

Contrail Filtering: A Mechanism for Efficient and Robust Activity Recognition

Katherine Deibel*
University of Washington,
Department of Computer Science & Engineering,
Seattle, WA, USA
deibel@cs.washington.edu

Abstract

Recognizing user actions is increasingly important in many ubiquitous computing applications, particularly for elderly care. Current probabilistic models do not scale due to the overhead for handling partial-orderings, time, and other activity properties. We present contrail filtering: an extension on Rao-Blackwell particle filtering that enables very descriptive activity tracking by simulating k^{th} -order Markov models with minimal overhead.

1 Introduction

In recent years, several projects have started developing systems (see [11, 12, 15]) for assisting the elderly in living independently and for easing the burden on caregivers. A critical feature for these systems is the ability to infer user intentions and actions from available sensor data. Ideally, activity recognition systems should be able to capture the incredible richness and complexity of daily activities while incurring a manageable overhead. Unfortunately, the overhead of these activity properties becomes quickly overwhelming.

An *activity model* is a set of activities and a description of how these activities are to be executed. For this description, *activity properties* are associated with individual or subsets of the activities. For examples of activity properties, consider the sample activity model shown in Figure 1. Upon waking up, a person has a set number of tasks that should be performed exactly once in essentially any order: eat breakfast, shower, get dressed, and other morning preparation tasks. Afterwards, they can do a set of activities any number of times: read a book, watch TV, talk on the

phone, as well as other normal daytime activities. (We will refer to this model throughout the paper and will sometimes mention activities not present in Figure 1). Even in this simple model, there are many activity properties to consider.

First, the duration of an activity is a valuable tool for distinguishing any events that trigger the same sensor activity. For example, the action *Get a Glass of Water* would set off the kitchen sensors for a much shorter time than the action *Do Dishes*. An inherent challenge in using durations is the length of durations. *Sleep*'s duration should be measured on the order of hours, while *Get a Glass of Water* should be measured on the order of minutes, if not seconds.

Consider also the partial-ordering at the beginning of the model. In order to guarantee that each activity in the partial-order is accomplished, we must keep some state that records which actions have and have not been finished yet. Although we refer to all morning activities as being part of a partial-order, there are

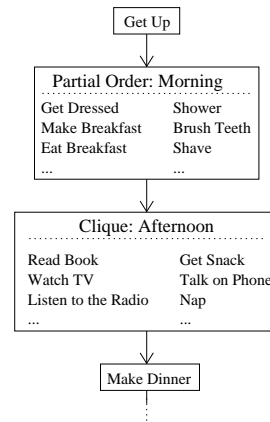


Figure 1: A partial daily activity model. Ellipses indicate more activities that are not listed.

*This research was funded in part by an NSF Graduate Research Fellowship.

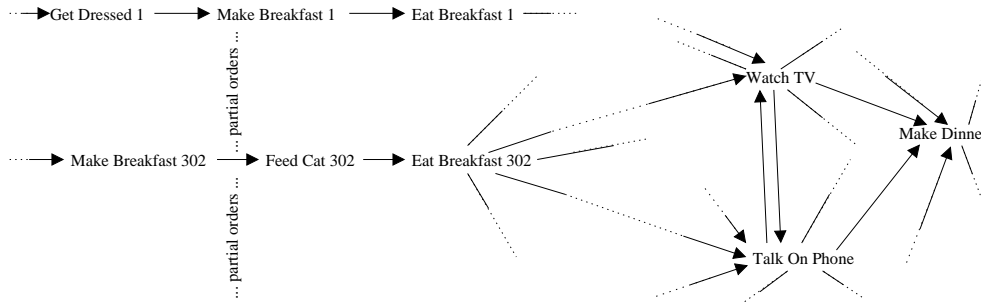


Figure 2: Part of a Semi-Markov Model for tracking the model in Figure 1. The states on the left are numbered to express all possible partial-orderings. The other states have no such ordering requirement and are thus unnumbered.

also internal orders within the larger ordering. For instance, a person cannot eat breakfast before making breakfast. For simplicity, an activity model could ignore these internal orderings,

Beyond these few examples, there are many other properties of activities than can be modeled to make the recognition more robust. As a recognizer handles more of these properties, the need for efficiency increases. This need can often be met through careful programming of the probabilistic inference engine that performs the tracking. Unfortunately, this is quite undesirable. We have no guarantee that the system maintainer will be an expert in probabilistic inference engine design. We need to prevent such non-experts from having to recode the inference engine for different models just to achieve efficiency.

Thus, to perform activity recognition, we desire three somewhat paradoxical properties:

- Robustness in our ability to describe activities
- Efficiency when performing the activity tracking
- An implementation that allows performance tuning without requiring extensive knowledge regarding inference techniques

Contrail filtering, the primary contribution of this paper, achieves all three. By extending on the Rao-Blackwell particle filter, the contrail filter permits robust activity tracking by simulating k^{th} -order Markov models with minimal overhead. Moreover, the implementation of a contrail filter automatically exposes to programmers the parts critical to efficient performance.

In the next section, we will briefly give an overview of current probabilistic techniques for activity recognition. We then describe contrail filtering in Section 3. Section 4 discusses the accuracy and performance of contrail filtering relative to other methods. Next, in Section 5, we consider a complete activity recognition system and how contrail filtering fits into it. Section 6

goes on to discuss related work, and we then outline future work and conclude in Section 7.

2 Current Probabilistic Methods

Because sensor data is inherently noisy, probabilistic techniques are a natural approach for activity recognition. Extensive research has been done in creating several types of probabilistic models, including Markov Models, Dynamic Bayesian Networks, and Particle Filters.

2.1 Markov Models

Markov models and their many variants (see chapters 1–2 of [14]) consist of a set Q of states and a transition function $f : Q \times Q \rightarrow [0, 1]$, where $f(x, y) = p$ indicates the probability of transitioning from state x to state y is p . Each state also produces an output that is observed. In Hidden Markov Models (HMMs), the observations themselves are noisy, thereby making inference the task of determining the state sequence from the observation sequence.

