MedSMan: A Streaming Data Management System over Live Multimedia

Bin Liu
School of Electrical and Computer Engineering
Georgia Institute of Technology
bliu@ece.gatech.edu

Amarnath Gupta
San Diego Supercomputer Center
University of California San Diego
gupta@sdsc.edu

Ramesh Jain
Department of Computer Science
University of California Irvine
jain@ics.uci.edu

ABSTRACT

Querying live media streams is a challenging problem that is becoming an essential requirement in a growing number of applications. Research in multimedia information systems has addressed and made good progress in dealing with archived data. Meanwhile, research in stream databases has received significant attention for querying alphanumeric symbolic streams. The lack of a unifying data model capable of representing multimedia data and providing reasonable abstractions for querying live multimedia streams poses the challenge of how to make the best use of data in video and other sensor networks for various applications including video surveillance, live conferencing and Eventweb. This paper presents a system that enables direct capture of media streams from sensors and automatically generates meaningful feature streams that can be queried by a data stream processor. The system provides an effective combination of extensible digital processing techniques and general data stream management research.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Retrieval models, Search process; D.3.3 [**Language Constructs and Features**]: Data types and structures

General Terms

Design, Experimentation, Languages

Keywords

multimedia, stream, continuous queries, languages, events

1. INTRODUCTION

Multimedia information retrieval has been a popular topic of research for quite some time [14, 6, 4, 8, 2]. Most multimedia information systems deal with archived video, audio, and images by using powerful DSP techniques, which remain a very active research area. In database community, *continuous queries*, which are persistent queries that are issued once and then logically run continuously over live and unbounded streams, have also become a major research area in data management (e.g., OpenCQ [12], NiagaraCQ [7], Aurora [1], Telegraph [13] and STREAM system [5]). Live sensors ranging from simple RFIDs to webcams become common in many applications, from homeland security to businesses, and produce enormous volumes of continuous sensor data called media streams. Expectedly, the large scale deployment of various sensors gives rise to more complex applications where multiple live streams of heterogeneous media must be jointly monitored to query interesting events [10]. A *media stream* is usually the output of a sensor device such as a video, audio or motion sensor that produces a continuous or discrete signal, but typically cannot be directly used by a data stream processor. To evaluate queries on media streams, one need to continuously extract content-based descriptors, commonly called *features*, and identify the corresponding media portions by evaluating queries on the generated *feature streams*, which are post-processed by one or more transformers and correlated to the media streams temporally and in terms of content.

The term *live multimedia* refers to the scenario where the multimedia information is not produced through manual editing, but is captured in a real-life setting by different sensors and streamed to a central processor. When multiple sensors of similar or different types are placed in an application, their placement in the physical environment and the models of the events that they are supposed to capture play very important roles in the extraction and assimilation of information. The basic premise behind delivering multimedia information is that while an individual media channel or a derived feature stream captures some aspects of such an event, it is the synchronous combination of all the streams that captures the entire intended semantics of the content and makes the event detection easier or more effective than only using one media or one aspect of that media.

Our research focuses on applying the knowledge about the location of sensors in a physical space and the role of spatially and temporally correlated information obtained from disparate sensors. An environment model is utilized to capture the physical placements and constraints on the information obtained from these sensors. The role of an event

MM'05, November 6–11, 2005, Singapore.

model in extracting and organizing information is also discussed. Then accessing and processing media streams are presented. We contend that a live multimedia management system (LMMS) should use the additional contextual information and semantic constraints to answer more complex queries over live media streams. The system deals with continuous queries on multiple live media streams by taking advantages of automatic and real-time multimedia information processing techniques. We previously studied media/feature stream generations and feature function implementations in [11], while this paper focuses on event modelling and stream query processing.

1.1 A Motivating Example

Consider a broadcast seminar application which remote participants can decide to "tune in" only when events of their interests occur. Such a professional conference room includes a podium connected with a projector that sends out a video stream V_P from the presentation projector, a video camera C_S and a microphone M_S reserved for a speaker. K seats are allocated at the table fitted with individual microphones $M_1 \ldots M_k$. A video camera C_P captures the podium, and n other cameras $C_1 \ldots C_n$ capture different segments of the conference room, so that a camera C_i $(1 \leq i \leq n)$ can be mapped to a specific subset $\mu(C_i)$ of the k participants. In this scenario highlights can be the relevant media streams for any time interval when the speaker presents or answers questions. The system is designed to evaluate continuous queries, such as the followings:

Query 1 Display the speaker's video when he is answering the $4th$ question about "multimedia".

Query 2 Display the speaker's video when he walks around and speaks.

Query 3 Connect to the speaker's audio only when he is speaking loudly for a while.

Query 4 Display the speaker's video from 1 minute before he walks around until he stops.

Query 5 Show content of V_P while the slide stream goes from slide-number 12 to 18.

2. THE MEDSMAN DATA MODEL

An event-based query approach provides a feasible way for users to access the related media stream content of interest at specific levels, and give a meaningful, more refined view of the world around them. However, the gap between the low-level sensor signal data and the high level event description is huge. Therefore, a unifying data model that can access heterogeneous media streams and provide effective interpretations as well as reasonable abstractions for them is necessary. This paper proposes a data model that consists of three components – the media and feature (MF) *stream model* modelling the data streams produced by sensors and feature transformer (or generation) functions, the *environment model* providing the prior knowledge of the physical setting from which stream data is collected, and the *event model* capturing the possible events that can occur in the environment.

2.1 The Environment Model

The environment $\mathcal{E}$ of a live multimedia schema (LMS) consists of a set L of *landmarks*, a set T of sensor types, a *placement* P of sensors, a map $\bar{\mu}$ called *the sensorium* for all sensors, together with $\mathcal{C}$, a set of sensing constraints.

A landmark is a named object that is fixed in space, and is defined by the pair o_name, o_extent, where o_extent is a point or region in space occupied by the object; thus the the podium p in our example occupying a specific cuboidal region within the seminar room is a landmark. We also assume that standard spatial predicates like `inside` and `overlaps` in 2D and 3D can be used as needed by the system.

