Functional Specification of Probabilistic Process Models*

Avi Pfeffer
Harvard University
avi@eecs.harvard.edu

Abstract

Agents that handle complex processes evolving over a period of time need to be able to monitor the state of the process. Since the evolution of a process is often stochastic, this requires probabilistic monitoring of processes. A probabilistic process modeling language is needed that can adequately capture our uncertainty about the process execution. We present a language for describing probabilistic process models. This language is functional in nature, and the paper argues that a functional language provides a natural way to specify process models. In our framework, processes have both states and values. Processes may execute sequentially or in parallel, and we describe two alternative forms of parallelism. An inference algorithm is presented that constructs a dynamic Bayesian network, containing a variable for every subprocess that is executed during the course of executing a process. We present a detailed example demonstrating the naturalness of the language.

Introduction

An important goal of artificial intelligence is to design agents that can handle complex processes that evolve over a period of time. For example, the CALO project is developing an intelligent office assistant that will be able to perform such tasks as planning a meeting or purchasing a laptop. These tasks require many stages and can go wrong in a variety of ways. For example, purchasing a laptop requires getting the purchase criteria from the user, soliciting bids, going through a cycle of refining the criteria if necessary and getting more bids, and, once an appropriate laptop has been found, getting the right managerial authorizations for the purchase and the final approval of the user. An agent that is designed to fulfil such a task must be able to keep track of the state of the task execution. Since the evolution of tasks is stochastic, this requires probabilistic monitoring of complex processes.

A probabilistic process model can be very useful in tracking the progress of a process. It will enable us to answer queries such as "What is the probability of successful completion of a process, given that a particular subprocess has failed?", "What is the probability that a subprocess will complete successfully given that it has been running for an hour?", "What is the probability that a subprocess will be executed, given the state of other subprocesses?", and "What is the expected running time of a process, given the state of its subprocesses?" In order to answer these queries, we need a probabilistic representation of the process model that adequately captures our uncertainty about the way the process evolves.

SPARK (Morley & Myers 2004) is a non-probabilistic process modeling language developed at SRI. In SPARK, processes pass arguments to their subprocesses. Each subprocess can be instantiated in many ways. In essence, a subprocess is a first order variable that can take arguments. Therefore, in order to compactly describe a probabilistic process model, we need a first-order probabilistic modeling language.

Some popular probabilistic representations for dynamic systems are inadequate for the task. Hidden Markov models (Rabiner & Juang 1986) collapse the entire state of the process into a single variable. Since the state of a process involves the states of all its subprocesses, we end up with an enormous state space. In addition, there is no way to talk about the different subprocesses as separately evolving entities. Dynamic Bayesian networks (DBNs) (Dean & Kanazawa 1989) do allow us to represent different subprocesses by different variables. Indeed, as we shall see, a DBN can be constructed containing every instantiation of every subprocess. However, this DBN is very difficult to construct manually. There may be thousands of possible instantiations of subprocesses, and each needs to be enumerated. The essential problem is that DBNs are a propositional representation, whereas process models are essentially first order. Hierarchical HMMs (Fine, Singer, & Tishby 1998) do incorporate hierarchy of processes, but they are also not a first order representation.

A more recent framework that supports modeling of first-order dynamic systems is Dynamic Probabilistic Relational Models (DPRMs) (Sanghai, Domingos, & Weld 2003). DPRMs go some of the way towards representing probabilistic process models, but they are not a complete solution. There is no notion of execution of a process. There is no sense in which a subprocess is executed as part of the execution of the parent. In particular, there are no notions of

*This work is supported by a DARPA subcontract through SRI International 27-000913-02.

parallel and sequential execution, which are important in a process modeling language.

An alternative expressive representation for probabilistic process models is Colored Petri Nets (CPNs) (Jensen 1997). CPNs combine Petri Nets with the power of programming languages. CPNs can model a general class of stochastic processes. In principle, since a CPN defines a probabilistic model, it could be used to monitor the state of hidden variables given observations. However, algorithms that perform this task have not been developed for CPNs. It is not clear whether a CPN can easily be converted to a DBN or similar representation that supports efficient monitoring.