Notice that we are making the classic first-order Markov assumption: the present depends only on the previous time slice. k^{th} -order Markov models (where the present depends on the last k time slices) can be created. However, by expanding the state space to Q^k , it is possible to represent such a model as a first-order Markov model.

Markov models assume the state changes with every clock cycle, Semi-Markov Models [9] extend Markov models by having the probability of switching into a new state depend on the time spent in the current state.

Markov models provide a simple language for describing the ordering of activities. For example, a por-

tion of a Semi-Markov model for activity tracking is shown in Figure 2. Each activity is its own state. In order to capture the ordering constraints, only certain transitions are given non-zero probabilities. Moreover, to handle partial orderings, copies of activities can be used, as demonstrated in the figure.

Although Markov models are robust in expressing ordering constraints, this robustness does not scale. A partial ordering of size n may require up to $n!$ states for the Markov model to fully express all of the possible orderings. Inference is linear in the number of states within the model. Moreover, as the number of states in the model increases, more data is required to learn the model and critically hinders editing of the model itself.

2.2 Dynamic Bayesian Networks

Dynamic Bayesian Networks (DBNs) [3, 10, 14] are an extension of Bayesian Networks. In a Bayesian network, the state space is represented in a factored form as a set of variables Z_1, \dots, Z_n . The variables correspond to nodes in a directed acyclic graph. Moreover, the probability of variable Z_i having a particular value depends on the values of Z_i 's parents. This probability model is often referred to as the Conditional Probability Distribution (CPD) for a node. Variables are also often separated into two types: observed and unobserved. Observed variables, like sensor readings, provide the basis for inferring the values of the unobserved variables.

DBNs extend Bayesian networks by incorporating time. For each time slice t , we have a set of variables $Z_t = \{Z_{1,t}, \dots, Z_{n,t}\}$. Under the Markov assumption, the state encoded in Z_t is dependent only on the previous state Z_{t-1} . Thus, a parent of a variable can

be in either the current or previous time slice. Like Markov models, k^{th} -order DBNs can be converted into first-order representations.

DBNs also generalize HMMs. The primary improvement is that the variable/node representation provides a factored representation of the entire state space. Figure 3 shows a DBN that can track the same activities as in Figure 2. Notice that the DBN does not have a separate variable for each activity as in the Markov representation.¹ Instead, the *Activity* node's possible values are the names of all the activities within the activity model. The *Memory* and *Duration* node provide the means of simulating the needed k^{th} -order properties to properly track the activities. For instance, the responsibility of maintaining the partial-ordering is delegated to the *Memory* node, which is represented as a bit vector where each bit indicates whether a certain activity has been completed. This node keeps track of which activities have and have not been completed in the partial ordering. As for the *Duration* node, it keeps track of how long the activity has been going on by counting down to zero via the method proposed by Murphy for simulating Semi-Markov models with DBNs [14]. When *Duration* reaches zero, *Activity* is required to change.

Despite this factored representation, inference in DBNs can still be computationally expensive in both time and space. Exact inference in DBNs is known to be NP-Hard [4], so approximation techniques have been developed (see chapter 4 of [14]). Unfortunately, these techniques are very sensitive to the structure of the network and often require specialized data structures for representing the CPDs to avoid memory problems. Even a slight change to the state modelled by the DBN could require a different sparse data structure or a different approximation algorithm altogether.

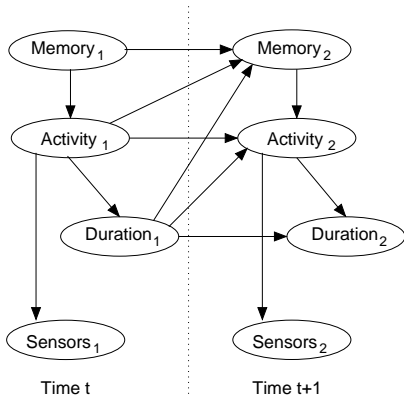


Figure 3: A DBN for tracking activity models like Figure 1 with durations and partial orders

2.3 Particle Filters

As is often the case with NP-Hard problems, Monte Carlo techniques for DBN inference have been found that outperform deterministic approximation algorithms. Particle Filtering [4] is one of the most widely used of these techniques.

Under particle filtering a collection of N particles is kept. Each particle contains a complete instance of the current state, *i.e.* a random sample of each variable in one time slice of a DBN. Thus, a particle represents a single possibility of what could be occurring.

To perform inference, N particles are created by

¹It is possible to construct a DBN with separate variables for each activity as was done in the Lumière project [8].

sampling some initial distribution. In each time step, one uses the probability distributions to move each particle forward one time slice. The CPDs used for this movement are often referred to as *movement functions*. Then, each particle is weighted by how likely it is to produce the current observations. The average sum of the weights gives the current belief of the DBN’s current state. Furthermore, to reduce the movement drift of the particles, the particles are resampled according to the likelihood weightings. After the resampling, the particles are usually clustered around the most likely current state. However, there will be outliers of more unlikely guesses, which might turn out to be the correct choice after all. By moving and constraining multiple guesses at once, particle filters readily adapt to changing states and noisy information.

Despite particle filtering’s remarkable success in many applications [4], it tends to perform poorly in high-dimensional spaces. To address this problem, Rao-Blackwell Particle Filtering (or RBPF) was developed. State variables are separated into two sets: those that are sampled, U , and nodes whose probability distributions can be efficiently calculated, V . Deterministic nodes, like the *Duration* counter in Figure 3, are good choices for variables for V . Variables that are not sampled are often said to be *rao-blackwellized*.

In RBPF, each particle carries both the values it has sampled for the U -nodes as well as distributions for the V -nodes. Although this adds overhead for storing distributions, significantly fewer particles are typically needed to achieve accurate inference. With a regular particle filter, tens of thousands of particles might be needed, but with RBPF, only a thousand might suffice. Since particle filtering’s performance is linear in the number of particles, this can be a significant speedup. However, choosing the right nodes to rao-blackwellize is critical to RBPF’s performance.