The sensor type depicts a classification over sensors, as well as the properties of the sensor, including its operating characteristics. Some of the characteristics are used to interpret the signal. Some other characteristics are obtained by standard signal transformations that produce a *feature stream* concurrently obtained with the direct sensor signal. For a camera one needs to know what RGB values would constitute a "black frame", or for a microphone, what FFT feature should be considered to be noise. A more detailed discussion on feature streams is in Section 2.3, while a property called the *expected sensitivity* of a device is defined as:

DEFINITION 1. *Let S be a non-cardinal scale of measurement with a comparison function $\leq_S$. Given a location ρ relative to the position of a sensor of type T, expected sensitivity I_S is a function that maps from ρ to S.*

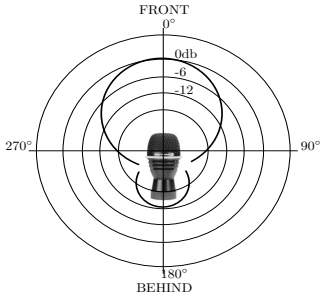

Figure 1: Expected sensitivity of a hyper-cardioid microphone.

The benefit of modelling a sensor's expected sensitivity can be seen in Figure 1, where the "pick up" and attenuation characteristics of microphones is depicted as a function of its radial and angular distance from the sound source. Therefore, if we know the location of a normal-volume speaking voice behind a hyper-cardioid microphone, we can estimate whether that voice is audible from the sensor stream of that microphone. The placement of a sensor s_i of type t_j in an environment is a set of functions to assign it: 1) a location in the same coordinate space as the landmarks, 2) a parameter called the "sensing direction" and optionally 3) a set of landmark objects directly sensed by the sensor. The placement P of all sensors, though not shown here, can be modelled as a spatial relation. Once the placement of sensors is known, the expected sensitivity can be utilized by composing it with sensor placement parameters. The result of the composition can be realized by a set of functions, collectively called the *sensorium* $\bar{\mu}$ of the environment. One function may, given a small region, compute the signal effectiveness of a specific sensor; another function can, given a landmark, determine a set of sensors (or a specific type) that cover the region at a specified level of signal effectiveness. In an alternate, coarser-grain implementation, the sensorium may be materialized as a relation called the *coverage map* shown in Table 1. Note that signal effectiveness for any sensor is an ordinal

Region	Sensor	Type	Effect	Landmarks
r1	s1	audio	loud	chair2
r1	s3	audio	faint	chair2
r1	s2	video	covered	chair2, door1
r2	s1	audio	loud	chair4
r2	s2	video	partial	chair4, window2
...	...	...	...	...

Table 1: A coverage map derived from sensor placement and expected sensitivity

scale that corresponds to its expected sensitivity. Pragmatically speaking, the user may also use a graphical tool to manually construct a coverage map instead of computing it from sensor specification.

Movable sensors fall into three different situations. If a sensor is attached to or follows along with a moving object (such as a microphone attached to a mobile person), we consider it a dedicated stream source. If a sensor is not fixed to a single region but switches between multiple regions, the coverage map needs to be updated accordingly. If a sensor changes the coverage regions continually, the coverage map needs to be recomputed. The latter two situations are beyond the scope of this paper.

Another aspect of an environment model is the set of device and domain constraints. The system should ensure that the returned stream segments as a query result must be valid for any query. For every sensor type, a set of *validity constraints* are defined and implemented as system-wide distinguished predicates `invalid(SensorStream)`, which are used as guard clauses for each related sensor or feature stream in a query. As a new sensor is added, one specifies new rules stating when the sensor input should be considered invalid.

```
invalid(camera_signal(X)) ← black-frame(X)
black-frame(X) ← frame-average(X) < 30
invalid(microphone_signal(X)) ← background_noise(X)
background_noise(X, interval) ← FFT_feature(X,Y),
window-average(Y, interval) < Threshold
```

All the above variables are universally quantified. For cameras, getting a black frame makes the frame unusable. For an audio channel, the s/n ratio can be determined by validating a window averaging function on the corresponding `FFT_feature` (i.e., a histogram where the X axis is a bin of frequency ranges and the Y axis is the coefficient) to test if the signal have much high frequency noise. The *Threshold* value of the sensor noise also depends on the specific application knowledge (e.g., a higher threshold is required for sound noise on a street than in an empty auditorium). Furthermore, the validity constraints should also be tested for the feature computation modules so that features are only computed from valid parts of the streams.

Synchronization constraints may also exist. For example, the microphone controller may allow only one microphone from the audience to be active at any time. One way to interpret this rule is – if one audience microphone already has a valid signal, then no other microphones should be tested until the active microphone becomes inactive for at least a period of τ.

```
check(microphone_signal(X))←active(microphone_signal(Y))
before(duration>τ) inactive(microphone_signal(Y)),X≠Y
```

The `check` is another distinguished predicate, which when true, initiates an action for the stream manager. The connective `before(duration > τ)` is used to model the temporal relationship between two predicates, parameterized by a duration constraint. A different method is used to specify constraints like "video cameras of the bridge may be on for two minutes at 15 minute intervals, unless some notable events (like a highlight incident) occur", where the relative order of their start-time may not be specified. We express these constraints as *default values* of sensor attributes `signal-duration` and `signal-frequency`.

The environment model serves as the setting against which both the event model and the stream model are specified.

2.2 Event Model

The term *event* has traditionally been used in the database community to designate a state-changing phenomenon like an update in a database, or turning-on of a boolean variable. In our setting, we refer to an *event* as a happening that occurs for a certain time interval and at a certain location. Every event, therefore, has a *start*, *end* and a set-valued *location* attribute. An event can belong to one or more *event classes*. The model allows a hierarchy H_E of event classes that admit multiple inheritance. A *subevent* E' of an event E is an event that occurs such that $E'.start \geq E.start$ and $E'.end \leq E.end$. However, **class-of**(E') need not be the same as **class-of**(E). If an event does not have any subevent, it is called *atomic*. The start and end times of a larger event is determined by the start and end times of all its subevents; the location attribute of an event can be defined by different aggregate functions over subevent locations. Two straightforward choices are union of location names (set-union semantics) and spatial-union of the extents of the atomic events (spatial-union semantics).