This paper presents a new language for representing probabilistic process models, called ProPL (standing for Probabilistic Process Language). The language is functional. We believe that a functional language naturally captures the evolution of processes. The relationship between a process and a subprocess is simply a function call. The parent process passes arguments to the subprocess as arguments to the function, and receives return values from the subprocess. In addition, ProPL provides constructs that represent the passage of time, and uncertainty over the way a process evolves. The language is currently restricted to discrete models; continuous models will be a matter for further study.

ProPL is based on IBAL (Pfeffer 2001), a probabilistic modeling language for static models. IBAL provides all the expressivity of a functional programming language; ProPL adds the features necessary to represent process models. Essentially, a process is modeled by writing a program describing the evolution of the process. This can be done quite naturally by adapting a process model written in a language such as SPARK. Indeed, two full-scale scenarios written in SPARK were adapted into ProPL programs. We present a fragment of the ProPL program for a laptop purchase scenario.

Once we have expressed a process model in ProPL, we need to be able to do inference on that model. We need to monitor the state of the process given observations about some of the subprocesses, and we need to reason about the future (for example, the probability of successful completion) based on the current state. Our approach is to construct a DBN that contains a variable for every instantiation of every subprocess in the model. We can then perform inference on this DBN using standard inference algorithms (Kjaerulff 1995; Boyen & Koller 1998; Doucet 1998).

The ProPL Language

IBAL

The basic idea behind IBAL is that a program describes an experiment that stochastically produces a value. IBAL provides a number of expression forms. These include:

Constants that describe experiments that always produce the same value.

Conditionals of the form "`if` e_1 `then` e_2 `else` e_3".

Stochastic choice of the form "`dist` $[p_1 : e_1, \ldots, p_n : e_n]$". This describes the experiment in which one of the possible subexpressions $e_1, \ldots, e_n$ is selected, with the probability that e_i is chosen being p_i.

Variable definitions of the form "`let` $x = e_1$ `in` e_2". This describes the experiment in which e_1 is evaluated, its value assigned to x, and then e_2 is evaluated using the assigned value of x.

Variables that have previously been assigned.

Function definition of the form "`fun`$f(x_1, \ldots, x_n) \rightarrow e$". This defines the (possibly recursive) function named f, taking arguments $x_1, \ldots, x_n$, with body e.

Function application of the form "$e_0(e_1, \ldots, e_n)$", where the value of e_0 is applied to the values of $e_1, \ldots, e_n$. Note that e_0 may be stochastic, which means there may be uncertainty about which function to apply.

Tuples and records that provide ways to aggregate information together into data structures.

In addition to these basic elements. IBAL provides a good deal of syntactic sugar.

ProPL

In IBAL, a program describes a stochastic experiment that generates a value. ProPL introduces two new concepts. The first is a process that executes over time. Time in this paper is discrete and synchronous. A program still defines a stochastic experiment, but now execution of the experiment takes place in time. Each subexpression that is evaluated corresponds to execution of a subprocess. Each subprocess has a particular point in time at which its execution begins. Then, there may be a period of time during which execution of the subprocess is in progress. Finally, execution is completed and the subprocess produces a value. The second notion introduced by ProPL is that of a process with state. Each subprocess that gets executed has a state that may vary over time. The state of one subprocess may depend on the previous state of other subprocesses. This is the natural way to represent dynamic Bayesian networks.

To capture the two notions, we say that every subprocess has state that varies from one time point to the next. The state can indicate the execution status of a subprocess executing in time. The state of a subprocess can be `NotBegun`, indicating that execution of this subprocess has not yet begun. It could be `InProgress`, indicating that execution has begun but has not yet completed. It could be `Complete` with a value, indicating that the execution of the subprocess has terminated and resulted in the given value. We also allow the value produced by a subprocess to vary over time, thereby allowing it to represent the current state of a subprocess with changing state. In other words, when a subprocess becomes `Complete` with a certain value, it may later become `Complete` with a different value. This may happen without the subprocess becoming `InProgress` again, as a result of changes in the values of its subprocesses. If the value of a subprocess changes, it does not mean that the previous value was wrong, or that the subprocess has multiple values, only that the state of the world has changed and therefore the value of the subprocess has changed. A subprocess that executes once and reaches a value that never changes is just a common special case of this. Its state will go through a period of `NotBegun`, followed by `InProgress`, before being `Complete` with a particular value.

