An Extended Publish/Subscribe Protocol for Transparent Subscriptions to Distributed Abstract State in Sensor-Driven Systems using Abstract Events

Eleftheria Katsiri[1], Jean Bacon[2] and Alan Mycroft[2]
[1] Laboratory for Communication Engineering, [2] Computer Laboratory,
University of Cambridge,
William Gates Building,
15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
ek236@eng.cam.ac.uk, {Jean.Bacon,Alan.Mycroft}@cl.cam.ac.uk

Abstract

Event-based middleware is emerging as the major paradigm for large-scale and widely distributed systems, especially sensor-rich environments. Here, many primitive events are low-level and effort has been directed towards defining more meaningful composite events, which are typically recognised by finite-state machines. We argue that finite-state-machines (FSMs) are insufficient for meeting the requirements of users in Sentient Computing environments; user intuition may be concerned with notions such as state and negation which FSMs cannot support with reasonable efficiency. We aim to support querying and subscribing transparently to distributed state, which necessitates an alternative model for Sentient Computing. We propose a state-based, temporal first order logic (TFOL) model whose implementation is based on a deductive knowledge-base.

Furthermore, we propose a generalised notion of an event, an abstract event, which we define as a notification of transparent changes in distributed state. An extension to the publish/subscribe protocol is discussed, in which a higher-order service (Abstract Event Detection Service) publishes its interface; this service takes a TFOL abstract event definition as an argument and in return publishes an interface to a further service (an abstract event detector) which notifies transitions between the values true and false of the formula, thus providing a more natural and efficient interface to applications.

1. Introduction

Our research focuses on Sentient Computing [10] which is a computing paradigm for *context-awareness* in sensor-driven systems. Sentient Computing systems monitor the stimuli provided by environmental sensors in order to *notify* Sentient applications, *in real-time*, of changes that are relevant to the rapidly changing context of the user. An example of a Sentient application is one where the user can define his requirements in terms of *abstract knowledge predicates*. Such predicates represent states of entities that share abstract spatial and temporal properties, which are of interest to the user. The application then receives notifications from the Sentient Computing system when the defined knowledge has been acquired, and as a result, it delivers relevant reminders. E.g., a user may ask to be reminded to return John's book, whenever he is in the same room as John.

Current trends in programming paradigms in the area of context-awareness, such as *ubiquitous* and *pervasive computing*, advocate a separation of concerns between the way the user perceives knowledge about the physical world and the way knowledge is produced and maintained within Sentient Computing systems. The user's view is *abstract and state-based*, i.e., events are perceived as changes of state. The user's view should be *transparent*, i.e., the required application functionality should be available irrespectively of the heterogeneity of the underlying distributed modelling (in terms of the physical entities it contains) and the heterogeneity should be concealed, even when the user is mobile. E.g., locating the *closest, empty, meeting room* should be feasible, both when the user is walking in the Computer Laboratory and within Engineering. Furthermore, to accomodate user requirements, Sentient Computing environments need to be *dynamically extensible in real-time*. New user requirements need to be satisfiable without taking the system offline, or recompiling existing applications, even when entities are added or removed from the underlying model.

This paper argues that in context-aware, sensor-driven systems such as Sentient Computing systems, events in their traditional sense are an unsustainable metaphor. We intro-

duce a generalised concept of an event, an *abstract event*, as a notification of changes of *abstract state* of model entities. Although we abide by the pub/sub protocol, we propose an extension to it, a higher-order service that allows *real-time dynamic extensibility*. This service allows for *abstract event detectors* to be created dynamically based on an abstract event definition in TFOL (Section 3.1). The abstract event detectors are implemented in accordance with our state model, by a deductive knowledge-base, which maintains the state of the entities of that component in an internal memory and *deduces* higher-level changes of abstract state, from changes in concrete state. This allows richer expressiveness in user requirements, as demonstrated in Section 7.

