Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2006-043 June 12, 2006

Schematic Querying of Large Tracking Databases

Gerald Dalley and Tomas Izo

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Schematic Querying of Large Tracking Databases

Gerald Dalley Tomáš Ižo
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
Email: {*dalleyg, tomas*}*@csail.mit.edu*

***Abstract*— In dealing with long-term tracking databases with wide-area coverage, an important problem is in formulating an intuitive and fast query system for analysis. In such a query system, a user who is not a computer vision research should be able to readily specify a novel query to the system and obtain the desired results. Furthermore, these queries should be able to not only search out individual actors (*e.g.* "find all white cars") but also find interactions amongst multiple actors (*e.g.* "find all drag racing activities in the city"). Informally, we have found that people often use sketches when describing activities and interactions. In this paper, we demonstrate a preliminary system that automatically interprets schematic drawings of activities. The system transforms the schematics into executable code that searches a tracking database. Through our query optimization, these queries tend to take orders of magnitude less time to execute than equivalent queries running on a partially-optimized SQL database.**

I. Introduction

Tracking systems are typically constructed with at least one of the following main purposes: to gather statistics about mobile and visually-observable phenomena, to detect anomalous and/or interesting activities in real-time, and to warehouse data to allow for later off-line querying. For this paper, we concentrate on the last of these. In particular, we are interested in building more intuitive and efficient interfaces for querying a database of tracking data.

For a querying environment, we believe that the following are important and interrelated goals:

- **Simplicity:** The query system should have a simple and intuitive manner of allowing the user to specify typical queries. It should also not require substantial amounts of time to specify new types of queries. For example, training a new and robust statistical activity detector may take hours or even months (including development time).
- **Expressibility:** The system should have a rich enough interface to allow for accurately describing complex activities.
- **Speed:** It should be able to find instances of the queried activity quickly. Note while that online detection typically only needs to run in realtime, much greater speeds are required for querying a database that has been populated by multiple cameras over a long period of time.
- **Refinement:** For all but the most trivial of situations, some iteration on the query will be needed to obtain exactly the desired results. The system should allow for easy refinement of queries.

In this paper, we present the groundwork for a system that targets these four goals. For our initial system, we provide a means for easily specifying the initial queries through drawn schematics. We then transform these schematics into lists of logical propositions which are passed to the low-level query system. Although our current system only recognizes a small number of types of propositions, it could naturally be expanded to include many more. The low-level query system then iterates over all possible combinations of observations that could satisfy the propositions to find the ones that actually do. To avoid testing large parts of the combinatorial space, we employ a cost-based query optimizer. In our final section, we describe some future enhancements that could be made to allow for query refinement.

At the present time, we are interested in sparse sensor networks where we model the world as a directed graph where the nodes are discrete locations with cameras and/or other sensors located on the edges. This model is directly applicable to large-area camera networks where cameras have a high zoom factor. It is also applicable to wide-angle far-field tracking where there are discrete locations where interesting events occur.

The main contribution of this work is to lay out an intuitive system for specifying and executing fast queries on activities involving multiple actors.

II. Prior Work

Our work builds off a history of work on database query optimization, single-person activity recognition, interaction recognition, sketch-based user interfaces.

The database community discovered early on that queries that involved JOINs could be optimized to avoid having to do a search through the database that is exponential in the number of JOINs required. A JOIN is a SQL construct roughly equivalent to a (sometimes slow) reference or pointer lookup in a procedural programming language. Selinger *et. al* enhanced the early System-R database management system to optimize queries that involved JOINs [1]. The query can be decomposed into steps where at each step, one JOIN is satisfied. The system examines all possible orderings of JOIN statements. Using statistics gathered on indexed data, they estimate the cost of satisfying the JOINs in that particular order. Depending on various query conditions and on which JOINs have already been satisfied, a new JOIN may be able to take advantage of data structures designed to speed up and/or eliminate searches

through the database tables. The cheapest expected ordering is then chosen, reduced to machine code, and executed. Nearly all modern database management systems use variants of this approach.