An event-schema S defined over an event hierarchy H_E is a copy of H_E adorned by occurrence constraints. Over an event hierarchy, we may define the event schema of a specific meeting to be an event with four subevents – "speaker introduction", "speech","questionnaire session" and "speaker thanks", where "questionnaire session" may be modelled as an arbitrary number of alternate occurrences of "audience question" and "speaker response". Suppose we put the constraints that there will be exactly one occurrence of each of these four subevents, and that they will occur consecutively. Furthermore, the duration of the speech will be 30 minutes and the total questionnaire session will be 10 minutes. Evidently, by this model, no audience member can interrupt the speaker during the speech, and the event lasts for 40 minutes. The event schema is shown in Figure 2.

The relationships among event instances are defined using three structures. The subevent relationship induces a tree T_E called the *event tree*, which is an instantiation of the event schema. An event tree is not constructed per sensor, but is a composite structure using multiple sensors. Moreover, the temporal relationship over all events is modelled as an interval tree T_I. In adddition, the transition between two events may occur by a named state-change. In the meeting example, the transition from the event "audience member 4 talking" to the event "speaker talking" may be named "speaker response". The state changes over the set of atomic events can be represented together as an edge-labelled transition graph G_T. Note that while T_E is an instantiation of event schema S created during system design, T_I and G_T are

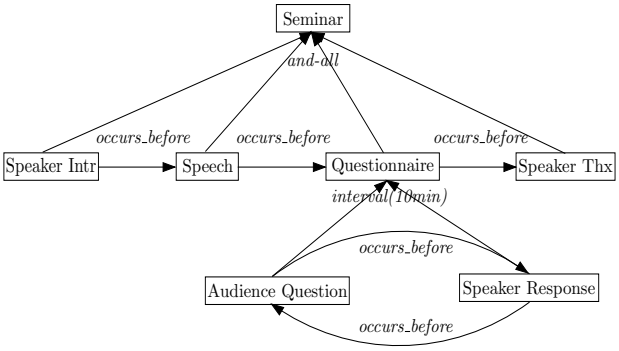

Figure 2: A seminar event schema.

occurrence structures and are produced at run-time by detecting event instances from data streams. Join operations can retrieve the stream contents occurred in T_I intervals. Similarly, the nodes of G_T contain pointers to stream contents. If an event that an audience member has spoken in the middle of the speech is detected, it will not correspond to the event schema, but will be recorded as an unexpected event in T_I, and in G_T. An important aspect of our model is that the occurrence structures T_I and G_T adhere to the event schema only loosely, thus allowing for deviations that naturally occur in real-life situations. It is possible for a system designer to accommodate for such deviations by assigning a *strictness qualifier* to a constraint:

```
event-duration(questionnaire(600)) non-strict
```

If the constraint is strict instead, the system would still record events but put them as instances of a special event class called *error-events*. The edge-labels for the deviant transitions of G_T may be defined using rules. For example, there may be a rule that an edge from the "speaker talking" event to an unexpected "audience member talking" will be labelled as an "interruption" edge, while returning from the unexpected state to the schema-defined state will be a "restoration" edge. If such a rule does not exist, G_T will have specially-marked ϵ-edges.

Event Operations In an LMMS, event operations are defined at two levels. At a higher level, the user can specify their event queries by constructing an event expression, much like the event expressions found in literature. At a lower level, events are defined in terms of stream operations, which is discussed in Section 2.3. In the LMMS model, an event can be named as discussed in the previous paragraphs, or it can be derived through an event expression.

DEFINITION 2. *An event expression is defined by the following rules:*

- *An empty event ϕ is an event expression.*
- *All named events e are event expressions.*
- *Events specified as a path expression over the event-subevent tree T_E or event classification hierarchy H_E are event expressions.*
- *Events specified as path expressions over the graph G_T are path expressions.*

Operator	Meaning
e_1 occurs_before e_2	e_1 ends before e_2 starts
and-any($e_1 \ldots e_n$, k)	any k of the n events occur
and-all($e_1 \ldots e_n$)	all n events occur
e_1 xor e_2	either e_1 or e_2 occurs but not both
e_1 overlaps e_2	the intervals of e_1 and e_2 overlap
e_1 not e_2	e_1 occurs and e_2 does not occur
interval(e_1)	there is an interval where e_1 occurs

Table 2: Event operators in our system

- *If e is an event expression, then expressions of the form (T* `before` *e) and (T* `after` *e) where T is a time-duration are also events whose start time is T units before (resp. after) the event e.*
- *If $e_1 \ldots e_n$ are event expressions then any formula formed by using the operators in Table 2 is an event expression.*
- *Nothing else is an event expression.*

Note that the operations in Table 2 generate event instances. Thus, $E = e_1$ `occurs_before` e_2 evaluates to an event that starts with e_1 and ends with e_2. Similarly, the expression $E = T$ `after and-all`(e_1, e_2, e_3) refers to the derived event E that starts with the earliest of the events and ends T units after all the three events have occurred. A query involving event expressions is typically specified through the special predicate `occurs(event, interval)` meaning that the event occurs in the interval. Typically, an event query will also situate events in terms of their relations with the objects specified in the environment model. The higher-level events are related to the observable streams, and event queries are illustrated through the examples in Section 4.4.

2.3 The Stream Model

The Media and Feature (MF) stream model works in the settings specified by the environment model, and serves as the foundation for the event model.

2.3.1 Formal Definitions

Because of their continuous nature and the stream dependency, both media and feature elements require explicit and exact timestamps. The time attributes provide valuable information for the stream generating and query processing. In our framework, the element of a media or feature stream is defined as a *tuple*, which consists of a logical sequence number (sqno) indicating its position in a stream, a temporal extent defined by a pair of *start* and *end* timestamps (t_s, t_e] (for a time-point attribute $t_s = t_e$, a single time point is represented as t_d), and any other media attribute. We give a series of conventions and definitions that make more precise the notions of media and feature streams.

