representations in XML and more efficient, optimized internal representations. Another aspect is the semantics of notifications, which is also likely to vary in heterogeneous environments [30], and any solution addressing heterogeneity in event-based systems should at least offer a starting point for including respective enhancements.

Supporting heterogeneity requires to transform notifications that cross the border of applications between external and internal representations. From the observations about decomposing applications in the previous section, we deduce the requirement that bundling of related components should not only encapsulate functionality but also delimit common syntax and semantics. For this purpose, we define event mappings and attach them to individual scopes to fulfill two tasks. First, they act as filters that explicitly permit only a specific set of events to be published and consumed, subsuming the interfaces of scopes as they are described earlier. Second, all notifications crossing a scope boundary are subject to transformation to map between internal and external representations. Scopes are an appropriate place to localize such transformations in the system, because any presentational differences directly give rise to distinguish the respective parts as different scopes.

5.2 Specification

We combine the two tasks and map an outer notification n, which comes from a superscope, to an inner notification n' which is forwarded to the subscopes. If a mapping results in the empty notification $\epsilon \notin \mathcal{N}$, it is not forwarded. The empty event ϵ is introduced to achieve a blocking behavior of the mappings. This blocking mechanism may be used to subsume filters into the mapping concept. Outgoing events are handled vice versa.

Event mappings are formally defined as relations on scope "boundaries." Briefly spoken, scope boundaries are the edges between the nodes in the scope graph G. With every such edge we associate two binary, asymmetric relations $\nearrow$ and $\searrow$ over the set $\mathcal{N}$ of notifications. Let n_1 and n_2 be two notifications. For any edge e and its associated relation $\nearrow_e$, the mapping $n_1 \nearrow_e n_2$ means that when "traveling" upwards along the edge (i.e., in direction of the superscope) n_1 is transformed into n_2. The relation $\searrow_e$ is defined analogously for the reverse direction.

Using the relations, we can now define a relation $\sim$ over $\mathcal{N} \times \mathcal{K}$ that extends the visibility $v(X, Y)$:

$$\begin{aligned}
&(n_1, X) \sim (n_2, Y) \Leftrightarrow \\
&\qquad \big(X = Y \wedge n_1 = n_2\big) \\
&\qquad \vee \big(\exists X' \in \mathit{super}(X)\,.\, \exists n'.\quad n_1 \nearrow n' \\
&\qquad\qquad\qquad\qquad \wedge \big[(n', X') \sim (n_2, Y)\big]\big) \\
&\qquad \vee \big(\exists Y' \in \mathit{super}(Y)\,.\, \exists n'.\quad n' \searrow n_2 \\
&\qquad\qquad\qquad\qquad \wedge \big[(n_1, X) \sim (n', Y')\big]\big)
\end{aligned}$$

In the previous definition, $\nearrow$ and $\searrow$ are the relations associated with the edge which is referenced by *super*. The recursive definition of $\sim$ can be best understood by looking at Figure 5. Intuitively, $(n_1, X) \sim (n_2, Y)$ means that notification n_1 can "flow" from X to Y and is received as notification n_2 (which might be different from n_1). The path on which n_1 flows to n_2 is similar to the visibility relation defined in Section 4, i.e., it can be characterized by a path from X up to a common superscope and then down to Y. The visibility of n_2 is additionally determined by the event mappings along this path and their possibility to block and discard notifications.

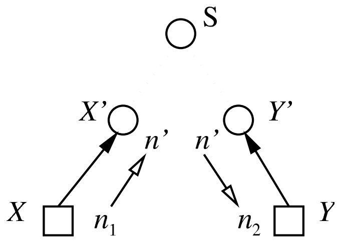

Fig. 5. Recursive definition of the relation $(n_1, X) \sim (n_2, Y)$.

We are now ready to define the semantics of a scoped event system with event mappings. Like the graph of scopes, the relations $\nearrow$ and $\searrow$ are required to be static in that a component's mappings are not allowed to change until all of its published events are notified; otherwise the visibility clause may corrupt the safety condition in the specification.

Definition 3 (scoped event system with event mappings). *A* scoped event system with event mappings *is a system that exhibits only traces satisfying the following requirements:*

– *(Safety)*

$$\Box\Big[\mathit{notify}(Y,n') \;\Rightarrow\; \big[\bigcirc\Box\neg\mathit{notify}(Y,n')\big] \wedge \big[\exists n.\, \exists X.\, n \in P_X \;\wedge\; \big((n,X) \sim (n',Y)\big)\big] \wedge \big[\exists F \in S_Y.\, n' \in N(F)\big]\Big]$$

– *(Liveness)*

$$\Box\Big[\mathit{sub}(Y,F) \Rightarrow \Big(\Diamond\big[\Box\big((n,X) \sim (n',Y)\big) \;\Rightarrow\; \Box\big(\mathit{pub}(X,n) \wedge n' \in N(F) \;\Rightarrow\; \Diamond\mathit{notify}(Y,n')\big)\big]\Big) \vee \Big(\Diamond\mathit{unsub}(Y,F)\Big)\Big]$$

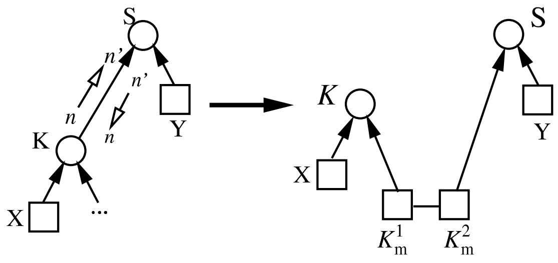

Fig. 6. Transformation of mappings into components.