Most of the recent work on event and activity recognition has involved using statistical techniques to overcome the brittleness of traditional logic-based systems. Bobick and Ivanov [2] used Hidden Markov Models (HMMs) to detect low-level events in video which are combined using a stochastic context-free grammar to recognize higher-level events. They apply this approach to single-hand gesture recognition. Brand and Kettnaker [3] uses HMMs to learn single-actor activities and a scene-wide composite activity (traffic flow patterns at a busy intersection). Oliver, Rosario, and Pentland [4] use synthetic data to learn coupled HMMs for detecting two-actor activities. Hongeng and Nevatia [5] learn more complex models for recognizing activities involving many actors. Their prototypical example is "stealing-by-blocking", where the victim enters the scene and drops off a bag. An accomplice approaches and converses with the victim while standing between the victim and his bag. Finally, the thief enters and takes the bag. To recognize this activity, they use Bayesian networks to recognize low-level activities and string these together with probabilistic finite state automata to recognize single-actor high-level activities. Multi-actor activities are recognized using a probabilistic adaptation of Allen's interval algebra [6].

More recently, there has been some renewed interest in using logic-based approaches. Ghanem *et. al* [7] begin with video that has manually-labeled primitive events and activities. To build a detector, they create a set of temporal ordering constraints that link low-level events and of interest. They then generate a Petri net to detect the described multiple-actor activities. Köhler [8] uses single-camera scenes where there are only a small number of actors visible at any point in time. On these actors, he defines a set of discrete spatial and temporal relationships such as "before" and "in front of". From his tracking database, he precomputes all spatial relationships between all actors present in each video frame. He then defines roles of actors such as the head and tail of a queue using spatio-temporal propositions.

Research in sketch-based user interfaces [9] suggests that we can interact more naturally with computers when the computer can understand more traditional modes of communication such as sketching. Yu and Boult [10] recently developed a preliminary system for recognizing two-actor activities using sketches. They track actors using GPS transponders and project their locations onto a city map. To specify a query such as "car dropping off a person", they use icons to draw the relative trajectories of a car and a person. From this initial sketch, they generate many sketches by varying the angle between the trajectories, for example. Once they have generated all the sketches, they compute an eigenspace of these sketch images. To recognize an activity, they plot the positions of the actors at each timestep on the map. These plots include a method of showing the recent trajectory history of the actors. Any areas of this map image that project well onto the eigenspace are recognized as examples of that activity.

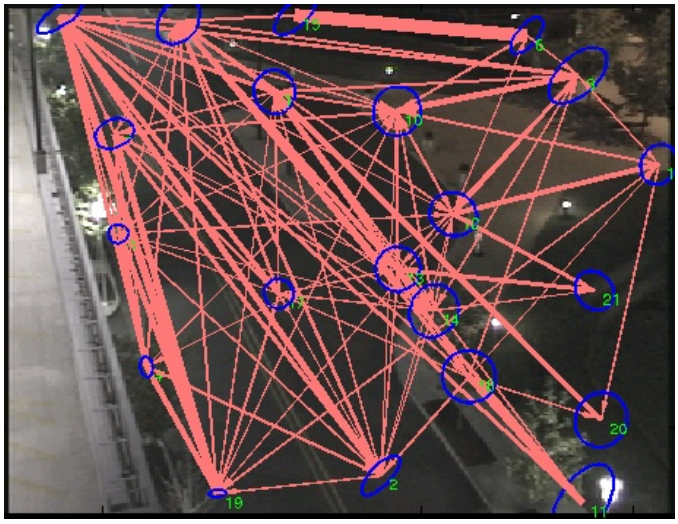

Fig. 1. **Far-field observers and links:** A far-field tracking scene with a two-way road (top to bottom), the entrance to a parking garage (on the left), and a pull-out area (in the middle) for deliveries and drop-offs to a nearby business (on the right). Source and sink locations have been clustered (blue/dark ellipses). In our schema, these 21 locations are interpreted as observers which generate a report anytime a tracked object passes through them. The pink/dark gray lines connecting the observers are the links along which objects have traveled. The clustering and link discovery were done by training on all of the clean tracks detected over the course of a day.