CONVENTION 1. *A sequence T, is said to be* ***continuously well-ordered*** *iff (1) T is well-ordered, and (2) for each time-unit $(t_{si}, t_{ei}]$, there must be one and only one directly following time-unit $(t_{si+1}, t_{ei+1}]$ in T, where $t_{ei} = t_{si+1}$. We refer to a continuously well-ordered set of time-units as a* ***continuous time set****. A corresponding tuple value holds in each time-unit.*

CONVENTION 2. *A sequence T is said to be **discretely well-ordered** if and only if T is well-ordered, i.e., the continuity clause does not apply for two consecutive units. We refer to a discretely well-ordered set of time-points as a **discrete time set**. At each time-point t_d, a corresponding tuple value holds.*

DEFINITION 3. *A **continuous media stream** is a sequence of tuples, each consists of a sequence number (m_{sqno}) uniquely identifying its position in stream, a pair of start and end timestamps $(t_s, t_e]$ whose domain is a continuous time set, and a media valued attribute v_m valid only during $(t_s, t_e]$.*

DEFINITION 4. *A **discrete stream** is a sequence of tuples, each consists of a sequence number (m_{sqno}) uniquely identifying its position in stream, a media or non-media valued attribute v_m, and a time-point t_d, defined on a discrete time set domain, indicating when v_m arrives intermittently.*

DEFINITION 5. *A **feature stream** is defined as a sequence of tuples, each consists of a sequence number (f_{sqno}) uniquely identifying its position in stream, a feature value attribute v_f, a time-point attribute t_f indicating when v_f is computed, and a set $\bar{m}_{sqno}$ identifying the media tuples or a set $\bar{f}_{sqno}$ identifying other feature tuples, from which a feature tuple is derived.*

2.3.2 Stream Semantics and Operator Classes

While being absent in the general purpose Data Stream Management System (DSMS) [3], direct stream-to-stream operators are necessary for media stream systems, since media and feature streams are (1) quite different in operations; and (2) dependent. Besides, we need to compose media stream(s) for presenting query results. The four subclasses of stream-to-stream operators, shown in Figure 3, are:

- MS-to-FS operators that generate feature streams from media streams;
- FS-to-MS operators that map back to the deriving media stream fragments from feature tuples;
- FS-to-FS operators that generate feature streams from existing feature streams;
- MS-to-MS operators that reformat a media stream or compose a composite media stream from multiple media streams.

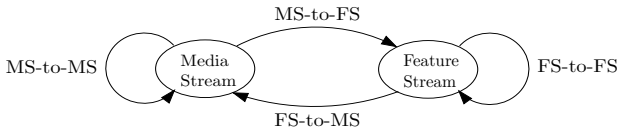

Figure 3: Stream operator classes and semantics.

Time Attributes and Order Our definitions in previous section require explicit time attributes in stream schemas. Furthermore, we assume all tuples in a stream are discretely well-ordered by time-points, or continually well-ordered by intervals. Note tuples from different streams may not be totally ordered, but partially ordered. For the presence of explicit time attributes, tuples in a relation produced from a stream by sliding window operators are also ordered in time, rather than a bag of unordered tuples. As we will present in later sections, the order of tuples is necessary to guarantee tuple continuity for queries over intervals.

Overlap Join In most multimedia applications, especially for audio-visual applications, the synchronization between different media streams needs to be precise to satisfy perceptual continuity, when viewed by the human user. This fact implies that media tuples from different media streams require very strict temporal relevance for join, so are feature stream tuples. Due to the unbounded nature of streams, continuous queries over streams are often defined in terms of sliding windows, either tuple-based or time-based [9]. Nevertheless, such windows do not take the tuple intervals into account. In those applications, a tuple (e.g., temperature reading) is instant-based, rather than interval-based. This is not true for typical media stream tuples, and may have problems (e.g., false join) in joining tuples with non-overlapping intervals. A loose window including multiple non-overlapping tuples cannot satisfy the strict temporal join constraint for media or feature tuples from different streams. Instead, a more precise and strict metric is required to join them. We apply an overlap join TSJ_1 defined in [16]: *All participating tuples that satisfy the join condition share a common time point.* Tuples whose temporal attributes overlap in time have the *highest temporal relevance*, thus can be joined. Those non-temporal attributes of tuples can only be joined after satisfying the premise that the corresponding temporal attributes are overlapping in time.

Dynamic window Either tuple-based window or time-based window has a fixed window size predefined in query language. However, there are cases in which *dynamic windows* (whose lengths are not predefined but detected at run time) are preferred, for both query semantic reason and cost efficiency. Suppose we want to join a keyword detected when a presenter is speaking a sentence (during period $P1$) with the same keyword detected from a presentation slide video when a specific slide is shown (during $P2$). Assume $P1$ overlaps with but is not identical to $P2$. Both $P1$ and $P2$ are unpredictable in terms of period ends and lengths. They are dynamically determined by some subqueries. Since the lengths of such periods are not deterministic, a fixed window (W) is not efficient – if W is short, it may not contain an entire $P1$ or $P2$ and require multiple sequential windows and additional post-processing; if W is too long, it may waste too much memory since typical media tuples are very space consuming. Instead, a dynamic window determined by period detection operations at run-time is desirable.

2.3.3 Algebraic Operators

Following the definitions in previous sections, media and feature stream operations are designed for the varying-interval media and feature tuples. The feature tuples carry pointers (i.e., sqno) to the deriving media tuples. The basic algebraic operators to manipulate them are defined in Table 3. Very often, media stream queries only concern with tuples in periods of interest, which are either fixed or dynamically determined by subqueries at run-time. The last three operators provide the capabilities for such dynamic period queries.

3. OVERVIEW OF THE SYSTEM

Figure 4 shows the overall system architecture. The application environment model provides application-dependent

Operator	Definitions	Stream
Scan	Get the next tuple of a same stream (push)	M/F
Selection	Produce a new stream R_s from stream R with a subset of its tuples	M/F
M_Projection	Create a stream by extracting a substructure from each media tuple	M
M_Fetch	Apply to one or a set of feature tuples to recursively retrieve the set of media fragments that produces the feature tuple(s) (pull)	M
F_Fetch	Apply to one or a set of feature tuples derived from other feature tuple(s) (pull)	F
O_Join	Join any tuple r_i from R_i with any tuple or set of tuples r_j, each from R_j ($m \geq 2$; $1 \leq i, j \leq m$; $i \neq j$), if r_i and all r_j have a common intersection in time	M/F
F_BJoin_1M	Join two feature streams generated from a same media stream taking sqno as joining attribute	F
F_mJoin_1M	Join multiple feature streams generated from a same media stream taking sqno as joining attribute	F
F_BJoin_2M	Join two feature streams from two different media streams joining on interval attributes	F
F_mJoin	Multi-way join m ($m \geq 2$) feature streams derived from different media streams	F
Compose	Reformat single media stream for output; or multiplex multiple streams to construct a composite media stream	M
M_PScan	Get the next tuple of a media stream, which falls into a specified period [*start, end*], and both period variables can be set or reset for repeated occurrences	M
P_Detect (p_predicate)	This class of period detection functions take as parameter a predicate on feature values, and return qualifying period(s); periods may repeat and the *start* and *end* are dynamically re-computed at run-time	F
P_Aggregate (period)	This set of aggregation functions (SUM, AVG, MIN, MAX, COUNT) perform over the given period, which is typically not fixed but returned by P_Detect functions	F