The difference between Definitions 3 and 2 is that the term $v(X, Y)$ is replaced by the term $(n, X) \sim (n', Y)$ and that the published event n is not necessarily the same as the delivered event n'. This formulation captures the notion that in addition to being visible with respect to scoping the flow of notifications must be permitted by the passed input/output interfaces, i.e., the event mappings. Also, the notification n' is the result of repetitive applications of the relations $\nearrow$ and $\searrow$ along the path implicitly defined by $\sim$.

Note that Definition 3 is a generalization of Definition 2. This is because a scoped event system can be regarded as one with event mappings where all event mappings are the identity relation (i.e., they do not change anything along the way). In such a system, $v(X, Y)$ is implied by the existence of a notification n such that $(n, X) \sim (n, Y)$. Note also that if a component has multiple superscope which in turn have a common superscope, a component attached to this superscope might receive more than one mapped version of a notification. Our specification does not rule out this case because a component might be interested in from which scope the notification comes from. If this is not desired, the specification must be strengthened appropriately.

5.3 Implementation

The implementation of a scoped event system with mappings $ES^{\mathcal{M}}$ is based on a scoped system $ES^{\mathcal{S}}$ and a transformation of the graph of scopes G that essentially follows the idea of adding activity to edges. Figure 6 sketches the transformation that creates G' by exchanging every edge (K, S) that does not apply the identity mappings $n \nearrow n$ and $n \searrow n$ for two extra mapping components K_m^1 and K_m^2. By inserting *one* K_m we would be able to add some form of activity to an edge. *Two* mapping components are required to constrain the visibility of the transformed notifications to the appropriate scopes.

Figure 7 describes the architecture of the implementation for the example system in Figure 6. A component X connected to $ES^{\mathcal{M}}$ is also directly connected to an underlying scoped event system $ES^{\mathcal{S}}$. Calls to $pub(X, n)$ of $ES^{\mathcal{M}}$ are forwarded to $ES^{\mathcal{S}}$ without changes, and vice versa, calls to $notify(X, n)$ of $ES^{\mathcal{S}}$ are forwarded to $ES^{\mathcal{M}}$.

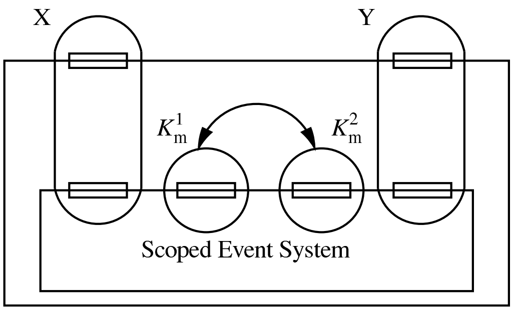

Fig. 7. Architecture of scoped event system with mappings.

In general, if a scope K is to be joined to a superscope S by calling $\mathit{jscope}(K, S)$, two mapping components K_m^1 and K_m^2 are created that communicate directly via a point-to-point connection. K_m^1 joins K, subscribes to all notifications published in K, transforms and forwards them to its peer. Furthermore, subscriptions in K have to be transformed before they are forwarded. The implementation relies on externally supplied functions that map notifications and filters/subscriptions between the internal and external representations in K and S, respectively. K_m^2 joins S and republishes all notifications it gets from its peer K_m^1. It subscribes in S according to the subscriptions forwarded by K_m^1, transforms any notifications received out of S, again with externally supplied functions, and forwards them to K_m^1 which republishes them into K.

5.4 Correctness of the Scoped Event System with Event Mappings

We must show that the algorithm from the previous section satisfies the requirements given in Definition 3, i.e., the safety and liveness condition. The correctness proof largely depends on the correctness of the underlying scoped event system $ES^{\mathcal{S}}$. We first state a helpful lemma.

Lemma 2. *If $(n, X) \sim (n', Y)$ holds, then in the implementation of $ES^{\mathcal{M}}$ exists a sequence $\rho = C_1, C_2, \ldots, C_m$ of components for which holds:*

1. *$C_1 = X$ and $C_m = Y$,*
2. *for all $1 < i < m$ holds that C_i is a mapping component, and*
3. *for all $1 \leq i \leq m-1$ holds that C_i and C_{i+1} either share a communication link or reside in the same scope of $ES^{\mathcal{S}}$.*

Proof: Assume $(n, X) \sim (n', Y)$ holds. From the definition of $\sim$ follows that there exists a sequence $\tau = X, D_1, D_2, \ldots, D_l, Y$ of complex components in the scope graph G. Since for all components Z, $\mathit{super}(Z)$ contains only scope components, all D_i must be scopes. There also exists a root scope D_j which is the "highest" scope in G (D_j is unique for every such path τ).