III. Tracking Databases

For our system, we assume that the low-level tracking has already been performed from the camera(s) and/or other sensors. Any necessary clustering and segmentation of tracks has also been done prior to using this query system. These systems populate our tracking database. This database consists of the following notable classes: observers, links, observations, observation links, and tracks.

Observers represent discrete locations of interest through which actors pass. For example, in a far-field tracking scenario, the observers correspond to the clustered source and sink locations (see Fig. 1). For sparse mid-field camera networks, we create one observer for each typical path through each camera. For example, if a mid-field camera is viewing a single street, we might create one observer per direction. In Fig. 2, we show a map of a network of such cameras. In both cases, an observer acts as a "smart tripwire", reporting timestamp and identity information whenever an actor passes through it.

A link exists between any two observers when there is a non-zero probability of an actor moving between those two observers. For example, in Fig. 1 the arrows indicate all of the learned links. Whether the links and observers form a complete graph or not depends on the tracking system employed, not on our query system. The links and observers together define the graph structure of the surveillance network.

In our model, whenever an observer sees an actor pass, it generates an observation record. This observation record may be annotated with information such as the observation time,

Fig. 2. **Citywide:** Road network used to generate simulated data. Red lines are streets (one directed edge each way), blue circles are intersections and green circles are sensors. Each sensor has one observer that reports each vehicle that passes in each road direction.

a color histogram, class label (e.g. truck vs. car vs. person), speed, and other features. Subsequent observations are linked together with observation links to form an entire track for each actor. An observation corresponds to a timestamp and an observation link corresponds to the time interval between observations.

For our experiments, we use two databases. Our first database was generated synthetically. Its aim was to simulate the output–over the span of one day–of a network of many smart tripwire-like cameras distributed across a large urban area with a complex road network. We represent the road network as a directed graph, in which nodes correspond to intersections and edges correspond to streets. Each sensor is located on an edge of the graph, corresponding to a camera placed on the side of the street with its axis perpendicular to the direction of traffic. Associated with each edge is a distribution over the transition time. For most edges, we assume this to be a Gaussian with the mean directly proportional to the real-world distance between the end nodes. Similarly, nodes are associated with some distribution over the stopping time. Our road graph contains 59 nodes and 221 directed edges (including loopback transitions for each node). We place 56 sensors along the edges of the graph (see figure 2).

In order to generate plausible sensor outputs, we designate certain nodes of the graph as residential, commercial, etc. We then simulate individual cars leaving from residential nodes and traveling to commercial nodes in the morning and in the opposite direction in the evening, returning to their homes. Similarly, cars tend to travel from commercial to other commercial or recreational nodes around noon, returning within a short period of time. In addition to this regular traffic, we add a large number of cars traveling along randomly generated paths through the road network. Whenever a car passes one of the sensors, we store an observation report that consists of the vehicle ID and the time stamp.

As the main motivation for our work is to provide an intuitive interface for the retrieval of specific activities, we inserted into our simulated road traffic a number of examples of ”anomalous” activities, which we describe in detail in the next section.

IV. Querying

A large database of tracking data collected for security purposes would not be very useful without a querying method intuitive enough to be used by a security professional who does not necessarily possess a doctorate in computer science. From our experience, people often use sketching (e.g. on a white board) combined with speech and gesturing to describe spatial and temporal relationships between actors involved in a specific activity. The aim of this paper is not to provide a comparably rich, multimodal interface, but rather to demonstrate the need for such an interface by showing how even a very rough approximation–such as a schematic drawing–can be used to specify a query prototype in an intuitive way.

Our schematic drawings contain four main elements: 1. Actors. These correspond to people, vehicles, etc. that are the participants in the activity. We represent actors by a stick figure icon (see Fig. 4). 2. Locations. These correspond to sensors and we represent them by rounded rectangles. 3. Transitions are drawn as lines with arrows specifying the direction. They denote actors traveling from sensor to sensor. A pair of transitions with endpoints on the same location are considered to be associated with the same actor iff they are anchored at points on the same horizontal line (e.g. a transition anchored at the top left corner has the same actor as one anchored at the top right corner). We initially establish the actor by drawing a transition from the actor’s icon to the first sensor at which the actor is to appear. Each transition has associated with it a variable that denotes the time at which the actor arrives at the given sensor. 4. Constraints. These are pair-wise constraints on the arrival times that the user can specify by typing them into the interface. In a multi-modal sketch-based approach, these constraints would be best described using words and gestures.

