Applying Formal Methods to Modeling and Analysis of Real-time Data Streams

Krasimira Kapitanova
University of Virginia, Charlottesville, VA, USA, krasi@virginia.edu

Yuan Wei
Zeno Software Inc., San Francisco, CA, USA, eric.wei@zenosoftware.com

Woochul Kang
Electronics and Telecommunication Research Institute, Daejeon, Korea
wchkang@etri.re.kr

Sang H. Son
University of Virginia, Charlottesville, VA, USA, son@virginia.edu

Received 11 February 2011; Revised 3 March 2011; Accepted 3 March 2011

Achieving situation awareness is especially challenging for real-time data stream applications because they i) operate on continuous unbounded streams of data, and ii) have inherent real-time requirements. In this paper we showed how formal data stream modeling and analysis can be used to better understand stream behavior, evaluate query costs, and improve application performance. We used MEDAL, a formal specification language based on Petri nets, to model the data stream queries and the quality-of-service management mechanisms of RT-STREAM, a prototype system for data stream management. MEDAL’s ability to combine query logic and data admission control in one model allows us to design a single comprehensive model of the system. This model can be used to perform a large set of analyses to help improve the application’s performance and quality of service.

Categories and Subject Descriptors: C.3 [Special-Purpose and Application-Based Systems]: Real-time and Embedded Systems; I.2 [Artificial Intelligence]: Knowledge Representation Formalisms and Methods

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Data Stream Analysis; Petri Nets; Operator Selectivity Estimation; Stream Query Modeling; Quality-of-Service Management

1. INTRODUCTION

Situation awareness (SA) has been recognized as a critical foundation for successful decision-making across a broad range of complex real-time applications, including aviation and air traffic control [Nullmeyer et al. 2005], emergency response [Blandford...
and Wong, 2004], and military command and control operations [Gorman et al. 2006]. SA is especially important for applications where the information flow can be high and poor decisions may lead to serious consequences (e.g., piloting a plane or treating critically injured patients). These applications need to operate on continuous unbounded streams of data to understand how incoming information and events could impact the system, both now and in the near future. The streaming data may come from various sources such as sensor readings, router traffic traces, or telephone records. Therefore, the capability to manage data streams has become an essential application requirement.

These applications also have inherent real-time requirements, and queries on the streaming data should be finished within their respective deadlines. Consider a surveillance system as an example. The system is expected to detect if a target enters the monitored area and alert the controlling party (e.g., human operators). If the detection does not occur within a certain deadline, the target may be missed. However, due to the dynamic nature of the input data streams, the stream queries may have unpredictable execution costs. First, the arrival rate of the data streams can be volatile, which leads to variable input volumes to the queries. Second, the content of the data streams may vary with time, which causes the selectivity of the query operators to change over time. The variable stream behavior and the irregular workload pattern make maintaining the desired level of quality-of-service (QoS) a challenging task.

Formal data stream analysis can help tremendously in achieving the desired levels of QoS for real-time data stream applications. Such analysis can provide query designers with a better understanding of the behavior of the real-time data streams they are working with. It could also be used to more accurately predict changes in the data arrival patterns and the query workloads. In addition, analysis of the data admission controller, responsible for determining the amount of input stream data to the queries, could aid the design of better controllers that can adapt faster to workload fluctuations. To perform such an analysis, we needed a specification language that would allow us to formally model real-time data streams, stream queries, and data admission control mechanisms. Such a language, however, was missing. To address this, we proposed to use MEDAL - a formal specification language, which in its nature is an enhanced Petri net. MEDAL can capture the structural, spatial, and temporal characteristics of a complex distributed real-time system, which makes it suitable for modeling and analysis of data streams.

In this paper we describe how MEDAL was used to model and analyze real-time data stream applications. First, each query plan could be specified with a MEDAL model. Second, MEDAL is also capable of modeling the data admission controller. MEDAL’s ability to model both the queries and the data admission control enabled us to combine the two main components of data stream applications - query logic and control - into a single comprehensive system model. Third, using the MEDAL query models to analyze different system properties such as the behavior of the data streams, the cost and selectivity of different query operators, and the real-time properties of the queries, can significantly improve the QoS of the applications. Further, MEDAL-aided analysis could also help admission control designers build better and more accurate prediction-based controllers.
There are three main contributions of this paper: 1) adapting MEDAL to model real-time data stream queries and stream management mechanisms, 2) integrating query logic and data control models into a comprehensive data stream application model, 3) introducing how MEDAL can be used for various types of analyses, such as query plan optimization and application timeliness.

The rest of this paper is organized as follows: We discuss related work in Section 2. Section 3 gives a short overview of MEDAL and some of its properties. Section 4 briefly introduces the components of our system model, namely, periodic query model, query plans, and the data admission controller. Section 5 describes how operator cost and selectivity predictions are used to improve the QoS of the system. Section 6 describes how MEDAL is applied to model stream queries and data admission. We demonstrate some of the types of analyses that MEDAL is used for in Section 7. Section 8 shows the MEDAL models of different data stream management configurations. We also evaluated how these configurations affect the QoS of a prototype data stream management system, RT-STREAM. Section 9 concludes the paper.

2. RELATED WORK

2.1 Query Optimization

There have been significant research efforts devoted to the query optimization problem in distributed Data Stream Management Systems (DSMSs) [Olston et al. 2003; Liu et al. 2005]. Adaptive filters have been proposed to regulate the data stream rate while still guaranteeing adequate answer precisions [Olston et al. 2003]. Liu et al. [2005] discuss the challenges for distributed DSMSs and propose dynamic adaptation techniques to alleviate the uneven workloads in distributed environments. Tu et al. [2006] use system identification and rigorous controller analysis to improve the benefits of input load shedding. Wei et al. [2007] apply just-in-time sampling to estimate the output size of query operators. The estimation results are used to control the intermediate query result propagation strategy.