The construction method of building G' from G implies that every consecutive pair of scopes (D_i, D_{i+1}) in τ is enhanced with two mapping components K_i^1 and K_i^2 which are joined by a communication link. The mapping components K_i^2 and K_{i+1}^1 of neighboring edges reside in the same scope D_{i+1}. The projection of τ to mapping components (and X and Y) yields a sequence $X, K_1^1, K_1^2, K_2^1, K_2^2, K_3^1, \dots, K_l^2, Y$ which is the witness for the sequence ρ of the lemma. $\square$

Proof of Safety. Assume that Y is a simple component and that *notify*(Y, n') of $ES^{\mathcal{M}}$ is called. We have to show that the three conjuncts of the implication in the safety property of Def. 3 hold.

From the algorithm description follows that *notify*(Y, n') of $ES^{\mathcal{S}}$ was called before, implying that n' is notified at most once and that n' matches an active subscription of Y. This proves the first and the third conjunct.

The second conjunct is proved by a backward induction on the path guaranteed by Lemma 2. The fact that Y is notified about n' implies that there is a component Z that has published n' which resides in the same scope. If this Z is not a mapping component, Z plays the role of X in the formula, $n' = n$, and the second conjunct follows immediately (this is the base case of the induction). The step case of the induction is as follows: Assume that a component Z'' along the path has published some notification n'' which from backward event mappings resulted from n'. Then there exists a component Z''' which is either in the same scope or connected by a communication link to Z''. In the first case, the step follows from the properties of $ES^{\mathcal{S}}$ and in the second case from the algorithm. This implies that $n \in P_X$ and that $\big((n, X) \sim (n', Y)\big)$, giving the second conjunct.

Proof of Liveness. The liveness property is proved by forward induction on the path guaranteed by Lemma 2 in a similar way as in the proof of the safety property. Assume that Y subscribes to F and never unsubscribes. Then assume that after subscribing, $(n, X) \sim (n', Y)$ begins to hold indefinitely. Then Lemma 2 guarantees a path between any publisher X of a relevant notification n and Y. A similar way of reasoning as in the safety proof implies that n is forwarded and transformed along the path resulting in n' which Y is eventually notified about.

6 Related Work

The ideas of the event-based style are used in a wide range of computer science areas, although terminology and meaning vary. The definition given in Sect. 2.1 is mainly inspired by the work in distributed event notification systems [25] and suits the comparison of related work, which can be found in the following areas:

- Rule-based systems,
- Active Database Systems,
- Distributed notification services,
- Coordination models,

– Software engineering
  (implicit invocation, software architecture, distributed debugging),
– GUI design.

Rule-based Systems. A rule-based system (RBS) consists of a set of if-then rules, a set of facts, and some interpreter controlling the application of the rules [57]. The evaluation and matching of conditions ('antecedents') and execution of an assigned action ('consequents') is similar to the event-based approach of publishing notifications, matching subscriptions, and delivering the notification data to the application program. Work on RBS typically concentrate on condition detection [41] and rule languages [69], whereas coordination models and especially event services look at the flow of notifications in distributed systems. On the other hand, these areas seem to complement each other with respect to rule or subscription construction and management. Rule groups and modularity are investigated by Browne et al. [16] and are used in a number of systems. However, the rule groups are connected by explicit procedural invocation and the modularity cannot be used to create new event-based components.

Active Database Systems. Event-condition-action (ECA) rules are used in active database management systems (ADBMS) as *knowledge model* to encode reactive behavior [86]: If a given event occurs and if the condition holds, the associated action is executed. Events, i.e., notifications in our terminology, can be raised by a variety of sources: data changes, transactions, clock ticks, or external source are common. Rule processing is controlled by an *execution model* that determines, for example, transactional contexts, coupling modes, and consumption modes [18].

The rules control and react to data changes and support the trend in application design to decouple coordination of shared databases from application computation; it is called 'knowledge independence' [44] here and this view resembles the coordination paradigm [82]. Internal and external use of rules is distinguished. The former implements classic database functionality like integrity constraints and the latter is used to encode application dependent constraints and triggers, e.g., bank account control.

Research on rule management concentrated on termination and confluence properties of rules [5] and only little work is published that addresses the engineering complexity of ECA-rule creation and management. Baralis et al. propose stratification of rules as a modularization technique in order to simplify termination testing [11]. Each rule is assigned to one stratum according to a given metric and the strata are ordered by priority so that rules with lower priority will not influence those of higher priority; termination testing can now be done for each isolated stratum in turn. The presented metrics are behavioral, assertional, and event-based stratification. The first subsumes the other two, and it groups rules based on functionality, separating levels of abstraction in a way. Assertional stratification tries to ensure a predefined postcondition with the rules of the stratum. Finally, event-based stratification orders rules so that lower strata do not produce events consumed by the upper ones. This approach can be seen as a simple design methodology, which also lends itself to termination analysis.

