Extending the Event-Based Programming Model to support Sensor-Driven Ubiquitous Computing Applications

Sean Reilly and Mads Haahr
Distributed Systems Group
Department of Computer Science
Trinity College Dublin
Ireland

***Abstract*—We propose an extension to the event-based programming model for sensor-driven ubiquitous computing applications. The extension consists of three abstractions: Multi-Event Handlers, Event Streams and Execution Policies. We believe that these additional abstractions simplify the task of writing applications in this domain. In this position paper we present the extensions and demonstrate our prototype middleware implemented in C.**

INTRODUCTION

Event-based programming is a popular paradigm for design and implementation of many types of computer applications. It has been used for applications as diverse as graphical user interfaces and large banking systems and it has enjoyed particular success in the domain of distributed systems. A large number of extensions have been proposed to event-based programming to make it more suitable for different domains [11], [12], [5], [14], [6], [2], [9], [4]. Such extensions tend to take the core event-programming model and add to it one or more abstractions which extend the model to be more suitable for a particular domain or to address a particular problem. One notable example is the addition of filters, pre and post constraints, zones, composite events and other abstractions which have been proposed to make the event model more suitable for large scale distributed systems [12], [2], [9], [4].

The extended event model, called the AESOP model, presented in this paper is primarily designed for the domain of sensor-driven ubiquitous computing applications. Such applications typically have one or more sensors each of which provides a stream of data which must be processed continuously. In particular, applications which perform sensor fusion or context generation typically exhibit these characteristics.

EVENT MODELS AND EVENT-BASED PROGRAMMING

The characteristics of the event model are perhaps best thought about in terms of communication. In the traditional client/server approach one client often interacts with one server. The communication is generally one-to-one. This is in contrast with the event model where there is generally more than one recipient of the event. Also in the client/server model the client usually (though not always) expects a response from the server. In the event model the communication is usually asynchronous and often anonymous, i.e., the communicating elements do not necessarily need to know about the receivers or producers of their events, just that an event has happened. This allows decoupling of the producer of the events from the consumer. In the event model all parties are permitted to initiate communication, i.e., send events and to receive communications, i.e., receive events. This is in contrast to the client/server model where the client initiates communication with the server. The communication model, i.e., one-to-one versus one-to-many and the asynchronous nature of the communication and the ability to decouple and modularise the elements of the application make the event model a tool of choice for developers and designers of distributed systems.

STEAM Event Model: A large amount of event-based programming models have been implemented [11], [3], [8], [14], [6], [7], [4]. A discussion of these event models is beyond the scope of this paper, however, we shall briefly introduce the STEAM event model as we have used it as a base on which to implement our extensions. STEAM [6] is an event-based middleware service that has been designed for the mobile computing domain, specifically for ad-hoc wireless networks. It provides an implementation of the implicit event model [7], i.e. consumers of events register their interest in an event type. This is in contrast with the peer-to-peer and mediator-based event models [7], [6] both of which rely on system wide services, which are not guaranteed in mobile ad-hoc networks. STEAM was designed for the domain of ubiquitous computing and in particular the domain of traffic management systems.

Extensions To Event Models

Filters And Constraints in the ECO model: Scalability is a major issue in distributed systems and several extensions to the event-model have been propose to better deal with it. Propagating events to all nodes in a large distributed system severely hampers the scalability of the system. To this end both the ECO[4] model and Cambridge Event Model [11] use some form of mechanism to limit the distribution of irrelevant or unneeded events. In the ECO model these are called *notify constraints* and in the Cambridge Event Model these are referred to as *filters.* The ECO model also provides another mechanism for reducing the unnecessary propagation

978-1-4244-3304-9/09/$25.00 ©2009 IEEE

of events. The concept of *zones* [9] is used to scope events by criteria defined by the application programmer. Only consumers within the zone will receive an event for that zone. Zones can overlap and change dynamically.

Composite Events

Composite Events [12], [5], [11] are also an extension to the event model for distributed systems to deal with the issue of scalability. In large-scale distributed systems large amounts of events may be produced who's existence on their own is of little relevance. Propagating all these events across the system can have a negative effect on bandwidth in the system. Instead events are matched with rules provided by the developer and once a certain pattern of events have occurred a composite event is then sent. The composite event detector is usually situated close to the source of the events in the network. This prevents the propagation of numerous lower level events through the network which reduces bandwidth consumption and increases scalability.

