Multi-State, Path-Dependent Models

Ross D. Boylan

January 18, 2010
11:13

Contents

1 Introduction

Some of the following definitions are essential, not merely for correct formatting but for avoiding a fatal FWEAVE bug triggered by **PathGenerator**⟨*A*, *B*⟩::*foo*( ) and others. The type must be **int** not **class** for the right production rules to kick in.

"mspath.C" 1 ≡
  @f *FooNode class*
  @f *PathGenerator int*
  @f *Array1D int*
  @f *Array2D int*
  @f *std int*
  @f *less int*
  @f *Covariates int*
  @f *Coefficients int*
  @f *LinearProduct int*
  @f *Coefficients int*
  @f *Id int*
  @f *IDList int*
  @f *test int*
  @f *AllocCounter int*

1.1 The Probability Process

Individuals, or, more generally, cases, occupy exactly one state at each time. They move between states over time, and we have histories of their observed states and covariates over time. The histories consist of observations at selected times; the state of the individual at other times is unknown.[1] There are a relatively small number of possible states; formally, the state space is discrete.

We assume that the observation times are independent of the underlying process, including covariates. Two possible exceptions are the first and last observation. The first time may be the origin time of the process, and the last time may be the exact time of entry into the final state (if "death"). Otherwise, we do not know the actual time of state transitions.

The observations may be wrong, so we have an observed state and an unobserved true state.

We assume that the histories of different individuals are independent.

We assume some probability process governs the evolution of true states for a particular individual. The probabilities of different transitions may depend on the current state, exogenous covariates (including time-varying ones) for the individual, and the history up to the time of the transition. Parameters describe the relation between covariates (incuding history) and the transition probabilities. In principle, there may be an individual-specific effect, though we do not currently handle that case.

For our study, history-dependent covariates of interest include age, time in state, time to reach state, recent alcohol exposure, and cumulative alcohol exposure. While conventional Markov models can be extended to include some history dependence on the covariates (e.g., alcohol consumption), they can not handle dependence on the path itself (e.g., time in state, time to reach state). And they handle variables with a known evolution through time (e.g., age) crudely, since they don't allow for the fact that they vary between observations.

We also assume a measurement error model, giving the chance of an observed state given a true state. In general, such a model may also depend on the individual, the covariates, and the history. In practice, we tend to use fairly simple models, though the program supports more complex ones.

We assume the process governing transitions among true states, the process governing what state we observe given a true state, and the process generating observation times are all independent of each other.

The purpose of this program is to compute the log-likelihood of a set of observations, given parameter values. Higher-level programs can use this computation to derive maximum-likelihood estimates of the parameters, along with associated information.

The likelihood so computed corresponds to a simple design in which we follow each case from the start of the process. Often we only see cases that have survived until the time we first observe them. In such a design one should compute the likelihood conditional on survival to first observation. The conditional likelihood is the naive likelihood divided by the probability of survival to first observation time.[2] This feature will not be in the initial implementation.

See §2.2, page 13, for the exact form of the model.

[1] Though the model may imply certain constraints on the possible states at those other times.

[2] Let D be the observed data for one case, and τ_1 the time of first observation. The naive approach is to compute $\Pr(D)$. Let S (survival) stand for the event that time of death exceeds τ_1. Then we should compute

$$\Pr(D|S) = \frac{\Pr(D \cap S) = \Pr(D)}{\Pr(S)}.$$

1.2 High Level Objects From the Problem Domain

The basic real objects for this problem are

States These are the discrete states people can occupy. At the moment, the main potential difference is simply the number of them, and perhaps their labelling.

People or, more generally, cases. Each has an observed sample path through the state space. In the original data, the time of each observation is a continuous variable. There may also be accompanying covariates. These may be either

- constant, observed initially and subsequently invariant;
- stepped, observed at the time of each state observation and assumed constant until the next observation;
- continuously varying, with some assumed change (e.g., linear in time) between observations.

We do not considered missing or mismeasured covariates.

True Process Model Describes how people move through the unobserved true state space. I think of this as being in continuous time, but we might consider a discrete version. This has significant properties such as whether it is Markov and what transitions are legal. It may have auxiliary constraints, such as that certain covariate effects are equal. We particularly want to cover two cases:

1. Time in current stage as a predictor of transition risk to next.
2. Time in previous stage as a predictor of transition risk to next, e.g., a short time in stage 0 implies greater hazard for the transition from 1 to 2.[3]

Notice that the second item is heading toward a random person effect; we might want to explore this eventually. SAS proc *NLMIXED* is good at that case.

Measurement Model gives the conditional probability of an observed state, given a true one. We assume this is independent of anything else about the process, including previous measurements.

Truncation Process implies that we don't observe some cases because they don't make it into the study. We will derive this from the true process model when we get around to dealing with truncation.

Parameters Of the different models.

[3] In the current implementation every transition must have the same covariates. For transitions out of the first state, the effects of time in previous state should be constrained to be 0.

1.3 High Level Objects To Implement the Computation

Then there are artifacts necessary to compute likelihoods in a discrete-time approximation. I'll initially state these with great generality.

Node A particular possible true state at a particular time for a person.

Path A real or hypothetical series of states for a particular person at particular times. May be incomplete. Paths are made of Nodes, but Nodes may be in several paths, given the branching structure of the problem. So looking backward in time a node is only on one path, but looking forward it may be in several.
Nodes will record all relevant features of their history, and also know about all their successor notes.

Node Data The elements of the history that need to be recorded for each node or time point(e.g., time in state). Obviously a function of the true process model.

Multiple Evaluations This is part of node data, but this time reflecting the fact that we could evaluate several sets of parameters at once for each run through the tree.

Traversal Strategy A method for generating nodes and paths in a particular sequence. It may not be necessary to create and hold all possible paths with all their nodes at once.

Time Point Generator Determines the time of steps in the path. Potentially this is different for each individual, and even for each path.

Successor Generator Determines the states of nodes which may occur next after a particular Node, given the time of the next step. This does not know about the probabilities of the different transitions. It's an open issue how this should interact with knowledge that might trim the tree. A simple approach would be to have a successor generator that knows nothing of such constraints, leaving it to others to trim.

Transition Probabilities Calculates the probability of each possible successor node, given the model.

Trimmer Knows which paths are inconsistent with the observed data. Note this is a function of several other parts of the model, potentially. Open issue: should it trim literally all impossible nodes, or just most of them? For example, given a large state jump in a short time and a model that only allows transitions a single step at a time, one can put tighter constraints on the intermediate steps than that they lie between the two observed states. The simpler alternative may be to permit some slack, and simply kill the path when it gets to a known impossible state.
Also note the possibility of solving the inverse problem of calculating the probability of all infeasible paths. Could be quicker in some cases, but doing this with the measurement model would require some adjustments.

Caching everywhere is possible

Qualitative Characteristics Each component should be able to answer questions that permit optimizations (e.g., every time step is same size), describe independence (e.g. process is Markov), or regularities.

Search Strategy Perhaps to accommodate the different possible optimizations flexibly, a standard search order for information could be instituded: node, person's time, person, general time, general.

Manager Knits all the pieces together and provides root object for the data.

1.4 Design Decisions

1.4.1 Architecture

At first glance, it is tempting to define each of the components above as an abstract class, and provide a basic set of protocols (messages) governing the interaction between them. Aside from the problems of providing an adequate, flexible abstract structure, C++ makes it difficult to implement such an approach. It is easiest to discuss why with a concrete example.

First, suppose we have different kinds of Nodes. For any node, we want to access previous and next nodes. So we could define a method *previous*( ) to return the prior node (really a pointer to it, so we can returnn *NULL* at the base of the tree). This could have type *AbstractNode*. However, for class **FooNode** we want to return a pointer to **FooNode**.

Thanks to some relatively recent changes to the C++ standard, it is permissible for a virtual function to have a return type that is more specific than the virtual function it overrides. So **FooNode** ::*previous* can return **FooNode** $*$ if **FooNode** is a subclass of *AbstractNode*.

Even here, we have two problems. Compilers may not yet reliably implement the new standard. And even if they do, we may need to downcast the result if we access through the *AbstractNode* interface.

Second, consider the problem of successor nodes. We mant want to return something that could iterate over them. And we might want the exact type of storage used for the nodes to be pluggable. If we want to use C++ library containers (aka the Standard Template Library), the different iterators have no necessary subclass relation. Suppose *nextFirst*( ) returned an iterator pointing to the first next node. **std**::*list* $<$ **FooNode** $>$ is not a subclass of **std**::*list* $<$ *AbstractNode* $>$. It's not entirely clear to me if the iterator types of these are subclasses of each other, since they likely have type pointer to Node.

Even more clearly, if I would like to return the entire *list* or some other container (e.g., in a *successorNodes*( ) call), I'm stuck. In this particular case that may not be likely, but consider the problem of returning the evaluation data associated with a Node. For example, if we evaluate multiple sets of parameters at once, the data might be a **double** for a single parameter set, a **std**:: **vector** ⟨**double**⟩ for others, and a *matrix* for a grid of parameters. These types have no shared class.