Every subprocess has an initial moment of execution, which is the point in time it changes from being `NotBegun` to something else. In the initial state of the world, all subprocesses are `NotBegun`. Not every subprocess goes through a stage of being `InProgress`. For some, the execution is instantaneous. For other processes, their execution has a duration, and they are `InProgress` for a certain amount of time. A subprocess may be `InProgress` because one of its subprocesses is `InProgress`, or it may have a duration in and of itself. To specify that a subprocess has duration, one of two syntactical forms are used. The form "`delay` e_1 `in` e_2", where e_1 is an integer expression and e_2 is an expression, is the same as e_2, except that the process has a duration as determined by the value of e_1. For example, in the expression "`delay` (`uniform` 5) + 4 `in true`", the value is `true`, but the subprocess represented by the expression is `InProgress` for 4 to 8 time units before it takes on the value `true`. The second syntactic form is "`wait` p `in` e", where p is a floating point number between 0 and 1 and e is an expression. This defines a geometric process whereby the subprocess begins in state `InProgress`, and at each time point it takes the value of e with probability p, otherwise it remains `InProgress`.

We need to be careful about specifying the semantics of `delay` and `wait` when the resulting expression e has changing state. For example, suppose computation of a `delay` expression begins at time 0 when e has one value, and before the delay is complete e changes to another value. Do we say that after the original delay time, the `delay` expression takes on the original value of e, and after a further delay it takes on the new value? Or do we say that it only takes on the new value, after the second delay time is complete. We opt for the latter interpretation. The general rule for `delay` is that whenever the value of e changes, we reset a counter that counts up to the delay time. The counter advances one tick per time unit. When the counter reaches the delay time, the entire `delay` expression takes on the value of e. This is an unambiguous and natural intepretation. It means that whenever the value of the `delay` expression changes, it takes on the current value of e and not some historical value.

For a `wait` expression, the issues are similar. Suppose, while computing "`wait` p `in` e", the expression e begins with one value and before the wait is completed it takes on another value. Here the natural interpretation is to say that at every time point, with probability p the entire `wait` expression takes on whatever the current value of e is.

All subprocesses depend on a (possibly empty) set of subprocesses to terminate before they begin. These other subprocesses, for which the first subprocess has to wait, are called the *waiters* of the first subprocess. A waiter may be required to take on specific values for the first subprocess to begin. For example, in an expression of the form "`if` e_1 `then` e_2 `else` e_3", the waiter of e_2 is e_1. This means that the test e_1 must terminate before the consequence e_2 is executed. Furthermore, in order for e_2 to begin execution, e_1 must take the value `true`. If e_1 takes the value `false` and does not change, e_2 is never executed.

In sequential execution, two subprocesses execute one after the other. The execution of the first subprocess must be completely finished before the second one begins. If the first subprocess is in state `NotBegun` or `InProgress`, then the second one is in state `NotBegun`. When the first subprocess becomes `Complete`, the second one begins.

The main language construct for defining sequential execution is the `let` ... `in` construct. In an expression of the form "`let` $x = e_1$ `in` e_2", e_1 is executed first. When the execution of e_1 has completely finished and it has produced a result, e_2 is executed. The subprocess e_1 is the waiter of e_2. Meanwhile, the waiters of e_1 are the same as the waiters of the entire `let` expression. An `if-then-else` expression also employs sequential execution, as described above.

ProPL distinguishes between two kinds of parallel execution. The main language construct for describing the first kind is the `let` ... `and` construct. In general, a `let` expression in ProPL may have the form

`let` $x_1 = e_1$`and` $x_2 = e_2 \ldots$ `and` $x_n = e_n$`in` e

This defines the variables $x_1, \ldots, x_n$ simultaneously. The subprocesses $e_1, \ldots, e_n$ are all executed in parallel. They all begin at the same time. Then, subsequent evaluation of the result subprocess e waits until all of $e_1, \ldots, e_n$ have completed. In other words, $e_1, \ldots, e_n$ are all waiters of e. The waiters of $e_1, \ldots, e_n$ are the same as the waiters of the entire `let` expression. Function applications behave similarly. In executing an expression "$e_0(e_1, \ldots, e_n)$", all the e_i are executed in parallel. Then when all have finished, the body of the function is applied to the values of the arguments.