3 Contrail Filtering

Dynamic Bayesian networks and Rao-Blackwell particle filtering, in general, provide an expressive and moderately efficient approach for modeling stochastic processes. The requirements for activity tracking, however, reveal critical limitations to these approaches. Contrail filtering, as described in this section, is designed to directly handle the challenges of activity recognition while reducing the need for specialized programming.

3.1 Challenges for Activity Tracking

In Section 1, we described a few of the complex properties that an activity can possess: partial orderings, durations, *etc.* Fortunately, DBNs can express almost any detail of state by the simple addition of more nodes. For example, a *BreakfastMadeAt* node could be added to Figure 3 to store what time breakfast was made. This information could then be used to guarantee that breakfast is eaten soon after its preparation. This added expressiveness does not come free, though.

Recall that DBNs assume that the network structure is stationary, so in any inference calculation, each and every node must be considered and updated. Consider the *Memory* node in Figure 3. When the agent is in the middle of a partial-ordering, the node’s influence is necessary. However, when the agent is doing the day activities where order does not matter, this node is not important but its influence still must be considered in calculations. More troubling that the programmer of the DBN must incorporate the node’s influence into *Activity*’s CPD. For ease of understanding, a programmer might sacrifice the expressiveness of the DBN.

More nodes in the network also add to the size of the particles. For each new node, the particle has to maintain either a sample or distribution for it. Resampling is greatly affected by this growth as the memory management and copying costs increase with the size of the particles. Each of these samples or distributions must also be updated for each time slice.

In general, DBNs and RBPF do a lot of unnecessary work. Information is considered when it shouldn’t be. Particles grow in size with each added piece of information they must lug around and keep updated at all times. As we have stated, specialized inference engines can ameliorate these overhead issues. These engines, though, are not necessarily amenable to change and reuse, particularly without expert programming.

3.2 Principles of Contrail Filtering

As we stated earlier, activity recognition requires paradoxically both expressiveness and efficient representation. (We ignore for the moment the third desired property regarding implementation). Contrail filtering works to achieve through two principles: determinism and parsimony.

Contrail Filter Principle 1 (Determinism). *For each possible activity, any property associated with that activity should be represented either deterministically or parametrically.*



Figure 4: A Contrail Filter Particle.

One of the challenges in using Rao-Blackwell particle filtering is choosing the proper nodes to rao-blackwellize. An important observation for activity tracking is that activity properties like duration and partial-orderings can easily be converted to parametric and/or deterministic functions. For example, the *Memory* node in Figure 3 is updated simply by setting the appropriate bit to 1 to indicate which part of the partial-order was just completed.

However, the real benefit of parametric representations is demonstrated with the *Duration* node. Using the DBN representation suggested by Murphy [14] for modeling durations, the node counts down to zero to indicate when to switch to a new activity. When this node is rao-blackwellized, the initial distribution is updated by shifting the distribution each time slice and renormalizing. For example, after one time slice, the distribution $(\frac{1}{4}, \frac{1}{2}, \frac{1}{4})$ would become $(\frac{2}{3}, \frac{1}{3}, 0)$. However, this update is easily calculated beforehand. We can define the probability of switching to a new activity as a function $f(a, t)$ where a is the current activity and t is the amount of time spent so far in the current activity. Instead of counting down, the *Duration* node counts up from one. By offloading the distributions for the *Duration* node to a global function, each particle only carries the value t and not a full distribution as before.

However, it is important to realize that Principle 1 is actually slightly more powerful than just rao-blackwellizing the nodes. Instead of choosing a single representation of a property, we can define different representations for the property based on the individual activities. For example, it might be the case that the duration of showering is best modeled as a Gaussian CPD, while making breakfast is better modeled as a discrete CPD. This approach also allows us to assign the influence of properties to specific activities. Boyan [1] has demonstrated the value of selectively using relative (it takes 5-10 minutes to shower) or absolute (a person watches the news from 6pm to 7pm) temporal constraints in solving Markov decision processes. Contrail filtering allows us to do this as well.

Contrail Filter Principle 2 (Parsimony). *Information from activity properties should only be created and considered when it is applicable and should be discarded when it no longer has any use.*

This principle directly attacks the complication in DBNs of having to consider *Memory*'s influence on ac-

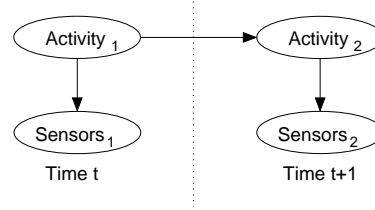


Figure 5: A DBN for a contrail filtering approach to activity recognition.

tivities that are not partial-orderings. Another aspect of this principle is the notion of information being created and carried till it is destroyed. At any point, a particle carries only the information it has to. Thus, we now expressively avoid the overhead concerns of Rao-Blackwell particle filtering. The space allocated by a particle automatically adjusts to exactly fulfill its needs.

3.3 Implementing a Contrail Filter

Contrail filtering is a new approach for implementing particle filtering. In Figure 4, we see the structure of a contrail filter particle. Like a particle in an RBPF engine, it is composed of two parts. The first part, the *particle body* is comprised of the sampled values. However, the second part, the *contrail*, is a dynamic collection of contrail objects. Each object corresponds to an activity property the particle finds relevant to keep around for the time being. For example, the checklist used to maintain the morning partial-ordering would be one of these objects.

Inference in contrail filtering is also very similar to inference in particle filtering. The underlying steps are still there: moving the particles, likelihood weighting, and resampling. In implementing the contrail filter engine, however, we make a significant break from a traditional implementation of a centralized probability engine. Under such a model, the engine itself would contain and coordinate the movement and probability updates of the particles. Instead, we view the engine as a data processor that computes different programs for the different values in the particle bodies.