2. Deficiencies of Current Event Models

In event-based modelling, the state of an entity is represented by means of an *event history* entity, which is in turn implemented by finite state machines (FSMs). However, events in sensor-driven systems are primitive, they are not always *significant*, and they only convey *positive, concrete* knowledge. Although each received event may affect the truth property of high-level, abstract knowledge, unless all changes are explicitly *deduced*, the semantic mapping between concrete and abstract knowledge is incomplete. On the other hand, transparent reasoning with state requires both reasoning with negative (missing) abstract knowledge as well as concealing the details of the universe of discourse from the reasoning tool. It is well known that FSMs are limited in both aspects. This is discussed in detail next.

Negation in Transparent Reasoning with State. *There does not exist a finite-state machine which can accept the expression $\neg P$, when there is no instance of the predicate $\neg P$ available in the system as a symbol.* This means, that a symbol that corresponds to either P or $\neg P$ must be explicitly generated for the FSM to be able to process it. This problem is interlinked with reasoning transparently with *parametric* state. We demonstrate that known methodologies for parametric FSM-based reasoning [1, 8] are not directly applicable here. In those methods, a parametric expression is modelled with an FSM with free variables and for each free variable in the initial parametric FSM, an identical, non-parametric FSM is spawned, whenever a symbol that instantiates the free variable occurs at a given state. In the spawned FSM, all instances of the parameter that corresponds to the symbol which has been encountered, are substituted with the actual symbol. However, in constrast to an event that occurs in a deterministic way, a state can be activated and de-activated, in responce to any received event. Therefore, each state S of the parametric FSM that models P needs to have a transition to a state S' that models $\neg P$ and which cancels the execution. Unless $\neg P$ can be directly extracted from a primitive event, $\neg P$ does not exist as a symbol in the system. The *Closed World Assumption* can be used to determine a set of states $\{Q_i, i = 1, 2, \cdots\}$ where $\neg P$ can be assumed to hold. This is possible only if $\{Q_i, i = 1, 2, \cdots\}$ is a set of states that represent concrete or deduced knowledge. Even so, creating the transitions from S to $\{Q_i, i = 1, 2, \cdots\}$ requires knowledge of the underlying universe of discourse, which makes the reasoning *non transparent*. Fig. 1 illustrates this with an example. In Fig. 1(a), if an event occurs that corresponds to user a_1 being inside region r_1, then the automaton of Fig. 1(b) will be spawned from the one in (a). In this automaton, the transitions from s_1 to s_3 must represent all events that report user a_1 exiting from region r_1. This means that there must exist one transition for each region in the universe which is different from r_1. If instead user a_1 had moved into r_2, the automaton, which would need to be constructed as a result, would be that of Fig. 1(c).

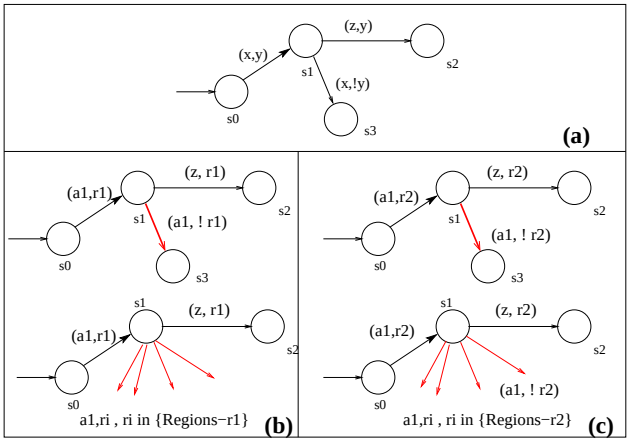

Figure 1. "Any two users are co-located".

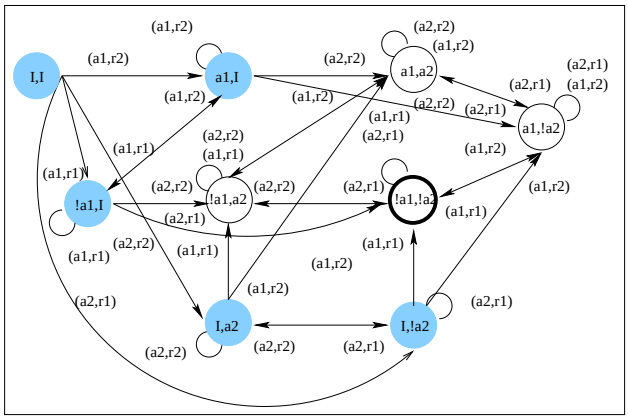

Figure 2. "Everybody is in r_2" with 2 users.