In the second kind of parallel execution, different subprocesses produce the value of the same variable. The subprocesses evaluate in parallel, and whichever finishes first is accepted. For example, one may try to contact someone by email or telephone. These two methods may be tried in parallel, and the first response produced is accepted as the value. This kind of parallel execution is described by the expression `first` $[e_1, \ldots, e_n]$.

The state of an expression may depend on other expressions at the previous time step. This is achieved using the `prev` expression form. Whenever `prev` appears in front of an expression, it indicates that the value of the expression from the previous time step is taken.

In a `dist` expression, one of a number of subexpressions is stochastically chosen for execution. When the expression is part of a process that is being executed, there are different possible ways to interpret the stochastic choice. One interpretation is that the stochastic choice is made at each time step. At each time step, a separate choice is made as to which subexpression gets chosen. Another interpretation is that the stochastic choice is made once and for all. Once the choice is made, the same subexpression is chosen at every time step. Rather than stipulate a particular interpretation, ProPL allows both interpretations. A `dist` expression is used for the first interpretation, in which a separate selection is made at each time step. The syntax "`select` $[p_1 : e_1, \ldots, p_n : e_n]$" is used for the second interpretation.

The question of which subexpressions get executed varies between the two interpretations. For a `select` expression, since the same subexpression is chosen at every time step, we say that only that subexpression gets evaluated. However, for a `dist` expression, since a different subexpression may be chosen at every time step, and since the subexpressions execute over time, the most natural thing is to say that every subexpression gets evaluated, and the value of one of them is chosen at each particular time point.

A crucial aspect of describing a probabilistic process model is describing what the observations of the process are. ProPL provides two methods for specifying when a process produces observations, corresponding to the two notions of process executing in time and process with state. For a process executing in time, one wants to be able to produce an observation whenever a subprocess begins executing. The syntax "`emit` s; e", where s is a string and e is an expression, achieves this. This expression behaves the same as expression e, except that at the moment it begins executing, the string s is produced as an observation. There may be multiple subprocesses that emit the same observation s; thus this mechanism allows for noise in the observations. The second form of observation is observation of a state variable, which happens at every point in time. The syntax "`observed` e" defines an expression which is the same as e, except that the state of the subprocess corresponding to the expression is always observed. Again, noise is supported because e might define a noisy sensor of some state variable.

Inference

Inference is performed in ProPL through constructing a dynamic Bayesian network (DBN). This DBN contains a node for every subexpression that is evaluated during the course of executing a program. Each expression has as parents its subexpressions from which it is computed. The conditional probability table (CPT) for an expression specifies the rule for computing the expression from its subexpressions. For example, an `if-then-else` expression will have as parents the `if`, `then` and `else` clauses, and its CPT will express the fact that when the `if` clause is true the expression takes on the value of the `then` clause, otherwise it takes on the value of the `else` clause.

In addition, every node in principle has its waiters as additional parents. The full CPT for a node, with all its waiters as parents, expresses the fact that if any of the waiters are `NotBegun` or `InProgress`, the node is `NotBegun`. Furthermore, if one of the waiters is required to take a certain value and has not taken that value, the node is `NotBegun`. Otherwise the node is computed from its subexpressions. When a node has many waiters, this scheme results in a large number of parents. Therefore the dependency on the waiters is decomposed and replaced with a single node. This node is the `or` of a series of nodes, where each represents whether a single waiter has not yet completed or has not achieved its required value. More precisely, let $w_1, \ldots, w_n$ represent the nodes for the different waiters of an expression e. We create a new node w defined by

$$\begin{aligned} w = \quad & (w_1 = \texttt{NotBegun} \lor w_1 = \texttt{InProgress}) \lor \\ & (w_2 = \texttt{NotBegun} \lor w_2 = \texttt{InProgress}) \lor \ldots \\ & (w_n = \texttt{NotBegun} \lor w_n = \texttt{InProgress}) \end{aligned}$$

If w is `true`, the node created for e is `NotBegun`; otherwise the node created for e depends on its subexpressions. The node w can be handled by a DBN engine equipped to deal with noisy-or nodes, or using the standard noisy-or decomposition (Heckerman & Breese 1996).