2.2 Selectivity Estimation

Selectivity estimation has been a fundamental problem in the database community since query optimizers use the estimation results to determine the most efficient query plans. Sampling [Haas et al. 1994], histograms [Poosala and Ioannidis, 1997], index trees [Comer, 1979], and discrete wavelet transforms [Press et al. 1992] are the most widely used selectivity estimation methods. Sampling has been used extensively in traditional database management systems (DBMSs) [Haas et al. 1994; Barbara et al. 1997; Chaudhuri et al. 2001]. It gives a more accurate estimation than parametric and curve fitting methods used in DBMSs and provides a good estimation for a wide range of data types [Barbara et al. 1997]. In addition, since sampling-based approaches, as opposed to histogram-based approaches, do not require the maintenance of an auxiliary data structure, they do not incur the overhead of constantly updating and maintaining that data structure. This is especially important in the context of data streams as the input rate of the streams is constantly changing. Wei et al. were the first to consider a sampling-based approach to evaluate the data stream query workloads.
and use the results to manage the query QoS [Wei et al. 2007]. In this paper we employed this sampling technique and used it as part of our prediction-based QoS management mechanism.

2.3 Modeling Stream Queries

The majority of work on data streams uses SQL or SQL-like semantics to define stream queries [Wei et al. 2006; Babcock et al. 2002]. Babcock et al. use standard SQL to model stream queries and extend the expressiveness of the language to allow the specification of sliding windows [Babcock et al. 2002]. However, SQL fails to express a number of essential characteristics specific to processing data streams, such as:

1) **Data dependency and correlation among different streams in heterogeneous systems;**
2) **Collaborative decision making:** Data stream systems are distributed, concurrent, and asynchronous. Therefore, detecting events usually requires spatial and temporal composition of ad-hoc readings;
3) **Modeling probabilities:** Individual data readings are often unreliable which results in non-deterministic decision making. In order to tolerate such non-determinism, queries might need to be defined using a probabilistic model. SQL has no explicit support in this regard;
4) **Graphical model:** SQL does not have natural graphical support.

Another method for modeling stream queries was proposed for the Aurora system [Abadi et al. 2003]. Aurora uses a graphical boxes-and-arrows interface for specifying data flow through the system. Compared to a declarative query language, this interface is more intuitive and gives the user more control over the exact series of steps by which the query answer is obtained. However, this approach lacks the ability to model probabilities, data dependencies and correlation, and collaborative decisions.

In this paper we used a formal specification language, called MEDAL [Kapitanova and Son, 2009], to model stream queries and data admission control. MEDAL combines features from Stochastic, Timed, and Colored Petri nets, which allows it to model system properties such as collaborative decision-making, temporal and spatial dependencies, heterogeneity, and non-determinism. These modeling capabilities render MEDAL very suitable for modeling stream query plans.

3. MEDAL

In this section we give a short overview of the specification language MEDAL and some of its properties. The MEDAL description of a real-time application is a 7-tuple structure \( F = (P, T, A, \lambda, \beta, H, L) \), where:

1) \( P \) is the set of all places. Places can represent application state or data, and are depicted as circles in a MEDAL model. \( P = S \cup E \), where \( S \) represents the data input places (\( T, L \), and \( A \) in Figure 1), and \( E \) represents the places for higher level events (place \( E \) in Figure 1).
2) \( T \) is the set of all transitions. Transitions model various types of actions or operators and are represented by rectangle bars (transitions \( T1, T2, T3, \) and \( T4 \) in Figure 1).
3) $A$ is the set of arcs. Arcs in MEDAL represent the flow of logic/control and are depicted as directed arrows.

4) $\lambda$ is the probability/weight function for the arcs and $\lambda : A \rightarrow [0, 1]$. With $\lambda$, MEDAL adopts features from Timed and Stochastic Petri nets which allow it to model probabilistic problems.

5) $\beta$ is the time guard function, $\beta : T \rightarrow \bigcup (r_1, r_2)$, where $r_1 \leq r_2 \in \mathbb{R}$. For a transition $T$, $\beta(T) = (\alpha_1, \alpha_2) \bigcup (\alpha_3, \alpha_4)$ indicates that this transition can only fire during the union closure of the given ranges. $\beta$ also acts as a persistency guard, i.e. we can use it to specify the amount of time a place can hold a token before this token becomes invalid.

6) $H$ is the threshold function for places and is defined as $H : P \rightarrow \mathbb{R}$. For example, $H(p) = c$ means that a token can enter a specific place $p$ only if its capacity is equal or higher than $c$.

7) $L$ is the spatial guard function for transitions, $L : T \rightarrow \mathbb{R}^+$. It is used to guarantee that the incoming data has been generated within the radius of interest.

Figure 1 shows the MEDAL model of an example explosion event detection system. An explosion is characterized by specific temperature, light, and sound values. Therefore, our detection application takes input data streams from three types of sensors - temperature, light, and acoustic - and uses the readings to determine if an explosion has occurred. In Figure 1, the temperature, light, and sound input data streams are represented with places $T$, $L$, and $A$, respectively. When the values from all three sensors exceed some predefined thresholds, transition $T_4$ fires and the application reports the detection of an explosion event, represented by Place $E$. The stream data processing must happen in real-time since an explosion needs to be detected as soon as possible in order to evacuate the dangerous area.

