A Modular Approach to Build Structured Event-based Systems

Ludger Fiege* Gero Mühl* Felix C. Gärtner
{fiege,gmuehl}@gkec.tu-darmstadt.de
felix@informatik.tu-darmstadt.de
Department of Computer Science
Darmstadt University of Technology
D-64283 Darmstadt, Germany

Keywords

Event-based Cooperation, Notification Services, Publish-Subscribe, Formal Specification

ABSTRACT

Event-based systems are developed and used as a coordination model 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 paper, we present the modular design and implementation of an event system which supports scopes and event mappings, two new and powerful structuring methods that facilitate engineering and coordination of components in event-based systems. The approach is based on a trace-based specification method adapted from temporal logic.

1. INTRODUCTION

Proliferation of computer networks led to increasingly complex information systems which are built out of heterogeneous, autonomous components. In such systems, computations are physically and logically distributed and have to be coordinated in order to reach a common goal. Different coordination models have been proposed in the literature, all of which try to integrate a number of components, but not all of them are scalable. For example, it was criticized that race conditions are possible in Linda [9] and its variants, resulting from the inherent concurrency of the model [2].

The model of event-based systems is increasingly often used in order to achieve scalability. In this model, the integrated components are only loosely coupled. Processes can act both as producers and consumers of events. Producers publish notifications about internal events but do not address any specific (set of) receivers. On the other hand, consumers specify the kind of data they want to receive by subscriptions, e.g., they subscribe by type or content of the transmitted notification. Publish/subscribe techniques directly implement this approach [15].

Specification of event-based systems.

There exist a considerable amount of work on event-based systems, and many concrete systems have been designed and implemented (e.g., SIENA [5], etc.). Unfortunately, it is very difficult to compare these systems because of different or informal semantics. For example, in the SIENA system [5], Carzaniga, Rosenblum and Wolf make a good effort at defining the semantics of subscription mechanisms. However, timing issues are explicitly excluded from the specification; delivery is "best effort." Processes are required to accommodate to race conditions; notifications may be delivered after cancellation of the respective subscriptions. No reasoning about any timing issues is possible according to the given specification. In most other systems, practitioner's approaches dominate and at most the formal semantics of the subscription languages are given [3], neglecting the semantics of the event service itself.

In other related work which follows the Actors paradigm [1], a pattern-oriented broadcasting mechanism is used, which is called *implicit invocation* in software engineering [8]. A formal specification of implicit invocation systems is presented in [6]. It may also be used to describe event-based systems, but only a static, predetermined binding of messages/notifications to methods is used, and the important aspect of dynamic subscriptions is excluded.

One of the contributions of this paper is that we provide a completely formal specification of the semantics of different types of event systems. The specifications are given using standard approaches from distributed algorithms, i.e., the specification language is adapted from temporal logic

*The authors' work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the PhD program "Enabling Technologies for Electronic Commerce" at Darmstadt University of Technology.

[13] and the specification itself is divided into safety and liveness conditions [12].

Structuring of event-based systems.

The event paradigm is a special kind of coordination model which is build around a shared data space, like Linda [9]. In comparison with Linda, the components are more loosely coupled, facilitating distributed deployment of independent components, but on the other hand, also complicating engineering of event-based systems. In order to cope with the inherent complexity, efficient abstractions are necessary like they are known in other areas of computer science. Former work on event-based systems, however, concentrated on the efficiency of implementation issues, disregarding the needs to facilitate coordination and engineering issues. The notion of visibility has proven to provide helpful abstractions in structuring complex systems. Information hiding [16] and transaction processing [10] are good and accepted examples of how complexity can be reduced by restricting the visibility of components and their actions.

We introduce the notion of *scopes* in event-based systems. 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.

Only a limited amount of initial work exists in the area of structuring event-based systems. The READY event notification service offers event zones as administrative domains [11]. They are used to bundle sets of 'specifications' (subscriptions and actions) or consumers in order to provide an uniform management interface. Research on Linda-like systems investigated structures of components. Agha and Callsen propose ActorSpaces to limit the distribution of messages [2]. The basic drawback of their approach is that even though previously unknown objects are intended to cooperate senders have to specify destination addresses. The sketched implementation is rather limited. In [14], Merrick and Wood introduce scopes to limit the visibility of tuples in Linda, but again, senders have to specify destination scopes. Furthermore, nesting of scopes is restricted to two levels.