In fact, for many nodes we do not need to provide the waiters as parents. For expressions that contain subexpressions, the waiters of the expression as a whole will be the same as the waiters for at least one of the subexpressions. For example, in an `if-then-else` expression, the waiters for the entire expression are the same as the waiters for the `if` clause. This is because as soon as the entire expression begins executing, the `if` clause begins executing. Therefore it holds that the entire expression is `NotBegun` if and only if the `if` clause is `NotBegun`. Therefore, instead of making the expression as a whole take the waiters as parents, we can make the `if` clause take them as parents. Similar considerations hold for all expressions with subexpressions. It is only for expressions without subexpressions, i.e. constant expressions and variables, that we need to provide the waiters explicitly. After providing the waiters for these primitive expressions, their effects will trickle down to all the expressions containing them that have the same set of waiters.

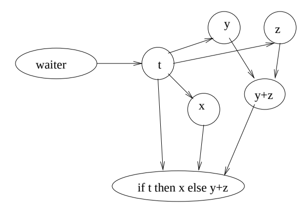

Figure 1: DBN fragment for "`if t then x else y+z`"

Figure 1 shows the constructed DBN fragment for the expression "`if t then x else y+z`". The entire expression has one waiter, but this is passed down to the `if` clause. The waiter becomes a parent of the variable `t`. Next, the `if` clause is the waiter of the `then` and `else` clauses, so `t` becomes a parent of `x`. In the `else` clause, the waiter is passed from `y+z` to its arguments `y` and `z`. The figure does not show the previous time slice since this expression does not contain dependencies on the previous time slice.

We turn now to the DBN construction for particular expression forms. We start with function application. The interesting thing about an expression of the form "$e_0(e_1, \ldots, e_n)$" is that we may have uncertainty about the value of e_0. Therefore we need to consider all possible values of e_0 in order to evaluate the resulting expression. We therefore create a parent corresponding to the body of each possible value of e_0. The expression e_0 is also a parent of the expression as a whole; it serves as a multiplexer to choose among the different bodies. A multiplexer node is a special case of context specific independence (CSI) (Boutilier *et al.* 1996), and can be handled by a DBN engine equipped to deal with CSI. Alternatively, it can be decomposed using a special purpose decomposition for multiplexers.

The actual arguments $e_1, \ldots, e_n$ in the function application become ancestors of the function bodies. They appear wherever a formal argument would have appeared. Fig-

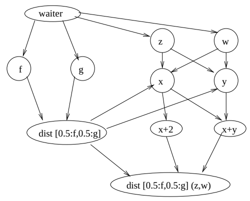

Figure 2: DBN fragment for "`dist[0.5:f, 0.5:g](z,w)`"

ure 2 shows the DBN construction for the function application "`dist[0.5:f, 0.5:g](z,w)`", where `f` is defined by `fun f(x,y) = x+2` and `g` is defined by `fun g(x,y) = x+y`. The two bodies are parents of the expression as a whole, as is the `dist` expression defining the function to be applied. The CPT for the expression as a whole is a multiplexer, with the value of `dist[0.5:f, 0.5:g]` determining whether the expression as a whole takes on the value of `x+2` or `x+y`. The actual arguments `z` is a parent of the formal argument `x`; the CPT of `x` just copies the value of `z` as long as its waiters are ready. Similarly with `w` and `y`. There is an edge from the waiter to `f` and `g` because it is passed down from the application to the `dist` expression to its subexpressions. Similarly there is an edge from the waiter to `z` and `w`. There is an edge from the `dist` expression and from `z` and `w` to `x` and `y` because the expression specifying the function to apply as well as the argument expressions are waiters of the body expressions.

Up to this point, all parents of a node have been in the same time slice. Intertemporal dependencies are introduced by several expression forms. In an expression of the form "`prev` e", the parent is the node representing e at the previous time slice. The CPT copies over the value of the parent.

In an expression of the form "`wait` p `in` e", at each time instance the expression as a whole receives the value of e with probability p. This is achieved by making its parents the node representing e at the current time slice, as well as the node representing the `wait` expression in the previous time slice. The CPT determines that with probability p, the `wait` expression takes on the value of e, while with probability $1-p$ it takes on its previous value.