THE AESOP MODEL

The AESOP (Architecture for fusing Events using Streams and executiOn Policies) model consists of events combined with *multi-event handlers*, *event streams* and *execution policies*. The multi-event handlers are functions that get executed in response to the arrival of one or more events. The execution policies are complex expressions that are used to decide what combination of incoming events trigger the execution of the multi-event handler. The event streams are length bounded streams of data used to supply the multi-event handler with parameters and to aid reasoning about historical sensor data.

Historically in computer science there has been a distinction made between data and control, with the two concepts traditionally being separated in the design of computer systems. In event-based systems data and control are merged in the one signal, with the event usually containing relevant data in addition to an implicit command. In traditional event based systems one event caused zero or more pieces of code to be executed, typically in the form of one or more callback functions.

In the AESOP model we extend this concept to allow a combination of one or more events to control the execution of the handler. The abstraction of execution policies provides the programmer with the means to specify exactly what combination of events should trigger the execution of the multi-event handler. The AESOP model is in essence a generalisation of the standard event-based programming model, which we think is particularly useful for sensor-driven ubiquitous computing applications.

Event Streams

Event Streams are the abstraction which facilitates all communication between multi-event handlers in the programming model. They serve as input to the multi-event handler and also as the output stream. Associated with each input stream to a multi-event handler is a sliding window. On creation of the event stream the programmer decides how long the sliding window should be. This maintains a list of previous elements in the event stream to the required length. The elements of the stream in the sliding window are exposed to the application developer. This allows multi-event handlers to reason about past elements of the stream in an application specific manner while dealing with a potentially infinite stream of sensor readings. Figure 1 shows an example of some sliding windows on event streams. In this example we have three event streams. The first event stream S1 has events with float data, S2 has events with integer data and S3 is shown as having events with an arbitrary type. The lengths of their sliding windows which are specified when initialising the event streams are 4, 6 and 3 respectively. As new events arrive they join the head of the queue pushing the last element out of the sliding windows range. In the example we can see an event with data 3.4 is arriving at the front of S1 pushing the event with value 1.2 out of the sliding window.

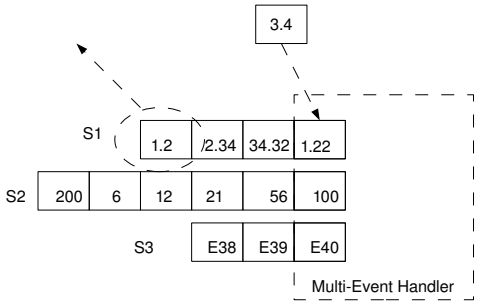

Figure 1. Event Streams and Sliding Windows

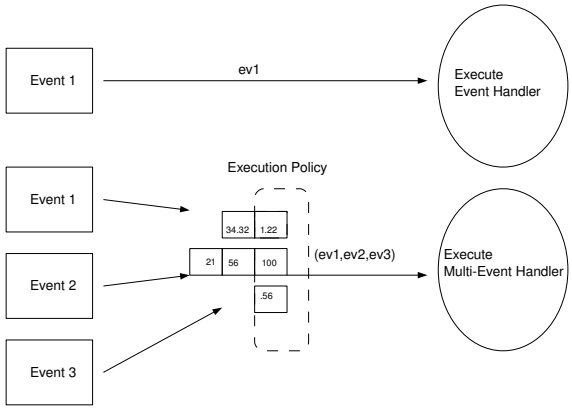

Figure 2. The AESOP Model, Multiple Events with Execution Policy

Multi-Event Handlers

Multi-Event Handlers are code fragments that get executed in response to new data in one or more event streams. They take as inputs one or more event streams and optionally output

an event stream. They are particularly suited to the creation of applications which rely on multiple sensor streams. These applications usually require that some analysis be performed periodically on the sensor streams and events may or may not be raised in response.

Multi-event handlers can be considered a generalisation of callbacks for traditional events. Figure 2 shows a traditional event handler which executes when one event arrives and a multiple-event handler in the same diagram. Multi-event handlers generalise the behaviour of the traditional event and execute in response to multiple events. In fact a traditional event is analogous to a multi-event handler with a single input event stream. We have found multi-event handlers to be particularly useful for implementing sensor fusion and context generation functionality. In sensor fusion applications multiple sensor readings are fused into a single higher order sensor. Multi-event handlers are particularly well suited to this as they can receive readings from multiple sensors and output a sensor stream. This output stream can be used as the input to other multi-event handlers allowing sensor data to be refined further.