Table 3: Definitions for stream algebraic operators

domain knowledge for both the MF stream model and the event model. At the stream processing level, a user creates definitions of media streams and declares their instances via Media Stream Description Language (MSDL). The raw streams captured from sensors are processed and transformed to user-defined media streams, which are transmitted to both the query engine and the feature generation component, where various feature streams are automatically generated via feature generation functions (FGFs) defined in Feature Stream Description Language (FSDL). The media streams and derived feature streams are queried by the query engine, which outputs the qualifying media portions to the composition component. The composed media results are returned to the user. To issue continuous queries over live media streams, a *Media and Feature Stream Continuous Query Language* (MF-CQL) is designed and implemented by exploiting the semantics of media and derived feature streams, as well as the inter-stream constraints among them. At the event processing level, a user creates definitions of event schema via Event Description Language (EDL). The event detection module takes feature streams together with their deriving media streams as inputs and detects event occurrences, with the knowledge and rules provided by the environment model and the event model.

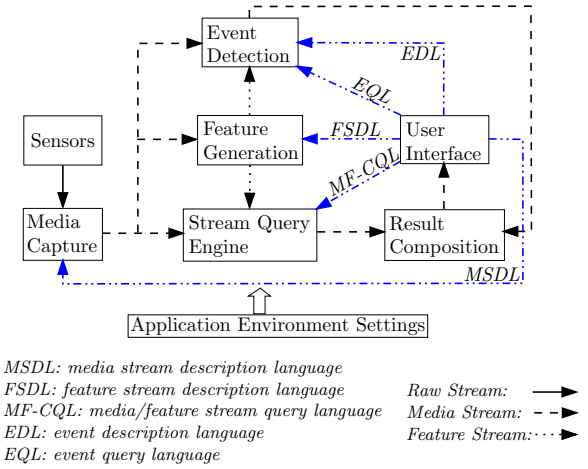

Figure 4: MedSMan system architecture.

4. QUERY PROCESSING

4.1 Media Stream Generation

The raw streams captured from sensors are bit streams without explicit media semantics. To correctly retrieve and interpret them, the details of *registering data* (such as media format, coding and size) must be known [15]. Particularly, de-multiplexing, decompression, or decryption procedures may be applied to retrieve individual media tracks from a raw stream. The heterogeneous nature (such as continuity, media type and data rate) of media streams greatly increases the processing complexity. However, some general principles exist on how to create various media streams from sensors and separate them into sequences of media tuples.

The *media stream generation model* $\mathcal{M}$ consists of a sensor type S_T, a sensor source S_S, a media tuple definition T_M, a set of sensor-dependent initialization parameters $\bar{P}$, and a media stream type M_T. S_T plays a significant role in determining other parameters.

A *Media Stream Description Language* (MSDL) is designed for defining various media tuples. Based on a specific T_M, a media stream instance conforming to particular S_T and $\bar{P}$ can be declared and initialized. For example, a video tuple definition and a video stream instance captured from a local port (vfw://) can be declared as:

```
create type frame { integer frame_num primary key,
        time frame_st, time frame_et, image content };
create media stream video1 of frame from
    sensortype cam sensorsource vfw://0 datarate 10.0;
```

Since *sqno* (i.e., frame_num in this example) works as a unique identifer for each tuple in a stream, it is declared as a primary key. Furthermore, most query operators require the explicit timestamp attributes t_s (i.e., frame_st) and t_e (i.e., frame_et). The sensorsource (S_S) defines a local or remote URL connecting the capturing sensor. The sensortype (S_T) cam indicates that the output video stream is a discrete media stream, and requires a datarate parameter denoting the

data rate of a video, which means each video frame spans an extent of $100ms$ on average. In contrast, an audio tuple definition and an audio instance captured from a local port (dsound://) can be declared as:

```
create type audioclip { integer clip_num primary key,
        time clip_st, time clip_et, audiobuffer clip };
create media stream audio2 of audioclip from
  sensortype mic sensorsource dsound:// capturebuffersize 40;
```

Different from cam, the mic (S_T) indicates a continuous audio stream, and requires a capture buffer size (e.g., $40ms$) to cut the stream into a sequence of audioclips.

4.2 Feature Generation

A feature stream is produced by one or more transformer functions operating on media streams or other feature streams. A feature tuple value is generated from its deriving media or feature tuple(s) by one or multiple specific FGFs, with different generation costs. A feature stream has tuples with complex-valued attributes, and a reference to the media/feature streams from which it is derived.

A *feature stream generation model* $\mathcal{F}$ consists of a feature tuple definition T_F, a set of deriving media or feature streams $\bar{D_S}$, a set of feature generation functions $\bar{F_G}$, and a set of optional parameters $\bar{P_F}$ controlling the data rate, delay, etc. We have designed a *Feature Stream Description Language* (FSDL) to create feature tuple definitions. Based on them, feature stream instances can be declared from deriving $\bar{D_S}$ by using particular $\bar{P_F}$.

The feature generation functions are implemented as interface between the FSDL and the underlying digital signal processing techniques. Thus, the FSDL is open for users to implement their own FGFs.