The schematic drawing interface works in the following way: 1. The user draws a schematic of the desired activity using the elements described above. We use the open source diagram drawing program Dia as the drawing interface. 2. We save the drawing as XML code. 3. We parse the XML code and extract the underlying graph structure of the drawing as well as the pair-wise constraints on observation times. 4. Using the graph structure and the constraints, we automatically generate a set of propositions to be used for the query. We describe these in greater detail in the following section.

The interface is perhaps best described by showing examples of how it would be used. We present three simple activity scenarios and show how a schematic drawing can be used to describe them: 1. U-turn. Intuitively, a car making a u-turn should generate two consecutive observations at the same sensor but in opposite directions. We can easily describe this activity by the drawing in Fig. 3. 2. Drag racing. This is an

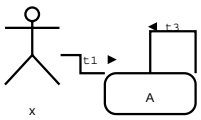

Fig. 3. Schematic drawing of a u-turn.

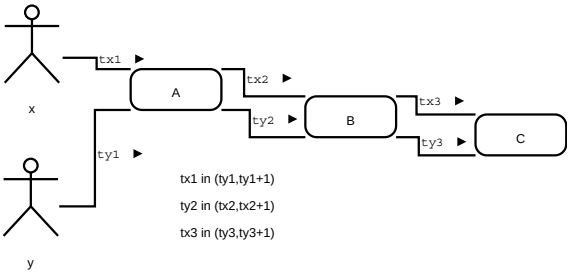

Fig. 4. Schematic drawing of drag racing.

example of an activity that might be a real security concern. Suppose we need to retrieve examples of cars racing each other. This could be similar to a following behavior, where a pair of cars appear at a number of consecutive locations with the second appearing within some short amount of time before or after the first at each location. We choose to define racing by a pair of vehicles appearing at three consecutive locations with one car leading at the first location, the other vehicle leading at the second location and the first vehicle leading again at the last location. The time constraints need to be specified by hand. (see Fig. 4) 3. Several cars meeting and traveling together. This is an example of a slightly more complex activity: three cars coming from different locations (call them A,B, and C), converging onto the same location (D) and traveling together to location E. In addition to the schematic, we also need to specify the time constraints that ensure the appearance of all three vehicles at the locations within some short amount of time.

V. Propositions

In the previous section, we described our schematic system and noted that it outputs a list of propositions. For our initial system, we support four types of propositions: track identity, sensor identity, observer identity, track following, and time intervals. We will describe each type of proposition that we currently support.

A track identity proposition is of the form $(a.track == b.track)$. This indicates that observations a and b both belong to the same track. This proposition does not imply any temporal or spatial constraints on the two observations. An sensor or observer identity proposition has the form $(a.sensor == b.sensor)$ or $(a.observer == b.observer)$, respectively. These indicate that both observations happened at the same location (sensor or observer, respectively) without implying any temporal constraints. A track following proposition is of the form $(a.next.next == b)$ and means that the tracker determined that observation b is the next observation after a in the track that contains both a and b. This implicitly implies that a happened before b, but makes no spatial implications. A time interval proposition has the form $(a.time < b.time + 5)$ or $(a.timein(b.time+1, c.time+5])$. In the first, observation a happened at least 5 seconds before observation b. No spatial or tracking propositions are implied. The second is a compound proposition that implies $(a.time > b.time+1)$ and $(a.time <= c.time + 5)$. No tracking or spatial propositions are implied.