Uncertainty in Existential and Universal Quantification. The FSM of Fig. 2, models the expression "Everybody is in

r_2" in a world with two users a_1, a_2 and two regions r_1, r_2. Until an event for each user a_1 and a_2 has occurred, the system has only *partial knowledge*. States in grey are *uncertain*. Uncertainty is inherent in reasoning with expressions that signify existential and universal quantification (see Section 7).

Dynamic Extensibility. The Closed World Assumption often leads to state explosion. E.g., assuming that user a_3 is added to the previous closed world, the FSM of Fig. 2 would need to be re-compiled to that of Fig. 3. That renders dynamic extensibility unsustainable.

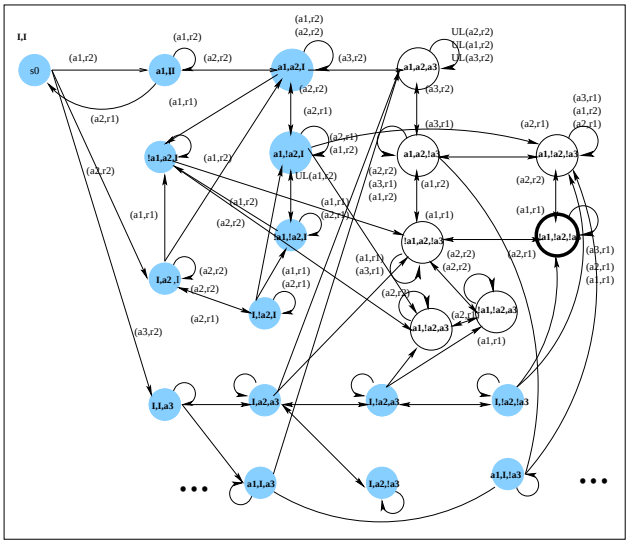

Figure 3. "Everybody is in r_2" with 3 users.

3. Abstract Events

Using the definitions and nomenclature proposed in [12] and extending the temporal notions of [11], we define a *state-based* model for distributed sensor-driven systems. We assume that a system consists of several physical *domains* such as an office domain. Each domain contains a set of physical objects such as mobile users and equipment. Modelled physical locations include *regions*, ranged over *rid*, and *positions (x,y,z)*. Each user *(u)* is associated with a *role* and each region is associated with a *regional attribute (rattr)* that describes relevant contextual information such as its functionality or ownership, e.g., kitchen, Alan's office etc. This allows the expression of *semantic abstractions* such as *the closest system administrator who is not busy.* Objects, locations, roles and regional attributes have *states* which are either concrete or abstract. Concrete state predicates represent state that is directly derived from the sensors and in this paper are always prefixed with L_(Low-level). Example of concrete state predicates are those that represent a user's position in terms of his co-ordinates, rooms in a building and nested locations such as floors. These are modelled [12] with the first-order logic predicates:[1]

$$L\_UserAtLocation(u, role, (x, y, z), timestamp)$$
$$L\_AtomicLocation(rid, rattr, polygon)$$
$$L\_NestedLocation(rid, rattr, rid\text{-}list)$$

E.g.,

$$L\_UserAtLocation(John, Phd, (13.45, 5.76, 1.75), 13{:}56)$$
$$L\_AtomicLocation(Room7, Meeting\text{-}room, Polygon37)$$
$$L\_NestedLocation(Floor4, [Room2, Room9])$$

A set of *function* predicates, as in Prolog, represent functions where the last variable represents the value of the function. The distance between a user's position and the centre of the polygon that defines a region is represented through the predicate $Distance(u, role, rid, rattr, v)$.