Tokens are abstract representations of data or occurrences of events. In MEDAL, a token is defined as:

$$Token = \{Type \ tp; \ Capacity \ c; \ Time \ t; \ Location \ l\},$$

where the type of a token indicates the type of data or event represented by the token. The capacity of a token is its value. It can either contain actual stream data or indicate the confidence that a particular event has occurred. Due to the high volume
of data, it is often assumed that it is not possible to store a stream in its entirety, nor is it feasible to query the whole stream history. Typically, the queries are executed on a window of data. A window on a data stream is a segment of the data stream that is considered for the current query. Therefore, to more accurately model the query execution behavior, rather than storing a single stream value, a token’s capacity contains a whole window of data. Time indicates when the token was created. The location represents where the data carried by the token was generated. This token representation allows us to encapsulate key stream data aspects into tokens.

4. SYSTEM MODEL
A data stream is defined as a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamps) sequence of data [Golab and Özsu, 2003]. A DSMS is a system especially constructed to process persistent queries on dynamic data streams. DSMSs are different from traditional DBMSs in that DBMSs expect the data to be persistent in the system and the queries to be dynamic, whereas DSMSs expect dynamic unbounded data streams and persistent queries. Emerging applications such as sensor networks, emergency response systems, and intelligent traffic management, have brought research related to data streams into focus. These applications inherently generate data streams, and DSMSs are well suited for managing the produced data.

4.1 Periodic Query Model
So far, DSMS research has mainly been focused on using a continuous query model [Carney et al. 2002; Abadi et al. 2003; Abadi et al. 2005; Girod et al. 2007]. In a continuous query model, long-running continuous queries are present in the system and new data tuples trigger the query instances. These incoming data tuples are then processed and the corresponding query results are updated. The continuous model performs well when the system workload is stable and the system has sufficient resources to finish all triggered query instances. However, since the number of query instances and the system workload depend directly on the input, which could be extremely volatile, this continuous query model is not appropriate for real-time applications that need predictable responses. Another drawback of this model is that, since the query execution is driven by the data rate of the system, the application does not have control over the frequency and the deadlines of the queries. For applications where some queries are more important than others, it might be desirable to have the ability to execute the important queries more often. This, however, is hard to accomplish in a continuous query model.

To address this, we have developed a periodic query model (PQuery) for data stream queries with timing constraints [Wei et al. 2006]. In this periodic query model, every query has an associated period. Upon initialization, a query instance takes a snapshot of the data streams at its inputs. The query input does not change throughout the course of the query instance execution, even when there are new data tuples arriving over the data streams. Instead, the newly arrived data tuples are processed by the next query instance. In this way, query execution is not interrupted or aborted by new incoming data. When an application receives the results of a periodic query instance, it is with the understanding that these results reflect the state of the system when
the query instance was initiated. In the periodic query model, the application can specify the query periods and deadlines. These parameters can be used to calculate the number of queries in the system at any given time, which allows us to estimate the query workloads much easier than with the continuous query model.

4.2 Query Plan and Query Execution

A DSMS contains long-running and persistent queries. When a query arrives in the system, it is registered and is triggered periodically based on its specified period. All queries are converted to query plans (containing operators, queues, and synopses) statically before execution. Queues in a query plan store the incoming data streams and the intermediate results between the operators. A synopsis is associated with a specific operator in a query plan, and stores the accessory data structures needed for the evaluation of the operator. For example, a JOIN operator may have a synopsis that contains a HASH JOIN index for each of its inputs. When the JOIN operator is executed, these hash indices are probed to generate the JOIN results.

Consider the following example scenario where a traffic monitoring system has been deployed to constantly analyze the traffic in a particular city and determine the most suitable times for delivering supplies to grocery stores. To achieve the desired situation awareness, the system analyzes data streams from speed and traffic sensors. We perceive events as the fundamental building blocks of a situation. Therefore, achieving situation awareness requires that we identify a particular set of events and perform our situation assessment and analysis based on their occurrence. One type of event that our traffic monitoring application is interested in is trucks that travel during light traffic in specific lanes, such as non-HOV lanes. For its consequent traffic analysis, the application calculates the average speed of these trucks. The SQL data stream and query specifications are given as follows:

**Stream**: Speed (int lane, float value, char[8] type);

**Stream**: Traffic (int lane, char[10] type);

**Relation**: Lanes (int ID, char[10] type, char[20] road);

**Query**: SELECT avg (Speed.value)

FROM Speed [range 10 minutes], Lanes, Traffic [range 10 minutes]

WHERE Speed.lane = Lanes.ID AND Lanes.ID = Traffic.lane AND Speed.type = Truck AND Traffic.type = Light

**Period** 10 seconds

**Deadline** 5 seconds

The query above operates on data streams generated by speed and traffic sensors and calculates the average speed of trucks in particular lanes during light traffic over a 10-minute window. The query needs to be executed every 10 seconds and the deadline is 5 seconds after the release time of every periodic query instance. The query plan that is generated based on this query is shown in Figure 2. It contains three types of
query operators (RANGE WINDOW operator, JOIN operator, and AGGREGATE operator) and two types of queues (one for storing the output of the RANGE WINDOW operators and one for storing the output of the JOIN operators).

After the query plan is generated, the operators are sent to the scheduler to be executed. Depending on the query model (e.g., continuous or periodic), a scheduling algorithm such as round-robin or earliest deadline first could be chosen so that the system requirements are met.