Rule patterns offer a mechanism to bundle a parameterized set of rules and capture a policy in an application-independent way [59]. They facilitate constructing complex reactive behavior out of simpler rules and in that offer a modularization technique, although recursive bundling of rules and the creation of new, first class components is not directly supported.

Notification services. The work presented in this article originated from the area of distributed event notification systems in which a considerable amount of work exists and many concrete systems have been designed and implemented (e.g., Siena [27], JEDI [32], Gryphon [9], and others [34,64,37]). Unfortunately, it is very difficult to compare these systems because of different or informal semantics, even though necessary tools are being developed recently [101].

For example, in the Siena system [24,27], Carzaniga, Rosenblum and Wolf make a good effort at defining the semantics of subscription mechanisms. However, the semantics of notification distribution is not clearly specified and has several flaws. For example, they demand that a notification is only delivered to a client if the client had a matching subscription at the time the notification was published. This is difficult to detect (and to implement) in a distributed system. Moreover, clients are required to accommodate to race conditions. For example, notifications may be delivered after cancellation of the respective subscriptions. Finally, in Siena, a client that unsubscribes to a filter implicitly unsubscribes to all filters that are covered by the former filter, too. This approach burdens the client to keep track of relations among the issued subscriptions.

In most other systems, practitioner's approaches dominate and at most the formal semantics of the subscription languages are given [4], neglecting the semantics of the event service itself. The formal specification presented in this article constitutes a basis to reason about event-based semantics and the correctness of implementations, as it is used in this paper.

Only a limited amount of initial work exists on structuring event-based systems. The Siena event notification service is a popular example of a service utilizing content-based filtering [27]. A thorough presentation of filtering semantics and design choices is given, focusing on network bandwidth efficiency but neglecting any engineering support. As for all other content-based filtering approaches, the filters may be used to realize visibility constraints, but these issues are not explicitly addressed, and additionally, the flat and global namespace of notification attributes impedes scalability.

An event-service for virtual reality applications is proposed by O'Connell et al. [79]. A hierarchy of *zones* is defined which limit the distribution of events for efficiency reasons. Events can only cross the boundary in downward direction and no other features of scopes are mentioned.

The READY event notification service introduce event zones which are used to partition components based on (either) logical, administrative, or geographical boundaries and which delimits the visibility of events [55]. Boundary routers connect zones and control the communication in between. But a component belongs to exactly one zone so that there is only a two-level hierarchy, and the system is structured only based on one specific point of view, prohibiting composition and mixing of aspects [56]. It is only mentioned that concepts from group communication [88] may be applicable, promising the flexibility of changing notification delivery semantics. READY is a prime example of an initial study of visibility issues that identifies the problem but fails to integrate the different aspects of visibility.

Heterogeneity issues are also addressed by READY: the boundary routers apply transformations on crossing notifications. Although similar to the event mappings presented in Sect. 5, the routers operate on a rather coarse and static granularity. While being well recognized in traditional request/reply systems, heterogeneity issues were mostly neglected in event systems and only recently gained attention [28,30]. The event mappings provide the facility to integrate these results into the scoping model.

The event channels of the Corba notification service [78] offer a structuring mechanism in that notifications are only visible within the channel in which they were published. Channels can be connected to integrate 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. Additionally, CORBA event management domains [54] support the management of a graph of interconnected channels. It provides a uniform management interface, but does not offer any of the advanced features like transmission policies.

In subject-based addressing schemes for notification delivery, a tree of subjects is used to partition and select notifications [80]. But the simplicity of the model results in severe disadvantages. Similar to selecting event channels, producers assign the subjects and the predefined tree of subjects constrain the view onto the system, impeding different points of view and thereby composition [56]. Nevertheless, the simplicity of the concept led to wide acceptance and a multitude of implementations that are also commercially available, e.g., from TIBCO [98], IBM (MQseries), and others. Most systems extend the basic characteristics and support additional features such as bridges connecting multiple busses, integration of transactional activities, and security considerations. Nevertheless, the expressiveness is rather limited and cannot be used to control visibility in an appropriate way.

The Java Message Service [95] coined the term topic-based subscription that encompasses a subject tree selection plus content-based filtering on a set of header fields and properties, similar to Corba notification and other content filtering systems (Siena).

Coordination models. As outlined in the introduction, the idea behind the event-based style directly corresponds to characteristics of the coordination paradigm. Different coordination models have been proposed in the literature, all of which try to integrate a number of components, but not all of them reduce component interdependencies, let alone offer scalability. For example, it was criticized that race conditions are possible in Linda [52] and its variants, resulting from the inherent concurrency of the model [3].

