A Methodology for In-Network Evaluation of Integrated Logical-Statistical Models

Anu Singh C. R. Ramakrishnan I. V. Ramakrishnan David S. Warren
Jennifer L. Wong
Department of Computer Science
Stony Brook University, Stony Brook, NY, USA
{anusingh,cram,ram,warren,jwong}@cs.sunysb.edu

ABSTRACT

Synthesizing high-level semantic knowledge from low-level sensor data is an important problem in many sensor network applications. Programming a network to perform such synthesis in situ is especially difficult due to the stringent resource constraints, unreliable wireless communication, and complex distributed algorithms and network protocols required to manipulate the data. Recently, a declarative programming language called Snlog [5] has been developed to address this problem. However, statistical reasoning for modeling noise in the context of sensor networks has not been addressed in Snlog. In this paper, we develop a methodology based on the PRISM [36] framework, which integrates logical and statistical reasoning, for specifying sensor network programs that deal with noisy data and tolerate faults in the network. The relationship between high-level (synthesized) and low-level (observed) data is captured by logical rules, while statistical models are used to specify computations in the presence of noise and faults. We illustrate our methodology with three examples: (i) estimating temperature at various points in a region, (ii) evaluating the trajectory of an object observed by a sensor network, based on the Hidden Markov Model, and (iii) evaluating most reliable communication paths between sensor nodes. We analyze the results of simulations as well as an experimental deployment to evaluate the practical feasibility of our approach.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Distributed Applications*; D.1.6 [**Programming Techniques**]: Logic Programming; D.2.10 [**Software Engineering**]: Design

General Terms

Languages, Design, Experimentation

SenSys'08, November 5–7, 2008, Raleigh, North Carolina, USA.


```
reach(S,D) :- link(S,D).
reach(S,D) :- link(S,T),
              reach(T,D).
link(a,b).
link(b,c).
link(b,d).
```

(a) Datalog

```
reach(@S,D) :- link(@S,D).
reach(@S,D) :- link(@S,T),
               reach(@T,D).
```

(b) Snlog

Figure 1: Reachability example

Keywords

Sensor networks, In-network analysis, Logical-Statistical models

1. INTRODUCTION

Sensor networks are multi-hop wireless networks consisting of a large number of sensor nodes each having very limited computing and energy resources. They are particularly useful in monitoring poorly accessible environments such as the ocean floors, disaster areas, etc.

A sensor network generates low-level sensor data that are typically represented by streams of numeric values. However, applications based on such networks often require that low-level data be automatically synthesized into high-level conceptual knowledge, such as "a vehicle has stopped moving", "several individuals have assembled in a place", etc.

Programming a sensor network application to synthesize such semantic knowledge remains a difficult task, since the programmer is burdened with low-level details related to distributed computing, careful management of limited resources, unreliable infrastructure, noisy sensor data and energy optimizations. Thus, developing a powerful programming framework for sensor networks is critical to realizing their full potential as collaborative monitoring systems. Ideally, we would like a user to be able to specify the desired computations associated with the application at a high-level, and automatically translate the specification into efficient distributed code that runs on individual nodes.

Towards such a goal, a declarative programming language Snlog was recently proposed [5]. Snlog is in essence an annotated Datalog program which is the function-free subset of the logic programming language Prolog. A Datalog program consists of rules of the form *head :- body* where *head* is an atomic formula and *body* is a conjunct of literals. The *head* is true whenever the *body* is true. A *head* with an empty *body* represents a fact that is unconditionally true. The set of all rules with identical names for their head, say *pred*, define the predicate *pred*.

Figure 1(a) is a Datalog specification for the *reach* predi-

cate in a network. The first rule says that node D is reachable from node S if there is a link from S to D. The second rule says that the other way D is reachable from S is if there is a link from S to some node T and D is reachable from T. The three facts specify the links in the network, namely, there are links from node a to node b (first fact), node b to node c (second fact) and node b to node d (third fact).

Executing a Datalog program amounts to posing queries and evaluating their truth using the rules and the facts in the program. For instance the query "?reach(a,X)" will return all the bindings for the variable X that make the query true. These will be b, c and d in the example. In relational database terminology the rules of a predicate constitute a (relational) view definition and query evaluation amounts to computing tuples in a relation, e.g. the tuples (a, b), (a, c) and (a, d) in the *reach* relation for the query above.

To facilitate distributed evaluation, Snlog programs are annotated with a location specifier. Specifically, one argument of the predicate is annotated with the "@" symbol to indicate that the value of this argument in any tuple specifies the sensor node where that tuple is stored. For example, in Figure 1(b) $link(@S, D)$ denotes that the tuple (S, D) in the *link* relation is located at the sensor node S. Similarly $reach(@S, D)$ and $reach(@T, D)$ denote the tuples (S, D) and (T, D) respectively hosted at nodes S and T. So at runtime, evaluation of the reach query using the second rule in Figure 1(b) will require a network communication from node T to node S so that node S knows what tuples (T, D) are in *reach*.

Although Snlog demonstrated the power and potential of the logic-style declarative programming paradigm for developing sensor network applications, it did not take into account an important aspect of sensor networks, namely, that sensor data is inherently noisy and the nodes are prone to failures. For instance, given a probability distribution for node failures, one would like to know the "*likelihood*" of reachability between nodes in the sensor network. Thus, it is important that the declarative programming approach facilitate specifications to capture aspects of uncertainty associated with the sensor network. Queries evaluated w.r.t. such specifications will return likelihood values (ranging from 0 to 1) instead of just yes/no answers.

Such a declarative framework is offered by PRISM, a language for specifying statistical models as logic programs [36]. In this paper, we present a methodology for the specification and evaluation of logical-statistical models of high-level events in sensor networks using PRISM. The noisy nature of the network is captured by the statistical aspect of the language while the logic encodes its high-level behavior. It should be noted that any statistical model that can be represented as a directed acyclic graphical model can be encoded in the PRISM language. Furthermore, the PRISM framework incorporates an EM learning algorithm [18] to estimate the distributions of random variables (statistical parameters) in a program from examples. In this paper, however, we focus only on evaluation of statistical models with known distributions, and do not consider the problem of learning the distributions.

In order to facilitate distributed evaluation, we extend PRISM with annotations analogous to those in Snlog programs. We then automatically generate Snlog programs that evaluate maximum likelihood queries over annotated PRISM models. The resulting Snlog programs can be directly evaluated on motes. We show the practical feasibility of our approach by encoding problems typically solved using sensor networks, such as estimation of temperature at locations within a region, and computation of communication routes. We also illustrate the power and flexibility of PRISM-based sensor network programming methodology by encoding and evaluating a problem that involves temporal reasoning, namely, that of evaluating trajectories using Hidden Markov Models.

The rest of the paper is organized as follows. Section 2 gives preliminaries on the PRISM framework. Section 3 describes modeling of sensor network applications in PRISM. Our methodology for translating centralized PRISM specifications to Snlog programs is given in Section 4. Experimental evaluation of trajectory detection in a sensor network using our modeling and inferencing approach is presented in Section 5. Related work is discussed in Section 6, followed by concluding remarks in Section 7.

2. PRISM FRAMEWORK

We briefly review the PRISM language using an encoding of Hidden Markov Models (HMMs) as an example. An HMM is a stochastic finite state automaton, where the transitions from a state have an associated probability distribution. When a state in an HMM is reached, an observation symbol is emitted; the emissions again have an associated probability distribution. HMMs are typically used to model temporal or sequential phenomena. In the context of sensor networks, the states may correspond to the events that we want to detect (e.g. position of a vehicle), and the observations may correspond to the actual sensor readings.

A PRISM program for an arbitrary HMM is given in Figure 2(a). PRISM programs have Prolog-like syntax: we use identifiers beginning with upper-case letters to denote variables, and terms consisting of integers, floating point numbers, and identifiers to denote data. A PRISM program consists of a set of rules, each of which partially defines a relation. Syntactically, each rule is of the form "*head* :- *body*", read as "head *if* body". The head of a rule is a predicate instance used to define a set of tuples in a given relation. The body of a rule is a sequence of predicate instances representing a conjunction of conditions under which the tuple defined by the head is in the given relation. For instance, the first rule of Figure 2(a) can be seen as defining tuples of the form `(0,S)` in the `hmm` relation whenever tuples of the form `(init,S)` are present in the `msw` relation (which is a special relation in PRISM, as explained below). Thus each rule defines a set of tuples in a relation. As usual, a PRISM program may have more than one rule defining a single relation. Then the set of tuples in the relation is the union of all tuples defined by each rule.