4.3 Data Admission Controller

In many real-time applications, partial results are more desirable than queries missing their deadlines. Therefore, the system might trade off data completeness for better query miss ratios at run time. We have designed an overload protection mechanism called a data admission controller which trades data completeness for better query miss ratios. The basic approach is to reduce the incoming data volume when the system becomes overloaded. The load shedding process is performed before the data stream tuples are processed by the queries. Operators perform data admission using random dropping. Though not covered in this paper, semantic dropping can be used to improve system performance if query semantics are considered. The system also allows the data sources to mark the important data tuples to make sure they get processed. The importance flag is marked by setting the highest bit of the data tuple timestamp. The data tuples with flags are admitted to the system regardless of the current data admission ratios. However, operators maintain the target data admission percentage by dropping more unmarked data tuples.

The data admission process is controlled with a proportional-integral (PI) controller as it is simple to use and provides an acceptable response time to workload fluctuations. A proportional-integral-derivative (PID) controller is not suitable in this situation because of the dramatic changes that might occur in the workloads of query systems.
from one sampling period to another. Adding a derivative control signal amplifies the random fluctuations in the system workloads [Hellerstein et al. 2004].

The data admission control architecture is shown in Figure 3. The query miss ratio (MR) is sampled periodically and compared against the miss ratio threshold specified by the application. The result is used by the PI controller to calculate the data admission control signal $\Delta P_{AC}$. The control signal is derived with the equation

$$\Delta P_{AC} = P_{MR} \times (MR_{ST} - MR_t) + I_{MR} \times (MR_{LT} - MR_t),$$

where $MR_{ST}$ and $MR_{LT}$ are the short-term and long-term query miss ratios sampled in the last sampling period. $MR_t$ is the maximum miss ratio allowed by the application. $P_{MR}$ and $I_{MR}$ are controller parameters that specify the significance of the short-term and long-term query miss ratios when calculating the data admission control signal. The controller parameters determine the behavior of the controller. The process of tuning these parameters, i.e. controller tuning, is not the focus of this paper. Readers are referred to [Lu et al. 2002] and [Lu et al. 2006] for details on the analysis and tuning of controllers.

In order to provide service differentiation, the system uses multiple data admission controllers. Each service class is associated with a designated data admission controller whose parameters have been specifically tuned for that particular service class. The use of multiple controllers is further discussed and evaluated in Section 8.

5. PREDICTION BASED QOS MANAGEMENT

When a system becomes overloaded, more queries start missing their deadlines. If the query miss ratio increases above the application threshold, the application can no longer satisfy its QoS requirements. When this happens, in order to decrease the deadline miss ratio, the data admission controller should adjust the amount of input data sent to the queries. To perform a correct adjustment, the data admission controller needs information about the future workload and the corresponding query execution times. We predicted the query workload using execution time profiling and input data sampling. To estimate the query execution time, we needed three parameters for each query - namely, the input data stream volume, the operator selectivity, and the execution overhead per data tuple for each operator. We made the assumption that the input data volumes for queries that are ready to execute were known. In the rest of this section we describe in more detail the prediction techniques employed by our QoS management mechanism.
5.1 Operator Selectivity Estimation

5.1.1 Operator selectivity. The operator selectivity measures the ratio between the size of the input that is passed to an operator and the size of output data remaining after the operator has been applied. The selectivity of an operator is defined as:

\[ \text{Selectivity} = \frac{\text{Size}_{\text{output}}}{\text{Size}_{\text{input}}} \]

As the operator selectivity varies, the size of the output changes even when the input volume stays the same. Thus, the query execution cost could change significantly when the data stream content changes. While possible to design the system based on the maximum system workloads, this is not practical nor efficient, because the transient system overload does not persist for long periods of time. Therefore, we dynamically estimated and updated the operator selectivity based on changes in the workload.

5.1.2 Selectivity estimation using sampling. We used sampling as the selectivity estimation algorithm since it is well-studied, easy to implement, and yields good estimation when handling high-rate data streams [Barbara et al. 1997]. A sampler query plan is constructed for every query in the system. These sampler query plans are the same as their corresponding real query plans. When a query instance is released to the scheduler, the sampler is executed first with sampled data tuples from the input. The data tuples are randomly selected from the real input according to a predefined sample ratio. The results of the sampler queries are used to estimate the selectivity and execution time of the operators in the real query plans.

5.2 Operator Overhead Prediction

5.2.1 Operator cost estimation. We made the assumption that all data, including incoming data streams and intermediate results, was stored in memory. Therefore, we could estimate the time it took for an operator to process a data tuple, excluding any additional delays caused by fetching data from the disk. Table I shows the average execution time per tuple for different types of operators. The experiments were performed on our real-time data stream management prototype system RT-STREAM. We could see that, with the exception of JOIN operators, the execution cost of most operators was relatively small and fairly predictable. The cost of a STREAM JOIN (a JOIN operator on two streams) varied from 3 µs to 100 µs because the number of data

<table>
<thead>
<tr>
<th>Operator</th>
<th>Average cost (µs)</th>
<th>Depends on selectivity?</th>
<th>Depends on synopsis size?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTION</td>
<td>0.16 - 0.3</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PROJECTION</td>
<td>0.16 - 0.2</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>JOIN</td>
<td>3 - 6</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>STREAM JOIN</td>
<td>3 - 100</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>0.16 - 0.3</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>EXCEPT</td>
<td>0.16 - 0.3</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>GROUP AGGREGATE</td>
<td>0.16 - 0.6</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table I. Operator cost and dependency on selectivity and synopsis size. The operator cost is the time it takes to process a data tuple.
tuples selected from the data streams could vary significantly. Joining one data tuple with thousands of data tuples from another data stream resulted in the high stream join cost we see in the table. Our experimental results showed that the cost of each operator can be estimated since it closely follows the formula obtained from cost analyses. Here, we presented our analyses for selection and join operators. The analyses for other operators are similar.