Execution Policy

In traditional single event handlers, when an event arrives the handler executes immediately. With multi-event handlers this might not always be the best course of action and we may want to delay execution of the multi-event handler until sufficient data is available or some combination of input events are available to the handler. In the AESOP model *execution policies* govern the way in which multi-event handlers decide when to execute, given new event stream data. When new data arrives in one of the streams there is a multi-event handler specific decision to be made. We can decide to execute the multi-event handler using this new event or wait until all streams have new events, or some combination of the two options. Execution policies are the abstraction used to represent these decisions in the programming model.

Execution policies are Boolean expressions. They use the presence of new data on the input streams of the multi-event handler in order to evaluate to true or false to determine if the multi-event handler should be executed. The contents of the events are not taken into consideration in the execution policies. Execution policies are evaluated whenever new data appears on an input stream for the multi-event handler.

The execution policy allows us to fine tune the control we have over the influence of the arriving data on the application. In the traditional event model, whenever an event is received the corresponding event handler executes. So, it is equivalent in this respect to having a multi-event handler with one event stream with an execution policy that executes whenever a new event arrives. As soon as we add the concept of additional event streams which may trigger the execution of the multi-event handler to the model we need the abstraction of execution policies to allow the application developer to specify exactly what actions they want to occur on arrival of the events.

With execution policies we can use arbitrarily complex expressions which specify exactly when we want our multi-event handlers to be executed. e.g. S1 AND S2 specifies that both S1 and S2 must have data, or perhaps if we only need one of sensor S2 or sensor S3 to execute the handler , S1 AND (S2 OR S3). This high degree of control in the execution of the multi-event handlers allows complicated applications to be written easily using the abstractions.

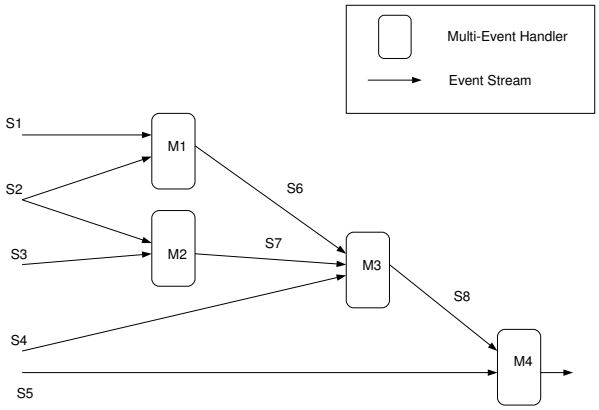

Figure 3. Composition of Multi-Event Handlers

Composition of Multi-Event Handlers

The ability to compose multi-event handlers into chains is a very useful and elegant feature of the AESOP model. The output of multi-event handlers are themselves event streams and can become the input to a second layer of multi-event handlers and so on. The composition is transparent to the multi-event handlers, source and destination multi-event handler need not be aware that it is part of a chain or producing event stream for use as inputs to other multi-event handlers. These events can go to multiple multi-event handlers and/or a traditional consumer of events. This is demonstrated in figure 3. Three layers of multi-event handlers are shown. The sensor data is continuously being refined and fused, creating higher level sensor streams such as S6, S7 and S8 which in turn can be used as inputs to other multi-event handlers.

Example System

As an example let us consider a simple implementation of a sensor fusion application. If for example the application relied on one very reliable sensor (sensor A) and one sensor which was prone to failures (sensor B) we could tackle the failures of sensor B using replication. We could build our application using a single sensor of type A and multiple sensors of type B. If we implemented our multi-event handler with an execution policy which executed when new data appeared from sensor A we could effectively ignore the surplus events provided by the multiple sensor Bs, in a highly transparent manner to both the multi-event handler, the sensors of type B and any applications built on top of the output.

This would deal with sensors which failed silently, i.e. sensors which on failure, stopped emitting data e.g., if the sensor was embedded in the environment and the application was travelling through the environment network partition might

occur quite frequently, resulting in sensors failing silently with respect to the application. This solution would not deal with noisy failure of sensors however and perhaps additional abstractions are required in the middleware to remove outliers or implement some voting strategy, although this in itself may be better implemented as a multi-event handler.