I could define each such structure as a subclass of a common class, but they would still have no common functions. The essential problem is that C++ inheritance works basically by method signature (including all argument types), not just name.

In cases where there is a commonality of names but not type signatures, we can not use standard inheritance in C++. However, this is, from one point of view, the problem that templates were designed to solve. As the system develops, we may use templates a lot. For a first cut they are unnecessary and complicating, so we'll tend to leave them out.

Template-based approaches likely will make components pluggable only at compile-time, not at run-time.

Consider particularly how to handle the path-dependent data (aka **ModelData**) and the **EvaluationData**. The latter might be an array of likelihoods, one for each set of parameters passed in. Model and Evaluation Data are roughly independent of each other. In each case there might be nothing (no path-dependent variables; no likelihood if just counting **Node**'s) or some particular kind of data (a **double**, **vector** ⟨**double**⟩).[4] Several approaches suggest themselves:

[4]The "nothing" part poses a challenge for templates. Consider **template** ⟨**class ModelData**⟩ **Node** { **ModelData** *d*; } ;. **ModelData** has to be something in that scenario—not nothing. However, conditional template instation does offer a way to make the inclusion of the **ModelData** *d* line conditional.

Most Complex Case Implementing only the most complex case and treating all others as special cases of it is simple and probably imposes relatively little overhead for the calculation of the simpler cases. It may be faster than approaches relying on virtual functions or dispatching. It saves all the hard thinking involved in the other approaches listed here, and is less demanding of the compiler. It minimizes the object could bloat that templates could create. It is somewhat unaesthetic, since it doesn't really do the hard work of factoring the problem. It may be inadequate for future demands (e.g., if **EvaluationData** is more than a **double**).

Policies This is a template technique in which a class includes all the types needed and static code to implement different policies (e.g., do or don't bother to computer path-dependent data). This is attractive in that it explicitly recognizes that certain combinations of classes must be used together. For example, a **Node** with path-dependent data requires a **PathGenerator** that knows how to make and evaluate such **Node**'s, along with an appropriate **Model**. Because it is compile-time, it offers fast run-time performance, though with some code bloat.

This is really only the beginning of a solution. Would I use templates or ordinary classes for most of the types involved? The policy-based functions only defer some issues. For example, the constructor for a **Node** needs to know the number of path-dependent variables in some cases but not others, so the caller of the static function still needs to switch on the type somehow (I suppose it could pass in a **this** argument which could sometimes be ignored).

Inheritance I could create classes by mixing in elements related to the evaluation data and the model data.

Composition I could build more complex objects (**Node**, **PathGenerator**) to include components specifically relating to the two parts. One problem is that some things, such as **Node** creation, need to know about both simultaneously.

Some implementations might involve needless combinatoric explosion. For example, some methods may dependent only on the type of **ModelData**. If the methods are templated on **Node**, or implemented in some other way, they might be repeated for each type of **EvaluationData** as well, even though the code would be the same.

Of course, it's not entirely clear the code would be the same. Even though a particular routine may only care about the type of **ModelData** it may pass the **Node** on to other routines that care about the **EvaluationData**. If so, there may need to be different code for each case, to get the correct **Node** type.

I considered double-dispatching as a possible solution. One disadvantage is that it increases the amount each class must know about the others. It also seems possible the same effects could be achieved with template policies.

Basically, I spent a lot of time thinking about the fancier implementations, and never came up with a definitive approach. So I'm going with the most straightforward approach, the first one listed above.

1.4.2 Dependencies

This problem, like many others, leads to a thicket of interdependencies among classes, including potential cyclic depedencies. As always, it is desirable to limit such dependencies. To clarify my thinking, let me list the types of dependencies:

Template A templated class may depend on another type, which may in turn be a template. Are template dependencies better or worse than others? Why use a template dependency rather than another form? See the discussion in the previous section for some considerations. While at first it seems a template dependency is stronger than a simple using dependency, it is weaker in the important sense of not depending on the exact types of method signatures. It permits interoperation of classes that share a common vocabulary (set of methods names) even when the types of the arguments don't match.

Using A class may use another class. The strength of the relation may range from tight (inheritance) to very weak (only holds a pointer or reference which is passed on). In the very weak case it may suffice to declare a class without defining it. In most others, we actually need the header file of the other class.

Typedef Sometimes we may use a **typedef**'d type. The suggestion that we can substitude a different type is often incorrect, however, so the dependency is pretty tight. For example, if we change the type **TEvaluationData** from a **double** to a matrix, a lot of code will need to change as well—including even the **R** interface.

Specific Templated Types If we use templates we must at some point fill in the specifics of the types. In some cases, even the implementation may be deferred to the specific type, and clients may be written to a particular template instantiation as well, leaving the general case unaddressed.

Abstract Classes These represent another way to control dependencies. For reasons given in the previous section, they may not help much in this case.

Combinations For example, a template may specify an abstract class, or inherit from an abstract class.

The general question is which of these forms to express relations in. Stated slightly differently, if a class depends on another, the choices are:

Direct Dependence Just include the header for the other class.

Template Include other class as a type only. What will dependencies look like for template instances?

Abstract Class Relate class to the abstract interface of the other class.

A variation is to use a typedef for the class; that doesn't really address how to get the header file. That issue is one of the limits of using typedef's. A weak form of dependence is one where only the name of the class is required, because it is an opaque pointer or reference.

As a specific example of how to limit dependencies, consider **Node**, which uses **ModelData**. It could obtain this from the **Model** class, but if we separately specify the type **TModelData** than **Node** and **Model** can both depend on it without depending on each other. The general principle is to avoid using a complicated type (**Model**) when only one part of it (**Model**::**TModelData**) is necessary.

1.4.3 Containers, Iterators, and Pointers

Often we will will want to refer to an object that is in some container. We may also want to get the next object. This section discusses how to manage those references.

The goal is make it easy to change the way the container operates without changing all its clients. There are at least three kinds of containers: containers of objects; containers of pointers to objects; and containers of pointers to pointers to objects. A linked list of pointers to data is an example of the last one. Correspondingly, iterators may refer to data indirectly or doubly indirectly (for the second, again consider the linked list).

Here are some ways to manage references:

Index Number This is only good for vectors, doesn't solve the double indirection problem, and requires supplementary information (namely the identity of the vector) to be useful. It is robust against resizing and easy to increment.

Pointer to Data Though easy and conventional (I mean a simple pointer, not an iterator), it is not necessarily incrementable to get the next item. Fragile on resizing.

Iterator This is incrementable and requires no additional context to get the data (though it does require context to bounds check). But it doesn't cope with single vs. double indirection. Iterators are fragile on resizing.

Smart Iterator One could subclass regular iterators to provide one with additional accessors to get the data, thereby hiding the single vs. double indirection problem. This could even incorporate bounds checking. However, it would require overriding quite a bit of code to return the smart iterators and/or carry the risk of getting a simple iterator.

Iterator and Smart Container This supplements the container with a couple of calls to dereference the iterator. So this requires additional context (namely the container) for use. An advantage is that it requires less programming than the smart iterators.

Only the last two solutions really deal with the problem. Fortunately, Thorsten Ottosen (nesotto *ptr_container* boost library, available from version 1.33, implemented a bunch of pointer containers that automatically perform indirection (and return indrected pointers) behind the scenes. Our source-tree includes a pre-release version for use with earlier versions of the library. A happy side effect is that we don't need to do anything special to get the standard $<$ operator to work properly on our collections.

If an object is *foo*, we'll use *iFoo* for an iterator to *foo* and *pFoo* for a simple pointer to *foo*.

Because iterators can be invalidated as containers grow, *pre-allocate containers, particularly vectors, so they won't need to grow.* Since we generally use containers of pointers, the issue is less pressing.

1.4.4 Threads

The core algorithm here of iterating through all possible paths is inherently sequential.[5] The assumption is that parallelism will be achieved at a higher level by farming out different cases, or different sets of parameters, to different threads.

Currently, the system is not thread-safe, because many components that should do not support multiple threads of execution. For example, if you attempt to use the same **Model** instance in two threads there will be trouble; **Model** stores thread-specific state in itself.

The implementation is thread-safe in the sense that you can use different objects of the same class in different threads. At least, that is safe if the rest of the library and infrastructure (e.g., the containers libraries) are safe in this sense.

The plan is to move all thread-specific information into **Environment**. First, some objects (**Path**, **TimeSteps**, *Covariate*) are inherently thread-specific and **Environment** owns them. They may contain thread-specific data without problem, because each thread should have its own copy of these objects.