To explain, consider the DBN in Figure 5. With contrail filtering, this network is sufficient for tracking activities and describing the particle body (which contains only a sample for the *Activity* node²). Contrail objects appear and disappear during filtering. In a DBN, values for a node are treated identically. The transition function assumes the same information

²For simplicity, we sample only one node in this paper. Multiple sampled nodes introduce variability in the order functions can be called. We intend to address this in future work.

```

if(value.shouldSwitch()) {
    value.onExit()
    value = value.moveParticle()
    value.onEnter()
}
else {
    value.onWait()
}

```

Figure 6: Algorithm for moving a particle in CF. *value* is the sampled variable within the particle body.

is present to decide how the particle should update. However, Principle 2 suggests treating each value of *Activity* differently. Thus, for each activity, we define a unique transition function that only uses the necessary information. Moreover, we also include functions that update the contrail.

Another way of viewing this is from an object-oriented programming perspective. Each value is a separate object that implements an interface for the CF engine to work with. When a particle needs to be moved, the CF engine uses only these interface methods. Through dynamic dispatch, the proper methods for the value the particle contains is called. In other words, the actual transition function is black-boxed from the engine’s point of view.

By hiding these calculations, activity objects effectively become miniature programs that are fed into the engine and processed. This achieves two important goals. First, it enables us to achieve Principle 2 for every value of *Activity*. Furthermore, changing the actual activity model no longer requires editing the engine. All changes, be they for efficiency or expressiveness, takes place within the value objects, the input into a contrail filtering engine. Since any programmer would have to input the activity model into the system, programmers are automatically exposed to the components of inference that are most valuable for them to code well.

Figure 6 shows the code the CF engine uses for moving a particle. The five functions within this code are part of the interface every value object represents. Figure 7 shows an example configuration file for the activity *MakeBreakfast*.

The first four items are variables local to this class that are used by the interface functions. Each represents a probabilistic distribution, but the first one, *SensorModel* . . . , bears special importance. Each line within the declaration identifies a sensor, an output from that sensor, and the probability of it occurring. Thus, contrail filtering, unlike a pure DBN representation as in Figure 3, has the ability to ignore sensors irrelevant to the activity.

The next six functions are the interface methods used by the contrail filter engine.

```

VALUE MakeBreakfast {
    SensorModel sensors(
        AtCounter = On, 0.87;
        InKitchen = On, 0.93;
    )
    DiscreteCPD po_cpt(
        EatBreakfast, 0.65;
        :
    )
    DiscreteCPD cl_cpt(
        WatchTV, 0.45;
        :
    )
    GaussianCPD dur(16, 2.17)
    void onEnter() {
        createDurationObject( MakeBreakfast, dur)
        if(!exists(PartialOrder::Morning))
            createPartialOrder(Morning)
    }
    void onWait() {
        updateDuration(Duration::MakeBreakfast)
    }
    void onExit() {
        PartialOrder::Morning.update(MakeBreakfast)
        destroy(Duration::MakeBreakfast)
        createTimeObject(BreakfastMadeAt, current_time())
        if(PartialOrder::Morning.done())
            destroy(PartialOrder::Morning)
    }
    boolean shouldSwitch() {
        return sample(Duration::MakeBreakfast)
    }
    value moveParticle() {
        if(exists(PartialOrder::Morning))
            return sample(PartialOrder::Morning.evaluate(po_cpt))
        else
            return sample(cl_cpt)
    }
    float weightObservation(Hashtable<sensor,value> obs) {
        return sensors.weight(obs)
    }
}

```

Figure 7: Configuration file for *MakeBreakfast*.

- *void onEnter()*: This function is called when a particle first enters into this activity. Notice that a Duration object and a PartialOrder object (if necessary) is added to the contrail.
- *void onWait()*: This function is called in any time slice in which the particle did not change its sampled value. In the example, only an update of the Duration object *MakeBreakfast* is performed.
- *void onExit()*: This function is called before the particle right before it changes its sampled value. Note that several events happen here. First, the PartialOrder object is updated to indicate that the activity has been completed. Next, the function destroys the duration object. Then, it creates the time object *BreakfastMadeAt* to record the current time. Finally, if the PartialOrder object has been com-

pletely filled in, it destroys the object.

- *boolean shouldSwitch()*:
This function returns true if the particle should transition to a new value/activity. In this case, this decision is dependent only on the Duration object. *sample(...)* is a helper function that returns a value from any probability distribution passed to it.
- *value moveParticle()*:
If it has been determined that the particle should transition, this function determines what it should transition to. For the *MakeBreakfast* activity, there are two distributions to consider. While the partial-ordering is still unfinished and in existence, it samples the *po_cpt* distribution which contains only activities from the morning partial-order. Otherwise, it samples the *cl_cpt* distribution, which contains only activities in the afternoon clique.
- *float weightObservation(Hashtable <sensor, value> obs)*:
Finally, since the sensor model for each activity is unique, this function calculates the particle’s likelihood weighting. *obs* is a hashtable which contains the current sensor observations indexed by the sensor names. The *SensorModel* method *weight(...)* uses this information then to return the appropriate likelihood weighting.

To help insulate the programmer from the inner workings of the contrail filter, contrail filtering includes a set of basic contrail object classes and helper functions. A class or function encapsulates the basic functionality for a feature desired in probabilistic inference or activity recognition. In Figure 7, some examples of these built-in classes are *Duration*, *PartialOrder*, and *Predicate*. Examples of these helper functions are *sample()* and *exists()*. We intend to eventually make it possible for programmers to implement their own contrail object classes and helper functions to extend the expressiveness beyond that of the built-in ones.

We now define a *contrail filter activity library* as a collection of value objects as seen in Figure 7. When a particle needs to change its sampled value(s), it dynamically dispatches the commands in this object to decide its movement as well as performing bookkeeping of the contrail. Because of this design, users can express many different activity libraries that can be monitored with the same probability engine.

4 Performance of Contrail Filtering

In order to justify the use of contrail filtering, we prove the soundness of the approach in this section. Furthermore, we show results that contrail filtering can produce noticeable improvements in time and memory usage as opposed to a Rao-Blackwell particle filter.