In large systems, delimiting of the visibility of notifications may not be sufficient because of heterogeneity issues and different administrative domains where syntax and semantics of events differ. It is most likely not possible to use *one* event model in the entire system. Different parts will use different representations and semantics of the transmitted events. The scoped event system model is extended to include *event mappings*, i.e., a possibility to transform events when crossing scope boundaries. Generalized scope interfaces are offered that leverage construction and maintenance of large systems.

The READY system [11] uses a similar mapping facility located in boundary routers connecting otherwise independent domains. However, in this way they operate on a rather coarse and static granularity. There exist some work on semantical mappings in the data management literature, which partly focuses on events [4, **?**].

In this paper, we present the design of an event system which supports scopes and event mappings. We proceed in three steps: The first step (described in Section 2) presents a precise specification of a simple event system and gives a possible distributed implementation. The offered semantics are similar to the basic functionality of existing event systems like SIENA. In the second step (presented in Section 3), we refine the specification of the simple event system to include scopes, and present an implementation which is built around a simple event system. In the third and final step (detailed in Section 4), the semantics of a scoped event system are extended to deal with event mappings. We present an implementation, which exploits the fact that we have already implemented an event system with scopes in the second 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. Furthermore, in conjunction with exact specifications it allows to deal with issues of correctness more easily. Due to lack of space correctness proofs can be found elsewhere [**?**].

2. SIMPLE EVENT-BASED SYSTEMS

In this section, we specify a simple event-based systems and show how to implement it.

2.1 Specification

A *simple event system* can be viewed as a transport mechanism for event notifications. Informally, components interacting with the event system signal event occurrences by invoking the *pub* operation of the system with the notification data describing this event as parameter. We further assume that notifications are unique in that they are distinguishable by some identifier, i.e., two consecutive *pub* operations with identical notification data result in sending two different notifications. The notification is conveyed by the event system and delivered to all connected components via an output operation called *notify*. Components register their interest in specific kinds of events by issuing subscriptions via the *sub* operation. This operation takes a *filter* (i.e., an event selector) as parameter, and every delivered event must match such a subscription filter. Each subscription must be revoked individually and separately by using the *unsub* operation. Otherwise, computability issues arise concerning matching and subtracting of filters in the specific subscription language.

Formally, the event-based system is viewed as a black box with an interface (see Figure 1). A set of clients interact with the system by invoking input operations *pub*, *sub* and *unsub*. The system can asynchronously notify a client by invoking an output operation *notify*. All these operations take parameters from 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*. We say that a notification n *matches* a filter F iff (if and only if) $F(n)$ is *true*, where $N(F)$ denotes the set of all notifications that match F: $N(F) = \{n \mid F(n) = \mathit{true}\} \subseteq \mathcal{N}$.

We specify the behavior of the event system by solely looking at its interface. We think of the interface as a

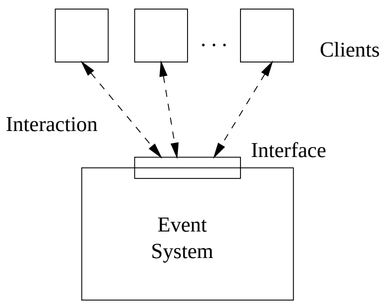

Figure 1: Black box view of an event system.

set of variables. A *state* of the interface is an assignment of values to these variables. Invoking operations at the interface results in atomic state changes so that individual behaviors of the system can be described as a sequence of states interleaved with operation names. We call such a sequence a *trace* of the system. For example, the trace

$$\sigma_1 = s_1, \mathit{sub}(X,F), s_2, \mathit{pub}(Y,n), s_3, \mathit{notify}(X,n), s_4, \ldots$$

describes that in the initial state s_1 component X subscribes to a filter F. After that, in the resulting state s_2, component Y publishes a notification n, which in turn results in state s_3. The next state s_4 results from component X receiving the notification of n, and so on.