**Selection operator cost analysis**

The following notations are used for the selection operator $O_{sel}$:

- $n$: input tuple volume,
- $s$: the selectivity of the operator,
- $C_p$: execution time to evaluate the predicates,
- $C_i$: execution time to insert the output tuple to buffer.

For the selection operator $O_{sel}$:

- Number of output tuples $= n \times s$
- Time to evaluate all input tuples $= n \times C_p$
- Time to insert the output tuples $= n \times s \times C_i$
- Total time $= n \times C_p + n \times s \times C_i$
- Average cost per data tuple $= \frac{n \times C_p + n \times s \times C_i}{n} = C_p + s \times C_i$

The costs $C_p$ and $C_i$ are expected to be constant for a particular set of predicates. Our experiments showed that when the selectivity is fixed to 0.2, the average cost of the selection operator ranged between 0.17 µs to 0.18 µs when the average input size is larger than 400 tuples per second. When the average input size is lower than that, the average cost can be as high as 0.2 µs per tuple due to the overhead of operator context switch. When the selectivity of an operator varies and the input volume is kept constant, the average cost per tuple is expected to increase linearly to the selectivity and the slope of the line corresponds to $C_i$. The experiments we performed confirmed this analysis.

**Join operator cost analysis**

For join operators, our system used symmetric hash join [Wilschut and Apers 1991; Hong and Stonebraker 1993]. The symmetric hash join keeps a hash table for each input in memory. When a tuple arrives, it is inserted in the hash table for its input. It is also used to probe the hash table of the other input. This probing may generate join results which are then inserted in the output buffer. The following notations are used for a join operator $O_{join}$:

- $n_L$: left input volume,
- $n_R$: right input volume,
- $s$: operator selectivity,
- $C_{LProbe}$: execution time to probe the left hash index,
- $C_{RProbe}$: execution time to probe the right hash index,
- $n_LHash$: execution time to hash the left input,
- $n_RHash$: execution time to hash the right input,
- $C_i$: execution time to insert the output tuple to the buffer.
For the join operator \( O_{\text{join}} \):

- Number of output tuples = \( n_L \times n_R \times s \)
- Time to process the left input tuples = \( n_L \times C_{L\text{Probe}} + n_L \times C_{L\text{Hash}} \)
- Time to process the right input tuples = \( n_R \times C_{R\text{Probe}} + n_R \times C_{R\text{Hash}} \)
- Time to insert the output tuples = \( n_L \times n_R \times s \times C_i \)
- Total time = \( n_L \times (C_{R\text{Probe}} + C_{L\text{Hash}}) + n_R \times (C_{R\text{Probe}} + C_{L\text{Hash}}) + n_L \times n_R \times s \times C_i \)

Of the three types of cost factors, hashing cost \( (C_{L\text{Hash}} \text{ and } C_{R\text{Hash}}) \) and insertion cost \( (C_i) \) are much smaller than the probing cost \( (C_{L\text{Probe}} \text{ and } C_{R\text{Probe}}) \) and generally remain constant. The probing cost, however, depends on the contention rate of the HASH JOIN index, which in turn depends on the input data volume and the allocated index size. Our experiments showed that when the hash index size was set to 200 kB, the hash index probing cost was between 4 \( \mu s \) to 6 \( \mu s \) if the input size did not exceed 3000 tuples. When the input size exceeded 3000 tuples, the probing cost increased rapidly due to hash contention. Therefore, if the hash index size was configured appropriately based on the expected number of data tuples in the index, the probing cost remained within a close range.

5.2.2 Maintaining cost constants using profiling. There are a number of constants that participate in the cost analysis of query operators. For example, when estimating the cost of a \( \text{SELECTION} \) operator, one of the necessary inputs is the execution time to evaluate the predicates. The system needs to know the precise values of such cost parameters in order to accurately estimate the execution time of the operators. The approach we used was to keep track of the execution time of each operator and compute these cost parameters periodically. Suppose that after a query instance, the execution time for an operators is \( T_{\text{exec}} \). The new cost parameter \( C_{\text{new}} \) is calculated based on the old cost parameter \( C_{\text{old}} \) and \( T_{\text{exec}} \). We used the single exponential smoothing formula:

\[
C_{\text{new}} = C_{\text{old}} \times (1 - \alpha) + T_{\text{exec}} \times \alpha,
\]

where \( \alpha \) is the smoothing factor and \( 0 < \alpha < 1 \). In order to give relatively higher weights to recent observations, we used values of \( \alpha \) close to 1, which caused a smaller smoothing effect and gave greater weight to recent changes in the data.

6. MODELING QUERIES AND CONTROL

An advantage of M\( E \)\( D \)AL is that it can be used to specify a comprehensive application model by combining the admission controller and the query plan models into a single M\( E \)\( D \)AL model. This was very beneficial for two main reasons. First, it allowed us to model the direct relationship between the data admission controller and the query inputs. Second, integrating the query plan models and the data admission control logic gave us a better understanding of the correlation between the query results and the stream input volume.

6.1 M\( E \)\( D \)AL Query Plans

M\( E \)\( D \)AL can seamlessly be applied to modeling real-time query plans. The same
query plan from Figure 2 is specified in Figure 4 using MEDAL. The different types of query operators are modeled with the help of transitions (transitions $T1$ - $T7$). Application state and data are represented using places. Places with interrupted borders are used to model data input (places $S$, $T$, and $L$). We introduced the use of shaded places (place $L$) in order to distinguish between input relations, which tend to be static for the most part, and input data streams. Therefore, the set of places $P$ in MEDAL can now be expressed as $P = SRE$, where $S$ represents the input streams, $R$ represents the input relations, and $E$ represents the higher level events and intermediate results.