4.3 The MF-CQL query language

We have designed a query language, MF-CQL, which is CQL [3] extended with additional syntax and shortcuts to express the extended semantics beyond DSMS. MF-CQL allows flexible expressions for period detections and period aggregations, which deliver higher levels of expressive power for media query applications. Currently, MF-CQL is a stream-only language. Considering the real-time requirement of most media stream applications, MedSMan's operations are triggered by individual (media or feature) tuple per time. For most media streams queries, query predicates are based on their derived feature streams. Then the qualifying fragments of the feature streams are mapped back to their deriving media stream fragments by using *sqno* or interval attribute. We designed syntax for efficiently expressing these queries. An abstract syntax of MF-CQL is:

```
select <Media Stream Attr List>
from  <Media Stream List>, <Feature Stream List>
[with  <Period Variable List>]
[when <Period Detection Operation>];
[where<Condition>];
```

The select-list specifies the target media or feature attributes to be output. The where-condition clause uses normal SQL expressions to specify predicates over feature or media attributes. The involved feature streams and deriving media streams must both appear in the from-clause. The when and with clauses are designed for computing dynamic periods which are determined by other subqueries at run-time.

Region	Sensor	Type	Effect	Landmarks
front	C_P	video	covered	podium
speaker	C_S	video	covered	speaker
speaker	M_S	audio	loud	speaker
left-side	M_1	audio	loud	chair1..chair3
right-side	C_1	video	covered	chair7..chair9
front	$Projector1$	video	covered	slides
...	...	...	...	...

Table 4: A coverage map for a seminar application

4.4 Detecting events from streams

The process of translating event expressions and operations to stream queries and operations we propose at the analysis level involves three steps:

- The selection of the candidate components for the translation. This means the selection of entities (expression and operation) which are extracted from the event expressions, and which correspond with stream entities.
- The choice of the translation rules allowing to go from the event level to the stream level, according to the candidate components found in the first step.
- The translation from the basic event structures to the suitable stream structures, preserving the original functionalities.

The set of the formal translation rules are composed of automatic and semi-automatic rules which are provided with both application-independent system knowledge and the application domain knowledge given by a system designer. One important point is to avoid losing information during the translation, which is not possible with an automatic translation without using any application-dependent knowledge provided by the environment model and the event model.

If a query is posed against the observable events that are atomic and at the leaf levels of our event hierarchy, it is computed by directly querying the relying streams. If a query is posed against the higher level events, it is decomposed to an equivalent query against atomic events. For a complex query the system will also need to optimize the evaluation of multiple atomic queries over the same set of media and feature streams. The decomposition algorithm and the optimization issues are beyond the scope of this paper and will be parts of our future work.

This paper discusses how single atomic event queries over one or more streams can be answered by two query examples in our seminar application. Table 4 describes the coverage map of sensor placement in our seminar application.

Example1 Query 4 requires to detect a composite event E_1 ("Speaker begins to walk and then stops") which consists of two sequential subevents as E_{11} ("Speaker begins to walk") and E_{12} ("Speaker stops"). E_1 can be expressed as E_{11} $occures\_before$ E_{12}. According to some translation rules created by using Table 4, the system knows that camera C_S covers the speaker's movement. The atomic events E_{11} and E_{12} can be directly detected using a movement feature stream, which is derived from C_S's output video stream (e.g., video1 defined in Section 4.1). Using FSDL, the movement feature stream is defined as:

```
create type mvFeature { integer mv_sn primary key,
```

```
        time mv_st, time mv_et, integer mv_pixel };
create feature stream mvFStream1 of mvFeature on video1
with   mv_sn:=getFrameNum(frame_num)
       mv_st:=getFrameTime(frame_st)
       mv_et:=getFrameTime(frame_et)
       mv_pixel:=getMovementNum(content);
```

On the other hand, the event operation "occurs_before" requires our system to set up a temporal structure in MF-CQL expression indicating the detection of E_{11} is followed by the detection of E_{12}. We need to know the exact *start* indicating E_{11}'s occurrence and *end* indicating E_{12}'s occurrence, and guarantee that *start* occurs before *end*, in order to detect each E_1's occurrence, which is a period event. When translating such a complex event expression into MF-CQL, both *start* and *end* are expressed and computed by two different subqueries using period variables supported by the with-clause in MF-CQL. For example, Query 4 can be expressed as:

```
select content from video1, mvFStream1
with   t1 as getFirstOccur(mv_pixel>8000) and
       t2 as getFirstOccur(mv_pixel<100)
where frame_st > t1-1m and frame_et < t2 and t1 < t2;
```

Note the translation from events to the concrete feature values needs the knowledge of the environment from the system administrator rather than the user. The administrator may manually set up a set of rules for the translation.

A getFirstOccur operator is designed to detect both the start and end of a period, which takes a predicate (typically in terms of feature values) as input, and returns the time of "first" qualifying tuple as the period endpoint. If the events of interest repeat, our algorithm allows resetting M_PScan to retrieve media tuples in future qualified periods. Figure 5 indicates qualified event periods denoted as P, P' and so forth. Although a single feature stream is used to define period variables in this example, the predicates in with-clause can use different features from different media streams.

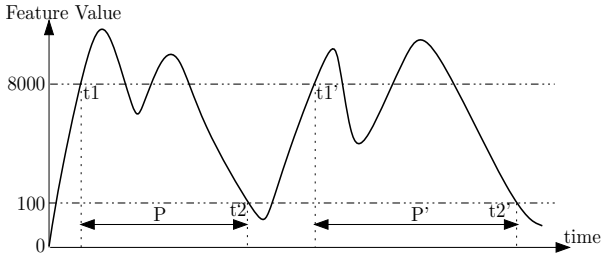

Figure 5: Periods determined by feature detection.

Example2 Query 3 requires to detect an interval-based event E_2 ("Speaker speaks loudly (i.e., greater than some threshold) for a while"). According to the coverage map, the microphone M_S captures the speaker's voice; thus E_2 can be detected using a soundEnergy feature stream extracted from M_S's output audio (e.g., audio2 in Section 4.1), which is defined as:

```
create type sdFeature { integer sd_sn primary key,
        time sd_st, time sd_et, double sd_energy };
create feature stream sdFStream3 of sdFeature on audio2
with   sd_sn:=getFrameNum(clip_num)
       sd_st:=getFrameTime(clip_st)
       sd_et:=getFrameTime(clip_et)
       sd_energy:=getSoundEnergy(clip);
```

However, this feature can only detect whether the sound energy for a single pre-defined clip interval ($40ms$) is greater than the threshold but can not guarantee the continuity for all clips during a qualified event period. Therefore, we design a period operation CONTINUE and use the following predicates to express this query in MF-CQL:

```
select clip, AVG(sd_energy)
from  audio2, sdFStream3
when  CONTINUE(sd_energy > 27.0) > 10;
```

The inner predicate evaluates if each individual tuple qualifies, while the outer predicate must guarantee both *length* and *continuity* of an event period (or sequence) of tuples qualifying to inner predicate. Obviously, the order and continuity between adjacent tuples must be considered. We also use the environment model knowledge to convert user specified time interval into the actual number of clips (e.g., 10) to be counted.