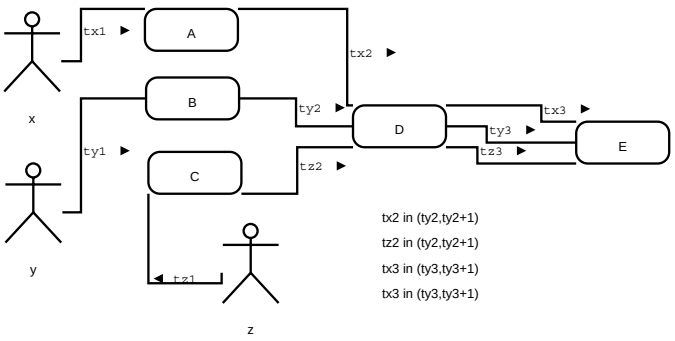

Fig. 5. Schematic drawing of converging cars.

Naturally, additional types of propositions could be added in the future. For example, unary propositions of the form $(a.class == truck)$ or $(a.time.day == Monday)$ would be extremely useful. Additionally, we expect to add support for propositions that reference observation links and not just observations.

In Fig. 6, we list the propositions for the four experiments we perform. Although these propositions are precise in what they define, in many ways they are less readily understandable compared to their schematic counterparts.

VI. Query Processing

Once we have created a set of propositions that define our query, we need to find all tuples of observations that satisfy those propositions. For this discussion, we will consider a moderately-sized wide-area database with approximately a million observations and the following propositions that define a "following" activity (see Fig. 6(b)).

In the most naïve implementation, we might choose to have six nested loops that try all combinations of observations for each of the variables $\{l1, l2, l3, f2, f2, f3\}$ and retain only those tuples that satisfy all ten propositions. This would require testing $10^{6\times6} = 10^{36}$ tuples, which is clearly impractical.

We initially began by transforming the propositions into SQL statements which were evaluated by a relational database management system (RDBMS). In particular, we have used MySQL and Microsoft Access. We found that even dramatically simplified versions of the following query took minutes to hours to satisfy due to difficulties in making the JOIN clauses efficient. While we would expect performance improvements through more extensive use of indexing and perhaps usage of higher-end database systems, we chose to develop our own query optimization and evaluation system.

$(x1.next.next == x2)\ (x1.sensor == x2.sensor)$

(a) **U-Turn:** (automatically generated from Fig. 3)

$(l1.next.next == l2)(l2.next.next == l3)$
$(f1.next.next == f2)(f2.next.next == f3)$
$(l1.observer == f1.observer)\ (l2.observer == f2.observer)$
$(l3.observer == f3.observer)$
$(f1.time\ in\ (l1.time, l1.time + 5])$
$(f2.time\ in\ (l2.time, l2.time + 5])$
$(f3.time\ in\ (l3.time, l3.time + 5])$

(b) **Following:** (manually generated) These propositions define following as seeing actor f appear at the same locations as actor l, three times in a row, all within 5 seconds of when l appears there.

$(x1.next.next == x2)\ (x2.next.next == x3)$
$(y1.next.next == y2)\ (y2.next.next == y3)$
$(x1.observer == y1.observer)\ (x2.observer == y2.observer)$
$(x3.observer == y3.observer)$
$(x1.time\ in\ (y1.time, y1.time + 5))$
$(y2.time\ in\ (x2.time, x2.time + 5))$
$(x3.time\ in\ (y3.time, y3.time + 5))$

(c) **Racing:** (automatically generated from Fig. 4)

$(x1.next.next == x2)\ (y1.next.next == y2)$
$(z1.next.next == z2)\ (x2.observer == y2.observer)$
$(y2.observer == z2.observer)$
$(x2.next.next == x3)\ (y2.next.next == y3)$
$(z2.next.next == z3)\ (x3.observer == y3.observer)$
$(y3.observer == z3.observer)$
$(x2.time\ in\ (y2.time, y2.time + 1))$
$(z2.time\ in\ (y2.time, y2.time + 1))$
$(x3.time\ in\ (y3.time, y3.time + 1))$
$(z3.time\ in\ (y3.time, y3.time + 1))$

(d) **Converging:** (automatically generated from Fig. 5)

Fig. 6. **Propositions used:** These sets of propositions were used to perform the experiments describedinthis paper. All except 6(b) were generated automatically from their schematics.

Our query system takes as a string the propositions and parses them into a set of proposition objects. We then use a greedy cost-based system to choose the satisfaction ordering of our propositions. This system generates Java source code that will perform the actual query. The source code consists of a series of nested loops, conditional statements, and object traversals that try to minimize the total expected query satisfaction time.