Note that the trace does not say anything about the exact "real-time" instances of when the operations are invoked, so our model reduces time to the relative ordering of operations within a trace. Note also that the trace does not require that n matches F. In fact, we can define a lot of useless traces. For example, the trace

$$\sigma_2 = s_1, \mathit{unsub}(X,F), s_2, \mathit{notify}(Y,n), s_3, \ldots$$

describes that X unsubscribes to a filter it has never subscribed to and that Y receives a notification although it never subscribed to anything. The task now is to find suitable restrictions on the set of all traces that resemble exactly what we expect an event system to do (e.g., that a delivered notification must match a previous subscription).

Let $\sigma = s_1, op_1, s_2, op_2, s_3, \ldots$ be a trace. For every operation op of the event system we define a predicate Op on traces in the following way: $Op(\sigma) = \mathit{true}$ iff $op_1 = op$, i.e., the predicate holds if the operation is the first one in the trace. For example, the predicate $\mathit{Sub}(X,F)$ holds for example trace σ_1 above. The formal language we use to specify sets of traces is built from the above predicates, the logical operators $\vee$, $\wedge$, $\Rightarrow$, $\neg$ and the "temporal" operators $\Box$ ("always") and $\Diamond$ ("eventually") which we borrow from temporal logic [13]. For example, the formula $\neg\mathit{Sub}(X,F)$ is true for a trace σ iff the first operation in σ is *not* $\mathit{sub}(X,F)$. The semantics of the temporal operators is defined as follows: Let Ψ be an arbitrary formula. Then

- $\Diamond\Psi$ is true for trace σ iff there exists an i such that Ψ is true for the trace $s_i, op_i, s_{i+1}, op_{i+1}, s_{i+2}, \ldots$
- $\Box\Psi$ is true for trace σ iff for all i Ψ is true for the trace $s_i, op_i, s_{i+1}, op_{i+1}, s_{i+2}, \ldots$

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 component 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.

In our formalization we assume that a set of *specification variables* is part of the interface. Specification variables are fictitious devices which are sometimes necessary to keep track of the internal history of the system within a specification. For example, if a component should never unsubscribe a filter to which it has not subscribed before, we need a way of telling what filters it is subscribing to in a given state. We assume three sets of specification variables at the interface: For every component $X \in \mathcal{C}$ we postulate

1. a set S_X of *active subscriptions* (i.e., filters to which X has subscribed and not unsubscribed yet),
2. a set P_X of *published notifications* (i.e., the subset of $\mathcal{N}$ containing all notifications previously published by X), and
3. a multiset D_X of *delivered notifications* (i.e., all notifications which have been delivered to X). A multiset is a set where identical elements can occur more than once. A special operation $\#(M,e)$ is available giving the number of occurrences of element e in multiset M.

We assume that these specification variables are initially empty and that they are updated by the system faithfully, e.g., whenever X subscribes to F it adds F to S_X. This makes it possible to formalize trivial well-formedness properties like that a component may only unsubscribe a filter to which it has currently subscribed, or that it may subscribe only to a filter which it has not currently subscribed.

Now we are ready to specify the behavior of a simple event system. Arguably, it captures only minimal requirements, however intuitive semantics is covered and it represents a basis for further refinements.

Definition 1. *A* simple event system *is a system that exhibits only traces so that every state satisfies the following requirements:*

$$\begin{aligned} \textit{Safety:}\quad & \#(D_Y,n) \le 1 \\ & \wedge \big(\mathit{Notify}(Y,n) \Rightarrow \\ & \qquad \exists X\ \exists F \in S_Y :\ n \in P_X \ \wedge\ n \in N(F)\big) \end{aligned}$$

and

$$\begin{aligned} \textit{Liveness:}\quad & \mathit{Sub}(Y,F) \Rightarrow \\ & \quad \Diamond\Box\big(\mathit{Pub}(X,n)\ \wedge\ n \in N(F) \Rightarrow \Diamond\mathit{Notify}(Y,n)\big) \\ & \vee \big(\Diamond\,\mathit{Unsub}(Y,F)\big) \end{aligned}$$

The safety condition states that no "wrong" events are notified to a component, i.e., events are delivered at most once, have been published sometime in the past, and the component must have an active subscription for them.

This condition has appeared in the same spirit in the literature [5] and is easily justified.