THE AESOP C-MAPPING

Our current implementation of AESOP is written in C on a Linux platform. We chose C because of the resource constrained devices on which we implement our work in the sports domain. We call our implementation C-AESOP. C-AESOP is implemented on top of the STEAM middleware [6]. The STEAM middleware provides an implementation of the implicit event model, i.e., consumers subscribe to event-types rather than with producers or with a broker [7]. The STEAM implementation uses multicast as its communication mechanism. The implementation of STEAM that C-AESOP uses is also implemented in C.

Event Streams in C-AESOP

In the C-AESOP implementation we introduce the notion of input streams and output streams. Input streams are used to provide values to the multi-event handler and store the sliding window of values. Output streams are used to transmit onwards the results of the multi-event handlers. In the STEAM middleware consumers of events register their interest by subscribing to event subjects. They then receive all subsequent events which match this event subject. In C-AESOP event streams are identified using universal identifiers, which correspond to the STEAM event subject. These identifiers allow us to bind the input and output streams to our event streams. Input and output streams are used exclusively by individual multi-event handlers. This is in contrast to the event streams in the abstract model which are shared among the entire system. Event streams correspond to streams of STEAM events in C-AESOP. Therefore multiple input streams can bind to a single event stream, by way of the event subject and multiple output streams can also bind to a single event stream by way of an event subject. The sliding window mechanism is implemented in C-AESOP and is defined over input streams as these are unique to individual multi-event handlers.

The event based implementation of C-AESOP provides a high degree of network transparency to applications constructed with it. Individual multi-event handlers can be on separate machines in a network, on the same machine or in the same process. To reduce network and battery usage efficient local communication has been implemented to exclusively deliver events locally on a machine. This can occur when the application developer explicitly tells C-AESOP that the events are to be delivered locally using the STEAM proximity *LOCAL*. This optimisation prevents them from being delivered on the network allowing significant resource savings while sacrificing some of the features of distributed events.

The data content of the stream events in C-AESOP are typed. This allows us to send data encapsulated in any arbitrary C struct or basic type as content in an event stream of the same type.

```
void average(double* d1,double* d2,double* d3,
    double* result)
{
*result = (*d1 + *d2 +*d3) /3;
}
```

Figure 4. Simple multi-event handler

Multi-Event Handlers in C-AESOP

The creation function for multi-event handlers takes as parameters an array of input streams, an output stream, an execution policy and a function pointer. The function pointed to in the function pointer has as many parameters as input streams with an additional parameter which is used for the result. The types of the parameters to the function are pointers to the corresponding type in the input and output streams. This allows the developer to write the logic for what to do each time the multi-event handler is called using the types as if they were writing a traditional C function. The C-AESOP middleware then manages the arrival of events, and supplies values to the appropriate parameters to the function. Any results of the multi-event handler by convention is assigned to the last parameter. When the middleware has executed the function it checks for a return value. If any value has been assigned to the result it then sends this value out on the output event stream. This allows for the instances where no value is returned by the multi-event handler - and so no onwards event is sent - and provides a clean and easy programming interface to the application developer.

Figure 4 shows an example of a simple function that could be passed to a multi-event handler to calculate the average of three double streams. Here we see three pointers to doubles d1,d2 and d3. These correspond to three input streams whose declarations are show in figure 5. The input streams in the example are declared with sliding windows of various lengths although this example does not show this historical data being accessed. Figure 5 also shows the output stream being declared. The creation of the multi-event handler itself is also shown. We pass the number of inputs, the array of inputs, the output stream, the function to be executed and the execution policy as parameters to the creation function. This is all the code required to create and begin the execution of this multi-event handler.