Other objects, such as **Model**, **Specification** and **Coefficients** could be shared across threads with careful implementation. These and similar classes will store their thread-specific state in **Environment**. The class of these state objects will be nested within the main class definition and called **State**, e.g., **Specification**::**State**. So there is one **Specification**, but possibly many **Specification**::**State**'s, one per thread.

It may be desirable to make some of these public, since the class involved may not be the only one that wants to peak at the information. For example, the **Specification** might want to know about the state of the **Model**.

See the discussion of **Environment**, §4.18, page 122, for more information.

1.4.5 Path-Dependent Variables and other Covariates

There are several types of covariates:

- Constant "covariates," such as race or sex that do not actually vary, except across cases;
- Observed, or partially observed, covariates that change over time. In some cases we might want to assume they vary continuously and adjust for that analytically; at the moment, we don't.
- Path-dependent covariates which depend on the particular evolution of the stochastic process (e.g., time in current state).

These different types of variables differ in how frequently they change, and thus in how frequently quantities derived from them need to be computed.

[5] Different branches of the tree could be handled in parallel.

User Specification of Path-Dependent Variables The easiest way for the user to specify path-dependent variables is to give each of them a name, and allow them to be included in the `covariates` term in **R**. Ideally, the specification could include analytic transformations of the variable such as taking logs, squaring, etc.

All the machinery of constraints and initial values would then carry over to the new variables.

Permitting this level of generality initially is undesirable. First, it's a lot of work. Second, supporting arbitrary analytic transformations would require callbacks to **R** from the code at each node evaluation, and would likely be very slow.

Internals There are two ways to look at how to handle path-dependent variables: from the top down, or the bottom up. From the top one considers the user interface and the attendant issues of managing constraints. From the bottom, one considers the speed of the core computations.

Because speed is so important, and because higher levels can manage the necessary transformations, the lower level concerns will drive the design. In particular, I will seek to minimize recomputations. I should note that it is not clear, *a priori*, that this will produce the fastest runs. The G5 architecture (and many other modern ones, I suspect) pays rather heavily for conditionals, so it's possible that fewer conditionals and more computation would be faster. Since testing so far has shown reducing computations pays big speed dividends, I will continue to favor it.

The first implication is that I do not want one *Covariate* that freely mixes path-dependent and regular covariates. To do so would require many recomputations involving values that don't change. So I will use 2 sets of **Coefficients** and associated **Covariates**.

Not all path-dependent variables may need to be computed in a given run; if the computation is expensive (as it may be if taking logs) I probably want to skip it.

Fun facts:

1. Many runs will not use all path-dependent variables. Some will use none.
2. Some path-dependent variables depend on other path-dependent variables (e.g., $\ln(x)$ requires x) and others require previous history.
3. Some of the variables we compute mmay not be covariates the user is interested in (e.g., they want $\ln(x)$ and don't need x).

To track this internally we need

- The set of required variables to compute.
- The set of variables to offer as covariates (and the order in which to offer them).
- Interdependencies. We need to assure that if b depends on a that we compute a first *and* that b knows how to find a.

1.5 C++ Language Notes

Templates can define functions, classes, or their members, but can not define types (e.g., **template** ⟨**class T** ⟩ **typedef std**:: **vector** < **T** > *TType* appears to be outside the language as defined in the first paragraph of section 14 of the standard).

Consider **template** ⟨**typename T** ⟩ **class Foo**.

Within its definition, **Foo** means **Foo**⟨ **T** ⟩ (standard section 14.6.1).

But names referenced through **T** , like **T** ::*Pointer* are assumed *not* to be types unless explicitly qualified by **typename**. so one must say **typename T** ::*Pointer aPointer*; or **typedef typename T** ::*Pointer* **TPointer**; (standard 14.6, paragraph 2).

It appears that default parameters in templates (I don't plan to use any) are considered definitions *outside* the template scope.

Suppose we a have a class member function definition outside of the main template declaration. If the return type involves **Foo**, or the arguments involve **Foo**, do they need to be qualified? I believe the return value is outside of scope, so must be qualified, as in **Foo**⟨ **T** ⟩. I also have a feeling that arguments, since they occur after the name is declared, don't need it. I can't find the discussion I saw, which said the return type was outside of scope. So I think a definition outside the scope of the initial declaration would look like this:
template ⟨**typename T** ⟩ **Foo**⟨ **T** ⟩
Foo⟨ **T** ⟩::*someFunction*(**Foo** *f*).

Template parameters can be used immediately, e.g., in return values **template** ⟨**typename T** ⟩ **T** **Foo**⟨ **T** ⟩::*function*( ) (14.6.1 paragraph 3).

static member functions may be referred to, like regular member functions, via an object reference (9.4 paragraph 2).

To access operators inside a member function use, e.g., (∗**this**)[*i*]. I tried many other constructs that failed, including { *i*], **this**→[*i*], and **operator** [*i*]. **operator** [ ](*i*) also works.

operator [ ] accepts exactly one argument (standard 13.5.5).

Suppose **class A** has member functions *foo* and *bar*, where *foo* calls *bar*. Suppose **class B**, a subclass of **A**, (re)defines *bar*. If *bar* is not declared virtual, **B** ( ).*foo*( ) will invoke **A**::*bar*. If it is declared virtual, it will invoke **B**::*bar*. Finally, if it is virtual, **static_cast**⟨**A**⟩ (**B** ( )).*foo*( ) again invokes **A**::*bar*.

If you attempt to define, for example, **operator** + as both a member and a free function you get an error that this is ambiguous. So if I want to redefine it for templates (e.g., sets), I need to do it as a free function, since that's how the templates do it.

To use < *utility* >'s automatic definition of comparison operators, say **using namespace std** ::*rel_ops*;. Great how they are hidden in a subspace.[6]

std::**less** appears to be allergic to adding reference or const qualifications to the type. The template definition adds those qualifiers automatically.

Selection of overloaded functions is done at compile time. If **class B** is a base of **class C** and a function is called through a **B**&, it will resolve to a function of **B**, if available, rather than a function of **C**, even if the dynamic type of the argument is **C**.

[6]For the templates in **std**::*rel_ops* to find your operators, they must be defined at class scope, the enclosing namespace's scope, or global scope. If they are defined in a nested scope, you must hoist them up with, e.g., **using B**::**operator** < ; in the upper level namespace (the one containing **B**).

2 First Cut

2.1 User Interface and Building Blocks

The user will see this as an **R** package, with an interface very similar to **msm** 0.3.3. This should make it easy to compare the results of this model with those of **msm**.

Underneath, we can re-use some of the machinery of **msm**, as well as much of its design about how to specify the problem. This means that we can specify arbitrary patterns of allowed transitions and measurement errors, and place constraints upon those values. We can have covariates affecting transition rates and measurement errors.

msm does things other than calculating likelihoods and estimating parameters; these routines do not.

2.2 Discrete Time Approximation

See §1.1, page 2, for the general setup of the problem. It is non-Markov, with transition probabilities depending on the history of the path.[7]

msm assumes an underlying continuous time model of transitions between states, and evaluates it analytically. But it is a Markov model, and that is critical to making the analytics tractable. The history dependence of the current problem precludes that approach.[8] Instead, we rely on a discrete time approximation to the process. Because there are a fixed number of time steps, we can enumerate all possible paths and compute their probabilities. The values of path-dependent variables are well-defined on each path, and so it's relatively straightforward to work with them.

Formally, divide the time path for an individual into N intervals with endpoints at $t_0, t_1, t_2, \ldots, t_N$.

Let D be the observed data for that individual; it consists of a set of observations $\{(o_m, \tau_m, x_m)\}$ where the m'th measurement is of state o_m at time τ_m. Measured covariates are x_m (which may be a vector) at that time. The data also include whether or not the final observation is an exact transition time (e.g., time of death is measured exactly).

Ordinarily there are many more time points than observations. Some auxilary model is necessary to relate the two. The current **R** implementation rounds observed τ to the nearest t, throwing out duplicates[9] It uses the last observed covariate values for steps that do not have an observation. The C++ code here assumes there will be at most one observation per time point.

D_X will refer to the exogenous data, $\{(\tau_m, x_m)\}$. D_Y refers to the endogenous data, $\{o_m\}$.

[7] Formally, measurement probabilities may also depend on the path. It seems unlikely anyone would use this feature. The current code has most of the hooks needed for such a facility, if desired.

[8] Conceivably, the current model, though much harder, has an analytic solution. We are not clever enough to come up with one, and it seems likely any such solution would be limited to special cases.

[9] Currently all but the last overlapping observation are thrown out, except in the first interval the first observation is retained.

2.2.1 The All-Paths Approach

Consider a single individual or case.[10] Let P_j refer to a particular path of true states across all time points. There are J possible paths consistent with the data under the transition and measurement models. Our approach is to enumerate all possible paths and compute the likelihood as

$$\Pr(D_Y|D_X) = \sum_{j=1}^{J} \Pr(D_Y \cap P_j|D_X).$$

There are usually many such paths, even though we observe only a single sequence of states through time. This is so for a couple of reasons. First, our observations cover only a small number of the time points on the paths, leaving many possible paths in between. For example, if there is a transition between states, that transition could happen anywhere between the time of the initial and the time of the final state. Each possibility generates at least one path. Second, since measurement is usually imprecise, each observed state corresponds to several possible true states. Each possible true state generates at least one path.