Abstract state predicates represent high-level state that is derived from concrete state by means of TFOL on properties of interest. A user's high-level location in terms of the region that contains him, a user's presence or absence and the fact that one or more people are co-located are examples of such predicates. Initially, when the system is started up, only concrete state predicates exist. Abstract state predicates in this paper are always prefixed with H_(High-level), $H\_UserAtLocation(u, role, rid, rattr, t)$.

A *timestamp* denotes the moment when a *current* abstract predicate instance becomes active. Previous instances are stored as *historical* predicates and are associated with an additional timestamp that denotes the moment when the predicate instance became inactive. Timestamps allow for *temporal abstractions* such as *now, today* and *temporal operations* such as sequence, iteration, equality and inequality with temporal intervals, over current and historical data.

Definition 1 *An abstract event is detected when an instance of an abstract state predicate which previously evaluates to true next evaluates to false and vice versa.*

3.1. Abstract Event Specification and Filtering

The *Abstract Event Specification Language (AESL)*[2] for creating abstract event definitions is a subset of TFOL that corresponds to Horn Clause Logic. *An abstract event definition* (AESL def) consists of one or more *implications* (Horn clauses). In case of only one implication, the RHS is the abstract predicate of interest. In case of more than one implication, the RHS of the last rule is the abstract predicate of interest.[3]

[1] As a convention, predicate attributes are in lower case and their values have the initial character capitalised, e.g., *polygon* is a variable whereas *Polygon27* denotes the value of the attribute *polygon* in a given co-ordinate system.

[2] AESL is a typed language, however types are ignored in this paper.

[3] As usual with these clauses all free variables are presumed to be implicitly universally quantified.

Example 1 *Locate the closest location to each user that has been empty for at least 5 min.*

Writing for brevity UL for $H\_UserAtLocation$, AL for $L\_AtomicLocation$, EL for $H\_EmptyLocation$, CL for $H\_ClosestLocation$, CEL for $H\_ClosestEmptyLocation$ and D for $Distance$:

$$
\begin{aligned}
&(\nexists u\ UL(u, rid, role, rattr) \wedge AL(rid, rattr) \\
&\quad \Rightarrow EL(rid, rattr)) \\
&D(u, role, rid_2, rattr_2, v_1) > D(u, role, rid_1, rattr_1, v_2) \\
&\quad \Rightarrow CL(u, role, rid_1, rattr_1) \\
&CL(u, role, rid, rattr) \wedge EL(rid, rattr) \\
&\wedge |EL(rid, rattr), CL(u, role, rid, rattr)|_{t>300} \\
&\quad \Rightarrow CEL(u, role, rid, rattr) \qquad (1)
\end{aligned}
$$

Eq. $|EL(rid, rattr), CL(u, role, rid, rattr)|_{t>300}$ follows the design of [16] and extends the sequence operator in [11]. It is used to select a pair of EL and CL predicates whose temporal distance is greater than 300 sec.

Filtering. Horn Clause logic is used for defining filters during client subscription. Filtering is equivalent to selecting a subset of instances of a specific predicate by specifying a set of constraints on its attributes. A filter that selects only the instances in (1) which are meeting-rooms, in respect to a system administrator, is shown in (2). Note that we encourage AESL definitions to use variables as arguments for predicates rather than constants. This is done for implementation optimisation and it ensures that the deduction of the abstract predicate, which is computationally expensive is performed once, and multiple instances of the predicate are selected later on, by filters. Section 3.2 discusses the implementation of AESL defs and filters in more detail.

$$(rattr = \textit{Meeting-room}) \wedge (role = \textit{Sysadm}) \qquad (2)$$

3.2. Abstract Event Detectors

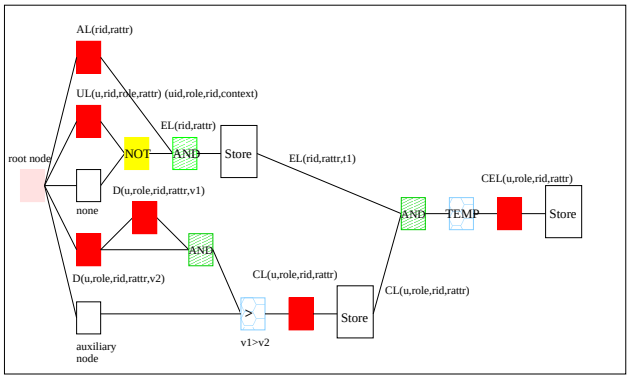

Figure 4. An abstract event detector for (1).