In a PRISM program the `msw` relation ("multi-valued switch") has a special meaning: `msw(X,V)` says that `V` is the value of a finite-domain random variable `X`. Thus, the `msw` relation provides the mechanism for defining and using random variables, thereby allowing us to weave together statistical and logical aspects of a model into a single program. For instance, consider the program in Figure 2(a) which defines a relation describing the sequence of states in a run of an HMM. The binary predicate `hmm` has two parameters: `hmm(I, S)` means that state `S` is at index position `I` in the state sequence. The program assumes that the corresponding sequence of observations is given by another relation `obs`

```
% hmm(I, S): S is the state at the
   I-th position in a run of the HMM

hmm(0, S) :-
    msw(init, S).
hmm(I, S) :-
    hmm(PrevI, PrevS),
    I is PrevI + 1,
    I > 0,
    msw(tr(PrevS), S),
    obs(I, A),
    msw(out(S), A).
```

(a)

```
% mle(hmm(I, S), P, E): E is the most likely state sequence ending
%   at state S at index I; P is the probability of E.

mle(hmm(I, S), <MAX(P)>, <ARGMAX(E,P)>) :-
    eval(hmm(I, S), P, E).

% eval(hmm(I, S), P, E): E is a likely state sequence ending
%   at state S at index I; P is the probability of E.

eval(hmm(0, S), P, [S]) :-
    distr(init, S, P).
eval(hmm(I, S), P, [S|PrevE]) :-
    mle(hmm(PrevI, PrevS), PrevP, PrevE),
    I is PrevI + 1,
    I > 0,
    distr(tr(PrevS), S, TransP),
    obs(I, A),
    distr(out(S), A, EmitP),
    P is PrevP * TransP * EmitP.
```

(b)

Figure 2: (a) PRISM model of an HMM; (b) Evaluation of most likely state sequence for HMMs

such that `obs(I,A)` means that `A` is the symbol at the `I`-th position in the observation sequence. Note that the state and observation sequences may unfold over time, hence the index `I` can be treated as a clock value (corresponding to discrete time). In the following discussion, we use notions of index and time instance interchangeably.

The first rule of `hmm` says that `S` is the state at the 0-th time instant (`hmm(0,S)` on the left hand side of the rule) whenever `S` is the value of random variable "`init`" (`msw(init, S)` in the body of the rule). Thus the random variable "`init`" represents the initial state distribution of the HMM. The second rule for `hmm` defines the conditions under which we can go to state `S` at position `I` from state `PrevS` at position `PrevI`, where `I` is `PrevI+1` (i.e. the next index in the sequence):

1. `msw(tr(PrevS), S)` which means that `S` is the value of random variable `tr(PrevS)`, i.e. the next state is generated using the random variable `tr(PrevS)`.

2. `obs(I,A)` which means that symbol `A` is at the `I`-th position in the observation sequence

3. `msw(out(S), A)` which means that `A` is also the value of random variable `out(S)`.

The family of random variables `tr(·)` and `out(·)` correspond to the transition and emission probabilities of states in an HMM: `tr(S)` gives the transition probabilities from state `S`, and `out(S)` gives the emission probabilities from state `S`.

The meaning of a PRISM program is given in terms of a *distribution semantics* [36], explained at a high level as follows. We can treat the PRISM program as a logic program defined over a set of facts (external database, or EDB, in Datalog terms) that define the `msw` relation. An instance of the `msw` relation defines one choice of values of all random variables. The probability of this instance can be found using the probability distribution of each random variable. A query over a PRISM program can be evaluated by first generating an instance of the `msw` relation, treating the PRISM program and this `msw` instance together as a single logic program, and evaluating the query over the resulting program.

Unlike a Prolog program, the PRISM framework can be used to compute the probability of each answer to a query when the distributions of random variables is known. Moreover, PRISM can be used to find the *most likely explanation* for an answer. In the following, we describe how the above computations are done in PRISM.

2.1 Evaluating the Probabilistic Models

In the case of HMM models, a typical query of interest is to find the most likely state sequence that can result in the given observation sequence generated by the sensor network. Viterbi's algorithm [34] provides an efficient way to find such a state sequence using dynamic programming. Given an HMM with $|S|$ states, and given an observation sequence of length N, Viterbi's algorithm has $O(|S|^2 N)$ time complexity. Conceptually, for each state S_j and time instant t, Viterbi's algorithm computes the value δ_{jt} which is the probability corresponding to the most likely state sequence ending at S_j for the observation sequence until time instant t. The value δ_{jt} can be recursively defined as [34]

$$\delta_{jt} = \max_{1 \le i \le n} (\delta_{i(t-1)} a_{ij}) b_j(O_t),$$

where a_{ij} is the state transition probability from S_i to S_j, and $b_j(O_t)$ is the probability of emitting the observation symbol O_t at state S_j. In our context of sensor networks, the observation symbol O_t is the vector of emissions by all nodes in the network at the time instant t, and a state corresponds to a low-level event and its location in the network. For instance, in the case of HMM for vehicle trajectories, there is a state for each "grid location" in the network where a vehicle is detected.

A logic program that computes the δ's using the above recursive equation is given in Figure 2(b). The top-level query `mle` evaluates the most likely state sequence over the HMM model encoded by `hmm`. This is done by finding likely state sequences and their probabilities (using `eval`) and, over all such sequences, selecting the one with the maximum probability (`<MAX P>` which selects the maximum `P`, and `<ARGMAX(E,P)>` which selects `E` that maximizes `P`). Predicate `distr` encodes the probability distributions associated with random variables; for instance, in `distr(X,V,P)`, `P` is the probability that random variable `X` has value `V`. It should be noted that evaluation based on caching (such as

```
temp(X0, I, T0) :-
   not sensor(X0),
   PrevI is I-1,
   neighbors(X0, Neighbors),
   neighbortemps(Neighbors, PrevI, NeighborTemps),
   msw(reg(X0, NeighborTemps), T0).

temp(X0, I, T0) :-
   sensor(X0),
   sense(X0, I, S0),
   PrevI is I-1,
   neighbors(X0, Neighbors),
   neighbortemps(Neighbors, PrevI, NeighborTemps),
   msw(sreg(X0, [S0|NeighborTemps]), T0).


neighbortemps([], I, []).
neighbortemps([Xi|Xs], I, [Ti|Ts]) :-
   temp(Xi, I, Ti),
   neighbortemps(Xs, I, Ts).
```

Figure 3: Temperature Estimation Model

the bottom-up evaluation [40]) do not repeat the same computation twice, thereby ensuring that the most likely state sequence is computed in time comparable to Viterbi's dynamic programming algorithm.

For every query result over a logic program, we can construct a derivation, i.e. a proof for that result. Note that there may be more than one proof for a result. In a probabilistic program, if the probability of each derivation is independent of the others, then we can find the probability of a query result as the sum of probabilities of all possible derivations for this result. In the case of the HMM program in Figure 2(a), the different state sequences ending at state `S` at time instant `I` that may result in the given observation sequence correspond to the different proofs for the query `hmm(I, S)`. Hence the probability corresponding to the most likely state sequence for a given observation is the maximum among the probabilities of the associated proofs. The program in Figure 2(b) finds the probability of the most likely proof for the program in Figure 2(a). As the similarity between the two programs suggests, the program to find the probability of the most likely proof can be automatically generated from given PRISM programs and queries.

3. MODELING SENSOR NETWORK APPLICATIONS IN PRISM

In this section we describe PRISM models of three sensor network applications: a temperature estimation model, a trajectory evaluation model based on HMMs, and a model to compute the most reliable communication paths in a network. We focus on the modeling aspects in this section; distributed in-network evaluation of these models form the topic of Section 4.

3.1 A Simple Temperature Estimation Model

We begin with a simple model for temperature estimation in a region of interest. We assume that the area over which we want to estimate temperature is discretized into a (suitably dense) finite set of points. Some (but not all) points in the area have a temperature sensor. We also assume that the temperature values form a finite set. We model temperature at each point (and at each instant of time) by a finite-domain random variable.

We first consider estimating temperature at points that do not have a sensor. If `X` is such a point, we say the temperature at `X` is dependent on the temperature estimates (at the previous time instant) at some surrounding points, called the *neighbors* of `X`. This relationship is modeled as a conditional probability distribution, and the statistical model for temperature estimation is encoded in the PRISM model shown in Figure 3. In the figure, the relation `neighbors` relates each point to the set of its neighbors. The predicate `neighbortemps` accumulates (in its third argument) the temperatures at a set of points (its first argument) at a given time instant (second argument). Thus the variable `NeighborTemps` will hold the value of the temperatures at neighbors of `X0` at time `I-1`. The last predicate, `msw`, in the definition of `temp` states that the temperature at point `X0` is determined by random variable `reg` whose probability distribution depends on the point `X0` and the temperatures at the neighbors.