The DBN construction for `delay` expressions is more complex. Recall that the semantics of "`delay` e_1 `in` e_2" is that every time the value of e_2 changes, a new delay is begun, and it is only when this delay is complete that the value of the `delay` expression takes on the value of e_2. We capture this as follows. First,we introduce a *Changed* node that is true if the value of e_2 has changed since the previous time step. Then we introduce a *Count* node that counts the time since the last change. If *Changed* is true it resets to zero, otherwise it increments the previous *Count*. Next, we introduce a *Target* node whose value is the length of the delay. If *Changed* is true it takes on the value of e_1, otherwise it keeps its previous value. We then introduce a *Ready* node which is true if the delay is complete. It depends on *Count* and *Target*. Finally, the node for the `delay` expression as a whole has as parents the node for e_2, the *Ready* node, and its previous value. If *Ready* is true, it takes on the value of e_2, otherwise it keeps its previous value. The DBN fragment corresponding to this construction is shown in Figure 3.

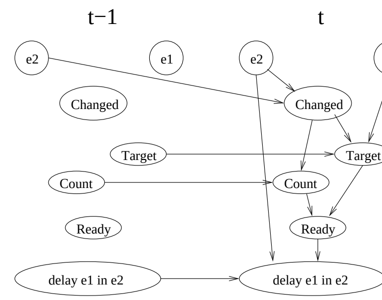

Figure 3: DBN fragment for "`delay` e_1 `in` e_2"

For `select` expressions, recall that the semantics is that the selection of which branch to take is made once and for all. One approach would be to make the selection in the initial time slice. However, this would have the effect that all future random choices are contained in the state at the very beginning. There would be no notion of a random choice being made during the course of evaluation of a process. Therefore, we make the selection at the time evaluation of the `select` expression begins. This is achieved by creating a *Just-Begun* node, which is true only at the moment that the waiters become finished. Then there is a *Selection* node, with parents *Just-Begun* and the previous *Selection*. If *Just-Begun* is false, *Selection* takes on its previous value, otherwise it is distributed over the possible selections according to the parameters of the `select` expression. *Selection* then serves as a multiplexer for choosing one of the subexpressions. Figure 4 shows a DBN fragment for this construction.

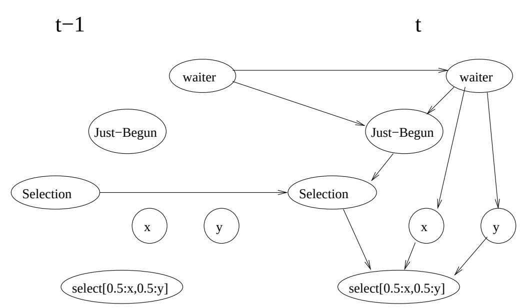

Figure 4: DBN fragment for "`select[0.5:x, 0.5:y]`"

The implementation of `dist` expressions is easier. The selection is made each time and there is no need for a *Just-Begun* node. A similar technique is used for `first` expressions, to remember the value of the first subprocess to complete. We create a chain of nodes, in which the first node in the chain is the previous value of the `first` expression. There is a successive node in the chain for each subexpression of the `first` expression. Each node in the chain (other than the first) has as parents the previous node in the chain, and the corresponding subexpression. The chain is value preserving. If the previous node has a value that is not `NotBegun` or `InProgress`, its successor will have the same value. Since the first node in the chain is the previous value of the expression, this ensures that once the expression is `Complete` with a value the value will be preserved. On the other hand, if the previous node in the chain was `NotBegun` or `InProgress`, and the current subexpression is `Complete`, the current node will take on the value of the subexpression. This ensures that the `first` expression will take on the first value of its subexpressions.

For emitted observations, a special node is created representing the observation. This node is made to be true whenever one of the processes that emits it begins. For this reason, it has as parents the current and previous states of all the processes that emit it. These parents are broken up using an `or` node, in a manner similar to the waiters.

Once the DBN has been constructed, we can use any DBN inference algorithm, such as exact inference (Kjaerulff 1995), the Boyen-Koller algorithm (Boyen & Koller 1998), or particle filtering (PF) (Doucet 1998). However, for any reasonably sized process, the constructed DBN will be too large for exact inference. Therefore an approximate inference algorithm is needed. We focus on PF. Although the DBN is very large, it is largely deterministic. The dimensionality of the stochastic choices need not be too high. Therefore, there is a reasonable chance that PF will work.