The liveness condition describes precisely under which conditions a notification must be delivered. The condition can be rephrased as follows: If a component Y subscribes to F, then there exists a future point in time where the publication of a notification n that matches F will lead to a delivery of n to Y. This can only be circumvented by Y unsubscribing to F.

For example, trace σ_1 above satisfies both safety and liveness conditions while σ_2 violates the wellformedness conditions stated above. As additional 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 (the intermediate states are omitted for brevity):

$$\begin{aligned}
\sigma_3 &= \mathit{sub}(Y,F), \mathit{pub}(X,n_1), \mathit{notify}(Y,n') \\
\sigma_4 &= \mathit{pub}(X,n), \mathit{sub}(Y,F), \mathit{unsub}(Y,F), \mathit{notify}(Y,n) \\
\sigma_5 &= \mathit{sub}(Y,F), \mathit{pub}(X,n_1), \mathit{pub}(X,n_2), \mathit{pub}(X,n_3), \ldots
\end{aligned}$$

Traces σ_3 and σ_4 violate the safety requirement because a notification is delivered to Y that does not match an active subscription. In trace σ_5 component Y subscribes to F and component X starts to publish a continuous sequence of notifications matching F. Since there is no notification in σ_5 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 we obtain a concise and unambiguous characterization of what types of actions must be produced by the system under which conditions. For example, if a component has subscribed to a filter F and later unsubscribes to it, the system does not have to notify the component about *any* events 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 an event is only necessary if the component 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 component may later unsubscribe to F.

2.2 Implementation

We now show how to implement the specification of a simple event system from Section 2.1. We base all our implementations in this paper 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. For simplicity, the communication topology is assumed to be acyclic and connected (see Figure 2).

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.

We note that there can be many different implementations of Definition 1, especially ones that are more efficient than ours. The purpose of this section is merely to show the possibility of implementing the specification and showing that our specification facilitates correctness arguments.

2.2.1 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.

2.2.2 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 $\mathit{notify}(Y,n)$.

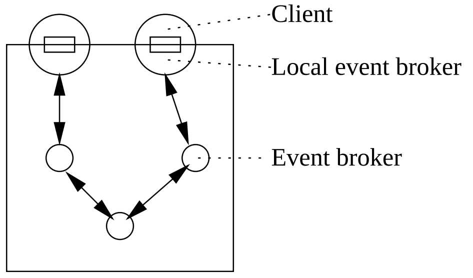

Figure 2: A possible implementation view of a simple event system.

3. EVENT-BASED SYSTEMS WITH SCOPES

We extend the specification of the simple event system presented in Section 2.1 and introduce the notion of *scopes*. For presentation purposes, we restrict our attention to *static scopes*, i.e., the scope hierarchy and membership cannot change once the first event has been published. This restriction is softened in Section 3.2.

3.1 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.

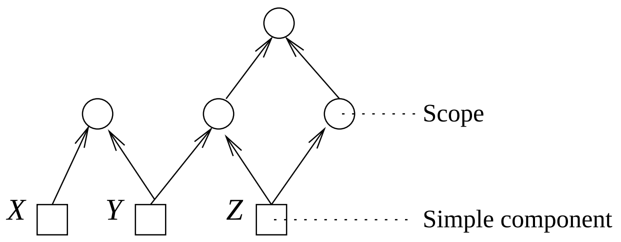

Figure 3: A graph of components

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 directed 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

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

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

Definition 2. *A* scoped event system *is a system that exhibits only traces so that every state satisfies the following requirements:*

$$\begin{aligned} \textit{Safety:}\quad & \#(D_Y, n) \leq 1 \\ & \wedge \Big[\mathit{Notify}(Y,n) \Rightarrow \\ & \qquad \exists X.\ \exists F \in S_Y.\ \ \big(n \in P_X\big) \wedge \big(n \in N(F)\big) \\ & \qquad\qquad \wedge\, v(X,Y)\Big] \end{aligned}$$

$$\begin{aligned} \textit{Liveness:}\ & \mathit{Sub}(Y,F) \Rightarrow \\ & \quad \Diamond\Big(\Box v(X,Y) \Rightarrow \Box\big[\mathit{Pub}(X,n) \wedge n \in N(F) \Rightarrow \\ & \qquad\qquad \Diamond \mathit{Notify}(Y,n')\big]\Big) \\ & \vee \Diamond\, \mathit{Unsub}(Y,F) \end{aligned}$$

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.