Now consider estimating temperature at the points with sensors. Note that since a sensor may be faulty, the actual temperature at a point may differ from the sensed value. We hence model the actual temperature at a sensed point as a random variable that is dependent both on the sensed value at that point as well as the estimated temperatures at its neighbors. The second rule in Figure 3 encodes this relationship using `sreg` random variable.

Using this model, one can compute the probability distribution of temperature (and hence the most likely temperature) at any point and at any instant in time.

Discussion: Any graphical model whose network structure forms a directed acyclic graph can be encoded and evaluated in PRISM. The above encoding of the temperature estimation model is one such instance. In general, let X_i be a random variable and *parents*(X_i) be the set of random variables that X_i is conditionally dependent on. Let *values*(X_i) be the set of values of the parents of X_i. Then `msw(rv(`*values*(X_i)`), `V_i`)` models the conditional dependency between X_i and its parents: the value V_i of X_i is given by a PRISM random variable `rv`, whose probability distribution is dependent on the values of X_i's parents. Given a set of facts that define the parent of each random variable X_i (i.e. the edges in the network structure of the graphical model), we can compute the *values* set in the same way we computed the temperatures at neighbors in the above temperature model. Thus we can encode any directed acyclic graphical model along the same lines as the PRISM program for temperature estimation. The PRISM framework can be used to evaluate queries over these models (e.g. value with the maximum probability, probability of a given value, etc.) as long as the dependencies are acyclic.

3.2 HMM-based Trajectory Evaluation

We consider the problem of evaluating, using a sensor network, the trajectory of an object moving over a region of interest. As in the temperature estimation problem, we divide the space over which the object moves into a (sufficiently dense) discrete set of points (called locations). Sensors are placed at some, but not all, locations. We assume that sensed data of each sensor ranges over a finite set.

We model the above problem as an HMM whose states correspond to locations. In other words, the state of the HMM at any instant of time corresponds to the location of the object. When the object is at a given location several sensors may fire. Note that the set of sensors that fire de-

pend only on the location of the object. Hence we do not directly account for interference or dependencies between sensor values. Any dependency is captured as a function of the object's location. Adding the Markovian assumption that the object's location at any given time depends only on its location at the previous instant of time, we naturally obtain an HMM.

The transitions of the HMM will be derived from the movement model of the object. For instance, we can assume that in one time step, the object can either stay in the same location, or move from one location p_0 to a neighboring location p_1. This would translate to a transition from state p_0 to p_1 in the HMM. The set of all sensors that fire at time t_0 is the observation symbol emitted by the HMM at time t_0. Thus the set of observation symbols of the HMM is the power set of observations of all sensor nodes. Note that the set of sensors that fire when an object is at a given location is not deterministic; some sensors may misfire (firing when the object is not nearby) while others may fail to fire. This uncertainty is captured by the emission probability of the HMM state. Also note that communication errors may prevent the observation of a sensor from being reported. The effect of communication errors due to white noise is also rolled into the emission probability.

Thus, our trajectory evaluation model is an instance of the HMM shown in Figure 2(a). Note that in a deployment of the sensor network, the actual location of the object is unknown: hence the states of the HMM are "hidden". What an observer can see are the sets of sensors that fired at each time. From this observation, we would like to reconstruct the position and trajectory of the object. Hence, given a sequence of observations, where each observation is the set of sensors that fired:

- We can compute the answer to query `mle(hmm(I,S),P,Seq)` which would give the most likely trajectory `Seq` of the object up to time I.
- We can compute the likelihood of answer `hmm(I,S)` which will give the probability that the object is at point S at time I.

Note that the above model is centralized: we have assumed that at each step, we observe all the sensors that fire, and proceed with the computation based on this observation. Moreover, the model assumes synchrony, in that the clocks of all sensors advance simultaneously without skew. We describe in Section 4 how we can drop these assumptions, and evaluate "most likely explanation" queries over the generalized model in a distributed fashion in the network itself.

3.3 Estimating the Most Reliable Communication Path

We now consider the problem of finding the most reliable communication path between all pairs of sensors in a sensor network. Observe that the problem of finding the set of all connected pairs of sensors (i.e. those with some path between them) can be solved by using the Snlog program in Figure 1(b). Figure 4 shows the PRISM model where each link is associated with a reliability measure.

Note that the PRISM model is very similar to the Datalog program for computing reachability in graphs (Figure 1(a)). The difference is that we consider in the PRISM model only those links that are up. Whether a link is up or not is

```
reach(S,D) :-
   link_up(S,D).
reach(S,D) :-
   link_up(S,T),
   reach(T,D).

link_up(S,D) :-
   link(S,D),
   msw(status(S,D), up).
```

Figure 4: Most Reliable Path: Simple Model

```
reach(X, Y, I) :-
    link_up(X, Y, I).
reach(X, Y, I) :-
    link_up(X, Z, I),
    reach(Z, Y, I).

link_up(X, Y, I) :-
  node(X, I, up),
  node(Y, I, up),
  link(X, Y).

% HMM model of node failure
node(X, 0, S) :-
    msw(init, S).
node(X, I, S) :-
    node(X, PrevI, PrevS),
    I is PrevI+1,
    I > 0,
    msw(tr(PrevS), S),
    obs(X, I, A),
    msw(out(S), A).
```

Figure 5: Most Reliable Path: a State-Based Model

determined by a random variable `status` associated with that link. Now, each explanation of the query `reach(S,D)` will correspond to a path in the network. The probability of an explanation will be the product, over each link in the path, of the probabilities that the link statuses are up. Thus the most likely explanation for `reach(S,D)` will correspond to the most reliable path in the network.

We can also derive a more elaborate model that takes into account temporal aspects of link failure. Consider a situation where sensor nodes may experience transient failures during which they do not communicate. Such failure modes can be modeled by HMMs. For instance, a simple two-state HMM can be used to model the situation where a node may fail at any time with a fixed probability, and a failed node may recover at any time with a fixed probability. We may not be able to directly observe whether a node has failed or not. However, at each time, a node may send a "hello" message (again with some fixed probability) when it is up. By observing a sequence of hello messages, we will be able to estimate the likelihood of the current state of a node. A link is up only when both the source and destination nodes are up. This model of link failure is encoded in PRISM as shown in Figure 5. Since now the link failure model is state-dependent, the most reliable path between two nodes may change over time. This is represented in the PRISM model by adding a new time parameter to the reach and link predicates.

Note that the PRISM model in Figure 5 is a combination of a statistical model (HMM for node failure) and a logical model (transitive closure over graphs). This model illustrates the power of our approach as compared to those based on Snlog alone (logic-only) or graphical models (statistics-only). This example also illustrates how we can refine an existing model at a high level by adding features (node reliability model, time, etc.).

4. IN-NETWORK EXECUTION OF PRISM MODELS

We describe a methodology for translating a centralized PRISM specification to an Snlog program. This process is done in two steps: (i) distributing the PRISM model and (ii) generating an Snlog program for in-network evaluation of queries over this model (most likely explanation, likelihood of an answer, etc.).

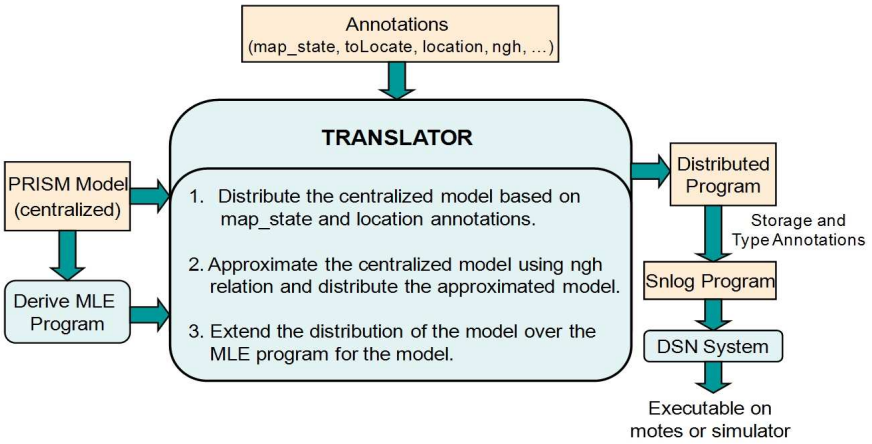


Figure 6: Architecture for In-Network Evaluation of Annotated PRISM Models

The distributed model is obtained by first adding location annotations to the model's predicates. The locations in the distributed model are *symbolic names* given to the computing nodes in a distributed network (such as sensor nodes in a sensor network). Locating a predicate means specifying the computing node where the tuples of that predicate would get evaluated and stored. Our translation also modifies the model using the network structure for localizing communication between the sensor nodes. The network structure is specified in form of a 'neighborhood' relation `ngh(A,B)` meaning that nodes `A` and `B` are neighbors.