Define $D(n)$ as the observed data through time t_n, and $P_j(n)$ the path through that time. s_{jn} is the state of the j'th path at step n, and o_m is the observed state at the m'th measurement. With the independence assumptions outlined at the beginning,

$$\Pr(P_j \cap D_Y|D_X) = \Pr(P_j|D_X)\Pr(D_Y|P_j, D_X) \quad (1)$$

$$\Pr(P_j|D_X) = \Pr(s_{j0}) \prod_{n=1}^{N} \Pr\left(s_{jn}|P_j(n-1), D_X(n), t_n\right) \quad (2)$$

$$\Pr(D_Y|P_j, D_X) = \prod_{m=1}^{M} \Pr\left(o_m|P_j(n_m), D_X(n_m)\right), \quad (3)$$

where there are M distinct[11] observations, and n_m is defined such that t_{n_m} is the time of observation m. Recall that N is the number of time intervals. So the second equation has as many terms as time intervals, while the third has only as many terms as observations.

The first equation says that the probability of a particular path and the data is the probability of the (unobserved) path times the probability of the data given the path. The next two equations give expressions for those two terms.

The second equation gives the transition model for moves between true states along the path. This is the probability of the initial state times the probability of each transition along the path. In general, the transition probability depends on the history of the data and path up to the point of transition. Most analyses use only selected aspects of that history. The implementation here makes the probability of entering s_{jn} depend on the previous state, $s_{j,n-1}$ and a set of covariates for time n. The covariates include both observational data and summary measures of the path history. Fuller discussion of the use of covariates at time n, as opposed to $n-1$, appears in §2.2.6, page 20.

The third equation gives the measurement model. While it too allows very general dependence on history, in practice it often reduces to

$$\Pr(D_Y|P_j, D_X) = \prod_{m=1}^{M} \Pr(o_m|s_{n_m}), \quad (4)$$

[10] The likelihood of the sample is the product of individual likelihoods, given the assumption of independence across individuals.

[11] If time τ of the observations is initially continuous, several observations may end up on the same discrete time point t. If so, the set of observations needs to be reduced before the main analysis.

the probability of an observed state depends on the true state at the same time.

The first three equations above are mostly consequences of the laws of probability, rather than statements of structure we are imposing on the model. The second equation does add a fixed initial probability, independent of covariates. The second and third equations exclude some exotic forms of history dependence.[12]

Our implementation allows only certain functional forms for the relations above; the point of this discussion is to explain why the resulting calculations produce the desired probability.

There are some fine points about the transition model. First, if the time of the final transition is observed exactly (which implies the resulting state is measured exactly), then only paths consistent with that timing are considered.

Second, at the initial point $n = 0$ there are no prior data to condition on. One must specify the initial state probabilities as an input to the model; obviously, they should sum to 1 across all possible states.[13]

Third, sometimes the first observation will be after t_0, the start of the process. This means we lack direct observation of state or covariates at the start; we may even not know t_0. We'll worry about that when it happens. On the upside, often we do know the state at t_0 even if we weren't there to observe it (e.g., "healthy"). We know fixed covariates too (race, sex),[14] and sometimes we can make assumptions about others (e.g., no risky behavior).

2.2.2 Derivatives

The preceding analysis may be expressed compactly as

$$p_j = \Pr(s_{j0}) \prod_{n=1}^{N} q_{jn} \prod_{m=1}^{M} r_{jm}. \tag{5}$$

p_j, the probability of path J, is a constant (usually 1) times the product of q_{jn}, the probabilities of the transitions at each step n of the path, and r_{jm}, the probabilities of the observed state for each observation m along the path. Note q_{jn} is the probability of a transition from $n-1$ to n.

Recalling that $\frac{dln(y)}{dx} = \frac{1}{y}\frac{dy}{dx}$, and thus $\frac{dy}{dx} = y\frac{dln(y)}{dx}$,

$$\frac{\partial p_j}{\partial \beta} = p_j \left\{ \sum_{n=1}^{N} \frac{\partial \ln(q_{jn})}{\partial \beta} + \sum_{m=1}^{M} \frac{\partial \ln(r_{jm})}{\partial \beta} \right\}. \tag{6}$$

If there is no measurement error, the r_{jm} terms drop out of both equations (5) and (6). If there is measurement error, but the error probabilities are fixed, those terms drop out of (6). β are the model parameters, so the previous expression gives the gradient. The exact functional form of the derivatives depends on the model details; they turn out to be convenient to compute, as we will see shortly.

[12] Specifically,

- The probability does not depend on the history of observed states.
- Transition probabilities don't depend on the future (see also §2.2.6, page 20).

[13] It's easiest to do this once for all cases, but this raises the possibility that some of the true states at $n = 0$ will be inconsistent with the observations. If so, no paths can start from those states. The remaining initial probabilities should be upweighted to 1. If none of the initial states are consistent with the data it is an error.

[14] well, usually fixed!

2.2.3 Discrete or Continuous Model Core?

This section considers two related questions. How should we choose the time steps—in particular, should they be uniform? And should the core model, used to derive the probabilities discussed above, be continuous or discrete?

Our first cut will use uniform time steps and a discrete core model. However, this is not the only possibility.

Using variable time steps, i.e., allowing the distance between t_i to vary both within a case and between cases, has some advantages. First, one can match the observation times exactly by ensuring that the they are among the t_i. In contrast, with a uniform grid (e.g., every 6 months) one must round observation time to the grid. This loses information. It may also allow the future to affect the past, if both are rounded to the same time. Finally, several observations may occur within a single uniform grid interval; one must throw some of them out or combine them, thus losing information again.

Second, if one is free to vary the grid size, one can use bigger steps when appropriate. For example, if one is computing the probability of truncation (death keeps someone from being sampled) for a disease that typically develops in adulthood, one might need to look at the process starting from birth. With a uniform grid size one must choose between a fine grain and a huge number of paths, or a coarse grain and poor approximation to the process. With a non-uniform grid, one could use very large steps in the early years and finer steps later on.

On the other hand, with a variable size grid it is harder to compute the transition probabilities; obviously in a sensible model they depend on the size of the step. A natural solution is to specify a core model in terms of instantaneous transition probabilities, and use it to derive the probabilities indicated in the previous section. One might choose to do this even with a uniform grid. It offers the side-benefit of being fairly directly comparable to the model **msm** uses.

There are several problems with this approach. They all come down to the fact that continous and discrete models are inconsistent.

Most obviously, continuous models say transitions can occur at any time, while discrete models say they can only occur at designated t_i.

The unfortunate implication is that any number of transitions can occur in a fixed interval under the continuous model. Even if the continuous model only allows instantaneous transitions between adjacent states, transitions between any state are possible over an interval. Translating this to a discrete model would require either allowing all possible transitions, and thus a huge number of paths, or lopping off some of the transitions that the continous model says are possible.

Additionally, in deriving transition probabilities over a finite interval, one would probably need to assume the process was Markov over that interval. That contradicts the assumption that the process is non-Markov.

A final, practical problem, is that a non-uniform grid requires one to compute transition probabilities over each different interval. With a uniform grid, one can use a single set of transition probabilities. This increases the computational burden of the non-uniform approach.[15]

A model cast directly in terms of equal-sized, discrete time steps suffers none of the internal contradictions mentioned above. To a certain extent this is because issues have been swept under the rug (at least if you

[15] However, in a non-Markov model one probably needs to compute the transition probabilities at each step anyway, so this factor may not make too much difference.

believe the "true" model is in continuous time). However, such models are well understood, so we'll use them initially.

The key point, at least for me, is that the correct analytic expression of the likelihood with a continuous-time core model is an infinite dimensional integral. Intuitively, the all-paths approach should approximate that integral as the mesh gets smaller. However, rigorously characterizing the relation between the true answer and the approximation is difficult. This difficulty casts a pall of uncertainty over the results of such an approach. We avoid such doubt by avoiding the approach.

2.2.4 Parametric Forms

Evaluation of the model comes down to evaluating a bunch of terms of the form $\Pr(s1|s0, \mathbf{X})$.

For transitions, $s1$ is the destination state and $s0$ the origin state; for measurement, $s1$ is the observed state and $s0$ the true state. $s0$ and $s1$ may be the same. $\mathbf{X}$ includes not only the observed covariates (lagged for transitions) but a constant term of 1 and all relevant parts of the the path history.[16]

The simplest parametric form is simply a fixed model in which values of $\Pr(s1|s0)$ are specified in advance for all possible combinations.[17] Covariates and history are ignored. We use this for our error model, although the program permits the more general possibilities to which we now turn.[18]

We will use either the logit or complementary log-log functional forms to specify the relation between a set of parameters, β, and covariates, $\mathbf{X}$.