Research on Linda-like systems investigated structures of components. Agha and Callsen propose ActorSpaces to limit the distribution of messages [3]. 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. Merrick and Wood introduce scopes to limit the visibility of tuples in Linda, but again, senders have to specify destination scopes [68]. Furthermore, scope nesting is restricted to two levels. Customization and adaptation of the semantics of tuple space operations is the major feature of tuple centers [81], following an idea similar to our transmission policies. Their general programmability and the automatic triggering of reactive behavior might lend them as an appropriate technique to implement scopes, if the necessary notification of components is programmed into the centers so that a single center implements an assigned scope coordinating the distribution of notifications to its sub- and super-components. With their general approach to implement tuple spaces they resemble the connectors idea of Sullivan and Notkin [92].

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's role and the need for externally provided configuration mechanisms that do not change instantiated components. The need to specify names or identities of tuple spaces is a major characteristic of many works on multiple tuple spaces [53,23]. In this way, the same negative arguments hold as for the manual selection of event channels and subjects, both draw coordination control into the application components. Lime [75] realizes a transparent access to multiple tuple spaces, 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 does not offer a general solution.

Different from Linda, event-based coordination media directly support the event-based style [20,19,101]. Moreover, control-driven models such as Manifold more strictly separate the coordination issues [83,82], though the event-based style is not directly supported, and in the case of Manifold even only used for configuration control. Commercially available are implementations of JavaSpaces [94], which also implement notification mechanisms to notify about newly inserted tuples that match a previously defined pattern.

Viroli et al. [99,100] considered formalizing event-based coordination media and proposed a grey-box approach to specification. The specification focuses on the comparability of different coordination media and not on issues of distribution. Combining their approach with ours in order to study distribution issues is worthwhile.

Software engineering. The event-based architectural style is identified in software architecture, which deals with the overall software system organization [48]. It constrains the possibly topology and is based on *implicit invocation* mechanisms [46,33]. This term describes the loose coupling from a software engineering point of view in that the invocation of a procedure is divided into three parts: the call on the caller's side, the procedure binding introducing a one-to-many indirection, and the (implicit) invocation of the mapped procedures. Handling of replies is also regarded, rendering implicit invocation a mere implementation technique that may be used to realize either anonymous request/reply or event-based cooperation models. The definition of the event-based style, however, intuitively matches the approach given in Sect. 2.2.

Garlan and Scott presented delivery policies for implicit invocation systems [47]. Four different delivery policies are distinguished: full (broadcast) and single delivery (1-of-n), parameter-based selection (filter), and a state-based policy. These policies are a subset of the transmission policies described in Sect. 4.4 and [38]. From a programmers point of view, the observer design pattern describes a publish/subscribe interaction [45], in which a subject manages a set of observes that are called whenever the subject changes.

Sullivan and Notkin introduce mediators [92] 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 [89] is an early work on tool integration and it is built around a centralized server that distributes messages. Messages sent to the server are selectively re-broadcasted to receivers that registered patterns matching the message. The original approach realizes 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 that is not bound to any application structure. Later extensions made it possible to limit the visibility of messages to a set of receivers, but did not support composition and interfaces.

The InfoBus [31] is a small Java API that facilitates communication between several JavaBeans or cooperating applets on a Web page. 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.

Cardelli and Gordon propose a process calculus for mobile ambients [21]. 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, similar to the approach of Bauhaus Linda [23].

Ported objects [67] 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, taking up the idea of control-driven coordination as in Manifold [82]. 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, extending the Manifold approach and missing all other scope features, though.

Evans and Dickman defined 'zones' in order to support partial system evolution [36]. The meta object protocol [60] 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 here.

An early application of the event-based style can be found in the distributed debugging literature [63,13].

GUI Design. The design and implementation of graphical user interfaces was one of the first areas applying event-based interaction schemata. The model view control and observer patterns [45] and the toolkits mentioned in the previous paragraphs are examples therefor. However, mostly a callback mode of interaction is used, in which an observer directly connects itself to a previously known source of events. Furthermore, these systems are typically not distributed.

7 Conclusions

The basic concept of visibility and the related problems are of fundamental nature to realize and control loosely coupled event-based systems. While some of the problems are identified and addressed in existing work, no other approach in the area of event-based systems is based on the notion of visibility.

We have introduced the notion of scopes as a powerful structuring mechanism for event-based systems. Scopes can help to hide internal configurations, to delimit administrative domains, and especially to make an application structure explicit. Further functionality is bound to this structure, controlling the visibility of events, their representation, and distribution. Transmission policies and event mappings offer the ability to 'program' scopes by adapting and customizing the event system to application-dependent needs, while separating customization from application functionality. The presented system model allows for coping with communication demands that deviate from the default event-based semantics and for providing support for heterogeneous processing environments.

We have also given a formal specification of event systems and scopes and event-mappings within a trace-based formalism adapted from temporal logic. The specifications are built up and refined in a modular way. The presented implementations follow this modularization and enables system designers to draw from existing work if it conforms to the underlying specifications. Conformance of the presented implementations is proven with informal arguments, showing the feasibility of the concept. The features are currently evaluated within REBECA, our prototype event system implementation [39].

A number of question for future work are raised. We wish to study systems with dynamic scopes and the integration of efficient network protocols to implement scopes. Other points are heterogeneity and the integration of appropriate supporting infrastructures, design methodologies for event-based systems, and development support in terms of languages describing scopes and tools facilitating composition and administration.