4.1 Soundness and Accuracy

Two questions inherent to any probabilistic model are “Is this approach sound?” and “How accurate is this approach?” Contrail filtering is sound and is as least as accurate as Rao-Blackwell Particle Filtering. To show this, we will prove that it is possible to convert a CF activity library to a full DBN representation that RBPF can then be applied to. Thus, all soundness and accuracy results for RBPF are directly applicable to CF. Moreover, improvements to the RBPF algorithm [4] are also likely to be applicable to CF. For example, contrail filtering, like RBPF, will require fewer particles than pure particle filtering.

Note again we consider only a CF library where a single node is sampled. It is a simple exercise to augment the following construction for multiple sampled nodes. Furthermore, we begin by describing the construction for a CF library under the condition that the calculation of a contrail object is independent of other contrail objects. We will discuss loosening this condition later in this section.

Theorem 1. *For a contrail filtering activity library \mathcal{L}_{CF} , if the calculation (creations, updates, and destructions) of each contrail object is independent of all other contrail objects, then \mathcal{L}_{CF} can be represented as DBN \mathcal{D} . RBPF can be applied to \mathcal{D} where the values sampled in \mathcal{L}_{CF} are sampled and the contrail objects in \mathcal{L}_{CF} are rao-blackwellized.*

Proof. Figure 8 shows the structure of the DBN \mathcal{D} that we will construct. The fundamental idea is to create a *Daemon* node that calculates when and where to switch activities. For every possible contrail object, an individual node is added to the network. These individual object nodes handle their own updates. We begin with describing the sensor models.

The *Activity* node’s possible values, as before, are the names of all possible activities within \mathcal{L}_{CF} . To mimic contrail filtering’s ability to be more concise with sensors, we create a separate node for each sensor. Consider a particular sensor *s*. For each activity *a*, we look at *a*’s declaration of the function

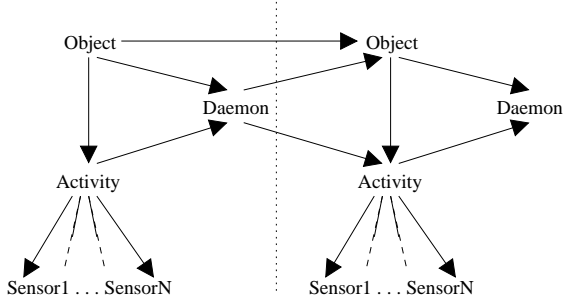


Figure 8: A DBN for representing a CF library with only one contrail object.

$weightObservations(\dots)$ in \mathcal{L}_{CF} . If this function does not ignore s , the function describes a 's influence for s . Otherwise, we use the sensor's general noise model. After doing this for each activity a , we have created the sensor's CPD.

To accommodate transitions, we add several nodes to the DBN. First, we create a separate node for each contrail object found in \mathcal{L}_{CF} . Each object node is influenced only by itself and the *Daemon* node, as seen in Figure 8. The *Daemon* node's CPD is based on the CF functions $shouldSwitch()$ and $moveParticle()$. The CPD uses the current value of *Activity* and the object nodes to calculate the appropriate $shouldSwitch()$ function, ignoring object nodes irrelevant to this calculation. If the result is to wait, *Daemon* has the value $(WAIT, a)$, where a is the current value of *Activity*. Otherwise, *Daemon* goes on to calculate $moveParticle()$ and thus has the value of $(a, SWITCH, b)$, where a is the previous activity and b is the new activity.

The *Activity* node's CPD is simple. If the *Daemon*'s value is $(WAIT, a)$, *Activity* has the value a . Similarly, if the *Daemon*'s value is $(a, SWITCH, b)$, then it has the value b . This explains why the *Activity* node is not influenced by itself from the previous time slice.

For an *Object* node, the CPD is created by combining all $onEnter()$, $onWait()$, and $onExit()$ functions described in \mathcal{L}_{CF} that involve the object. For each activity a , the CPD is as follows. When *Daemon* has the value $(WAIT, a)$ and a 's $onWait()$ function is associated with the object, the CPD performs the update in $onWait()$. Otherwise, the object's value remains unchanged. When *Daemon* has the value $(a, SWITCH, b)$, it performs activity a 's $onExit()$ function if appropriate and b 's $onEntry()$ function if appropriate. If the exit function calls for the destruction of the object and the object is not recreated by the enter function, then *Object* is assigned the value of *null* to indicate that it does not exist.

The construction of \mathcal{D} creates complex CPDs to simulate every function contained in \mathcal{L}_{CF} . Despite the complexity of these CPDs, the *Daemon* and the *Object* nodes are deterministic, allowing them to be easily rao-blackwellized. It follows that \mathcal{D} expresses, through RBPF, the same probabilistic model as \mathcal{L}_{CF} . \square

The challenge of having contrail objects interact is that a DBN requires the network is acyclic. Consider all the contrail objects in \mathcal{L}_{CF} . For every pair of different objects x and y , if x influences y during an update, then draw a directed edge from x to y . If the resulting dependency graph is acyclic, then following theorem is proven by a simple extension of the previous proof.

Theorem 2. *For a contrail filtering activity library \mathcal{L}_{CF} , if the dependencies among contrail objects forms a directed (possibly disconnected) acyclic graph, then \mathcal{L}_{CF} can be represented as DBN \mathcal{D} . RBPF can be applied to \mathcal{D} where the values sampled in \mathcal{L}_{CF} are sampled in \mathcal{D} and the contrail objects in \mathcal{L}_{CF} are rao-blackwellized.*

Thus, under a reasonable restriction, CF is as sound as RBPF. However, it should be noted that in contrail filtering, the dependency graph between contrail objects can contain cycles. For example, for activity a , x influences the update of y , but in activity b , the opposite holds true. Such flexibility promotes greater expressiveness within contrail filtering. Whether or not this flexibility introduces unsoundness is a topic for future research.

4.2 Time and Space Efficiency

Given that contrail filtering can be as sound as Rao-Blackwell particle filtering, we now discuss CF's better time and memory performance. To do this, we considered the following activity model: a partial-order of size PO_1 followed by a clique of size CL , and another partial-order of size PO_2 . Furthermore, each activity had could last for 1 to T time slices. This is a plausible model for activity tracking. For example, on a computer, a user might have a set of initial startup tasks (turn on, check e-mail, etc.), then work throughout the day (edit document, send e-mail, etc.), and then do some final shutdown tasks (close applications, turn off, etc.) Note that the DBN in Figure 3 is capable of tracking this model

To gather sensor traces to track, we assigned each activity its own unique sensor output. A random noise model was added to this sensor model. Probabilities for transitions and durations were also randomly gen-

erated. Finally, sensor traces were then gathered from the generated model.