5. IMPLEMENTATION

The prototype of MedSMan is implemented using Java (JDK 1.5.0). We use APIs provided by Java Media Framework (JMF2.1.1) and OpenCV (integrated with the Java based query engine via Java Native Interface (JNI)) for real-time audio/video capturing and feature generating. The stream description and query languages are implemented using Java Compiler Compiler (JavaCC).

Our implementation consists of two major components as media/feature stream generation [11] and stream querying execution. The former permits a designer to directly capture various live data streams from different sensor devices, and form media streams consisting of logical media tuples. Then, more meaningful feature streams can be automatically derived from media streams for the query purpose.

Stream level queries are parsed and generate physical query plans. MedSMan runs physical plans in an individual tuple triggering approach for media and feature stream query execution, thus reduces query delays. Either new arriving media tuple or feature tuple triggers the entire physical plan to run. Figure 6 shows the architecture of MF-stream query execution. Each input *MF-CQL statement* is parsed into a *query*, which then creates a logical query *plan*. Based on both query and plan, a *QueryDesc* is generated, which starts a *QueryDesc Manager* and registers itself into the involving *Media Stream Desc* or *Feature Stream Desc*, depending on the triggering stream. Typically, M_Scan and F_Scan are the bottom triggering nodes. Each *QueryDesc Manager* works as an independent thread, and continually triggers its plan execution when driven by new arriving tuples from registering media or feature queue managers. Each queue manager can concurrently drive multiple queries. Inside each query plan, the nodes of each subtree are executed in a bottom-up, left-to-right manner, except for join operators (e.g.,F_BJoin1M or F_BJoin2M) – they must execute in a symmetric manner, i.e., every tuple in each join stream will trigger the join operation.

6. EXPERIMENTS AND ANALYSIS

Experimental Set-up We run a number of query examples varying in media streams and feature streams. Our experiments run on a XP machine with dual $2.4GHz$ CPUs and $2GB$ RAM. The experiments use the instances of video

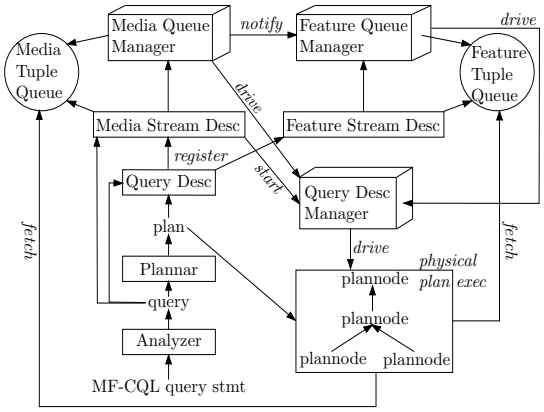

Figure 6: Stream query execution.

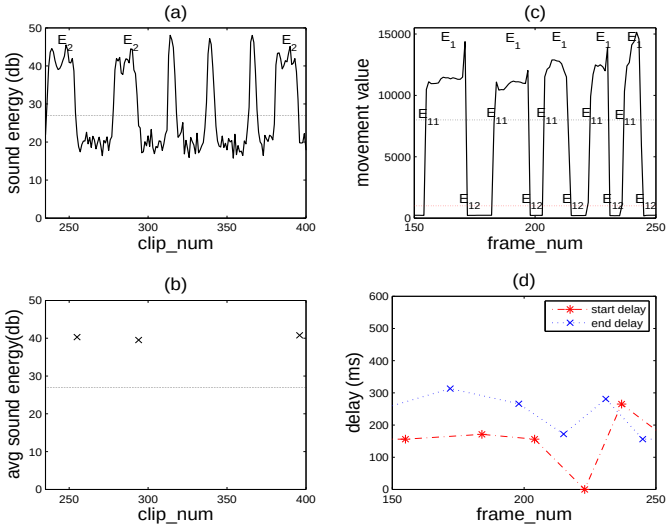

Figure 7: Period queries.

(with the format of RGB, 400x320, 10fps, Length 24-bit), audio (with the format of LINEAR, 4410 bit,Stereo, 2Channels) and the derived feature streams in previous sections.

We evaluate performances for different query types in terms of query delays. For one satisfying tuple in a media stream, its query delay is defined as the time difference between when it enters the system and when it leaves the topmost operator. Query delay is both media stream dependent and individual tuple dependent, because (1) different media tuples have different tuple extents and *feature computation delays* (FCD); (2) a particular media tuple may be joined with multiple tuples from other streams; thus (3) different tuples in one stream joined with a common tuple in another stream may have different delays.

For Example2, Figure 7 (a) shows three detected E_2 occurrences with qualifying interval of 19, 16 and 15, respectively. The other three detected periods are not qualified events since their interval is less than 10 clips. Figure 7 (b) shows the average sound energy feature values for the three detected event occurrences, respectively.

For Example1, Figure 7 (c) shows the atomic event occurrence pairs of E_{11} and E_{12}, each of which indicates the start and end for one composite E_1 occurrence. Figure 7 (d) shows the E_1's start (i.e., E_{11}) detection delay and end (i.e., E_{11}) detection delay, respectively.

One advantage of our system is that queries can be easily composed and evaluated against multiple features derived from single or multiple media streams.