Execution Policies in C-AESOP

The present implementation of C-AESOP does not implement the arbitrary Boolean expressions in the abstract model. Currently we implement a subset of all the conceivable execution policies. There are three different execution policies implemented, which we call *greedy*, *lazy* and *individual*. These policies can be scoped at the multi-event handler level or the stream level. A multi-event handler with a greedy execution policy executes as soon as any of its streams receive new data using the most up to date values for the other streams. This results in every instance of new sensor data triggering an execution of the multi-event handler. A multi-event handler with a lazy execution policy waits until all of the streams

```
stream* inputs[3];
inputs[0] = (stream*) create_double_input_stream(1,"DoubleInput1",LAZY);
inputs[1] = (stream*) create_double_input_stream(20,"DoubleInput2",LAZY);
inputs[2] = (stream*) create_double_input_stream(3,"DoubleInput3",LAZY);
stream* output =  (stream*) create_double_output_stream(20,"DoubleOutput");
meh_t* average_meh =create_meh(3,inputs,output,average,INDIVIDUAL);
```

Figure 5. Creating Event Streams and Multi-Event Handlers

have new data. It then executes with the most up to date values from the sensor stream. This lazy approach can result in sensor readings being discarded, as the multi-event handler always executes with the newest value it has received on a particular stream. The individual policy is used by multi-event handlers to pass the decision on whether to execute on to the streams and allow each one to specify whether it is greedy or lazy. This allows a combination of the execution policies to be used for the multi-event handler e.g. we can specify that when a particular stream or group of streams which are inputs to the multi-event handler get new data, we want to execute the multi-event handler. We do this by specifying them as GREEDY on creation and specifying the remaining streams as LAZY.

Execution policies in C-AESOP can be provided when we are creating input streams and when we are creating multi-event handlers. The C-AESOP middleware then executes the multi-event handlers according to the execution policies provided with the multi-event handlers execution policy taking precedence over the streams policy. In practice it would be unusual to specify both as the input stream is unique to the multi-event handler in question, so in practice input streams must only specify an execution policy if the multi-event handler's policy is set to INDIVIDUAL. As stated previously specifying execution policies for individual streams allows us to exercise finer-grained control over the execution of the multi-event handler.

Special Streams

In keeping with the event stream abstraction some special streams are provided by the C-AESOP middleware. Frequently it is desired to perform some activity with a set frequency. To aid with this a timing stream is provided which can be used to execute the multi-event handler in a periodic fashion. This stream can be combined with execution policies and other streams in the standard manner. It allows programmers to periodical execute a multi-event handler. It can be used for example to create an output of a set frequency from a combination of asynchronous sensors.

There is a second type of special stream provided to the application. These streams have a fixed value and a length of one and are used to provide parameters to multi-event handlers. They can be of an arbitrary type as declared by the programmer. They exist to support reuse of multi-event handler functions and are not taken into account when evaluating execution policies.

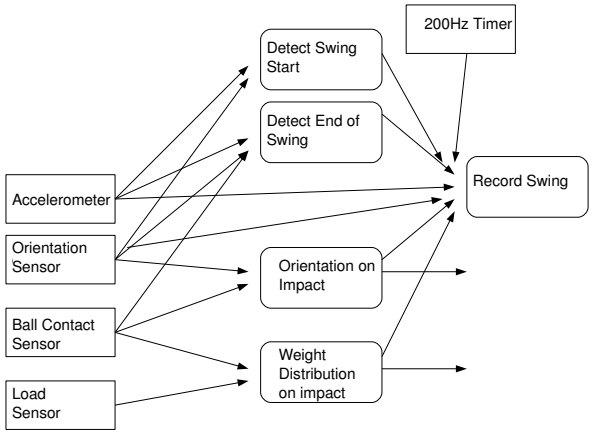

Figure 7. Squash Application

EXAMPLES

In order to demonstrate the abstractions and our implementation of them in a C based middleware we will describe an example application in the sports domain. Squash is a sport played indoors in a small static area between two athletes using rackets and a ball[13]. We have augmented a squash racket with several sensors in order to provide advice to the athlete on how to improve their swing. The angle of contact between the face of the squash racket and the ball is of interest to advanced squash players. An "open" racket face, i.e. with the racket face tilted upwards, is desired when hitting most shots as this causes the ball to bounce lower and therefore increases the work rate of the opposition player. We have augmented a squash racket with an orientation sensor and a sensor which detects ball contact. Figure 7 shows the events and multi-event handlers used in the application. When a ball contact event occurs the orientation of the racket is calculated and the resulting orientation event is produced. We also detect the start and end of swings in separate multi-event handlers and use these events to record the athletes swings for later analysis. Figure 6 shows a simple multi-event handler which produces events with the value of the orientation of the racket on contact with the ball. This multi-event handler has two inputs: an orientation event stream and a ball contact event stream. The ball contact input is designated with an execution policy of GREEDY while the orientation event stream is designated LAZY. Therefore whenever a ball contact event arrives we send a new event representing the orientation of the racket on impact.