Each AESL definition is compiled into one or more abstract event (AE) detectors, which are structured as a deductive knowledge-base and which can perform semantic operations on instances of knowledge predicates that correspond to TFOL. They are implemented as Rete networks [7] and they consist of nodes and arcs. Every time a sensor creates a new instance of a concrete state predicate, corresponding tokens are created and propagated through the arcs to the nodes. Each node checks whether the received tokens correspond to a particular condition, e.g., if they are of class $H\_UserAtLocation$. It then forwards the tokens that satisfy the check on to the the child nodes. *Two-input* nodes concatenate tokens where shared variables are bound correctly. *Store* nodes, act as buffers for the current and historical instances of a predicate type and forward all stored instances on to the child nodes. This allows for temporal reasoning. When a token is forwarded to the final node, an instance of the abstract predicate that is being defined is created or deleted accordingly and an "activation" or "de-activation" abstract event is triggered, respectively. An abstract event detector for (1) is portrayed in Fig. 4.

Abstract event detection can be distributed so that each implication in an AESL def. is implemented by a separate detector, forming an hierarchical topology similar to SIENA [6] and HERMES [14]. The optimal placement of the detectors in the system can be determined as appropriate [16].

A filter is implemented as an AE detector with a linear complexity. Filters can be combined whenever there is a shared condition. E.g., the filter of (2) can be combined with that where $(rattr = \textit{M}\text{eeting-room}) \wedge (role = \textit{Ceo})$ as shown in Fig. 5.

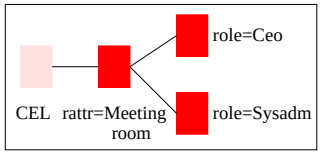

Figure 5. Filter combination.

4. An AED Service

We propose an extension to the pub/sub protocol, a *high-order service* which we call *Abstract Event Detection (AED) Service*, where subscribers do not just subscribe to event notifications as in the traditional form of this protocol, but to the establishment and configuration of an abstract event detector for a new abstract event of interest. The AED Service, acts as a mediator between the subscriber and the publisher and is responsible for detecting abstract events from primitive events. It interacts with publishers and subscribers using the pub/sub primitives *(Subscribe(), Notify())* according to the following extension to traditional pub/sub:

the AED Service publishes its interface using an event service such as the CORBA Notification Service, to all the subscribers and publishers of primitive events. Subscribers register their interest in subscribing to abstract event types of interest by subscribing at a dedicated *"AbstractEventSubscriptionListener"* (*AESListener()*) interface, at the AED Service. Each subscription carries the subscriber id, an AESL definition and a filter.

AEDListener.**subscribe**$(subscriberId, AESL\ def, filter)$

The AED Service uses the AEDListener (Fig. 6) to listen for subscriptions of the above type. For each received subscription, it checks in the *abstract event repository* whether an event type with the same name or AESL definition exists and if so, adds the subscriber to the list of subscribers for this event type. If it does not exist already, it registers the new event type with the underlying event service, so that the abstract event is available for filtering. The AESL definition is made available at the abstract event repository along with the new abstract event type. Next, the AESL definition and filter are extracted in order to construct an abstract event detector, that detects an abstract event of the requested type, as explained in Section 3.2. Using again the AESL definition, the primitive events of interest are selected and the underlying notification service is used for subscribing to the primitive events which are then translated appropriately and forwarded as inputs to the abstract event detector.