In Snlog, location information is mixed with the definition of predicates: one of the arguments of a distributed predicate, distinguished with an "@" symbol, gives the location of the tuple. In contrast, we separate the location definitions from the predicate definitions themselves. This has several advantages. First of all, this separates the logic of the model from its implementation. Secondly, separate location definitions makes it possible to specify this information for some predicates, and automatically infer this information for the rest. Thirdly, queries over PRISM models are implemented using meta programs, i.e., logic programs that view the PRISM programs as data. Separate location definitions lets us infer the location information for meta programs based on the location information of the PRISM models.

To evaluate PRISM models in the network in a distributed fashion, we generate Snlog programs that correspond to the queries of interest. When generating the final Snlog program, we use the location definitions and generate appropriate location annotations ("@" variables in Snlog). The distributed PRISM model may also carry additional annotations used by Snlog, including type declarations (relation name, types of parameters, primary keys) and materialization policies (how long the tuples are valid, number of tuples stored in a relation, etc.), suppressing rule execution, and/or prioritizing tuple processing.

The architecture of our methodology and the steps involved in the translation of PRISM models to distributed Snlog programs are summarized in Fig. 6.

4.1 An illustration of translation methodology: the HMM example

We describe our methodology for translation of PRISM specifications to Snlog through the HMM example.

HMM Trajectory Evaluation We adapt the PRISM model of HMM (Fig. 2(a)) for evaluating trajectories as follows. The sensed data `An` from each sensor `N` at time `I` is represented by a predicate of the form `sensor(N,I,An)`. From this data, we build the observation at time instant `I` using the following definition for `obs(I,A)`:

```
obs(I, <VEC(N, An)>) :-
    sensor(N, I, An).
```

Here the aggregate operation `VEC` is used to construct a vector of all sensor readings. The complete definition of the HMM with this definition of `obs` is in Fig. 7(a).

Global

```
hmm(0, S) :-
    msw(init, S).

hmm(I, S) :-
    hmm(PrevI, PrevS),
    I is PrevI + 1,
    I > 0,
    msw(tr(PrevS), S),
    obs(I, A),
    msw(out(S), A).

obs(I, <VEC(N,An)>) :-
    sensor(N, I, An).
```

(a)

Local

```
hmm(0, S) :-
    msw(init, S).

hmm(I, S) :-
    hmm(PrevI, PrevS),
    I is PrevI + 1,
    I > 0,
    msw(tr(PrevS), S),
    local_obs(S, I, A),
    msw(out(S), A).

local_obs(S, I, <VEC(N,An)>) :-
    map_state(S,L),
    ngh(L,N),
    sensor(N, I, An).
```

(b)

Figure 7: HMM models for trajectory evaluation (under synchrony assumption).

From this basic centralized model, we derive a distributed model of the HMM as follows. We first add location definitions for predicates in our model. We define a predicate called `map_state` to map states of the HMM to the computing units (nodes) in the network. A tuple of the form `map_state(S, N)` means that the state `S` of the HMM is mapped to node `N` in the network. Using `map_state`, we specify the location of `hmm` tuples as follows:

```
location(hmm(I,S), N) :- map_state(S, N).
location(sensor(N,I,An), N).
```

The above location specifications state that tuples of the form `hmm(I,S)` will be computed and stored at node `N`; and

Synchronous

```
hmm(Time, S) :-
      clock(0, Time),
      msw(init, S).

hmm(Time, S) :-
      hmm(PrevTime, PrevS),
      clock(PrevT, PrevTime),
      T is PrevT + 1,
      T > 0,
      msw(tr(PrevS), S),
      clock(T, Time),
      local_obs(S, Time, A),
      msw(out(S), A).

local_obs(S, Time, <VEC(N,An)>) :-
      map_state(S,L),
      ngh(L,N),
      sensor(N, Time, An).
```

(a)

Asynchronous

```
hmm(Time, S) :-
      map_state(S,N),
      localClock(N, 0, Time),
      msw(init, S).

hmm(Time, S) :-
      hmm(PrevTime, PrevS),
      map_state(PrevS, L),
      localClock(L, PrevT, PrevTime),
      T is PrevT + 1,
      T > 0,
      msw(tr(PrevS), S),
      map_state(S,N),
      localClock(N, T, Time),
      local_obs(S, Time, A),
      msw(out(S), A).

local_obs(S, Time, <VEC(N,An)>) :-
      map_state(S,L),
      ngh(L,N),
      localClock(L, T, Time),
      localClock(N, T, NTime),
      sensor(N, NTime, An).

location(localClock(L, T, Time), L).
```

(b)

Figure 8: Synchronous and Asynchronous HMM models with explicit time.

that the sensed values will be stored at the sensor itself. By default, the other predicates used in the specification of the PRISM model will be located at all sensor nodes.

Note that definition of `obs` means that every sensed value needs to be known to every sensor node. However, each sensor can sense only over a relatively small area; hence only a set of sensors near an object are likely to report non-zero readings. We can hence derive an alternative HMM model by considering only locally observed sensor readings. We do this by defining a predicate `local_obs`, using the structure of the sensor network. We assume that the predicate `ngh` gives the neighborhood information for each node. Using `ngh`, we define `local_obs` as follows:

```
local_obs(S, I, <VEC(N,An)>) :-
      map_state(S, L),
      ngh(L, N),
      sensor(N, I, An).

location(local_obs(S, I, A), L) :-
      map_state(S, L).
```

While the first rule above defines the `local_obs` predicate, the `location` rule specifies where the tuples of this predicate will be located. With this definition, observe that each node needs to know only the sensed values of neighboring sensors. The distributed program after this step is given in Fig. 7(b). This model ensures that sensed values are communicated only to neighboring nodes, and hence is suitable for distributed evaluation.

It must be noted that while the distributed model may localize communication, a single sensor node may have a large number of "neighbors" if the sensing regions of many nodes overlap, i.e. if the nodes are densely packed. Note also that the correlation between neighboring sensors' readings is naturally captured by this model. In this model, the sensor readings (at a given time) at neighboring nodes are not directly dependent on each other but are linked via the hidden state variable in the HMM.

The above model still assumes that all sensors are synchronous in the sense that they share a single global clock (hence the shared time instance `I`). In a sensor network, the sensor nodes operate independent clocks. To faithfully evaluate this model, we would need that all these clocks be synchronized, perhaps using a distributed clock synchronization protocol. Alternatively, we can develop an asynchronous model that can be evaluated without clock synchronization.

4.2 Adding time to the model

We logically characterize asynchronous computation in our model by first explicitly specifying a global clock, and then modifying the specification to use local (asynchronous) clocks. Temporal models such as HMMs represent transitions in time; hence time is an integral part of the model. In the previous subsection (Sec. 4.1) the parameter `I` in `hmm(I, S)` represented the logical time for the model and assumed synchrony among the nodes in the distributed model (all nodes agreed on same logical time I at any point in time). Instead of using a logical clock, we turn this parameter into an explicit global clock.

Assume there is a global clock and its value at discrete time instants is given through the relation `clock(T, Time)`, where `T` is a logical time instant and `Time` is the actual global clock time at that instant. The synchronous HMM model with explicit time is given in Fig. 8(a). The model `hmm(I, S)` without an explicit actual clock is now written as `hmm(Time, S)` in the timed model where `clock(I, Time)` holds.