Example and Results

As an example, we show part of a ProPL program describing the process of purchasing a laptop. The program was produced by hand translating a SPARK description of the process. In the ProPL program, each of the stages in the process is represented by a function, and breaks up naturally into subprocesses. At the top level is a `purchase` function, which is performed by obtaining criteria from the user and then purchasing a laptop with the given criteria.

```
purchase() =
  emit ‘‘beginning purchase’’;
  let (criteria,s) = get_criteria() in
  s &
  purchase_laptop(criteria)
```

Note that an observation is emitted at the beginning of this and most other functions. This is because the CALO system that executes this process always knows which part of the process it is executing. Next, the `purchase_laptop` process breaks down into finding a laptop that meets the criteria, completing a requisition form for the found laptop, obtaining the appropriate authorizations, placing the order and informing the user about the order. Each subprocess returns both an actual result and a status flag. The execution of subsequent processes only continues if the status flag is true.

```
purchase_laptop(criteria) =
  emit "beginning purchase_laptop";
  let {select,s1} = find_laptop(criteria) in
  s1 &
  let {form,s2} = complete_form(select) in
  s2 &
  let {actual_auth,perceived_auth} =
    obtain_auth(form,select) in
  perceived_auth &
  let s4 = place_order(select,form) in
  s4 &
  inform_user() &
  actual_auth
```

Note that `obtain_auth` returns two status flags, to allow for uncertainty about whether authorizations were actually received. The system might believe authorizations were received when they actually were not, or vice versa. To capture this, the first flag `actual_auth` indicates whether the authorizations were actually received, while the second flag `perceived_auth` indicates whether the system thinks they were received. It is the second flag that determines whether the system will execute the rest of the process, but the first flag determines whether the process is successful.

Authorizations may be obtained from one or two managers. Note that when obtaining authorizations from two managers, the `let-and` construct is used, to indicate that they are obtained in parallel. The subprocess `get_auth` returns two status codes, as described above. Getting an authorization requires sending an email and obtaining a reply.

```
get_auth(manager) =
  let s1 = send_email(manager) in
  if s1
  then obtain_reply(manager)
  else {false, false}
obtain_auth_one_manager() =
  emit "beginning obtain_auth_one_manager";
  get_auth('manager1)
obtain_auth_two_managers () =
  emit "beginning obtain_auth_two_managers";
  let {r1,s1} = get_auth('manager1)
  and {r2,s2} = get_auth('manager2)
  in (r1 & r2, s1 & s2)
obtain_authorizations (form, selection) =
  select [0.5 : obtain_auth_one_manager(),
          0.5 : obtain_auth_two_managers()]
```

Sending an email and obtaining a reply are primitive actions. It is here that uncertainty and time enter the system. These actions might fail to complete correctly. We also have uncertainty over how long they take. The model for `send_email` is

```
send_email(recipient) =
  select [0.8 : delay uniform 100 in true,
          0.2 : delay 100 in false]
```

Sending the email may terminate correctly, in which case the time it takes is uniform between 0 and 99 units. Alternatively, it may timeout after 100 units. The model for `obtain_reply` is a little more complex, as it has four possibilities corresponding to the two status flags. It also includes a noisy observation. The observation corresponds to whether the system thinks a reply was sent. Whatever the result, the delay until a reply follows a geometric process.

```
obtain_reply(replyer) =
  wait 0.05 in
  select [0.6 : emit "acc"; {true,true},
```

```
            0.1 : emit "rej"; {true,false},
            0.25 : emit "rej"; {false,false},
            0.05 : emit "acc"; {false,true}]
```

We can imagine a more sophisticated model for `obtain_reply`, in which obtaining a reply is only possible when the replyer is attentive to email.

```
obtain_reply(replyer) =
  if attentive(replyer)
  then wait 0.05 in ...
  else wait 0 in obtain_reply(replyer)
attentive(person) =
  if prev (attentive(person))
  then dist [0.1 : false, 0.9 : true]
  else dist [0.9 : false, 0.1 : true]
```

The complete ProPL description of the laptop purchase scenario is 443 lines of code. This was produced from a SPARK description that is 734 lines long. As mentioned earlier, the translation from SPARK to ProPL was done by hand. The translation took about four hours. We ran our inference algorithm on the program. The constructed DBN has 7208 nodes in a time slice. While this is a large network, most of the nodes deterministically compute very simple functions. Note that this DBN would be very hard to construct by hand, because of its size and because of the special techniques used in its construction described earlier.