We are in the process of adding load sensors to the squash players shoes. This would allow us to analyse the athletes

```
void orient_on_impact(Orient_str* orient,Impact_str* impact, void* result){
    return orient;
}

main(){
...
    inputs[0] = (stream*) create_orient_str_input_stream(1,"ORIENT",LAZY);
    inputs[0] = (stream*) create_impact_str_input_stream(1,"IMPACT",GREEDY);
    meh_t* detect_orient_on_impact_meh =create_meh(2,inputs,output,orient_on_impact,INDIVIDUAL);
...
}
```

Figure 6. Example Multi-Event Handler

weight distribution as they perform the swing which would allow us to infer details about the stance of the player during the swing. At present we use the application to record the swings of athletes and to provide feedback on the number of ball impacts the athlete makes with an open racket face.

Future Work

Sliding windows are the only history mechanism in the present model, although various other mechanisms such as tumbling windows[10] may be included in the future in a similar manner to history mechanisms which have proved useful in the domain of data stream managements systems, e.g. in scoping the length of join operations [1].

At present the storage policy for event streams is to store all data on the stream. Other data storage policies such as storing event data in the sliding window only for data that has been used in the multi-event handler may also be required.

Conclusion

We have presented three abstractions that extend the event-model: multi-event handlers, event streams and execution policies. The extensions are intended to ease programming of sensor-driven ubiquitous computing applications. Multi-event handlers are a generalisation of traditional event handlers which take as inputs one or more events. Execution policies are used to define precisely what combination of input event streams causes the execution of the multi-event handler. Sliding windows over event streams are provided as a mechanism to help reason about historical sensor data in an application specific manner. Composition of multi-event handlers is also supported in the AESOP model.

We have implemented our extensions, which we call C-AESOP, on top of the STEAM event middleware. Although STEAM uses an implicit event model, we believe the abstractions presented are general enough to be used with all event models. We have demonstrated C-AESOP using an application in the ubiquitous computing domain.

References

[1] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 2002. ACM Press.

[2] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic support for distributed applications. *Computer*, 33(3):68–76, 2000.

[3] Object Managment Group. *The Common Object Request Broker: Architecture and Specification, V2.1.* Object Management Group, 1995.

[4] Mads Haahr, René Meier, Paddy Nixon, and Vinny Cahill. Filtering and scalability in the eco distributed event model. In *Proc. of the 5th Int. Symposium on Software Engineering for Parallel and Distributed Systems (ICSE/PDSE2000*, pages 83–95, 2000.

[5] Guoli Li and Hans arno Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *In ACM/IFIP/USENIX 6th International Middleware Conference*, pages 249–269, 2005.

[6] René Meier and Vinny Cahill. Steam: Event-based middleware for wireless ad hoc network. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 639–644, Washington, DC, USA, 2002. IEEE Computer Society.

[7] Rene Meier and Vinny Cahill. Taxonomy of distributed event-based programming systems. *Comput. J.*, 48(5):602–626, 2005.

[8] Sun Microsystems. *Javabeans api sepcification, version 1.01*, July 1997.

[9] Karl O'Connell, Tom Dinneen, Steven Collins, Brendan Tangney, Neville Harris, and Vinny Cahill. Techniques for handling scale and distribution in virtual worlds. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 17–24, New York, NY, USA, 1996. ACM.

[10] Kostas Patroumpas and Timos Sellis. *Current Trends in Database Technology*, chapter Window Specification over Data Streams, pages 445 – 464. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006.

[11] Peter R. Pietzuch and Jean M. Bacon. Hermes: A distributed event-based middleware architecture. *Distributed Computing Systems Workshops, International Conference on*, 0:611, 2002.

[12] P.R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44–55, Jan/Feb 2004.

[13] S. Reilly, P. Barron, V. Cahill, K. Moran, and M. Haahr. *Digital Sport for Performance Enhancement and Competitive Evolution: Intelligent Gaming Technologies*, chapter A General-Purpose Taxonomy of Computer-Augmented Sports Systems. Hershey, PA: IGI Global, 2009. (In Press).

[14] Gradimir Starovic, Vinny Cahill, and Brendan Tangney. An event based object model for distributed programming. In *Dublin City University*, pages 72–86. Springer-Verlag, 1995.