Example3 Suppose we want to query an event like "A person raises a book". This query can use a face detection feature and a color-based object detection feature (by passing the image of the book cover as a reference (ref_image) into the query) extracted from each frame (content) of a common video (video1), which are defined as:

```
create type fdFeature { integer fd_sn primary key,
            time fd_st, time fd_et, integer fd_num };
create feature stream fdFStream3 of fdFeature on video1
with   fd_sn:=getFrameNum(frame_num)
       fd_st:=getFrameTime(frame_st)
       fd_et:=getFrameTime(frame_et)
       fd_num:=getFaceDetectNum(content);
create type odFeature { integer od_sn primary key,
            time od_st, time od_et, integer od_pixel };
create feature stream odFStream2 of odFeature on video1
with   od_sn:=getFrameNum(frame_num)
       od_st:=getFrameTime(frame_st)
       od_et:=getFrameTime(frame_et)
       od_pixel:=getObjDetectNum(content, ref_image);
```

Figure 8 (a) and (b) shows the face detection result and the book reference image, respectively. Then, this event query can be expressed using both features as:

```
SELECT content FROM video1, fdFStream3, odFStream2
WHERE fd_num=1 AND od_pixel>1000;
```

(a) (b) (c)

Figure 8: Querying a person raising a book.

Figure 8 (c) shows a detected event occurrence. Figure 9 (a) shows the total query delay (with average of 740.6ms), frame interval (with average of 261.5ms) and two FCDs (with averages of 89.9ms for object detection and 259.3ms for face detection) for each frame. Since the face detection feature delay costs more than the specified frame interval (i.e., 100ms), the video capturing thread is delayed and the actual frame rate is slowed down. We can reduce the feature delay impaction by skipping some feature generations at the cost of losing some query accuracy.

Example4 A more complex example is to use features derived from different media streams. Suppose we want to query "A person walks and talks". This event consists of two subevent as "a person walks", which can be detected using the movement feature from video1, and "a person speaks", which can be detected using the sound energy feature from audio2. Then, we can compose this query as:

```
SELECT content FROM video1,mvFStream1,audio2,sdFStream3
WHERE mv_pixel>8000 AND sd_energy>27.0;
```

Query delays for each video frame and each audio clip are shown in Figure 9 (b). Note each video frame overlaps with multiple sequential audio clips. The average delay of the video frames is $393.1ms$, and the average *max* delay of the audio clips is $622.5ms$ for the oldest one waiting in queue, since an optimization is implemented by making a slower feature (mv) tuple trigger the faster feature (sd) tuples waiting in queue in order to reduce an extra *Scan* operator and queue buffers.

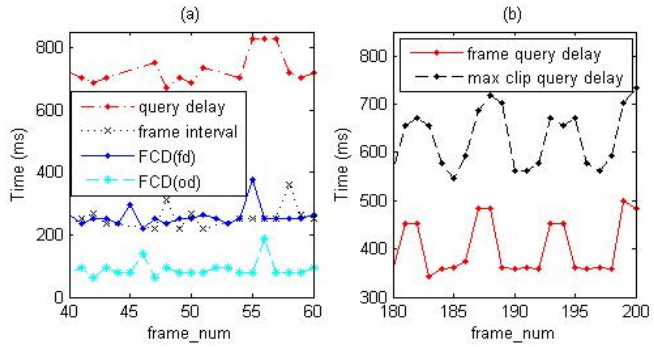

Figure 9: Query delays.

Obviously, FCDs play a significant role in determining dynamic tuple intervals, total query delays and the actual capturing rates for media streams. Another observation from experiments is that we need some "massaging" mechanism for presenting the media query results in order to reduce the jitter for better human perception.

7. CONCLUSIONS AND FUTURE WORK

This paper presents our approach to dealing with continuous querying over live heterogeneous media streams by effectively combining extendible digital processing techniques with a general media stream management system. To bridge the two different areas, a unifying data model for media and feature streams is designed, along with description languages managing media and feature streams and a query language manipulating complex queries. Moreover, an event-based query approach can provide a more feasible way for users to access the related media stream content of interest at specific levels and give users a meaningful, more refined view of the world around them, with the domain knowledge provided by an environment model. Due to the increasing use of media data in many emerging applications, we believe this is an area of significant interest to researchers in stream data as well as multimedia data. In the near future, we will continue to work on the hierarchical event model to enable users to directly compose event-based queries. This event model should take into account the temporal, spatial and logical relationships inside an event or between events; thus a formal event query language is necessary for expressing events and event-based queries, which will be translated into the lower-level stream operations. On the other hand, we will investigate how to efficiently serve multiple queries over incoming streams by utilizing common elements of different queries for optimization.

8. REFERENCES

[1] D. Abadi, D. Carney, U. Cetintemel, and et al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, August 2003.

[2] J. Amores, N. Sebe, P. Radeva, T. Gevers, and A. Smeulders. Boosting contextual information in content-based image retrieval. In *Proceedings of the 6th ACM SIGMM international workshop on Multimedia information retrieval*, pages 31–38, 2004.

[3] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. Technical report, Stanford University, Oct. 2003.

[4] W. G. Aref, A. C. Catlin, and et al. Vdbms: A testbed facility for research in video database benchmarking. *ACM Multimedia Systems Journal, Special Issue on Multimedia Document Management Systems*, 9(6):575–585, June 2004.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Symposium on Principles of Database Systems*, pages 1–16, Madison,Wisconsin, May 2002.

[6] A. F. Cardenas and P. A. Michael. Image Stack Stream Model of Multimedia Data. September 2002.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: a scalable continuous query system for internet databases. In *International Conference on Management of Data*, pages 379 – 390, Dallas, Texas, May 2000.

[8] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: The qbic system. *IEEE Computer*, 28(9):23–32, Sept. 1995.

[9] L. Golab and M. T. Ozsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, June 2003.

[10] R. Jain. Experiential computing. *Communications of the ACM*, 46(7):48–55, July 2003.

[11] B. Liu, A. Gupta, and R. Jain. A live multimedia stream querying system. In *Proceedings of the 2nd international workshop on Computer Vision Meets Databases*, pages 35–42, June 2005.

[12] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):610–628, 1999.

[13] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continously adaptive continous queries over streams. In *ACM SIGMOD International Conference on Management of Data*, Madison,Wisconsin, 2002.

[14] A. W. Smeulders, M. Worring, S. Santini, A. Gupta, and R. Jain. Content-based image retrieval at the end of the early years. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(12):1349–1380, December 2000.

[15] R. Steinmetz and K. Nahrstedt. *Multimedia computing, communications and applications*. Prentice Hall, 1995.

[16] A. U. Tansel, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1993.