Consider first a case involving binary outcomes, so that p is the probability of moving up one state (vs staying put) or an observation being misclassified by down one state (vs being correct). The logit is

$$\log\left(\frac{p}{1-p}\right) = \mathbf{X}\beta, \text{ which implies} \tag{7}$$

$$p = \frac{1}{1+e^{-\mathbf{X}\beta}}. \tag{8}$$

The complementary log-log is

$$\log\left(-\log(1-p)\right) = \mathbf{X}\beta; \tag{9}$$

$$p = 1 - e^{-e^{\mathbf{X}\beta}}. \tag{10}$$

The corresponding multinomial extensions, with H possible outcomes and p_h the probability of the h'th outcome, are

$$p_k = \frac{e^{\mathbf{X}\beta_k}}{\sum_{h=0}^{H} e^{\mathbf{X}\beta_h}} \tag{11}$$

for the logit and

$$p_k = \frac{1 - e^{-e^{\mathbf{X}\beta_k}}}{\sum_{h=0}^{H} 1 - e^{-e^{\mathbf{X}\beta_h}}} \tag{12}$$

for the complementary log-log.[19]. Allowing one set of β_h for each outcome gives too many parameters. We will treat the no change/no measurement error category as the omitted category; its coefficients are 0, and the other coefficients are contrasts with it.[20]

[16] In principle it could also include the history of the data, such as the maximum, minimum, or average value acheived by a covariate, the rate of change in the covariate, etc. The program doesn't currently implement that.

[17] $\sum_s \Pr(s|s0)$ must $= 1$.

[18] Currently, you can not use a fixed model for transitions.

[19] Peter should check

[20] Our interface, borrowed from **msm**, does not even allow specification of terms for the diagonals of the transition and misclassification matrices.

msm uses the binary logit for its misclassification model.[21] It does not use the complementary log-log, and since it is cast in terms of instantaneous transition rates its handling of transition probabilities differs from our approach.

Our program currently implements the fixed and multinomial logit forms.

2.2.5 Derivatives

The derivative of a fixed term is trivially zero. Let's consider the multinomial logit from equation (11), generalized:

$$q_{jn} = \frac{e^{\mathbf{X}\beta_{s(j,n-1),s(j,n)}}}{\sum_{h=0}^{H} e^{\mathbf{X}\beta_{s(j,n-1),h}}}, \tag{13}$$

with q_{jn} the probability of the transition made on path j at step n. This is a transition from state $s(j, n-1)$, the state at step $n-1$ of path j, to state $s(j, n)$. For simplicity, $\mathbf{X}$ is not subscripted, but it may vary with the time n and the path j (the path matters if there are path-dependent covariates).[22] $\beta_{a,b}$ gives the *vector* of coefficients that are relevant for transitions from state a to b.

This discussion will be cast in terms of the transition probabilities, but the mathematics applies equally to observation probabilities with a multinomial logistic model, substituting r for q and interpreting k as indexing observations.

Recalling that equation (6) requires the derivatives of logarithms, we can proceed to obtain them:

$$\begin{aligned}
\frac{\partial \ln(q_{jk})}{\partial \beta_{a,b}} &= \frac{\partial}{\partial \beta_{a,b}} \left[\mathbf{X}\beta_{s(j,n-1),s(j,n)} - \ln\left(\sum_{h=0}^{H} e^{\mathbf{X}\beta_{s(j,n-1),h}} \right) \right] && (14)\\
&= \frac{\partial}{\partial \beta_{a,b}} \left[\mathbf{X}\beta_{s(j,n-1),s(j,n)} \right] - \frac{\sum_h \frac{\partial}{\partial \beta_{a,b}} e^{\mathbf{X}\beta_{s(j,n-1),h}}}{\sum_h e^{\mathbf{X}\beta_{s(j,n-1),h}}} && (15)\\
&= \begin{cases} 0 - 0 & \text{if } a \neq s(j,n-1) \\ 0 - \frac{\mathbf{X} e^{\mathbf{X}\beta_{a,b}}}{\sum_h e^{\mathbf{X}\beta_{s(j,n-1),h}}} & \text{if } a = s(j,n-1) \text{ and } b \neq s(j,n) \\ \mathbf{X} - \frac{\mathbf{X} e^{\mathbf{X}\beta_{a,b}}}{\sum_h e^{\mathbf{X}\beta_{s(j,n-1),h}}} & \text{if } a = s(j,n-1) \text{ and } b = s(j,n) \end{cases} && (16)\\
&= \begin{cases} 0 & \text{if } a \neq s(j,n-1) \\ -\mathbf{X} q_{jn}(b) & \text{if } a = s(j,n-1) \text{ and } b \neq s(j,n) \\ \mathbf{X}(1 - q_{jn}) & \text{if } a = s(j,n-1) \text{ and } b = s(j,n) \end{cases} && (17)
\end{aligned}$$

The last step uses $q_{jn}(b)$ to refer to the multinomial probability of transitioning into state b from time $n-1$ on path j.[23]

The sum of these terms times p_j gives $\partial p_j / \partial \beta_{ab}$, as specified in equation (6).

[21] That is, each misclassification is treated separately from all the others. In principle this can produce probabilities summing to greater than one across all misclassifications, with an implied negative probability for observation without error.

[22] In the current implementation, observed covariates from $n-1$ are the ones governing the transition from $n-1$ to n.

[23]
$$q_{jn} = q_{jn}\left(s(j,n)\right).$$

2.2.6 Time of Covariate vs. Time of Jump

Given the discrete-time approximation, how do we deal with time-varying covariates (whether path-dependent or otherwise)? In the transition from t_{n-1} to t_n we could use any value of the covariates in the interval, or we could attempt to solve analytically using the complete assumed path. The latter case is not only complicated but requires dropping out of the discrete time model into a continuous time one. So we're not doing it for now.

What follows are some considerations for and against particular choices within that interval.

In t_{n-1}'s favor, note that if we use any later value for a covariate we may run into trouble if the process actually influences the value of the covariate. To take a slighly silly example, if you're dead your blood pressure is 0, so if we used blood pressure at t_n it would be an excellent predictor of death.

This argument is not definitive. First, if one of our independent variables is actually dependent, simply fiddling with the timing of observations is not likely to be sufficient.

Second, taking the discrete model literally means that the jump happens exactly at t_n (or perhaps the associated interval). This makes it natural to use covariate values from t_n as well.

t_{n-1} has other problems. Suppose one wishes to use log of time in state as a covariate. At the start (t_{n-1}) of an interval, time in state can be zero, so it is impossible to take the log. Other transformations might produce legal but extreme value for 0, and so end up giving excessive weight to the observations at jump time. No value after t_{n-1} can be 0 for time in state, and the further from t_{n-1} one goes the less extreme the value will be.

Often our observations will be relatively far apart compared to our time steps. If we use the previous observation for the covariate, we may be using a value that is quite distant compared to the value at t_n.[24]

Finally, if one thinks of the discrete model as an approximation to a continuous one, in some formulations the midpoint value for a covariate (its value at $(t_{n-1} + t_n)/2$) may be the best approximation to the average effect of the covariate in the interval.[25] The midpoint also does not suffer from the 0 time in state problem, though it will produce more extreme log values than the endpoint.

The current implementation uses observed covariates at t_n to calculate the transition rate from t_{n-1} to t_n. This scheme seems more likely to match user expectations about how the variables are used: the covariates and state come from observation at the same time. Users may preprocess the input covariates if they want lags.

For time in state and related path-dependent variables the user may specify an offset relative to t_{n-1} to get the effective value of the variable; we have used half the step size as the offset, effectively using the midpoint of the interval.

[24] If we have observations at t_1 and t_{20}, this problem of distant values remains a problem at, e.g., t_{18}, even if picking the endpoint solves it for t_{20}.

[25] The quality of the approximation depends on the exact functional forms under consideration.

2.3 Approach to Paths

An earlier implementation in SAS first generated all possible paths and then computed the likelihood of each. An advantage of this approach is that the paths can be reused for each iteration of the parameters. However, the storage required for the paths is potentially quite large, possibly exceeding physical or even virtual memory. This also requires much redundant calculation, because many paths have common parts, but each path likelihood is caclulated separately from the start of the path. Since generating paths is relatively quick, and calculating likelihoods along them is relatively slow, this didn't seem the best tradeoff.

Another approach would explicitly structure the paths as a branching, non-recombining tree. This would save on calculations, but would impose some overhead managing the tree, and even more in time coding it. It also potentially uses lots of memory, though there are ways around it.

The general approach here is a hybrid. We use a tree-like logic to generate paths, but only create one path at a time and then analyze its likelihood. Then we move onto the next path. So we don't actually have to manage a tree, but we can benefit from much of the reduced computation it involves, since we only need to recompute the path parts changed as we generate each new path. A drawback of this approach is that each parameter iteration requires regenerating all the paths.