Each time an abstract event is detected, *Notify()* is invoked in order to publish the abstract event. *Notify()* publishes both the abstract event and the AESL def. This protects the service from malevolent event subscription in case of duplicate subscriptions to the same abstract event type with incorrect AESL defs. The *AEUListener()* listens to *Unsubscribe()* requests and removes the client that corresponds to the unsubscribe event from the notification list for that abstract event type.

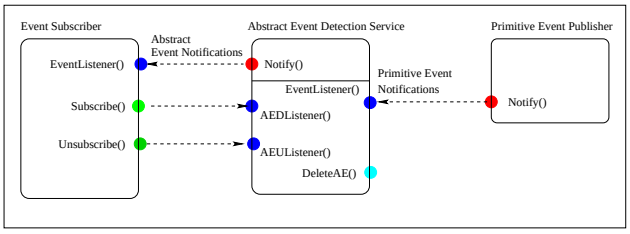

Figure 6. The AED Service.

The following call allows a subscriber to register interest in the events of type $H\_ClosestEmptyLocation$. The AESL definition in (1) and the filter in (2) are applied during subscription.

AEDListener.**subscribe**(a, Φ_1, Φ_2)

Φ_1 and Φ_2 correspond to (1) and (2) respectively, where a is the subscriber's Interoperable Object Reference (IOR).

4.1. AED Service Properties

The AED Service can be distributed to implement distributed event detection (Section 3.2). It supports a garbage collection process for unused abstract predicates, when no subscribers are interested in a particular event type. This is implemented through the *DeleteAE()* interface. It also supports a satisfiability checking mechanism that ensures the correctness of the abstract event specifications e.g., conflicting requests are detected, such as "Whenever the system administrator is at a different room from the health and security officer, notify me", where the system administrator is the same person as the health and security officer.

5. Implementation and Scalability Analysis

A prototype scalability analysis for the example of the $H\_ClosestEmptyLocation$ in (1) compares the traditional pub/sub, where *instances* of the concrete state predicates $L\_AtomicLocation$, $H\_UserAtLocation$ and $Distance$ are available directly as primitive events, with the extended pub/sub, which uses the AED Service. We assume that the AED Service subscribes to the primitive events from the respective publishers and uses the AESL defs to create the abstract event detector of Fig. 4. Assuming an architecture that consists of k domains, each containing on average n users and m rooms and assuming that the user moves at a rate of $1\ event/sec$, then as long as an AED component is placed close enough to the primitive event sources, the overall event bandwidth generated from the above scenario can be reduced to $O(n) notifications/sec\ (per\ domain)$ vs. $O(n^2) notifications/sec\ (per\ domain)$ for the case of the traditional pub/sub. The overall, worst-case computational cost is $O(n^2)\ tests/sec$, using the dual-layer knowledge-base architecture of [12].

6. Related Work

There seems to be little attention to applying distributed systems theory to the special behaviour of ubiquitous sensor-driven networks. SIENA [6] is a distributed event architecture consisting of *event brokers*. It aims to optimise message traffic by deciding on the optimal places where message processing should take place and it takes advantage of overlapping subscription filters in order to reduce computational costs. HERMES [14] is similarly distributed and aims to improve the efficiency of the delivery of event notifications by using peer-to-peer routing techniques for creating overlay broker networks [15]. OASIS [4, 5] extends HERMES with *role-based access control*. Significant

work has also been done in the area of *event composition* [2, 3, 13, 16, 17] i.e. the combination of primitive events into composite events by applying a set of *composition operators*. There, composite events are regarded as regular expressions with extensions for sequencing, concurrency and time.

The seminal notion of abstract events in sensor-driven systems appears to have originated in [3] and it is mentioned again later on in [2]. However, this early intuition is not directly applicable to distributed sensor-driven systems as it simulates the process of abstracting a region from a given position by means of a location service such as SPIRIT [9] and does not investigate further logical abstractions, such as TFOL reasoning. Our work allows the dynamic creation of abstract events in an asynchronous manner.