### 6.2 Data Admission Control

There is an array of real-time data stream applications that use feedback control to adjust to the constantly changing environment and improve system performance [He et al. 2003; Lin et al. 2006]. To model feedback control, MEDAL employs a theoretical control approach which adopts the controller synthesis paradigm from control theory. Given a model of the system’s dynamics and a specification of the desired closed-loop behavior, the objective is to synthesize a controller to achieve the specified behavior. This approach relies on the clear distinction between the system and the controller and requires that the information flow between them is explicitly modeled. We met this requirement by designing two separate MEDAL models - one for the query plan and one for the data admission controller. The two models were then composed into a single application MEDAL model which allowed us to analyze the interactions between the system and the data admission controller.

The feedback control loop in Figure 4 is shown in bold. Transition $T8$ compares the query deadline miss ratio to a predefined miss ratio threshold. If the observed miss ratio is higher, controller $C$ calculates the new data admission ratio and propagates it to the system. Place $C$ is a very high-level representation of a controller. A more detailed model of the controller’s logic can be designed with MEDAL and used to
replace Place C. Due to space limitations, we do not discuss the details of modeling the internal controller logic.

MEDAL can be used to model different levels of abstraction. For example, analogously to using Place C to model the entire controller logic, we could abstract away the query plan from Figure 4. The query plan could be modeled with a single place, similarly to how Figure 3 models a DSMS. This is especially helpful during the controller design phase, since it allows us to treat the query as a simple linear system without considering how the internal parts of this system interact. Using this simplified model of the query plan, the appropriate values for the $P_{MR}$ and $I_{MR}$ controller parameters are determined through system identification [Ljung, 1999]. Once the controller is designed, we could expand the query plan model and use it to additionally adjust the controller to better reflect the query properties. The same abstraction approach could also be applied when there are multiple registered queries. The whole query set is abstracted away using a single place, which can consequently be expanded once the controller is designed.

Another requirement introduced by the control theoretic approach is that changes in the system are observable [Lee and Markus 1967]. MEDAL achieves observability by explicitly identifying and modeling the parameters that need to be monitored. In Figure 4, the query results are delivered to controller C through transition $T_8$. As mentioned above, this transition determines the difference between the query deadline miss ratio and the threshold specified by the application. Therefore, transition $T_8$ allows us to explicitly model the query miss ratio and thus achieve the required observability.

The data admission control needs to be implemented at the query level so that different queries can have different data admission ratios even if they share the same data stream source. The two queries in Figure 5 share the same data stream input.

![Figure 5. MEDAL model of the data admission control implementation. This design enables different queries to have different data admission ratios even when they share the same data stream source.](image-url)
S. The incoming data tuples of data stream $S$ are first processed by the stream source operator, represented by transition $T_1$. After the stream source operator, the data stream tuples are in the system. The data admission process for the queries is carried out at the range window operators. In the example shown in Figure 5, range window operators Op 1 and Op 2, represented by transitions $T_2$ and $T_3$, perform the data admission for Query 1 and Query 2 respectively. With this design, different queries can use different data admission ratios.

7. MEDAL ANALYSIS

7.1 Query Optimization Analysis

The MEDAL model of a data stream system could be used to analyze query plans, controller logic, and the interactions between them. This analysis could help designers identify possible query plan optimizations and design more suitable data admission mechanisms. Consider, for example, the following scenario: For the query model in Figure 4 we have estimated the cost of each of the query operators. In addition, based on historical workload information, we have also estimated the average selectivity of the operators. The data admission controller has been designed to decrease the data admission rate by x%, where x is a function of the difference between the miss ratio threshold and the current deadline miss ratio. We can employ the MEDAL model of this application to determine if the controller logic has the desired effect on the deadline miss ratio. Using the available historical data as input to the MEDAL model, we can estimate the new deadline miss ratio, and thus evaluate the controller’s efficiency. This is a very simple high-level example, but it showcases one of the possible MEDAL system analyses.

7.2 Real-time Stream Analysis

MEDAL can also be used for real-time analysis of data stream network applications. For example, a safety requirement for the explosion detection application could be that explosions are detected within 5 seconds of their occurrence. If the application takes longer to detect an explosion and send an alert, there might not be enough time to evacuate. The MEDAL model of the explosion application could be used to help determine if the application is able to meet this requirement. During the design of the

![Figure 6](image-url)

Figure 6. MEDAL can be used to perform real-time analysis of data stream applications.
application, the sampling periods for the temperature, light, and acoustic sensors have been set to 3, 1, and 4 seconds, respectively. In addition, based on knowledge about the system and the environment, the designers have determined that it takes 2 seconds, in the worst case, to process the data, run the explosion detection algorithms, and calculate the result. Since the rest of the transitions in the model represent simple computations, the execution times associated with them could be neglected for this particular application analysis.

Figure 6 shows the time information associated with each transition. Analyzing this model reveals that detecting an explosion could take 6 seconds in the worst case: 4 seconds for sound detection and 2 more seconds for data processing. This, however, means that the application may not be able to meet its real-time requirements. Therefore, either the acoustic sensor sampling period has to be changed, or the system should be altered such that analyzing the incoming data does not take more than 1 second. Similar real-time analysis could be performed for much more complex systems.

8. EVALUATION

In this section we show how MEDAL is used to model different data stream management configurations. We created MEDAL models for the following scenarios:

(1) All queries belong to the same query service class.
(2) There are three query service classes in the system and no data admission control is applied.
(3) There are three query service classes in the system and they share a single data admission controller.
(4) There are three query service classes in the system and each service class has its own designated data admission controller.