Acknowledgments

We gratefully acknowledge the comments of the anonymous reviewers who provided helpful comments leading to significant improvements of the presentation, and Mira Mezini and Alejandro Buchmann for their contributions to the development of the scoping concept.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.
2. M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan. 1993.
3. G. Agha and C. J. Callsen. ActorSpace: an open distributed programming paradigm. *ACM SIGPLAN Notices*, 28(7):23–32, July 1993.
4. M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC 1999)*, pages 53–61, Atlanta, Georgia, USA, 1999.
5. A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: termination, confluence, and observable determinism. In M. Stonebraker, editor, *Proceedings of the ACM International Conference on Management of Data (SIGMOD 1992)*, pages 59–68, San Diego, CA, USA, 1992. ACM Press.
6. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
7. F. Arbab and C. Talcott, editors. *5th International Conference on Coordination Models and Languages (COORDINATION 2002)*, volume 2315 of *LNCS*, York, UK, 2002. Springer.
8. J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic support for distributed applications. *IEEE Computer*, 33(3):68–76, 2000.

9. G. Banavar, M. Kaplan, K. Shaw, R. Strom, D. Sturman, and W. Tao. Information flow based event distribution middleware. In W. Sun, S. Chanson, D. Tygar, and P. Dasgupta, editors, *ICDCS Workshop on Electronic Commerce and Web-based Applications/Middleware*, pages 114–121, Austin, TX, USA, 1999.
10. E. Baralis. Rule analysis. In N. W. Paton, editor, *Active Rules in Database Systems*, Monographs in Computer Science, chapter 3, pages 51–67. Springer-Verlag, 1999.
11. E. Baralis, S. Ceri, and S. Paraboschi. Modularization techniques for active rules design. *ACM Transactions on Database Systems (TODS)*, 21(1):1–29, 1996.
12. J. Bates, J. Bacon, K. Moody, and M. Spiteri. Using events for the scalable federation of heterogeneous components. In P. Guedes and J. Bacon, editors, *Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, Sintra, Portugal, Sept. 1998.
13. P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, Feb. 1995.
14. J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra.* North-Holland, 2001.
15. J. C. Browne et al. A new approach to modularity in rule-based programming. In *Proceedings of the 6th International Conference on Tools with Artificial Intelligence (ICTAI '94)*, pages 18–25, New Orleans, LA, USA, Nov. 1994. IEEE Computer Society.
16. J. C. Browne et al. Modularity and rule-based programming. *Intl. Journal on Artificial Intelligence Tools*, 4(1):201–218, 1995.
17. M. Broy and E.-R. Olderog. Trace-oriented models of concurrency. In Bergstra et al. [14], chapter 2.
18. A. P. Buchmann. Architecture of Active Database Systems. In N. W. Paton, editor, *Active Rules in Database Systems*, Monographs in Computer Science, chapter 2, pages 29–48. Springer-Verlag, 1999.
19. N. Busi, A. Rowstron, and G. Zavattaro. State- and event-based reactive programming in shared dataspaces. In Arbab and Talcott [7], pages 111–124.
20. N. Busi and G. Zavattaro. On the expressiveness of event notification in data-driven coordination languages. In G. Smolka, editor, *Proceedings of 9th European Symposium on Programming Languages and Systems (ESOP 2000)*, volume 1782 of *LNCS*, pages 41–55, Berlin, Germany, 2000.
21. 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, Lisbon, Portugal, 1998. Springer-Verlag.
22. N. Carriero and D. Gelernter. Linda in context. *Communication of the ACM*, 32(4):444–458, Apr. 1989.
23. N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems, ECOOP'94 Workshop*, volume 924 of *LNCS*, pages 66–76, Bologna, Italy, 1995. Springer-Verlag.
24. A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks.* PhD thesis, Politecnico di Milano, Milano, Italy, Dec. 1998.
25. A. Carzaniga, E. Di Nitto, D. S. Rosenblum, and A. L. Wolf. Issues in supporting event-based architectural styles. In *Proceedings of the Third International Workshop on Software Architecture (ISAW '98)*, pages 17–20, Orlando, FL, USA, 1998.