We implemented three algorithms: RBPF1, RBPF2, and CF. The first two algorithms were Rao-Blackwell particle filters. Both used the same general inference engine that did not include any special code for efficient performance. RBPF1 sampled both the *Duration* and *Activity* nodes in Figure 3, but kept a distribution for the *Memory* node. RBPF2 sampled only the *Activity* node, but rao-blackwellized both *Memory* and *Duration* by keeping around durations. In all implementations, *Memory* was represented as a fixed-length array of booleans. The *Duration* node in RBPF2 was similarly represented as a fixed-length array of integers.

Our CF implementation allowed two optimizations through the input file. One optimization was the discarding of the *Memory* object when a particle is in the clique. The input also represented the duration probabilities as a parametric function, and thus required the particle to maintain a duration counter when durations were applicable. We note that the memory management in our code was quite simple and could be considerably improved. Also, to guarantee fairness between the algorithms, we implemented CF by modifying the RBPF code as minimally as was necessary. In fact, via the construction suggested in Theorems 1 and 2, it can be shown that CF and RBPF2, given the same random seed, both algorithms return the exact same probabilities for the *Activity* node for every time slice. This equality between the two algorithms was verified before data collection began.

Figure 9 shows the average size of a particle in the three algorithms for two models.³ Both models have the same partial-order size; only the maximum duration T differs. The two plots for CF show what we would expect. In both the first partial-order and the clique, CF uses significantly less memory than either of the RBPF algorithms. The spikiness of the CF plots is due to some particles believing that they had entered the second-partial order when in fact the data sequence was still in the clique.

Note that for this activity model, CF uses no more memory (36 bytes) than RBPF1, while RBPF2 always uses more bytes. RBPF2’s memory usage is very dependent on the value T since it has to keep an entire distribution for *Duration* in every particle. Both CF and RBPF1 maintain just a single integer for the duration. CF’s duration counter mimics (via the parametric representation of duration) the same probabili-

³All results shown here were run on a 550MHz Pentium III processor using the g++ compiler, version 2.96. No compiler optimizations were used.

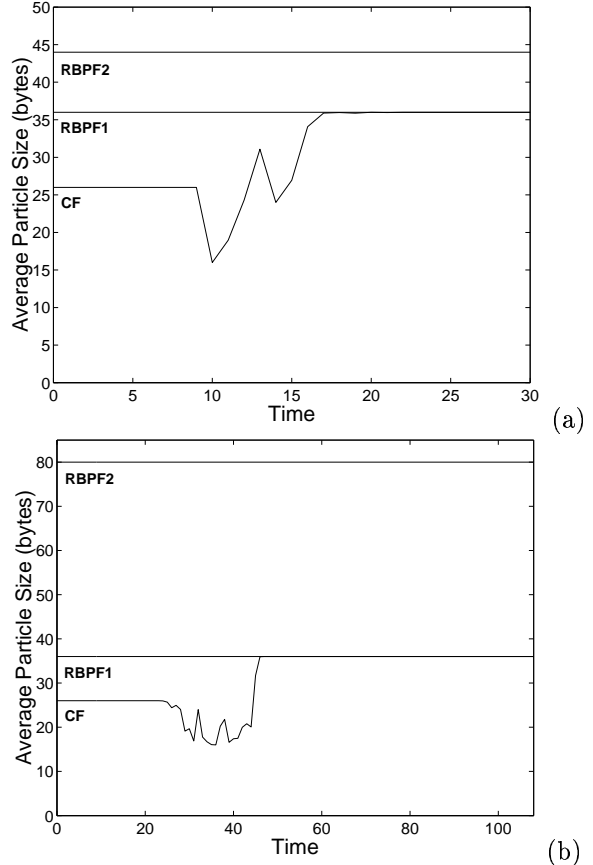


Figure 9: Average particle size over ≈ 300 runs for each algorithm.

- (a) $PO_1 = 10, CL = 10, PO_2 = 20, T = 1$
- (b) $PO_1 = 10, CL = 10, PO_2 = 20, T = 10$

ty distribution as RBPF2’s, while RBPF1’s is just a random guess.

Figure 10 shows the effect of memory efficiency on the runtimes. Again, we show results for two models. The reader should be aware that the two data sequences were of significantly different length. As expected, the number of particles used significantly impacts the runtime. Notice that in Figure 10b, there are no results for RBPF1 with 512 and 1024 particles since despite several hundred attempted runs, RBPF1 never finished processing the sequence due to all particles acquiring zero likelihoods. This is due to RBPF1’s sampling of the *Duration* node. RBPF2, which rao-blackwellized that node, does process the sequence successfully several times for these two particle counts.

In both graphs, there is little or no difference between the processing times for the three algorithms. Despite our constant deletion and creation of memory, the CF algorithm takes the same amount of time as