3 Fundamental Definitions

⟨ basic types 3 ⟩ ≡
@o basic.h

```
#ifndef basic_h
#define basic_h 1

    // for Matrices
#include <valarray>

    // for my indirection operator
#include <functional>

    // for ModelData
#include <vector>

#include <iostream>
#include <ostream>
  using std::cout;
  using std::endl;

  namespace mspath
    {
    ⟨ Simple Matrices and Vectors 3.2 ⟩
    ⟨ Fundamental Types 3.1 ⟩   // must follow vectors, since use them
    ⟨ StatePoint 3.1.0.1 ⟩
    ⟨ Debug Tools 3.3 ⟩
    ⟨ boost compiler support 3.4 ⟩
    }

    // cheat for template compilation
  ⟨ basic.cc 5.18 ⟩

#endif   // basic_h
```

This code is used in section 9.

3.1 State Space

We begin with the state space. Logically the states are like **enum**'s, but considering that we are interfacing with other, non-C code, and that we may want to exploit the ordering of states, there seems little point in defining them that way.

True states are members of the set $0, \ldots, n$, where n is typically a small integer. Observed states may also take negative values, indicating, ironically, that the state is not observed. For a time in which covariates are observed, but the state is not, the "observed state" is negative (by convention, -1).

Warning: state numbering differs between the **mspath** *function in* **R** *and the C++ code here}. The* **R** *calculator object subtracts 1 from the states before calling C++.*

Several considerations went into these decisions. Compatibility with **msm** *guided the* **R** *interface. However, in this code we use state as an index into vectors and matrices or comparision with collection size, and in C++those are 0-based and have type* **unsigned int** *aka* **size_t** *for the standard C++ library.*

The use of negative values for observed state is theoretically impure; it would be cleaner to have an abstract state class with separate subclasses for observed and unobserved state. However, the additional overhead on such a low-level, frequently used variable is undesirable. One alternative would be to treat state as an **int** *throughout, but that conflicts with C++ conventions discussed in the previous paragraph. I adopt a middle road of using two different types:* **State** *for the theoretical state, and* **ObsState** *for the observed state.*

Time is conceptually continuous, and is so for the input observed path. Given that we may not have a fixed time step size, we really must treat internal time as continuous too. But if we are not careful, we may construct a time that we think matches an observed time, but doesn't. Either ensure constructed times exactly match observed times (in the sense of $\equiv$) or use appropriate types of comparisons. *Note that some code converts time to an integer step number and works with that, and some works with rational number types to avoid rounding problems.*

Finally, we have some types that may eventually become template parameters.

⟨ *Fundamental Types 3.1* ⟩ ≡

```
using std::size_t;

typedef unsigned int State;
typedef int ObsState;

typedef double Time;
typedef int Id;    // identifies cases

  // FUTURE TEMPLATE PARAMETER CANDIDATES

  /* All of the following would seem to naturally go in the header files of the corresponding classes.
     We pull them out here to avoid loops or nastinesses in the way classes depend on each other. */

  // Info needed to evaluate the model.
  // currently, just a single probability
typedef double EvaluationData;

  // Info needed by the model.
  // This holds path-dependent data.
typedef Double1D ModelData;

  // Index into observations of Data.
typedef size_t IObservation;
```

This code is used in section 3.

StatePoint *captures the state of the probability process at one particular instant. Note this uses the low-level* **Time** *concept, rather than* **TimePoint***, which is more deeply embedded in the program structure.* **StatePoint***'s ordinarily are invariant once created. Very knowledgeable clients can change the state with setState( ), but doing so is asking for trouble. For example, any data calculated with the old values will be invalid. Currently this feature is only used in simulation.*

The binary comparison operators are to aid testing, in particular sticking **Path***'s in Set's. That requires sorting the* **Node***'s. Barton and Nackman recommend defining such inequality relation as external functions (e.g, pp. 161–2, 545 of* **Scientific and Engineering C++***).*

⟨ *StatePoint 3.1.0.1* ⟩ ≡

```
class StatePoint
  {
public:
  StatePoint(const State& s, const Time& t) : myState(s), myTime(t)
    { }
  const Time& time( ) const
    { return myTime; }
  Time time( )
    { return myTime; }
  const State& state( ) const
    { return myState; }
  State state( )
    { return myState; }
  // setState is a dangerous operation; use with care.
  void setState(State s)
    {
    myState = s;
    }
  //operators
  friend bool operator ≡(const StatePoint& lhs, const StatePoint& rhs)
    { return lhs.myState ≡ rhs.myState ∧ lhs.myTime ≡ rhs.myTime; }
  friend bool operator <(const StatePoint& lhs, const StatePoint& rhs)
    { if (lhs.myState < rhs.myState)
        return true;
      else if (lhs.myState > rhs.myState)
        return false;return lhs.myTime < rhs.myTime; }
  friend std::ostream & operator <<(std::ostream & ostr, const StatePoint& sp)
    { ostr << "StatePoint(" << sp.state( ) << ", " << sp.time( ) << ")";return ostr; }
protected:
  State myState;
  Time myTime;
  };
```

This code is used in section 3.

3.2 Arrays

These classes are designed to hold the basic vector and matrix data, particularly as transferred in from C.

I spent some time debating whether to use **vector** *or* **valarray**. *The former is a full member of the templatized containers, with the wealth of facilities that brings. But the latter is designed specifically for fast numerical work and work with numeric libraries. It appears that I can &***valarray** [*i*] *as an iterator, so I can still use the algorithms facility if I want.*

I develop these classes as subclasses so I can add functionality, for example begin( ) and end( ) if I want greater container compatibility. And clearly I need something more for matrices, as the standard library provides no such class (though it indicates **valarray** *may be used to build one.*

Note all indexing is 0-based, and that the vectors index with [*i*] *while the matrices use* (*i*, *j*).

⟨ *Simple Matrices and Vectors 3.2* ⟩ ≡
  // **#include** ⟨**valarray**⟩ required in global namespace

  ⟨ *2D Subset of 1D 3.2.1* ⟩
  ⟨ *Simple Vector 3.2.0.1* ⟩
  ⟨ *Simple Matrix 3.2.0.2* ⟩

This code is used in section 3.

A vector is straightforward. I picked a name that doesn't clash with **std:: vector** .

⟨ *Simple Vector 3.2.0.1* ⟩ ≡

```
template ⟨typename T⟩
class Array1D : public std ::valarray⟨ T ⟩
{
public:   // next directive keeps my def of [ ] from hiding all the others
  using std::valarray < T > ::operator [];
  Array1D (const T *d, size_t n) : std::valarray⟨ T ⟩(d, n)
    {}
  Array1D ( )
    {}
  Array1D (size_t n) : std::valarray⟨ T ⟩(n)
    {}
  Array1D (const T & v, size_t n) : std::valarray⟨ T ⟩(v, n)
    {}
  Array1D (const std::valarray < T > &theVals) : std::valarray⟨ T ⟩(theVals)
    {}
  Array1D& operator =(const std::valarray < T > &rhs)
    {
    std::valarray < T > ::operator =(rhs);
    return *this;
    }
    // indirection
  Indirect1Dto2D⟨ T ⟩ operator [](const Array2D⟨size_t⟩& theIndirect) const
    {
    return Indirect1Dto2D⟨ T ⟩(*this, theIndirect);
    }
  template ⟨typename U⟩
  const Array1D& setRaw(U *p, size_t n);
}
;
  template ⟨typename T⟩
  template ⟨typename U⟩
  const Array1D⟨ T ⟩& Array1D⟨ T ⟩::setRaw(U *p, size_t n)
    {
    this→resize(n);
    for (size_t i = 0; i < n; i++)
      (*this)[i] = static_cast⟨ T ⟩(p[i]);
    return *this;
    }
  typedef Array1D⟨int⟩ Int1D;
  typedef Array1D⟨bool⟩ Bool1D;
  typedef Array1D⟨double⟩ Double1D;
  typedef Array1D⟨size_t⟩ TIndirect1D;   // use for subsetting any of the above
```

template ⟨**typename** **T** ⟩
std::*ostream* & **operator** <<(**std**::*ostream* & *s*, **const** **Array1D**⟨ **T** ⟩& *a*);

This code is used in section 3.2.

The matrices that are passed in have all the first row, then all the second, etc.