The explicit-time model reveals where the assumption of synchronicity comes in, and how this assumption can be dropped. In particular, consider the predicate `local_obs(S, Time, Obs)` which is used to collect, for a given state `S` and

global time `Time`, the set of observations `Obs` at its neighbors. This set of observations is taken at the same time instant `Time`. In an asynchronous model, the observations at each sensor may be done at different points in time, and moreover, each sensor node may have its own notion of time. We model this by associating a local clock with each sensor node, and relating these local clocks via "global logical time", which can be considered as a clock at a particular frame of reference. We use a predicate called `localClock(L, T, Time)` to relate the location `L` of a clock and its value `Time` to a fixed global logical time `T`. This relation, `localClock`, is *locally consistent*, i.e., given two tuples (l, g_1, t_1) and (l, g_2, t_2), $g_1 \geq g_2 \Longrightarrow t_1 \geq t_2$ and *communication consistent*, i.e., if a message is sent from l_1 at local time t_1 and received by l_2 at local time t_2, then for all tuples (l_1, g_1, t_1) and (l_2, g_2, t_2) in the `localClock` relation, $g_1 < g_2$.

Using this `localClock` relation, we can rewrite the `local_obs` predicate as:

```
local_obs(S, Time, <VEC(N,An)>) :-
    map_state(S,L),
    ngh(L,N),
    localClock(L, T, Time),
    localClock(N, T, NTime),
    sensor(N, NTime, An).
```

In the above definition, `Time` is the local clock at location `L`, and `NTime` is the local clock at a neighbor `N`. The relationship between the two local clocks is that they represent the same global logical time instant `T`. The asynchronous model of HMMs is given in Fig. 8(b).

It should be noted that `localClock` is a modeling notion that lets us characterize the possible computations in an asynchronous deployment. In the final deployment, we omit the explicit use of `localClock`, using the latest set of sensor readings as the observation vector at the current instant.

4.3 Maximum Likelihood Evaluation (MLE) over the Distributed HMM Model

The maximum likelihood evaluation (MLE) program can be automatically generated for a PRISM model as illustrated previously in Section 2. Given an HMM model and the MLE program over the model, we add the following annotations to the set of annotations used for the distribution of the model, and use our translation methodology to generate a distributed program for the MLE program.

```
modelParam(mle(M,A1,A2), M).
modelParam(eval(M,A1,A2), M).
location(Pred, S) :- modelParam(Pred, M),
                     location(M,S).
```

Note that the `mle` and `eval` predicates operate over the HMM model, treating it as data, and hence the location of `mle` and `eval` tuples are taken from the location of the underlying `hmm` tuples. The MLE program for the asynchronous distributed HMM model, using the explicit location annotation as in Snlog, is shown in Fig. 9.

Using the above translation methodology we derive distributed programs for MLE over PRISM models. The distributed version of an MLE program for a PRISM model is converted to an Snlog program by adding type declarations and materialization policies for the predicates in the distributed program.

```
mle(@N, hmm(Time, S), <MAX(P)>, <ARGMAX(E,P)>) :-
    eval(@N, hmm(Time, S), P, E).

eval(@N, hmm(Time, S), Prob, S) :-
        map_state(S, N),
        localClock(@N, 0, Time),
        distr(init, S, Prob).

eval(@N, hmm(Time, S), Prob, PrevS) :-
        mle(@L, hmm(PrevTime, PrevS), PrevP, PrevE),
        map_state(PrevS, L),
        localClock(@L, PrevT, PrevTime),
        T is PrevT + 1,
        T > 0,
        distr(tr(PrevS), S, TransP),
        TransP > 0,
        map_state(S, N),
        localClock(@N, T, Time),
        local_obs(@N, S, Time, A),
        distr(out(S), A, OutP),
        Prob is PrevP * TransP * OutP.
```

Figure 9: MLE over HMM model with explicit time.

4.4 Generality of the Approach

We have illustrated our approach for deriving sensor network programs from high-level models using the PRISM model of HMM as an example. The HMM example was used to illustrate the manual steps that can be taken to derive a distributed asynchronous model from a simple centralized model. It should be noted that models for most sensor applications, including the temperature sensing example in Section 3, are inherently distributed. For such models, we need to only specify the location annotations. Once we have an annotated PRISM model, the program for evaluating maximum likelihood evaluation (MLE) queries over it can be automatically generated. The MLE program, in turn can be be automatically translated into an Snlog program that can then be compiled and deployed for in-network evaluation. For instance, for the "most reliable path" example described in Section 3 (Fig. 5), the following location definitions are sufficient:

```
location(reach(X,Y,I), X).
location(link_up(X,Y,I), X).
location(node(X,I,S), X).
location(obs(X,I,A), X).
```

First of all, note that the models are inherently distributed and hence the location definitions above will result in local communication. Secondly, the model is still synchronous (e.g. `link_up` needs the state of two different nodes at the same time instant). We can derive an asynchronous model following the same steps we took to make the HMM model asynchronous. Once we introduce local clocks, we can generate the `MLE` query over this model and evaluate it in a distributed fashion in the network itself.

5. EXPERIMENTAL RESULTS

Our trajectory detection experiments were based on the distributed asynchronous model for HMM in PRISM shown in Fig. 8(b). We automatically derived the program to evaluate maximum likelihood evaluation (MLE) queries over this model, shown in Fig. 9. This program was then automatically translated into Snlog, and added Snlog-specific type and materialization declarations. (These declarations can

be easily lifted to the PRISM model level as well and then derived by automatic translation, but this was not done in our current implementation). The DSN system [4] was then used to derive nesC [13] code from the Snlog program. The generated code was compiled using the nesC compiler and with the addition of DSN runtime support a binary image was generated for execution using TinyOS [7]. Due to unavailability of interfaces to light sensors in the DSN system, we could not directly use binary image generated using DSN for light-sensor network deployment; we hand-generated a nesC program from the intermediate Snlog program and used that for deployment. We used both the hand-coded and automatically-generated nesC programs in our simulation experiments, and the two programs produced nearly identical results, indicating that our deployment results will extend to the automatically-generated programs too.

The HMM model parameters used in our simulations and experiments include: *initial state probabilities:* any of the states on the borders of the grid is equally likely to be the initial state; *state transition probabilities:* from a given state all of the neighboring states (1-grid point away in our setting), and the state itself are equally likely to be the next state; and *emission probabilities:* for the deployment experiment the emission probabilities were learnt using labeled data from a real network setting (the same as used for our deployment experiments), and in our simulation setup nodes emit binary observations which are correct with probability 0.95.

5.1 Simulation

5.1.1 Setup

In this section we describe the different network scenarios and settings used to evaluate the HMM-based trajectory detection model as described in Sec. 3.2. The states in the HMM are mapped to sensors in the network. We consider two scenarios: when the mapping is 1-1, i.e. the number of states and sensor nodes are same, and when there are more states than sensor nodes in which case the mapping from states to sensors is many-to-one. In the latter case, a node is responsible for performing computation for more than one state in the model.

In general, lossless data transmission protocols are necessary to ensure that the results computed by distributed inference is same as that computed by a centralized inference algorithm. In a probabilistic setting, since the information being exchanged are estimated values, simpler, perhaps lossy, data transmission protocols suffice. When developing a model, errors due to communication failure can be rolled into the probability distributions in the model. However, an iterative model like HMM leads to significant contention for medium since many nodes may attempt to communicate simultaneously. We hence experimented with the following communication modes to analyze their effect on error rates:

- Mode 0: all nodes send all their messages in every iteration.
- Mode 1: messages are sent selectively i.e., a message is sent only when its present observation results in a high probability value that the node's state is the current state. Roughly speaking, only the more informative nodes send as in [37]. This mode reduces bandwidth and contention, and consequently reduces communication error rates.
- Mode 2: nodes corresponding to highly likely states resend their messages after performing Mode 1 of communication. This "double send" mode increases bandwidth compared to Mode 1, but significantly reduces the possibility that important messages are lost.

Mode	#Msgs	Msg Loss Rate %	Error Rate %	Std Dev
0	558720	7	11.20	0.75
1	19400	3	7.80	1.17
2	27152	2	0.60	0.49

Table 1: Effect of Communication Modes on Detection Error Rates

We exploit the fact that we compute probability values, and hence our computations are relatively tolerant to communication errors. In contrast, Chu et al's [3] approach uses replicated dynamic probabilistic models to minimize communication from sensor nodes to the network's base station. The main idea is to exploit spatial correlations across sensor nodes without imposing unnecessary communication between sensor nodes. Baysail [39] uses a Bayesian approach to estimate missing values as precisely as possible, as well as estimate the process parameters for the model generating those values. While it is possible to obtain more accurate inference without excessive communication costs using either of these approaches, our experimental results given below indicate that simple threshold-based message suppression is sufficient to maintain low error rates for probabilistic inference.

The robustness of our approach is validated in the presence of clock drifts and node failures— two factors that are not considered by the PRISM model. We considered both transient and permanent node failures, where transient failures may be due to skips (where a node becomes inactive without losing state information) or reboots (where a failing node becomes active again after a reboot).