We also show the effect of these four configurations on the QoS of the system. We implemented the configurations in RT-STREAM, which is a real-time prototype system for data stream management developed at the University of Virginia [Wei et al. 2006]. RT-STREAM supports the periodic query model and utilizes the prediction based QoS management techniques described in Section 5.

8.1 Experimental Setup

We conducted the performance evaluation using a synthetic workload. The settings for the synthetic workload experiments are shown in Table II. We tested the system performance with both short and long workload bursts. The experiments were carried out on a single machine running Redhat Linux 8.0. The machine was equipped with a 2.8 GHz Pentium 4 hyperthreading processor and 1 Gigabyte DDR 3200 SDRAM memory.

There were 12 streams registered in the system and four queries associated with each of the data streams. The data streams used in the experiments are variable-rate data streams and the average data tuple arrival rate of one data stream is 200 tuples/sec. The arrival of the data tuples conforms to Poisson distribution and the data rate shown is the average arrival rate. The data tuples in the streams are assigned with special values so that the selectivity values are configurable. In the experiments, we
set the \texttt{SELECTION} query selectivity to 0.1, the \texttt{STREAM-TO-STREAM JOIN} selectivity to 0.01, and \texttt{STREAM-TO-RELATION JOIN} selectivity to 0.1. The sampling period of the data admission controller was set to 1 second to give fast response to workload fluctuations. The total run time of one experiment was 300 seconds.

The data streams and relations used in the experiments had the following schema:

\begin{verbatim}
Stream S (ID : integer, value : float, type : char(8));
Relation R (ID : integer);
\end{verbatim}

We conduct experiments on the following four systems:

(1) STREAM: STREAM is a data stream management system developed at Stanford University [Motwani et al. 2003]. STREAM uses a round-robin scheduler and supports continuous queries. The same set of queries without real-time query specifications were executed on the STREAM prototype in order to compare its system performance to that of RT-STREAM.

(2) RT-STREAM: Data admission control is not used in this system.

(3) RT-STREAM-DACS: RT-STREAM with a single data admission controller for the whole system.

(4) RT-STREAM-DACM: RT-STREAM with multiple data admission controllers - one for each query service class.

To evaluate the system, we studied the system performance under normal and extreme workload. As shown in Figure 7, the system workload was configured in such a way that under a normal workload, the system CPU utilization was 80%. To evaluate the system performance under a heavy workload, we created two workload bursts during the experiment. These workload bursts were twice as large as the
normal workload and needed 160% of the CPU processing capacity. As shown in Figure 7, the first workload burst was short. It began at the 60th second and lasted for only 10 seconds. The second workload burst was longer. It started at the 180th second and continued for 60 seconds. We chose such heavy workload fluctuations to test the systems' performance under extreme overload situations. All results are based on more than 10 runs and the 90% confidence intervals are less than 10% of the corresponding data values.

8.2 Single Service Class
The first set of experiments evaluated the system performance when all queries belonged to the same service class. The experiment results are shown in Figure 8 and Figure 9. As we can see in Figure 8, when the system was overloaded, the latencies of continuous queries increase monotonically with time. The maximum latency reaches as high as 23 seconds. The latency of the system continues to grow even after the
burst ends at the 240th second. The reason is that a large number of data tuples are accumulated in the queues and the system needs to process these tuples before it can process new incoming data. Even though the length of the queue decreased after the workload burst, the latency still increased since there were more data tuples per second during the workload burst, and it took more than one second to process the data tuples accumulated during the burst.

In the periodic query model, the latencies of the queries were bound by the specified query deadlines. As shown in Figure 9, similarly to the latency of the continuous queries, the short-term miss ratio of the periodic query continued to increase until the burst went away at the 240th second. After that, the short-term miss ratio began to drop. When data admission control was introduced in RT-STREAM-DAC, the system can respond to the workload fluctuations much faster. In the experiment, the data admission process decreased the system query miss ratio to below 5% within 10 seconds. The process was fairly fast given the magnitude of the workload fluctuations. The data completeness of the system was maintained around 63% during the long workload burst. After the burst, the data completeness was restored to 100% within 10 seconds.

Figure 9. Periodic query miss ratio and data completeness for RT-STREAM. All queries in the system belong to a single service class.

Figure 10. MEDAL model of the RT-STREAM-DAC configuration with one query service class.
The MEDAL model for the RT-STREAM-DAC configuration with one query service class is shown in Figure 10. There is one data admission controller, $C$, which controls the volume of data that enters the system. If the control loop is removed from this MEDAL model, the resulting model is that of the default RT-STREAM configuration with one service class and no admission control.

### 8.3 Differentiated Services

Differentiated services are required by many applications. In case of overload, the system has to guarantee that the most important set of queries get processed. The second set of our experiments tests the service differentiation capability of the system. In this set of experiments, 12 streams and the associated queries are divided into three service classes - Class 0, Class 1, and Class 2 - where the Class 0 queries are the most important. Each service class has 4 data streams and 16 queries.

The results from this experiment are shown in Figure 12, Figure 14, and Figure 16. Figure 11 shows the MEDAL model of RT-STREAM without data admission control. The three query service classes share data stream $S$. Since there is no data admission control, there is no control loop that uses the query miss ratios to estimate the suitable data admission ratios for each query service class. The system performance without data admission control is shown in Figure 12. Without data admission, the miss ratios of Class 1 and Class 2 queries keep increasing until the workload burst disappears at the 240th second.