If we wished to satisfy "(l1.next.next ==l2)" as the first proposition, we would create a loop of the form (shown in pseudo-code for simplicity):

1: **for all** $Observation\ l1 \in allObservations$ **do**
2: $Observation\ l2 = l1.next.next$;
3: ...code for satisfying subsequent
4: propositions goes here..
5: **end for**

Satisfying this proposition has an expected cost of 1×10^6 because we must loop over all observations. If we then wished to satisfy $(l2.next.next == l3)$, our system knows that the variable $l2$ has already been bound and thus only the following code: Observation l3 = l2.next.next; code for the 3rd proposition goes here would need to be injected in order to satisfy this second proposition and bind variable $l3$. Because no looping was needed, given that the first proposition has been satisfied, the expected cost is 1. Note that had we attempted to satisfy $(f1.next.next == f2)$ as the second proposition, our total expected cost would have been multiplied by 1×10^6 instead of 1.

To satisfy a track identity, observer identity or track following proposition, the expected cost is 1×10^6 if neither variable is bound, 1 if one variable is bound (we perform the object traversal to fetch the unbound variable), or 1 if both variables are already bound (we simply check to make sure that the proposition holds).

For the time interval propositions, we assume a uniform distribution of observation times. Given a time-based index, in many cases, the cost of iterating through a small time window is small since the number of observations to consider is small and these observations may be found quickly ($O(\log_2 N)$, where N is the total number of observations in the database) and traversed in linear time.

When assembling the query, we examine all unsatisfied propositions and find the one with the smallest expected satisfaction cost, given the current free and bound variables. We greedily select the cheapest proposition and satisfy it by binding any remaining variables that apply to that proposition and checking that the bindings satisfy the proposition's conditions. This process repeats until all propositions have been satisfied.

At this point, we inject the source code required to pass the results out of the query. We then compile the Java source at runtime into bytecode and execute it. The compilation of the "following" query takes a few seconds or less on a Pentium 4, 3GHz machine, and running this query take about a half a second when we only count the number of results. If we actually create a data structure to store all the results, without taking too much care for efficiency, the entire process including parsing, compilation, execution, and display of the results in an unoptimized GUI takes tens of seconds.

While modern RDBMSs have used similar techniques for decades [1], we found that we gained several extra features by implementing this query system on a database of live objects rather than using an existing SQL database:

- Finer control over optimization settings. For example, we can tune more tightly estimates over how long certain propositions should take under different conditions.
- Finer control over how results are streamed to the client.
- Future abilities to modify the query on-the-fly. For example, we imagine presenting tuples to the user as they are found. The user could then refine the query as it is being executed.
- Ability to directly take advantage of object traversal without even needing to even do an index lookup: $(a.next.next == b)$, $(a.observer == b.observer)$, *etc.*
- Ease in taking advantage of a special temporal index for $(a.time\ in\ (b.time, c.time])$. In particular, after implementing our system, we examined the query optimization in MySQL and had a difficult time convincing it to use a temporal index to speed up the queries even though it

should result performance improvements of a few orders of magnitude.

VII. Results

We ran the four queries described in section V on both of our databases. For the far-field scene (see Fig. 1), there are a total of 1730 observations of 865 tracks made by 21 observers. Due to artifacts in the data, our U-turn query returns 67 detections. If modified to only require two observations per actor, our following query reports 6 activities. If similarly modified, the racing detector detects one activity. Four simplified convergence activities were detected.

For the simulated city scene (see Fig. 2), there are a total of 16,311 reports of 1100 vehicles made on 112 observers. Using the U-turn query as defined, we detect 971 activities. By adding a proposition, $(x2.time < x1.time + 1)$, to avoid detecting people that return along the same path after several hours, only 30 results remain. If we require the vehicles to also retrace their path by a node, we drop to 11 detections. This type of exploration is easy to do. 133 suspected racing activities were detected.