26. A. Carzaniga, D. S. Rosenblum, and A. L. 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.
27. 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.
28. S. Chakravarthy and R. Lee. ECA Rule Support for Distributed Heterogeneous Environments. In *Proceedings of the 14th International Conference on Data Engineering (ICDE '98)*, page 601, Orlando, FL, USA, 1998. IEEE Computer Society Press.
29. P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys (CSUR)*, 28(2):300–302, 1996.
30. M. Cilia, C. Bornhövd, and A. P. Buchmann. Moving active functionality from centralized to open distributed heterogeneous environments. In C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, editors, *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS '01)*, volume 2172 of *LNCS*, pages 195–210, Trento, Italy, 2001. Springer-Verlag.
31. M. Colan. *InfoBus 1.2 Specification*. Lotus, 1999.
32. 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*, 27(9), 2001.
33. J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. In *Proceedings of of the 6th International Symposium on the Foundations of Software Engineering (FSE-6)*, Lake Buena Vista, FL, USA, Nov. 1998. ACM Press.
34. G. E. Dong Zhou, Karsten Schwan and Y. Chen. Jecho–interactive high performance computing with java event channels. In *Intl. Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, CA, USA, 2001.
35. P. T. Eugster, R. Guerraoui, and C. H. Damm. On objects and events. In L. Northrop and J. Vlissides, editors, *Proceedings of the OOPSLA '01 Conference on Object Oriented Programming Systems Languages and Applications*, pages 254–269, Tampa Bay, FL, USA, 2001. ACM Press.
36. 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, Jyväskylä, Finnland, 1997. Springer-Verlag.
37. F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In T. Sellis and S. Mehrotra, editors, *Proceedings of the 20th Intl. Conference on Management of Data (SIGMOD 2001)*, pages 115–126, sanat Barbara, CA, USA, 2001.
38. L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. Engineering event-based systems with scopes. In B. Magnusson, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *LNCS*, pages 309–333, Malaga, Spain, June 2002. Springer-Verlag.
39. L. Fiege and G. Mühl. Rebeca Event-Based Electronic Commerce Architecture, 2000. http://www.gkec.informatik.tu-darmstadt.de/rebeca.
40. 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.
41. C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

42. M. J. Franklin and S. B. Zdonik. A framework for scalable dissemination-based systems. In A. M. Berman, M. Loomis, and T. Bloom, editors, *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '97)*, Atlanta, GA, USA, Oct. 5–9, 1997.
43. M. J. Franklin and S. B. Zdonik. "Data In Your Face": Push Technology in Perspective. In L. M. Haas and A. Tiwary, editors, *Proceedings ACM SIGMOD International Conference on Management of Data (SIGMID '98)*, pages 516–519, Seattle, USA, 1998. ACM Press.
44. O. Friesen, G. Gauthier-Villars, A. Lefebvre, and L. Vieille. Applications of deductive object-oriented databases using del. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 1–22. Kluwer Academics, 1994.
45. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, USA, 1995.
46. D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In S. Prehn and W. J. H. Toetenel, editors, *VDM '91: Formal Software Development Methods*, volume 551 of *LNCS*, pages 31–44, Noordwijkerhout, The Netherlands, 1991. Springer-Verlag.
47. D. Garlan and C. Scott. Adding implicit invocation to traditional programming languages. In V. R. Basili, R. A. DeMillo, and T. Katayama, editors, *Proceedings of the 15th Intl. Conference on Software Engineering (ICSE '93)*, pages 447–455, Baltimore, MD, USA, 1993. IEEE Computer Society Press / ACM Press.
48. D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.
49. F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1), Mar. 1999.
50. F. C. Gärtner. *Formale Grundlagen der Fehlertoleranz in verteilten Systemen.* PhD thesis, TU Darmstadt, 2001.
51. D. Gawlick. Messaging/queuing in Oracle8. In *Proceedings of 14th International Conference on Data Engineering (ICDE '98)*, Orlando, FL, USA, 1998. IEEE Press.
52. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.
53. D. Gelernter. Multiple tuple spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Parallel Architectures and Languages Europe (PARLE '89)*, volume 366 of *LNCS*, pages 20–27, Eindhoven, The Netherlands, 1989. Springer-Verlag.
54. O. M. Group. Management of event domains. Final Adopted Specification, 2000. dtc/00-08-01.
55. R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the READY event notification service. In P. Dasgupta, editor, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, Middleware Workshop*, Austin, TX, USA, May 1999.
56. 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, Washington, DC, USA, 1993.
57. F. Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28(9):921–932, Sept. 1985.
58. ISO/IEC. Open distributed processing–reference model (odp-rm). International Standard ISO/IEC IS 10746, May 1995.