We also experimented with different network topologies: a regular grid topology as well as random topologies.

5.1.2 Results

We performed simulations for our generated Snlog program for maximum likelihood evaluation over an HMM model for trajectory detection of a moving object in grid-structured and random topology 144-node sensor network. The simulations were performed using TOSSIM [21]. In order to validate our deployment results, we performed simulation on two types of programs, Snlog compiled using DSN, and Snlog hand-converted to nesC. The trajectories are given in terms of the grid points of the grid overlaying the network. For simulation, the sensor readings (binary 0 or 1) were generated based on the original trajectory of the object. The error rates in trajectory detection are computed as the percentage of points in the computed trajectory that do not match with the points in the original trajectory. The message loss rates mentioned in the following results indicate the number of application messages that were sent over the network but not received. Message loss rate is computed using the message reception rate which we calculate as the

Nodes %	Permanent Failure		Temporary Failure			
			Skip 10 iterations		Reboot in any iteration	
	Error Rate %	Std Dev	Error Rate %	Std Dev	Error Rate %	Std Dev
20	28.20	0.84	3.67	0.82	1.33	0.52
15	18.80	0.83	2.83	0.98	1.27	0.46
10	14.60	0.89	1.40	0.89	1.20	0.44
7	7.83	0.75	1.02	0.71	0.8	0.47

Table 2: Effect of Node Failure on Detection Error Rates

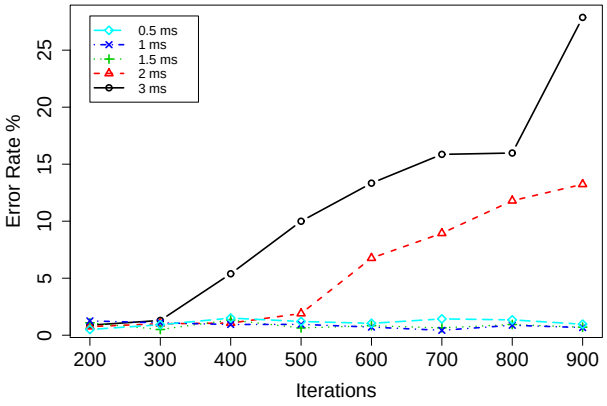


Figure 10: Effect of Clock Drift (20% of nodes with clock drifts, in millisec. per sec.; 1 iteration = 1 sec.)

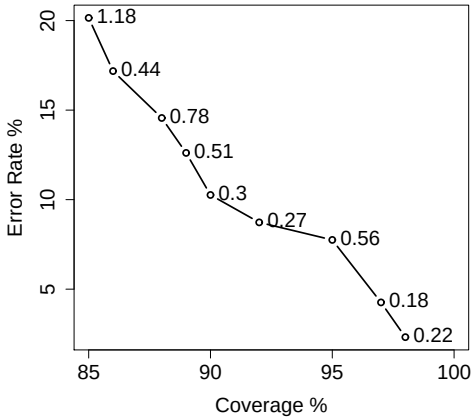


Figure 11: Effect of Sensor Coverage (Random topologies)

Mode	Event 1		Event 2	
	Error Rate %	Std Dev	Error Rate %	Std Dev
0	13.6	0.55	12.8	1.4832
1	9.8	0.84	8.6	1.34
2	3.8	1.3	3.6	0.55

Table 3: Simultaneous Detection of Two Events

total number of application messages received by all nodes in the network divided by the total number of application messages sent over the network. Overall, Snlog compiled using DSN, and hand-converted nesC programs showed very similar results in simulation for computing trajectories. Each simulation had 1000 iterations, i.e. the trajectory length was 1000 points.

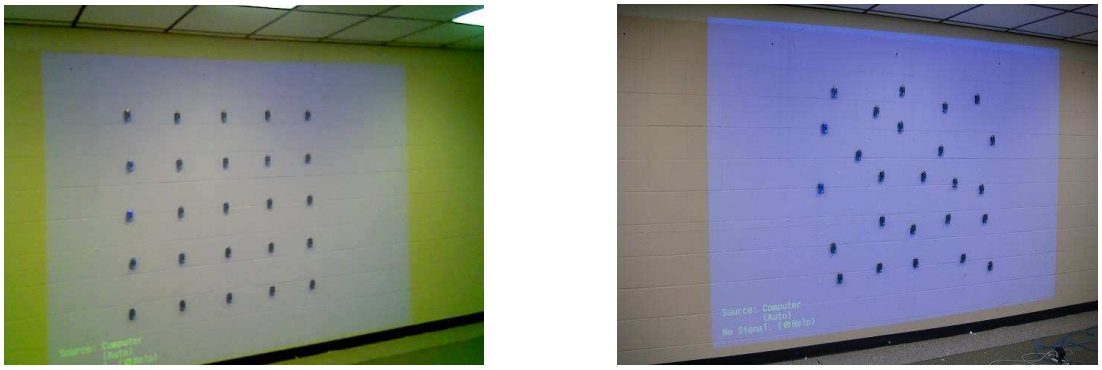
Unless otherwise specified, all results were taken using the distributed HMM model, for detecting a single object in the region, with one sensor node per point (1-1 mapping). The results were obtained over 50 simulation runs.

Table 1 gives simulation results for the correctness of our trajectory detection programs (generated Snlog compiled using DSN and hand-converted nesC) for a single moving object under the three different modes of communication described in the setup. The error rate and its standard deviation, total number of application messages sent, and message loss rate are given in the table. In mode 1 and mode 2, the number of transmitted messages are reduced and are thus suitable for energy-constrained sensor nodes. The error rates are also less for the mode 2 communication strategy as evident from results in Table 1. This suggests that mode 2 communication strategy is lightweight in terms of message communication, and is more accurate because important messages, i.e. those that contribute most to the answer for the application that is being evaluated, are less likely to be lost. We also measured the energy consumption for radio and CPU for our application using PowerTOSSIM [38] for the TelosB [32] energy model. The energy consumed for radio and CPU for Mode 0 communication was 2391.6 mJ and 6.522 mJ, respectively for 60 seconds of simulation.

Table 2 presents simulation results for trajectory detection of a single event in the presence of node failures (up to 20% of randomly chosen nodes fail) at random times during the evaluation. Nodes may fail either permanently (die) or temporarily (reboot or transient failures). The event detection algorithm (maximum likelihood evaluation over HMM Model) is resilient to temporary node failures. Interestingly, it appears that temporary failures due to reboots contribute to fewer errors than failures due to skipping of iterations. In case of permanent node failures, the error rates grow higher with increase in the percentage of permanent node failures.

We also experimented with clock drifts to observe their effect on error rates (Fig. 10). Clocks of 20% randomly chosen nodes were allowed to drift over time. We observed that slow rate of clock drifts does not have an adverse effect on the accuracy of the results. As expected, over longer periods of simulations, the drifts get accumulated and increase the error rates.

Results of our simulations with random topologies are shown in Fig. 11. The topologies were randomly generated and the error rates were measured for topologies with varying sensing coverage. The uncovered areas spanned over few adjacent grid points in the region. The error rates are pro-



(a) Regular topology (b) Random topology

Figure 12: Sensor setup: Showing 25 sensors deployed on a wall.

portional to the percentage of uncovered area. The numbers next to the points of error rates in the graph of Fig. 11 represent the standard deviation for the error rates.

Table 3 evaluates the correctness of trajectory detection for two simultaneously moving objects within a region. The column headings 'Event1' and 'Event2' in Table 3 indicate two separate events (moving objects) being simultaneously detected by the maximum likelihood evaluation over two HMM models, one model for each event. The error rates are slightly higher than that for single-object trajectory detection but are reasonably low.

The above simulation results were for network setting when the number of HMM states modeling the network and the number of sensor nodes are equal and the mapping from states to nodes is one-to-one. We also evaluated trajectory detection for many-to-one mapping of states to sensor nodes. The trajectory is represented at a finer level when the states in the model are more than the sensor nodes in the network. For many-to-one mapping (4-to-1 mapping in this case) of states to sensor nodes, mode 2 communication and single event detection, the message loss rate is 2.5% and error rate is 10.8% (with 1.13 standard deviation).

These simulation results suggest that our approach to modeling sensor networks using probabilistic models and performing inference over these models using maximum likelihood evaluation is feasible and robust to network failures.