3.2 Dynamic Scopes

In Definition 2 we have assumed a static scope hierarchy. The case of dynamic scopes is however 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: $\mathit{cscope}(S)$ and $\mathit{dscope}(S)$ to create and destroy a scope S, $\mathit{jscope}(X,S)$ and $\mathit{lscope}(X,S)$ 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 the worst case, X has to be blocked until n is delivered, which is unfavorable.

There are two possibilities to solve this problem. The first is to postulate that a client may only leave a scope if all of its published notifications have been delivered. Under this assumption, Definition 2 makes sense with dynamic scopes, too. The second possibility is to weaken the definition and allow the delivery of a notification if publisher and receiver were visible *at the time the notification was published.* Since this substantially changes the safety semantics we have chosen not to pursue this direction here. A discussion of the different possibilities is left for future work.

Note that the liveness part of Definition 2 is perfectly compliant to dynamic scopes.

3.3 Implementation

We present a possible implementation of the previous specification which uses a simple event system 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.

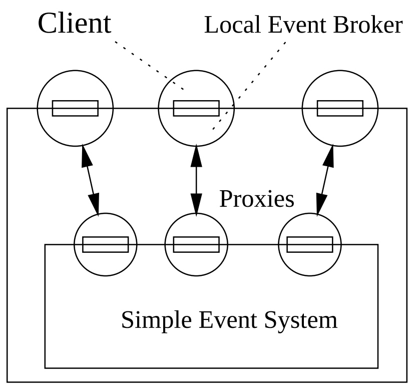

Figure 4: A possible implementation of a scoped event system.

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 distributed system. Again we call the part of the client process which handles these calls the *local event broker* of that client. Furthermore, for every client there is an additional process at the interface of the simple event system which we call the client's *proxy*.

Although we do not deal with dynamic scopes here, the presented algorithm can easily be extended to include dynamic scopes as of Section 3.2. 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.

3.3.1 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.

3.3.2 Algorithm

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

Calls to $sub(X, F)$ and $unsub(X, F)$ 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 $sub(Prox_X, \tilde{F})$.

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 $notify(Y, n)$. Otherwise the notification is discarded.

4. SCOPED EVENT-BASED SYSTEMS WITH EVENT MAPPINGS

We now provide a specification and an implementation for a scoped event system with event mappings. 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.

4.1 Specification

Scopes are components and they publish and consume notifications about events just as simple components do. But their behavior should not be merely a sum of their constituent components. The expressiveness of the graph of scopes is greatly extended if scopes are able to influence the set of events communicated through them. For this purpose, we define event mappings which are attached to individual scopes and which fulfill two tasks. First, they act as filters that explicitly allow only a specific set of events to be published and consumed, describing the interface of the scope. Second, all events crossing a scope boundary, which encapsulates its subscopes, may be transformed to map between internal and external representations. For example, mappings may be used to accommodate application-specific syntactical or semantical differences in data representations, like currencies in data types or constraint views on published data required by security issues.

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 $v(X, Y)$:

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

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 3, 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.

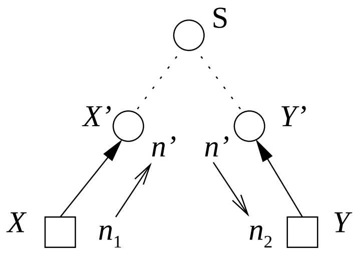

Figure 5: Recursive definition of the relation $(n_1, X) \sim (n_2, Y)$.

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.