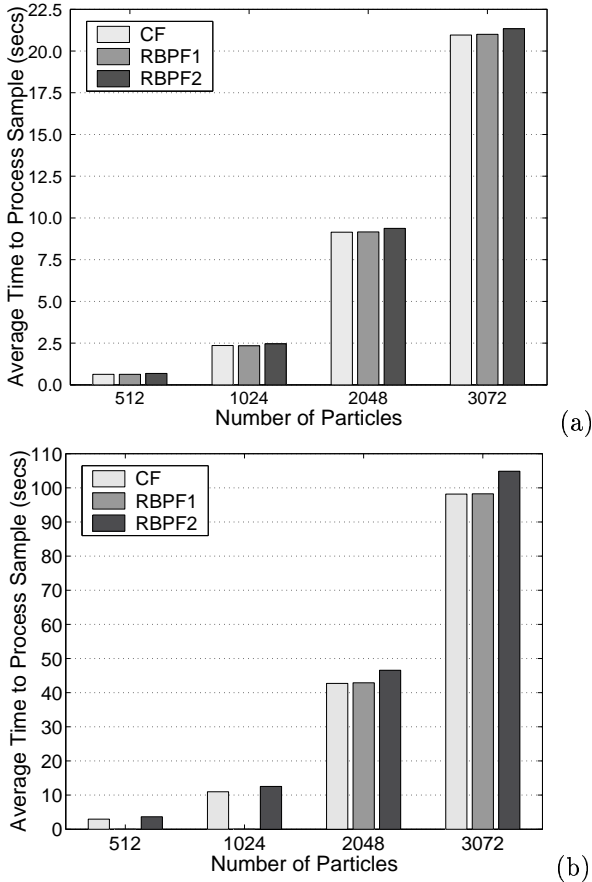


Figure 10: Average runtimes (≈ 50 runs per algorithm and particle count) for each algorithm. L is the length of the sequence.

- (a). $PO_1 = 10, CL = 10, PO_2 = 10, T = 10, L = 69$
 (b). $PO_1 = 10, CL = 10, PO_2 = 30, T = 30, L = 306$

the others. Two factors compensate for this memory management overhead. First, by having particles of smaller, varying size, copying particles during the resampling step is slightly faster. Secondly, a CF particle only updates its distribution for the *Memory* node when the particle is in a partial-order and is switching to a new activity. Both RBPF1 and RBPF2 have to update it every time slice.

Both graphs also show that CF and RBPF1 slightly outperform RBPF2. This is particularly evident in Figure 10b for 2048 and 3072 particles. This is as expected. By rao-blackwellizing the *Duration* node, RBPF2 tracks the model better than RBPF1, but it has to update the full distribution for the that node in every time slice. Meanwhile, CF and RBPF1 update only a single value every time slice (recall that CF uses a parametric representation for the durations).

These memory and runtime results are quite promising. As we stated earlier in this section, CF and

RBPF2 return the exact same probabilistic results. Contrail filtering, however, does it at essentially the same speed (a little better in fact) and with significantly better memory overhead. However, we will admit that there are no guarantees that contrail filtering will always outperform Rao-Blackwell particle filtering. For larger models that express many activity properties, the overhead of the memory management in contrail filtering might overcompensate the efficiencies inherent in our approach. Still, these early results suggest that the CF approach is both viable and likely to scale to larger, more complicated activity models.

5 A Complete Activity Recognition System

Non-expert usability is one motivation we gave in Section 1 for developing contrail filtering. While CF does remove a user from interacting with the underlying inference engine, programming skills and knowledge of probabilities are still required to achieve an efficient activity model representation. This is a particular concern for our interest in developing these systems for elderly care. Caretakers, not programmers, are likely to be the ones updating the activity models within these systems. Due to this, we believe that contrail filtering has a powerful role as part of a complete activity recognition system,

Figure 11 shows our proposed architecture for an activity tracking system. To understand how this system functions, we will explain how a caretaker would input the *Shower* activity from Figure 1. Through the interface, the caretaker is provided with two sets of information. The first is the box labelled Activity Constructs (AC). The AC module contains a set of activity properties: partial-orderings, durations, pre-conditions, *etc.* To the caretaker, these are just labels that are placed on activities to explain to the system

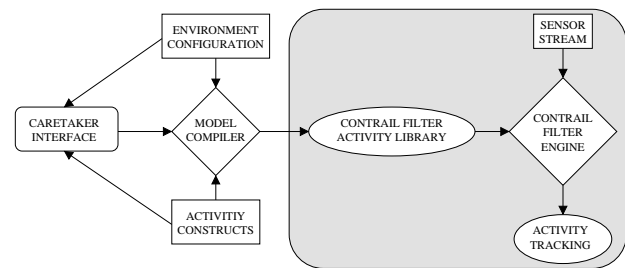


Figure 11: A proposed activity tracking system. The gray box represents our contribution of contrail filtering.

how the activity plan should be executed. The other is the box labelled Environmental Configuration (EC). This EC module contains information about the environment the activities will occur in. Besides just a map of the environment, it provides the caretaker with a list of the available sensors and a reasonable set of outputs for each sensor. This insulates the caretaker from having to consider sensor noise.

Thus, to add the *Shower* activity to the model, the caretaker would input the following through the caretaker interface:

```
Activity: Shower
Pre: Exercise Duration: 5 to 10 minutes
Shower Temperature: 80 to 100 °F
Location: Bathroom
Bathroom Light: On
Next: Brush Teeth, Get Dressed
      Shave, Use Toilet
```

This representation has several nice features. First, although sensors can produce false readings, the user only has to note the sensor readings that should occur. Notice as well that no probabilities have been assigned to the duration or to the transitions to the next activities. The user could weight them if so desired, but the system will ultimately refine these probabilities through learning.

Once the caretaker has entered all the activities within model to be tracked, the caretaker's input is sent to the Model Compiler. The compiler receives input from the AC and EC modules as well. This input includes items like sensor noise models and definitions of the activity constructs. Since we have limited the number of activity properties, we believe it will be possible to define rules converting the caretakers input into a CF activity model. For example, since *Shower* is part of the partial-order Morning, the compiler would create a partial-order object for morning activities. Within this object, it could specialize the code to guarantee that showering never occurs before having exercised. Similarly, within the value declaration for *Shower*, the compiler would produce code for a duration object for *Shower*. It also creates the *weightObservations(...)* function using the appropriate sensor noise models. Once the contrail filter activity library has been assembled, activity tracking can begin.

In all respects, the gray box in Figure 11 could be replaced with an RBPF inference engine or any other probabilistic inference engine. Note that is up to the Model Compiler to produce an efficient representation for input into the inference engine. However, contrail filtering's implementation places the controls for performance efficiency at the input level. Other models

place them within the engine. Thus, the performance of our system depends solely on the quality of the activity library produced by the compiler.