To test the performance, we ran experiments in which the network was simulated for 20 time steps to obtain ground truth. At the same time, PF was run to obtain an approximate representation of the probability distribution over the state of the system at each point in time. We then queried the probability that the laptop purchasing process terminated successfully, given the state after 20 time steps. We also queried the expected time to completion of the process.

We averaged the results over twenty experiments. We used 1000 particles, and the average time for one iteration of PF was about 1 minute. The average error in predicting the probability of success was 7.238%. The average relative error in predicting the time to completion of the process was 7.231%. These are surprisingly good results given that PF normally has a lot of trouble in high dimensions. It seems to be the case that since most of the nodes are deterministic, the effective dimensionality of the domain is much lower.

We also successfully implemented a meeting scheduling SPARK domain in ProPL. The task requires contacting each of the participants about their availability, attempting to find a time that meets all the constraints, selecting a meeting time, and asking all the participants to confirm. Each of these steps can result in failure. The constructed DBN for this scenario has 2820 nodes. The average prediction error was only 0.0275%, and the relative time error was 9.05%.

Discussion and Conclusion

One thing that is needed in coding ProPL programs is models of primitive actions. Ideally these would be learned from examples of the actions taking place. Such a learned model would need to specify the probability of success of an action, the distribution over execution times of the action given that it terminates successfully and given that it fails, and the probability distribution over the value returned.

It would be nice to make the translation from SPARK to ProPL as automatic as possible, can be automated, but there are aspects of the translation that require human intervention. There are design decisions that are made with regard to what elements to include in the probabilistic model and what to leave out. For example, a decision was made that the exact specifications of the laptops returned by a web query are unnecessary; all we need to know is the number of laptops returned. This simplifcation made the model feasible to work with, and it could not have been made automatically.

In future work, we would like to extend ProPL to continuous and asynchronous time. We would also like to incorporate interrupt-driven processes. Finally we would like to allow models in which subprocesses are interleaved with each other, rather than executing in parallel.

In conclusion, we have presented a language for describing probabilistic process models and shown by example that this language is easy to use. We have also developed an inference algorithm for the language and applied the inference algorithm successfully to the example.

References

Boutilier, C.; Friedman, N.; Goldszmidt, M.; ; and Koller, D. 1996. Context-specific independence in Bayesian networks. In *UAI*.

Boyen, X., and Koller, D. 1998. Tractable inference for complex stochastic processes. In *UAI*.

Dean, T., and Kanazawa, K. 1989. A model for reasoning about persistence and causation. *Computational Intelligence* 5:142–150.

Doucet, A. 1998. On sequential simulation-based methods for Bayesian filtering. Technical Report CUED/F-INFENG/TR. 310, Cambridge University Department of Engineering.

Fine, S.; Singer, Y.; and Tishby, N. 1998. The hierarchical Hidden Markov Model: Analysis and applications. *Machine Learning* 32.

Heckerman, D., and Breese, J. 1996. Causal independence for probabilistic assessment and inference using Bayesian networks. *IEEE Transactions on Systems, Man and Cybernetics* 26(6).

Jensen, K. 1997. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Springer-Verlag.

Kjaerulff, U. 1995. dHugin: A computational system for dynamic time-sliced Bayesian networks. *International Journal of Forecasting*.

Morley, D. N., and Myers, K. L. 2004. The SPARK agent framework. In *AAMAS*.

Pfeffer, A. 2001. IBAL: A probabilistic rational programming language. In *IJCAI*.

Rabiner, L. R., and Juang, B.-H. 1986. An introduction to hidden Markov models. *IEEE Transactions on Acoustics Speech, Signal Processing* 3(1):4–16.

Sanghai, S.; Domingos, P.; and Weld, D. 2003. Dynamic probabilistic relational models. In *IJCAI*.