DEFINITION 3. *A* scoped event system with event mappings *is a system that exhibits only traces so that every state satisfies the following requirements:*

Safety: $\#(D_Y, n) \leq 1$

$$\begin{aligned}&\wedge \Big(Notify(Y, n') \Rightarrow \\ &\qquad \exists X.\, \exists F \in S_Y. \quad (n \in P_X) \wedge (n' \in N(F)) \\ &\qquad\qquad \wedge \left[(n', X) \sim (n, Y)\right]\Big)\end{aligned}$$

Liveness: $Sub(Y, F) \Rightarrow$

$$\begin{aligned}&\Diamond\Big(\Box\left[(n, X) \sim (n', Y)\right] \Rightarrow \\ &\qquad \Box\left[Pub(X, n) \wedge n \in N(F) \Rightarrow \Diamond Notify(Y, n')\right]\Big) \\ &\vee \Diamond Unsub(Y, F)\end{aligned}$$

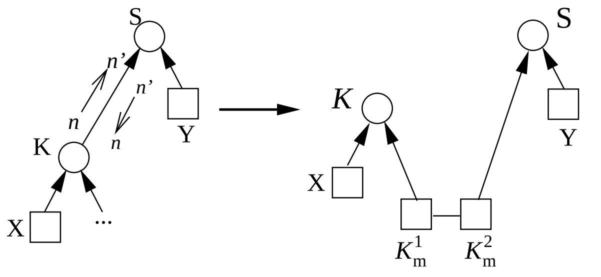

Figure 6: Transformation of mappings into components.

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'. Similar to the visibility, this formulation captures the notion that in addition to being visible with respect to scoping, the event mappings must additionally allow the flow of notifications. 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 $(n, X) \sim (n, Y)$.

4.2 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^1_m and K^2_m. 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.

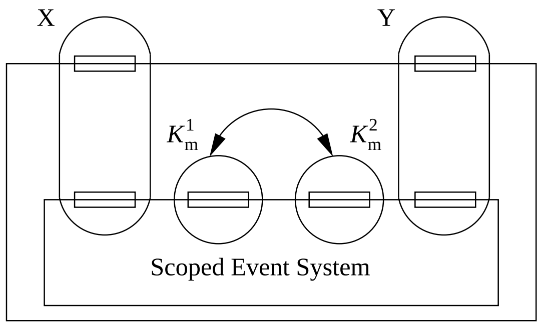

Figure 7: Architecture of scoped event system with mappings.

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 $ES^{\mathcal{M}}$.

In general, if a scope K is to be joined to a superscope S by calling $jscope(K, S)$, 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. CONCLUSIONS

We have introduced the notion of scopes as a powerful structuring mechanism for event-based systems. Scopes can help to hide internal configurations or delimit administrative domains. In conjunction with event mappings, scopes can even provide support for heterogeneous processing environments. We have also shown how to design and implement scoped event systems by providing modular and unambiguous specifications and provably correct implementations. In future work we wish to study the open specification questions concerning systems with dynamic scopes. Additionally we will evaluate our design within Rebeca, our prototype event system implementation [7].

6. REFERENCES

[1] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., USA, 1986.

[2] Gul Agha and Christian J. Callsen. ActorSpace: an open distributed programming paradigm. *ACM SIGPLAN Notices*, 28(7):23–32, July 1993.

[3] Marcos Aguilera, Robert Strom, Daniel Sturman, Mark Astley, and Tushar Chandra. Matching events in a content-based subscription system. In *PODC: 18th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 53–61, 1999.

[4] C. Bornhövd and A.P. Buchmann. A prototype for metadata-based integration of internet sources. In *11th International Conference on Advanced Information Systems Engineering (CAiSE'99)*, volume 1626 of *LNCS*, Heidelberg, Germany, June 1999. Springer-Verlag.

[5] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[6] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. In *Proceedings of of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, Lake Buena Vista, Florida, November 1998. ACM.

[7] L. Fiege and G. Mühl. Rebeca Event-Based Electronic Commerce Architecture, 2000. http://www.gkec.informatik.tu-darmstadt.de/rebeca.

[8] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In Søren Prehn and W. J. (Hans) Toetenel, editors, *VDM '91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 31–44. Springer-Verlag, 1991.

[9] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[10] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[11] R.E. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the READY event notification service. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop*, Austin, Texas, USA, May 1999.

[12] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[13] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[14] Iain Merrick and Alan Wood. Coordination with scopes. In *Proceedings of the 2000 ACM Symposium on Applied Computing 2000*, pages 210–217, Como, Italy, March 2000.

[15] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus—an architecture for extensible distributed systems. In Barbara Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 58–68, New York, NY, USA, December 1993. ACM Press.

[16] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.