Because our simulator is designed with lunch, work, and residential zones, our following query returns 157,685 overlapping following activities. If a particular type of following were being queried, additional propositions would be required, for example we might only want to see the activities where one car passes through a particular set of interesting sensors. For similar reasons, the number of converging activities was extremely high: 793,171. Extensions to our system become critical here to extract the desired results.

In some performance tests, we used simulated data with 66 observers and nearly a million reports. The "following" query took 1.5 seconds to compile and approximately a half a second to execute when only counting the number of results. Due to inefficiencies in our Java code, tens of seconds were required when actually storing a full list of results.

VIII. Discussion

In our system, we use logical propositions. In doing so, we readily recognize that no simple system of such propositions is likely to result in a robust detector. For example, in section IV, we discuss a drag racing activity. Our definition can detect such activities, but it also has a very high false-positive rate. Other less-narrow activity specifications result in even worse signal-to-noise ratios. However, because or system is extremely fast, we can deal with these false positives by either allowing the user to refine the query and add additional constraints or by passing off the results to another process. To demonstrate this, we added the temporal proposition to the U-turn query for the simulated data and reduced the number of results by nearly three orders of magnitude.

Because our propositions imply fixed-length tuples of observations, we also envision exporting feature vectors to be used with common machine learning algorithms such as clustering and classifiers.

In the future, we expect to make the following additional contributions:

- **Sketch-based:** We would like to leverage the work of the sketch understanding community to allow the input schematics to be sketched by hand and include the capability for multimodal input.
- **User studies** to determine how well the system works for real users and how people choose to describe activities when a computer is not involved.
- **User-guided:** When results are returned by the query, we envision doing the following:
  - **Clustering:** In several instances in section VII, we received a very large number of results. These results could be made more useful by employing machine learning techniques such as clustering. For example, we might want to highlight times and areas where convergence activities tend to occur to gather statistics and/or find anomalous ones.
  - **Structural query modification:** For long-running queries involving many observations, we might modify the structure of the search on-the-fly to return the results of most interest first.
  - **Refinement:** As the query is being processed, we would like to be able to add, remove, and modify constraints from the query.

In summary, we have developed a system for querying databases using schematic drawings as a front-end. In this system, we automatically convert the schematics to a set of propositions. These propositions are then parsed and a cost-based query optimizer reduces them to Java bytecode. We demonstrated the usage of this system on a real far-field scene and a synthetic city-sized sensor network. We also did performance tests on a much larger set of simulated data to test the system's scalability. In doing so, we saw that our system performed well in terms of speed, and we suggest additional improvements to help reduce the false positive rate.

References

[1] P. Selinger *et. al*, "Access path selection in relational database management systems," in *International Conference on Management of Data*. ACM SIGMOD, 1979, pp. 23–34.

[2] A. Bobick and Y. Ivanov, "Action recognition using probabilistic parsing," in *CVPR*. IEEE, 1998, pp. 198–202.

[3] M. Brand and V. Kettnaker, "Discovery and segmentation of activities in video," *PAMI*, vol. 22, no. 8, pp. 844–851, Aug. 2000.

[4] N. Oliver, B. Rosario, and A. Pentland, "A bayesian computer vision system for modeling human interactions," *PAMI*, vol. 22, no. 8, pp. 831–843, Aug. 2000.

[5] Hongeng and Nevatia, "Multi-agent event recognition," in *ICCV*. IEEE, June 2004.

[6] J. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, Nov. 1983. [Online]. Available: http://doi.acm.org/10.1145/182.358434

[7] N. Ghanem *et. al*, "Representation and recognition of events in surveillance video using petri nets," in *Event Detection and Recognition Workshop at ICCV*. IEEE, June 2004.

[8] C. Köhler, "Selecting ghosts and queues from a car trackers output using a spatio-temporal query language," in *ICCV*. IEEE, June 2004.

[9] R. Davis, "Position statement and overview: Sketch recognition at mit," in *AAAI Spring Symposium on Sketch Recognition*, 2002.

[10] L. Yu and T. Boult, "Understanding images of graphical user interfaces: A new approach to activity recognition for visual surveillance," in *Event Detection and Recognition Workshop at ICCV*. IEEE, June 2004.