I tried to define the types TRow and **TCol** *as* **std**::*slice_array* < **T** >*, but that is not intended for user consumption. It's not clear to me how optimized* **valarray***'s constructed from slice's are, but at the moment I'm using them.*

I get fatal `FWEAVE` *errors without the next two items.*

"mspath.C" *3.2.0.2* ≡
@f *vector* `$_EXPR_`
@f *T* `$_EXPR_`

⟨ *Simple Matrix 3.2.0.2* ⟩ ≡

```
template ⟨typename T ⟩
class Array2D
  {
public:   // types
  typedef std::valarray < T > TRow; typedef std::valarray < T > TCol ;
    // constructors
  Array2D ( )
    { }
  Array2D (const T *p, size_t rows, size_t cols) : myCols(cols), myData(p, cols * rows)
    { }
  Array2D (size_t rows, size_t cols) : myCols(cols), myData(cols * rows)
    { }
  Array2D (const T & initial, size_t rows, size_t cols) : myCols(cols), myData(initial, cols*rows)
    { }   // from indirected 1D!
  Array2D (const Indirect1Dto2D⟨ T ⟩& theI) : myCols(theI.indirect( ).ncols( )),
        myData(theI.base( )[theI.indirect( ).rawData( )])
    { }
    // setup
  template ⟨typename U⟩
  const Array2D& setRaw(U * vector , size_t rows, size_t cols);
    // accessors
    // Must use ( ) not [ ] since standard says latter is single argument
  T & operator ( )(size_t row, size_t col)
    { return myData[row * myCols + col]; }
  T operator ( )(size_t row, size_t col) const
    { return myData[row * myCols + col]; }
  size_t nrows( ) const
    { if (myCols > 0)
        return myData.size( ) / myCols;
      else
        return 0; }
  size_t ncols( ) const
    { return myCols; }
  TRow row(size_t r)
```

```
    { return myData[std::slice(ncols( ) * r, ncols( ), 1)]; }
  TCol col(size_t c)
    { return myData[std::slice(c, nrows( ), ncols( ))]; }
  const TRow row(size_t r) const
    { return myData[std::slice(ncols( ) * r, ncols( ), 1)]; }
  const TCol col(size_t c) const
    { return myData[std::slice(c, nrows( ), ncols( ))]; }
    // container-like protocol
  size_t size( ) const
    { return myData.size( ); }
  void resize(size_t r, size_t c)
    { myData.resize(r * c); myCols = c; }
    // operators (defined as needed)
  Array2D& operator -=(const T & x)
    {
    myData -= x;
    return *this;
    }
  Array2D operator -(const T & x) const
    {
    Array2D a(*this);
    a -= x;
    return a;
    }
#if 0
    // couldn't get this to work
    // following really only needed for T = size_t
  template <typename U>
  friend class Array2D<U>;
#endif
    // this is only for use by the c'tor with Indirect1Dto2D
    // and it should only be called for T ≡ size_t.
  const std::valarray < T > &rawData( ) const
    {
    return myData;
    }
protected:
  size_t myCols;
  std::valarray < T > myData; } ;
  typedef Array2D<int> Int2D;
  typedef Array2D<bool> Bool2D;
  typedef Array2D<double> Double2D;
  typedef Array2D<size_t> TIndirect2D;    // for subsetting
  template <typename T >
  template <typename U>
```

```
const Array2D⟨ T ⟩& Array2D⟨ T ⟩::setRaw(U * vector , size_t rows, size_t cols)
  {
  myCols = cols;
  size_t nElements;
  nElements = cols * rows;
  myData.resize(nElements);
  for (size_t i = 0; i < nElements; i++)
    {
    myData[i] = static_cast⟨ T ⟩( vector [i]);
    }
  return *this;
  }
;
template ⟨typename T ⟩
std::ostream & operator <<(std::ostream & s, const Array2D⟨ T ⟩& a);
```

This code is used in section 3.2.

3.2.1 Weird Indirection

The next snippet defines a class that is purely for internal use by the **Array1D** *and* **Array2D** *classes to allow convenient subsetting.*

Indirect1Dto2D *supports taking a 2D subset (or expansion) from a bunch of 1D values.*

```
⟨ 2D Subset of 1D 3.2.1 ⟩ ≡
    // forward declare a return type
  template ⟨typename T ⟩
  class Array2D;

  template ⟨typename T ⟩
  class Array1D;

    // no client should use the following class
  template ⟨typename T ⟩
  class Indirect1Dto2D
    {
  public:
    Indirect1Dto2D(const Array1D⟨ T ⟩& theBase, const Array2D⟨size_t⟩& theIndirect) :
        myBase(theBase), myIndirect(theIndirect)
      { }

    ;

    const Array1D⟨ T ⟩& base( ) const
      {
      return myBase;
      }

    const Array2D⟨size_t⟩& indirect( ) const
      {
      return myIndirect;
      }

  protected:
    const Array1D⟨ T ⟩& myBase;
    const Array2D⟨size_t⟩& myIndirect;
    };
```

This code is used in section 3.2.

3.3 Debugging

Here are some little aids to debugging, active if DEBUG is defined.

⟨ *Debug Tools 3.3* ⟩ ≡

```
#ifdef DEBUG
#define TRACE(stuff) std::cout << stuff
#else
#define TRACE(stuff)
#endif
```

This code is used in section 3.

3.4 Boost Compiler Support

The code was developed with **Boost** *1.31 and 1.32.*

The ptr_container library is in the **Boost** *library from release 1.33 on; the source is included in this package in case you are using an earlier version. Both the main code and the test framework use other parts of* **Boost***; you will need some version of the library, including a compiled version of its unit test library.*

The interface to the **Boost** *test library stabilized with release 1.34, I believe. The test library is not needed for building the* **R** *package, so even earier versions of* **Boost** *may be suitable in that case. You will need to edit the header here to avoid failure in that case.*

Adapting to later versions of **Boost** *generally requires extending the definitions immediately below and in* `mspath/src/test/main_test.cc`*.*

Useage: **#ifdef** *BOOST_1_32* .

Implementation note: originally I tried to define all of the following as a macro (that is, one macro GET_BOOST_VERSION would produce the desired code when invoked. This was to reduce inclusion of `boost/version.hpp` *to those situations in which it was necessary. Unfortunately, it didn't work, I think because the body was never evaluated.*[26]

I would also prefer a warning to an error, but there doesn't seem to be a way to do it within the standard.

I tried to get the version number in the output message, but I couldn't get it to expand.

⟨ *boost compiler support 3.4* ⟩ ≡

```
#include <boost/version.hpp>
#if BOOST_VERSION / 100 ≡ 1031
#define BOOST_1_31 1
#elif BOOST_VERSION / 100 ≡ 1032
#define BOOST_1_32 1
#elif BOOST_VERSION / 100 ≡ 1033
#define BOOST_1_33 1
#elif BOOST_VERSION / 100 ≡ 1034
#define BOOST_1_34 1
#define BOOST_AUTO_UNIT_TEST(x) BOOST_AUTO_TEST_CASE(x)
#elif BOOST_VERSION / 100 > 1034
#define BOOST_1_35PLUS
#define BOOST_AUTO_UNIT_TEST(x) BOOST_AUTO_TEST_CASE(x)
#else
#error mspath requires Boost 1.31 or later
#endif
```

This code is used in section 3.

[26] maybe because it was all one line. Maybe because nesting endif doesn't work.

4 Interfaces

We begin our long march through the headers, with the higher level ones first.

4.1 Manager

The **Manager** *has overall responsibility for performing the calculations and returning the results. It knits together everything else.*

Manager *assumes that the probabilities of initial true states apply to the first step in the generated path, which in turn is from the first observation for a case. Thus, if you have some assumed starting state and time that is not in your observed data (e.g., birth), you should create a corresponding data point.*

The **Manager** *own most of the top-level objects in the system; they in turn own other objects. Some objects refer to these top-level objects, but they don't own them.* **Manager** *also creates many of the objects, although I'm moving toward separating object creation out.*

Manager *includes a mix of thread-specific information, such as the* **Environment**, *and material that can be shared across threads (e.g.,* **Model**). *It would be natural to have* **Manager** *control multiple threads of execution.*

⟨ *Manager.h 4.1* ⟩ ≡

```
@o Manager.h
#ifndef Manager_h
#define Manager_h 1

#include <memory>

#include "basic.h"
#include "Data.h"
#include "Model.h"
#include "Path.h"
#include "TimeSteps.h"
#include "Node.h"
#include "PathGenerator.h"
#include "Environment.h"
#include "SimpleRecorder.h"
#include "EvaluatorRecorder.h"
#include "Evaluator.h"
#include "FixedTimeStepsGenerator.h"

  namespace mspath {
        class Manager
    {
      public:
      ⟨ Manager Typedefs 4.1.0.1 ⟩
      ⟨ Manager Constructors 4.1.0.2 ⟩
      ⟨ Manager Actions 4.1.0.3 ⟩
      ⟨ Manager Simulation 4.1.0.7 ⟩
    protected:
      ⟨ Manager Helpers 4.1.0.6 ⟩
      ⟨ Manager Protected Accessors 4.1.0.5 ⟩
      ⟨ Manager Data 4.1.0.4 ⟩
    } ;    // end class Manager
    }    // end namespace mspath
  #endif    // Manager_h
```

This code is used in section 9.