59. G. Kappel, S. Rausch-Schott, W. Retschitzegger, and M. Sakkinen. From rules to rule patterns. In P. Constantopoulos, J. Mylopolous, and Y. Vassiliou, editors, *Proc. of the 8th Intl. Conference on Advanced Information Systems Engineering (CAiSe'96)*, volume 1080 of *LNCS*, pages 99–115, Heraklion, Crete, Greece, 1996. Springer-Verlag.
60. G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol.* MIT Press, Cambridge, MA, USA, 1991.
61. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, Mar. 1977.
62. Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Proceedings of the IFIP Congress on Information Processing*, pages 657–667, Amsterdam, 1983. North-Holland.
63. C.-C. Lin and R. J. LeBlanc. Event-based debugging of object/action programs. In R. L. Wexelbalt, editor, *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, volume 24(1) of *SIGPLAN Notices*, pages 23–34, Madison, WI, USA, 1988. ACM Press.
64. C. Ma and J. Bacon. COBEA: A CORBA-based event architecture. In J. Sventek, editor, *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS-98)*, pages 117–132, Santa Fe, NM, USA, 1998. USENIX Association.
65. T. W. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.
66. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.
67. J. McAffer. Meta-level programming with CodA. In W. Olthoff, editor, *Proceedings of the European Conference for Object-Oriented Programming (ECOOP '95)*, volume 952 of *LNCS*, Aarhus, Denmark, 1995. Springer-Verlag.
68. 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.
69. D. P. Miranker, L. Obermeyer, L. Warshaw, and J. C. Browne. Venus: An object-oriented extension of rule-based programming. Technical report, University of Texas at Austin, 1998.
70. G. Mühl. Generic constraints for content-based publish/subscribe systems. In C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, editors, *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS '01)*, volume 2172 of *LNCS*, pages 211–225, Trento, Italy, 2001. Springer-Verlag.
71. G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems.* PhD thesis, Darmstadt University of Technology, 2002.
72. G. Mühl and L. Fiege. Supporting covering and merging in content-based publish/subscribe systems: Beyond name/value pairs. *IEEE Distributed Systems Online (DSOnline)*, 2(7), 2001.
73. G. Mühl, L. Fiege, and A. P. Buchmann. Filter similarities in content-based publish/subscribe systems. In H. Schmeck, T. Ungerer, and L. Wolf, editors, *International Conference on Architecture of Computing Systems (ARCS)*, volume 2299 of *Lecture Notes in Computer Science*, pages 224–238, Karlsruhe, Germany, 2002. Springer-Verlag.
74. G. Mühl, L. Fiege, F. C. Gärtner, and A. P. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In A. Boukerche and S. Majumdar, editors, *The Tenth IEEE/ACM International Symposium on*

Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002), Fort Worth, TX, USA, October 2002. IEEE Press.

75. 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, Phoenix, AZ, USA, May 2001.
76. Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Version 2.3. Object Management Group, Framingham, MA, USA, 1998.
77. Object Management Group. *CORBA Components.* OMG, Framingham, MA, USA, 1999. orbos/99-07-01.
78. Object Management Group. Corba notification service. OMG Document telecom/99-07-01, 1999.
79. K. O'Connell, T. Dinneen, S. Collins, B. Tangney, N. Harris, and V. Cahill. Techniques for handling scale and distribution in virtual worlds. In A. Herbert and A. S. Tanenbaum, editors, *Proceedings of the 7th ACM SIGOPS European Workshop*, pages 17–24, Connemara, Ireland, Sept. 1996. ACM SIGOPS.
80. 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, Asheville, NC, United States, Dec. 1993. ACM Press.
81. A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, Nov. 2001.
82. G. A. Papadopoulos and F. Arbab. Coordination models and languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*. Academic Press, Aug. 1998.
83. G. A. Papadopoulos and F. Arbab. Modelling Activities in Information Systems using the coordination language Manifold. In K. M. George and G. B. Lamong, editors, *Proceedings of the ACM Symposium on Applied Computing (SAC '98)*, pages 185–193, Atlanta, GA, USA, 1998. ACM Press.
84. G. A. Papadopoulos and F. Arbab. Configuration and dynamic reconfiguration of components using the coordination paradigm. *Future Generation Computer Systems*, 17(8):1023–1038, June 2001.
85. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.
86. N. W. Paton and O. Diaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63–103, 1999.
87. A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
88. D. Powell. Group communication. *Communications of the ACM*, 39(4):50–53, Apr. 1996.
89. S. P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.
90. A. Ricci, A. Omicini, and E. Denti. Objective vs. subjective coordination in agent-based systems: A case study. In Arbab and Talcott [7], pages 291–299.
91. K. J. Sullivan and D. Notkin. Reconciling environment integration and component independence. In R. N. Taylor, editor, *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 22–33, Irvine, CA, USA, 1990. ACM Press.

92. 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.
93. Sun Microsystems, Inc. Enterprise javabeans specification, version 2.0. Proposed Final Draft, 2000. http://java.sun.com/products/ejb/index.html.
94. Sun Microsystems, Inc. JavaSpaces Service Specification version 1.1, 2000.
95. Sun Microsystems, Inc. Java Message Service Specification 1.1, 2002.
96. C. Szyperski. *Components Software, Beyond Object-Oriented Programming.* Addison-Wesley, 1997.
97. G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.
98. TIBCO, Inc. TIB/Rendezvous. White Paper, 1996. http://www.rv.tibco.com/.
99. M. Viroli and A. Omicini. Tuple-based models in the observation framework. In Arbab and Talcott [7], pages 364–379.
100. M. Viroli, A. Omicini, and A. Ricci. On the expressiveness of event-based coordination media. In H. R. Arabnia, editor, *The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, Las Vegas, Nevada, USA, 2002.
101. M. Viroli and A. Ricci. Tuple-based coordination models in event-based scenarios. In J. Bacon, L. Fiege, R. Guerraoui, H.-A. Jacobsen, and G. Mühl, editors, *First Intl. Workshop on Distributed Event-based Systems (DEBS'02)*, Vienna, Austria, 2002. IEEE Press. Published as part of the ICDCS '02 Workshop Proceedings.
102. T. W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the Boolean model. *ACM Transactions on Database Systems*, 19(2):332–364, 1994.