5.2 Deployment

5.2.1 Setup

In our setup for trajectory evaluation, we used a simulated light source as the object to be tracked. The tracking region was a square with side of 63 inches on a wall. We superimposed a 10×10 grid in this region, so that the distance between two grid points was 7 inches. A light source was created with the help of a powerpoint slide. Two concentric circles of increasing radii, 5.25 inches and 16 inches, having increasing intensity of grey color radially outwards, were drawn on the slide. The rest of the slide was dark grey. The slide was projected (using a projector placed 16 feet infront of the wall) over the region on the wall so as to simulate a light source. A trajectory of the light source was formed by having the circular object change its positions on the slide. The experiments were done in a dark room and the projected slide on the wall was the only light source in the room. The average sensor reading for the light source centered over a mote was 98. The average reading 14 inches (2 grid points) away was 80, and was below 20 further away from the center. The light intensity of the light source at various grid positions is shown in Fig. 13.

The light source (object) randomly transitioned from one grid point to a neighboring grid point at 1 second intervals over the 10×10 grid. The trajectory lengths were set to 35.

We experimented with two sensor network deployments— regular and random topologies— with 25 TelosB motes in the region (i.e. attached to the wall). For regular topology, the motes were laid out evenly spaced through the region forming a 5×5 grid of sensors (see Fig. 12(a)). For random topology, the motes were laid out randomly within the region.

As mentioned before, since the current implementation of Snlog did not provide light sensor interfaces, we used a hand-converted nesC program in the deployment experiments. As remarked in the previous subsection, the automatically-derived Snlog program and the hand-converted nesC program gave similar results using simulation. The program deployed on the motes evaluated, in-network, the most likely trajectory of the light source in the region. It used the HMM model for trajectory evaluation with 100 states (10 points). The nesC program for trajectory evaluation used in deployment experiments occupied 20.5 KB ROM (code) and 3.54 KB RAM (data) on a TelosB mote.

The projected light source was moved over the region, while the motes were running an iterative MLE algorithm. During an iteration (1000 ms duration), each mote senses light (called observation), transmits its observation and the probability values for its mapped points from the previous iteration to its neighboring motes in real time. It then performs local computation over received observations and probability values from its neighboring motes. An observation for a mote is computed by averaging over 20 sensor readings, each sampled every 10 ms. The sensor observation was then converted to binary 0 or 1 (based on a threshold value 75), for use in local computation in every iteration. The sensors communicated data to each other in every iteration in a TDMA manner to avoid collisions. At the end of each experiment all the motes sent their computed data to a basestation (mote) connected to a laptop, and the collected data was analyzed to estimate the trajectory of the moving object.

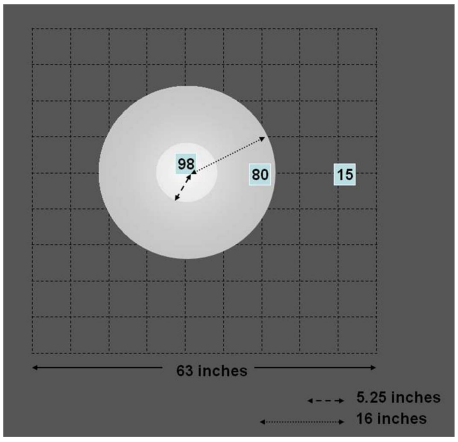


Figure 13: Simulated light source (showing decrease in light intensity radially outwards).

For the regular topology, the mapping between states and motes (sensors) is shown in Fig. 14(a). There are 100 points and each point models a state in the HMM model and the motes (sensors) are marked as encircled numbers 1-25. Each mote is responsible for performing computation and real-time communication for 4 states in the model (marked within a rounded square in Fig. 14(a)). For random topologies, the mapping between states and motes (sensors) was such that each mote performs computation for the states corresponding to the four nearest grid points to the mote (illustrated in Fig. 14(b)). This might lead to some states being mapped to multiple motes and some states unmapped.

5.2.2 Results

We ran 20 experiments, for mode 2 communication, where nodes transmit selectively. For each experiment, we compared the original trajectory with the computed trajectory For trajectories of length 35 grid points, with the regular topology, the computed trajectory differed from the original trajectory on average in 4 grid points. The differences were off-by-one errors, with the predicted point and the actual point being next to each other on the 10×10 grid. The deviations in the computed trajectory points did not have any adverse effect on the entire computed trajectory because the deviations continued for, on average, only one grid-point, and in the worst case, two grid points. Fig. 14(a) shows an example of comparison between the original and computed trajectories. The original trajectory is marked by dashed lines and the computed trajectory is marked by solid lines with arrows on the lines indicating the direction of the trajectories. In this figure, the two trajectories differ in 2 points. The areas marked by a 'star' symbol in Fig. 14(a) indicate the regions where the computed trajectory deviates from the original trajectory. For random topologies, the error rates were slightly higher than for the regular topologies. The error rates differed mostly because of the uncovered regions on the grid. An example of the same original trajectory as Fig. 14(a) which was computed with a random sensor node deployment is shown in Fig. 14(b). The bigger 'star' symbols indicate higher deviations in the computed trajectory from the original one. The average error rates using grid-based topology and uniform random topologies were 11.34% and 15.26%, respectively.

Overall, the experimental platform performed within 5% of the simulated scenarios. For example, for grid-based topologies, mode 2 communication and 4-to-1 mapping of the model states to sensors, the average error rate for trajectory detection in experiments was 11.34%, comparable with the average error rate of 10.8% for simulations. For random topologies also, the error rates observed through experiments were of the same magnitude as that observed in simulations.

6. RELATED WORK

The importance of programming sensor networks at a high-level has given impetus to research on this topic spanning a gamut of approaches such as operating system prototypes [7, 13], programming abstractions [41, 42], procedural languages [15] and distributed database frameworks [2, 24].

A high-level declarative programming paradigm has been used recently for specification of network routing protocols [23], overlay architectures [22] and programming sensor networks in functional [27] and logic programming styles [6, 5]. In particular the latter [6, 5] describes Snlog, a dialect of the logic programming language Datalog, for declarative programming of sensor networks.

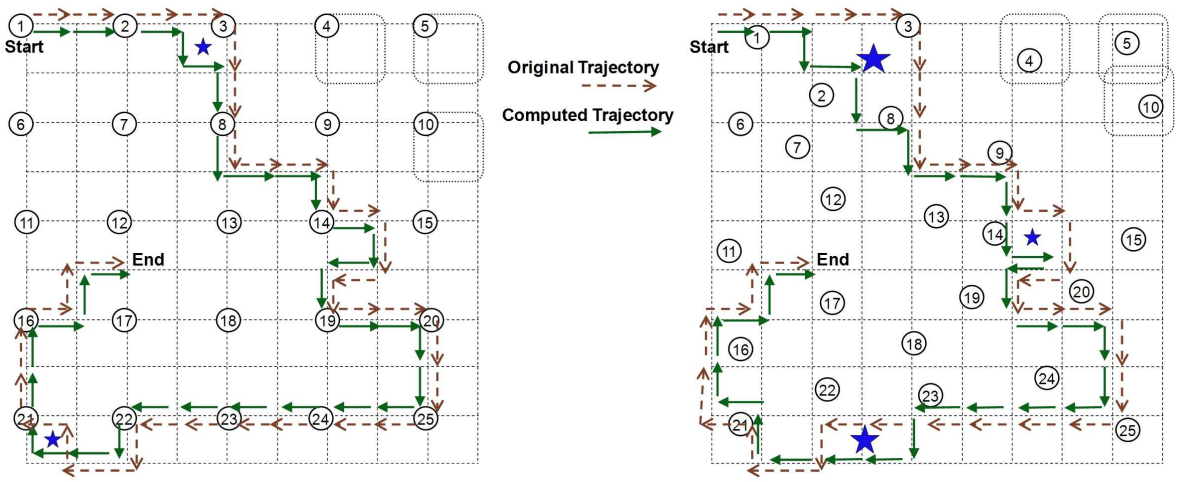
In the context of sensor networks, data is inherently noisy in nature and the nodes are prone to failures, and thus, it is important that the programming approach facilitate reasoning with uncertain or noisy data. The aforementioned works on programming sensor networks do not address the uncertainty aspects of sensor data.

Probabilistic models for dealing with noisy sensor data and doing in-network inferencing over these models have been reported (e.g. [37, 30]). But the use of a programming paradigm that allows the specification of statistical models has not yet been explored for programming sensor networks.