The key to achieving the system in Figure 11 will be creating a large set of constructs for the AC module that can be easily transformed into a CF activity library. This library will not always be as efficient or expressive as a manually created one. However, activity properties will still be represented deterministically, and information will be stored parsimoniously. Together, these produce the efficient representation we desire.

6 Related Work

The Dynamic Object-Oriented Bayesian Networks [6, 13] created by Avi Pfeffer are an approach to representing DBNs via object-oriented programming. Nodes are defined as objects with various sub-variables. The objects permit reusing the same structure in various networks for potentially large, complex domains, much like the reusability of basic contrail object constructs. DOOBNs are like CF in that they provide a probabilistic modeling language that can ease the burden of creating networks. Unfortunately, no efficient inference method has yet to be proposed for DOOBNs.

Bui has proposed the Abstract Hidden Markov Model [2] as an approach to performing policy recognition. These models are a special case of the Hierarchical Hidden Markov Model [5]. Both models compact the activity recognition through the use of hierarchy and abstraction. Furthermore, DBN representations of these Markov models reduces the time complexity of inference, but the complex structure of the DBN representations is a major constraint on scaling these approaches.

Goldman and Geib [7] have developed a rather different activity modeling technique based on Poole's probabilistic Horn abduction [16]. Using Poole's combination of logic and probability, their system keeps around an *active set* of activities that are permitted to occur according to various ordering constraints. This approach is very similar to CF's principle of parsimony. However, the logical representation required to maintain Horn clauses can easily become complex, limiting the general use of this approach.

As part of the Autominder [15] assisted care project, Pollack has proposed the Quantitative Temporal Bayesian Network [15] to better handle time-related constraints. The model's combination of time nets and DBNs suffers from exponential time complexity, although ongoing work aims to reduce this. More dis-

trekking is the proven unsoundness of QTBN inference.

7 Conclusions and Future Work

In conclusion, conrail filtering provides an efficient representation for simulating k^{th} -order Markov models and is particularly suited. We have demonstrated its potentially greater expressiveness and performance over Rao-Blackwell particle filtering, popular approach for performing Bayesian inference. We feel that upon further study, conrail filtering will prove to be a powerful and successful approach for deploying activity tracking in real world application.

Conrail filtering, as a probability model, has many avenues for future research. Allowing for multiple nodes to be sampled is one priority. In particular, smoothing and learning techniques developed for particle filtering [4] need to be adapted to conrail filtering to make it truly applicable to real world applications. Once these have been implemented, conrail filtering could then be applied towards an activity tracking situation involving real sensor data like the datasets currently being analyzed by the Assisted Cognition project [11]. The full activity tracking system proposed in Section 5 is also a critical step for future development.

8 Acknowledgements

Thanks to Dieter Fox, Henry Kautz, Lin Liao, and Don Patterson for the countless discussions. We are also grateful to Brian Youngstrom for his technical support and to Steve Wolfman for his helpful comments on this paper.

References

- [1] J. BOYAN AND M. LITTMAN, *Exact solutions to time-dependent MDPs*, in NIPS, 2000, pp. 1026–1032.
- [2] H. BUI, S. VENKATESH, AND G. WEST, *Policy recognition in the abstract hidden markov model*, Journal of Artificial Intelligence Research, 17 (2002), pp. 451–499.
- [3] T. DEAN AND K. KANAZAWA, *A model for reasoning about persistence and causation*, Computational Intelligence, 5 (1989), pp. 142–150.
- [4] A. DOUCET, N. D. FREITAS, AND N. GORDON, eds., *Sequential Monte Carlo Methods in Practice*, Springer Verlag, 2001.
- [5] S. FINE, Y. SINGER, AND N. TISHBY, *The hierarchical hidden markov model: Analysis and applications*, Machine Learning, 32 (1998), pp. 41–62.
- [6] N. FRIEDMAN, D. KOLLER, AND A. PFEFFER, *Structured representation of complex stochastic systems*, in AAAI/IAAI, 1998, pp. 157–164.
- [7] R. GOLDMAN, C. GEIB, AND C. MILLER, *A new model of plan recognition*, in Proceedings of the Conference on Uncertainty in Artificial Intelligence, 1999, pp. 245–254.
- [8] E. HORVITZ, J. BREESE, D. HECKERMAN, D. HOVEL, AND K. ROMMELSE, *The lumière project: Bayesian user modeling for inferring the goals and needs of software users*, in Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence, 1998, pp. 256–265.
- [9] R. A. HOWARD, *Dynamic Probabilistic Systems*, John Wiley and Sons, New York, 1971.
- [10] C. HUANG AND A. DARWICHE, *Inference in belief networks: A procedural guide*, Intl. J. Approx. Reasoning, 15 (1996), pp. 225–263.
- [11] H. KAUTZ, O. ETZIONI, D. FOX, D. WELD, AND L. SHASTRI, *Foundations of assisted cognition systems*, tech. rep., University of Washington, Computer Science and Engineering, March 2003.
- [12] C. KIDD, R. ORR, G. ABOWD, C. ATKESON, I. ESSA, B. MACINTYRE, E. MYNATT, T. STERNER, AND W. NEWSTETTER, *The aware home: A living laboratory for ubiquitous computing research*, in Proceedings of the 2nd International Workshop on Cooperative Buildings, October 1999.
- [13] D. KOLLER AND A. PFEFFER, *Object-oriented bayesian networks*, in Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence (UAI-97), 1997, pp. 302–313.
- [14] K. MURPHY, *Dynamic Bayesian Networks: Representation, Inference, and Learning*, ph.d. dissertation, University of California, Berkeley, 2002.
- [15] M. POLLACK, C. MCCARTHY, S. RAMAKRISHNAN, I. TSAMARDINOS, L. BROWN, S. CARRION, D. COLBRY, C. OROSZ, AND B. PEINTNER, *Autominder: A planning, monitoring, and reminding assistive agent*, in Seventh International Conference on Intelligent Autonomous Systems., 2002.
- [16] D. POOLE, *Probabilistic horn abduction and bayesian networks*, Artificial Intelligence, 64 (1993), pp. 81–129.