Figure 14 and Figure 16 show the miss ratio and data completeness of our two data admission schemes. The first one is called RT-STREAM-DAC, which stands for data admission control with a single controller. In this scheme, there is only one data admission controller and all queries in the system share the same data admission controller. The MEDAL model for this scheme is shown in Figure 13. Each service...
class has its own set of queries. As the queries are executed, the controller receives information about the missed deadline ratios for the three classes and determines whether more or less data tuples should be admitted into the system. The decision of the controller is then propagated to the three service classes.

Figure 12. RT-STREAM miss ratio for three different service classes: class 0, class 1, and class 2, in order of query importance.

Figure 13. MEDAL model of the RT-STREAM-DAC admission scheme. There are three query service classes and they share a single data admission controller.
The results for RT-STREAM-DAC are shown in Figure 14. We can see that the system handles the workload fluctuations very well. The miss ratios of Class 0 and Class 1 queries remain 0 throughout the experiment and the miss ratio of Class 2 queries is restored to below 5% within 10 seconds after the bursts occur.

The second data admission scheme is RT-STREAM-DACM. Each of the service classes and each service class has its own designated controller.
Applying Formal Methods to Modeling and Analysis of Real-time Data Streams

classes in this scheme has its own designated data admission controller. Figure 15 shows the MEDAL model for this data admission scheme. Although all three of the service classes receive data from the same data stream \( S \), the data admission for each service class is determined independently. Each of the three classes, Class 0, Class 1, and Class 2, has its own data admission controller, \( C_0 \), \( C_1 \), and \( C_2 \), respectively. The controllers receive information only about the deadline miss ratio for the queries from their corresponding class.

The results for the RT-STREAM-DACM controller scheme are shown in Figure 16. The data completeness of the Class 0 and Class 1 queries remains very close to 100% and the miss ratio of the Class 0 queries remains at 0. The miss ratio of Class 1 queries has some fluctuations around the 200th second. It is restored to zero within 5 seconds. However, this comes at the expense of the Class 2 queries. During the long workload burst, almost all incoming data for the Class 2 queries is dropped.

Comparing Figure 14 with Figure 16, we can clearly see the QoS tradeoffs in the system. If the application is willing to tolerate low data completeness ratios, the query miss ratios of different service classes can be substantially improved. If the miss ratios of the lower service class queries can be sacrificed, the service quality of the higher class queries can be preserved.

9. CONCLUSIONS AND FUTURE WORK

In this paper we showed how MEDAL can be used to model and analyze real-time data stream queries, QoS management mechanisms, and the relationships between them. Unlike previous work, where query models and system control logic are designed and analyzed separately, MEDAL allowed us to merge these two components into a single comprehensive system model. The advantage of this combined model is that it can be used not only to predict the workload and estimate the query cost, but also to model and analyze the interactions between the input and output of the query plans and the data control mechanism, which gives us a much better understanding of the system.
In our performance evaluation we used a synthetic workload to test the system’s behavior under heavy workload and workload fluctuations. Our results revealed the benefit of using a periodic query model as an alternative to the continuous query model for real-time applications. We also designed and implemented data admission as an overload protection mechanism to reduce the system workload in case of an overload and study the QoS tradeoffs presented by the admission controller. Our results showed that when there was a dedicated controller for each query service class, if the application is willing to sacrifice the miss ratios of the queries from the lower service classes, the higher service class queries can benefit significantly.

For future work we plan to design an automated data admission controller that can preemptively alter the data admission rates and thus maintain high data completeness while ensuring low deadline miss ratio. This data admission controller will take advantage of context information such as spatial and temporal data characteristics, and stream behavior patterns extracted from historical data. We will also implement a MEDAL tool which will allow system designers to conveniently build and analyze data stream applications. This tool could be used to perform operator cost analysis and selectivity estimation. It will also help model dependencies between the data stream system and the surrounding environment, which could be extremely useful for context-aware workload predictions.

ACKNOWLEDGMENTS

This work was supported in part by NRF WCU R33-2010-10110 and by the IT R&D program of MKE/KEIT 10035708.

REFERENCES


Krasimira Kapitanova is currently working towards a Ph.D. in Computer Science at the University of Virginia. She received her B.S. degree in Computer Science and Technologies from Technical University Sofia, Bulgaria, and an M.C.S. degree from the University of Virginia. Her research interests include formal event description in sensor networks, QoS management, and information management and security.

Yuan Wei finished PhD from University of Virginia in 2006. After that, he built large scale data warehouse query engines in Microstrategy Inc and large scale MySQL server clusters in Google Inc. In 2009, he founded Zeno Software Inc to build high speed data stream processing systems for hedge funds.

Woochul Kang received his PhD degree in computer science from the University of Virginia in 2009. His research interests include cyber-physical systems, real-time embedded systems, embedded databases, large-scale distributed systems, feedback control of computing systems, and wireless sensor networks. Currently, he is a senior research staff at Electronics and Telecommunication Research Institute (ETRI), South Korea. He is investigating a distributed middleware architecture that enables efficient and timely access to real-time sensor data in large-scale distributed cyber-physical systems. He also developed Qplus real-time operating systems and its development toolkit.

Sang Hyuk Son is a Professor at the Department of Computer Science of University of Virginia. He received the B.S. degree in electronics engineering from Seoul National University, M.S. degree from KAIST, and the Ph.D. in computer science from University of Maryland, College Park, in 1986. He is on the executive board of the IEEE Technical Committee on Real-Time Systems, for which he served as the Chair during 2007-2008. He is currently serving as an Associate Editor for IEEE Transactions on Computers and Real-Time Systems Journal. His research interests include real-time and embedded systems, database and data services, QoS management, wireless sensor networks, and information security.