A number of proposals combining logic with probability have appeared in the research literature [1, 16, 29]. The primary focus of these early works was on understanding the semantic underpinnings of the combination. Consequently, questions about programming using probability-logic combination formalisms were not addressed. Recently Markov Logic Networks (MLN) have been proposed as a formalism for the integration [35]. But there is no programming infrastructure for computing with MLNs, and programming in full FOL is not practically feasible, especially for non-trivial real-life applications.

Several proposals incorporating probabilities into logic programs have appeared [8, 9, 20, 26, 28, 33]. But all of these works are limiting in their scope. For instance [8, 20, 28] impose syntactic restrictions on the programs; [33] imposes the acyclicity condition on the clauses; the range restrictions imposed in [26] exclude even simple predicates such as membership. In contrast the PRISM language [36] does not impose such restrictions. Moreover in PRISM the probability distributions as embodied by the `msw` predicates can be learned [17].

IBAL [31] is a functional language for specifying and combining statistical models, and supports parameter learning, and provides constructs for direct specification of a large class of graphical models. BLOG [25] is a language based on Bayesian logic that can be used for building probabilistic models with unknown components. These languages can be used to encode statistical models that cannot be encoded in PRISM. However, they focus on the probabilistic aspects of



(a) Regular topology (b) Random topology

Figure 14: Example original and computed trajectory

modeling and programming, but do not attempt to combine non-probabilistic reasoning (deduction) with probabilistic reasoning. In contrast, our PRISM-based approach can be used to uniformly treat probabilistic problems (such as those described in this paper) and non-probabilistic ones (e.g. spanning tree construction).

There has also been work on relational learning for modeling uncertainty [12, 14, 10, 19]. These works generalize the propositional Bayesian networks to first order, but logical reasoning is not their focus.

7. CONCLUSION

In this paper we described a methodology for the specification and in-network evaluation of statistical models encoding the transformation of low-level noisy sensor network data into high-level knowledge of activities in the network. It is based on the PRISM language, which integrates both logical and statistical reasoning within one declarative programming framework. Any statistical model whose network (conditional dependency) structure is a directed acyclic graph can be encoded as a PRISM program. We illustrated the utility of our approach using three non-trivial sensor network problems. We have constructed a tool to automatically generate Snlog programs from annotated PRISM models, which can then be directly deployed in sensor networks. Experimental evaluation through simulations and deployment provide evidence that our approach is feasible and robust to noisy sensor data. The results from the hardware deployment confirmed our simulation analysis and performed within 5% of the simulation results.

There are several avenues for future research along the lines pursued in this paper. We mention two of them here: The PRISM specifications and their evaluations described in this paper illustrate the encoding of statistical models in a logic framework. But the full power of logical reasoning has not yet been brought to bear on these models – e.g. methods to reason with missing and inconsistent information. Doing so will truly integrate logical and statistical reasoning and get the "best of the two worlds". The other problem concerns the estimation of the probability distributions. In PRISM such estimations, represented by the msw predicates are done within the logic using a generalization of the Expectation-Maximization algorithm [11]. An interesting and intriguing topic will be the translation of this algorithm to do in-network learning of these distributions from sensor data.

Acknowledgements

We thank Himanshu Gupta for his valuable insight and contributions to this project. We also thank the Sensys'08 referees and program committee for critical comments that helped improve this paper. This work was supported in part by NSF grants CNS-0627447 and CNS-0721665.

8. REFERENCES

[1] F. Bacchus, A. J. Grove, J. Y. Halpern, and D. Koller. From statistical knowledge bases to degrees of belief. *Artif. Intell.*, 87(1-2):75–143, 1996.

[2] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Lecture Notes In Computer Science*, volume 1987, pages 3–14, 2001.

[3] D. Chu, A. Deshpande, J. M. Hellerstein, and W. Hong. Approximate data collection in sensor networks using probabilistic models. In *22nd International Conference on Data Engineering (ICDE)*, pages 48–60, 2006.

[4] D. Chu et al. Declarative Sensor Networks. http://db.cs.berkeley.edu/dsn/, 2007.

[5] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *5th ACM Conference on Embedded networked Sensor Systems (SenSys)*, pages 175–188, 2007.

[6] D. Chu, A. Tavakoli, L. Popa, and J. Hellerstein. Entirely declarative sensor network systems. In *Demo session in VLDB*, 2006.

[7] D. Culler et al. TinyOS. http://www.tinyos.net, 2004.

[8] J. Cussens. Loglinear models for first-order probabilistic reasoning. In *Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 126–133, 1999.

[9] J. Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245–271, 2001.

[10] C. M. D. Heckerman and D. Koller. Probabilistic entity-relationship models, PRMs, and plate models. In

ICML-2004 Workshop on Statistical Relational Learning and its Connections to Other Fields, pages 55–60, 2004.

[11] A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Royal Statistical Society*, 39(1):1–38, 1977.

[12] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1300–1309, 1999.

[13] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, pages 1–11, 2003.

[14] L. Getoor, N. Friedman, D. Koller, and B. Taskar. Learning probabilistic models of link structure. *J. Mach. Learn. Res.*, 3:679–707, 2003.

[15] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 126–140, 2005.

[16] J. Y. Halpern. An analysis of first-order logics of probability. *Artif. Intell.*, 46(3):311–350, 1990.

[17] Y. Kameya and T. Sato. Efficient EM learning with tabulation for parameterized logic programs. *Computational Logic*, pages 269–284, 2000.

[18] Y. Kameya, N. Ueda, and T. Sato. A graphical method for parameter learning of symbolic-statistical models. In *Discovery Science*, pages 264–276, 1999.

[19] D. Koller and A. Pfeffer. Learning probabilities for noisy first-order rules. In *15th International Joint Conference on Artificial Intelligence*, pages 1316–1321, 1997.

[20] L. Lakshmanan and F. Sadri. Probabilistic deductive databases. In *International Symposium on Logic Programming*, pages 254–268, 1994.

[21] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire tinyOS applications. In *1st international conference on Embedded networked sensor systems (SenSys)*, pages 126–137, 2003.

[22] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev.*, 39(5):75–90, 2005.

[23] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *SIGCOMM*, pages 289–300, 2005.

[24] S. R. Madden, J. M. Hellerstein, and W. Hong. TinyDB: In-network query processing in TinyOS. http://telegraph.cs.berkeley.edu/tinydb, 2003.

[25] B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: probabilistic models with unknown objects. In *IJCAI*, pages 1352–1359, 2005.

[26] S. Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in inductive logic programming*, pages 254–264, 1996.

[27] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *International Workshop on Data Management for Sensor Networks (DMSN)*, pages 78–87, 2004.

[28] R. Ng and V. Subramanian. Probabilistic logic programming. *Information and Computation*, 101:150–201, 1992.

[29] N. J. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28(1):71–88, 1986.

[30] M. Paskin, C. Guestrin, and J. McFadden. A robust architecture for distributed inference in sensor networks. In *4th international symposium on Information processing in sensor networks (IPSN)*, pages 55–62, 2005.

[31] A. Pfeffer. IBAL: A probabilistic rational programming language. In *IJCAI*, pages 733–740, 2001.

[32] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *4th international symposium on Information processing in sensor networks (IPSN)*, pages 364–369, 2005.

[33] D. Poole. Probabilistic horn abduction and bayesian networks. *Artificial Intelligence*, 64:81–29, 1993.

[34] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Readings in speech recognition*, pages 267–296. Morgan Kaufmann, 1990.

[35] M. Richardson and P. Domingos. Markov logic networks. *Mach. Learn.*, 62(1-2):107–136, 2006.

[36] T. Sato and Y. Kameya. Prism: A symbolic-statistical modeling language. In *Fifteenth International Joint Conference on Artificial Intelligence*, pages 1330–1335, 1997.

[37] J. Schiff, D. Antonelli, A. G. Dimakis, D. Chu, and M. J. Wainwright. Robust message-passing for statistical inference in sensor networks. In *6th international conference on Information processing in sensor networks (IPSN)*, pages 109–118, USA, 2007.

[38] V. Shnayder, M. Hempstead, B. rong Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *2nd International Conference on Embedded networked sensor systems (SenSys)*, pages 188–200, 2004.

[39] A. Silberstein, G. Puggioni, A. Gelfand, K. Munagala, and J. Yang. Suppression and failures in sensor networks: a bayesian approach. In *33rd international conference on Very large data bases (VLDB)*, pages 842–853, 2007.

[40] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., 1990.

[41] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *ACM Symposium on Networked Systems Design and Implementation (NSDI)*, pages 29–42, 2004.

[42] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *International Conference on Mobile Systems, Applications, and Services*, pages 99–110, 2004.