7. Conclusions and Future Work

Current event models have limitations in reasoning transparently with distributed state, which is due to the incomplete mapping between the application requirements in abstract knowledge and the concrete knowledge of the implementation domain, as well as the insufficiency of finite automata to deal with negation and quantification. In order to address these limitations we introduce the concept of abstract events as changes of abstract state and we describe a higher-order service that takes as an argument an abstract event definition and in return publishes an interface to a further service, an AE detector, which notifies transitions between the values *true* and *false* of the defined predicate. Detectors can be generated automatically. Because abstract events are high-order, the notification bandwidth is reduced. Furthermore, greater expressiveness in the subscription and querying language is achieved. Expressions that are implemented transparently by the proposed model include all expressions that negate knowledge from a source of context such as *not exists, nobody, nowhere, not seen, absent, empty, idle* as well as those that quantify abstract and concrete context and model entities: *somebody, somewhere, anybody, everybody, everywhere, exists, for all* $\langle context \rangle$, *there is* $\langle context \rangle$. We have created a prototype implementation of the proposed system. Our system can be applied to stored events as well. Future work involves addressing the lack of global time appropriately e.g., interval timestamps have been used in [16]. Future work will also consider sensor failure which is a common source of uncertainty in sensor-driven systems, as well as extending the reasoning beyond boolean logic.

References

[1] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using Events to Build Distributed Application. In *2nd International Workshop on Services in Distributed and Network Environments, Whistler, British Columbia, Canada*, June 1995.

[2] J. Bacon and K. Moody. Toward Open, Secure, Widely Distributed Services. *Communications of the ACM*, 45(6), June 2002.

[3] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. M. Neil, O. Seidel, and M. Spiteri. Generic Support for Distributed Applications. *IEEE Computer*, 33(3):68–76, Mar. 2000.

[4] J. Bacon, K. Moody, and W. Yao. A Model of OASIS Role-Based Access Control and its Support for Active Security. *ACM Transactions on Information and System Security (TISEC)*, 5(4):492–540, Nov. 2002.

[5] A. Belokosztolszki, D. Eyers, P. Pietzuch, J. Bacon, and K. Moody. Role-Based Access Control for Publish/Subscribe Middleware Architectures. In *International Workshop on Distributed Event-Based Systems (DEBS03), San Diego, CA*, June 2003.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, Aug. 2001.

[7] C. L. Forgy. RETE: A fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[8] N. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *18th VLDB Conference, Vancouver, British Columbia, Canada*, Aug. 1992.

[9] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The Anatomy of a Context-Aware Application. In *Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking, Seattle, WA*, pages 15–19, Aug. 1999.

[10] A. Hopper. The Royal Society Clifford Paterson Lecture: Sentient Computing, 1999.

[11] D. Ipina and E. Katsiri. A Rule-Matching Service for Simpler Develpment of Reactive Applications. In *IEEE Distributed Systems Online*, Nov. 2001.

[12] E. Katsiri and A. Mycroft. Knowledge-Representation and Abstract Reasoning for Sentient Computing. In *Workshop on Challenges and Novel Applications of Automated Reasoning, CADE-19, Miami Beach, FL*, July 2003.

[13] M. Mansouri-Samani and M. Sloman. GEM: A Generalised Event Monitoring Language for Distributed Systems. *Distributed Systems Engineering Journal*, 4(2), June 1997.

[14] P. Pietzuch and J. Bacon. HERMES: A Distributed Event-Based Middleware Architecture. In *the First International Workshop on Distributed Event-Based Systems (DEBS02)*, pages 611–618, July 2002.

[15] P. Pietzuch and J. Bacon. Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware. In *International Workshop on Distributed Event-Based Systems (DEBS03), San Diego, CA*, June 2003.

[16] P. Pietzuch, B. Shand, and J. Bacon. Composite Event Detection as a Generic Middleware Extension. *IEEE Network Magazine, Special Issue on Middleware Technologies for Future Communication Networks*, Jan. 2004.

[17] M. Spiteri. *An architecture for the Notification Storage and Retrieval of Events*. PhD thesis, University of Cambridge, 2000.