⟨ *Manager Typedefs 4.1.0.1* ⟩ ≡

```
typedef Data TData;
typedef TimeSteps TTimeSteps;
typedef Model TModel;
typedef Node TNode;
typedef Path TPath;
typedef StateTimeClassifier TStateTimeClassifier;
typedef SuccessorGenerator TSuccessorGenerator;
typedef Environment TEnvironment;
```

This code is used in section 4.1.

This constructor may not do down the line, with repeated calls with same data and varying model parameters.

⟨ *Manager Constructors 4.1.0.2* ⟩ ≡

```
Manager(TData *pData, TModel *pModel, int stepNumerator = 1, int stepDenominator = 1,
        bool isExactTimeAbsorb = false) : mypData(pData), myEnvironment(pData, new
        Path(pModel→nPathDependentVariables( ))), mypModel(pModel),
        myTimeStepsGenerator(stepNumerator, stepDenominator), mySuccessorGenerator(pModel),
        mypStateTimeClassifier(0), myIsExactTimeAbsorb(isExactTimeAbsorb), mypRecorder(0),
        mypEvaluator(0)
  { if (isExactTimeAbsorb)
      mypStateTimeClassifier.reset(new PickyStateTimeClassifier(pModel));
    else
      mypStateTimeClassifier.reset(new StateTimeClassifier(pModel)); }
```

This code is used in section 4.1.

Kick off the overall computation or adjust its parameters.

go( ) returns results:

1. *Log likehood (0 if counts only requested).*
2. *Number of cases.*
3. *Number of good paths.*
4. *Number of unique good nodes.*
5. *Number of bad nodes.*
6. *Number of good nodes, including repeats and nodes that were on bad paths.*

Note that although all the counts are (long) integers, we return 6 doubles.

See the discussion of **SimpleRecorder** *for more on these concepts (§4.20.1, page 141).*

The default do_what of 0 computes counts; any other value (conventionally, 1) computes likelihood as well.

setSubset( ) and setAll( ) turn subsetting on and off, respectively. It is off by default. Computations made after these calls will only consider the active subset (which may be the whole set) of the data.

For setSubset( ) the argument is a list of **Id***'s, which must be in the same order as the* **Id***'s in the* **Manager***'s* **Environment***'s data( ). The* **Manager** *will take ownership of the list's memory; clients should* not *attempt to free that memory or use pSubset after the call.*

⟨ *Manager Actions 4.1.0.3* ⟩ ≡

```
    // kick off the computations
  void go(double *results, int do_what = 0);

    // change the active subset of cases

    // argument is a list of ID's, in same order as in Data
    // future computations will only use this subset.

  void setSubset(SubsetDataIterator::IDList & pSubset)
    {
    environment( ).setSubset(pSubset);
    }

    // use all possible cases

  void setAll(void)
    {
    environment( ).setAll( );
    }

    // use only by knowledgable clients
    // I take ownership of pointer
    // New model is assumed to differ only in parameters, not structure.

  void setModel(std::auto_ptr⟨Model⟩pm)
    {   /* Currently environment.clear( ) takes care of releasing memory, and should be somewhat
           faster. In the future, there might be things in the environment we want to keep. Then it would
           be good to activate the following code. */
  #if 0
    if (mypModel.get( ))
      mypModel→release(&(environment( )));
  #endif

    mypModel = pm;    // Since old model is now gone, must point the next items
      // at the new one.
    stateTimeClassifier( ).setModel(&(model( )));
    successorGenerator( ).setModel(&(model( )));
    environment( ).clear( );
    }
```

This code is used in section 4.1.

I may want to move to pointers for everything; always use the accessor function to avoid depending on current data style.

Be careful about reordering the definitions, as some of the later ones require earlier ones to initialize, and initialization is in order of the list below.

⟨ *Manager Data 4.1.0.4* ⟩ ≡

```
std::auto_ptr<Data> mypData;    // just so I clean up
TEnvironment myEnvironment;
std::auto_ptr<Model> mypModel;
FixedTimeStepsGenerator myTimeStepsGenerator;
TSuccessorGenerator mySuccessorGenerator;
std::auto_ptr<TStateTimeClassifier> mypStateTimeClassifier;
bool myIsExactTimeAbsorb;
std::auto_ptr<SimpleRecorder> mypRecorder;
std::auto_ptr<Evaluator> mypEvaluator;    // optional
std::auto_ptr<AbstractPathGenerator> mypPathGenerator;
```

This code is used in section 4.1.

Hide whether I'm using pointer from myself. Also provide a way to switch the model.

⟨ *Manager Protected Accessors 4.1.0.5* ⟩ ≡

```
TEnvironment& environment( )
  { return myEnvironment; }

Model& model( )
  { return *mypModel; }

FixedTimeStepsGenerator& timeStepsGenerator( )
  { return myTimeStepsGenerator; }

AbstractPathGenerator& pathGenerator( )
  { return *mypPathGenerator; }

SimpleRecorder& recorder( )
  { return *mypRecorder; }    // may be a subclass

Evaluator& evaluator( )
  { return *mypEvaluator; }

TStateTimeClassifier& stateTimeClassifier( )
  { return *mypStateTimeClassifier; }

TSuccessorGenerator& successorGenerator( )
  { return mySuccessorGenerator; }

bool hasLikelihood( ) const
  {
  return mypEvaluator.get( ) ≠ 0;
  }
```

This code is used in section 4.1.

These functions help implement go( ).

⟨ *Manager Helpers 4.1.0.6* ⟩ ≡
```
void setupCount( );    // setup for getting simple counts
void setupLikelihood( );
void mainOperation( );
void getResults(double *results);
```

This code is used in section 4.1.

This does a simulation and returns the results to the caller, which is responsible for destruction of the returned object. Since we use **R***'s random number generator, stream management, such as setting the seed, should be done in* **R***.*

⟨ *Manager Simulation 4.1.0.7* ⟩ ≡
```
std::auto_ptr < RandomPathGenerator::Results > simulate( );
```

This code is used in section 4.1.

4.2 PathGenerator

These classes know how to generate paths through the underlying true states.

They work with a set of **TimeSteps** *that have already been created in the* **Environment** *by a* **TimeStepsGenerator***. These class does not set up the* **Environment** *or iterate it to the next case; they handle the paths for one particular case.*

⟨ *PathGenerator.h 4.2* ⟩ ≡
```
@o PathGenerator.h
#ifndef PathGenerator_h
#define PathGenerator_h 1

#include <stdexcept>
#include <vector>    // needed by RandomPathGenerator

#include "basic.h"
#include "Environment.h"
#include "MSPathError.h"
#include "NodeFactory.h"
#include "Recorder.h"
#include "StateTimeClassifier.h"
#include "SuccessorGenerator.h"

namespace mspath {⟨ AbstractPathGenerator interface 4.2.1 ⟩
⟨ PathGenerator interface 4.2.2 ⟩
⟨ RandomPathGenerator interface 4.2.3 ⟩


}    // end namespace mspath
#endif    // PathGenerator_h
```

This code is used in section 9.

4.2.1 AbstractPathGenerator

nextTimePoint( ) encapsulates finding the next time point. This routine assumes we wish to extend the path by one. Since it is heavily called; making it non-virtual and inlining it noticeably improve performance. See benchmarks at the end for specifics.

See §5, page 171 for the meanings of the different standard actions.

⟨ *AbstractPathGenerator interface 4.2.1* ⟩ ≡

```
class AbstractPathGenerator
  {
public:
  AbstractPathGenerator(Environment *pEnv) : mypEnvironment(pEnv)
    {}
  virtual ~AbstractPathGenerator( )
    {}
    // standard actions
  virtual void startSession( );

  virtual void startCase( );
  virtual void startTree(const StatePoint& sp, double probability) throw
      (std::runtime_error, std::logic_error);
  virtual void finishCase( );

  virtual void finishSession( );


protected:    //accessors
  Environment& environment( )
    {return *mypEnvironment; }
  const Environment& environment( ) const
    {return *mypEnvironment; }
    // helpers
  const inline TimePoint& nextTimePoint( ) const
    {
    Path::size_typen = environment( ).path( ).size( );    // 0-based index
    return environment( ).timeSteps( )[n];    // so this already +1
    }
    // data – reference, not ownership
  Environment *mypEnvironment; } ;
```

This code is used in section 4.2.

4.2.2 PathGenerator

This is the standard PathGenerator that makes all possible paths, and calls a **Recorder** *as it proceeds to capture information about the paths.*

⟨ *PathGenerator interface 4.2.2* ⟩ ≡

```
class PathGenerator : public AbstractPathGenerator
  {
public:
  typedef AbstractPathGenerator Super;

  PathGenerator(Environment *pEnv, Recorder *pRec, StateTimeClassifier *pSTC,
      SuccessorGenerator *pSG) : AbstractPathGenerator(pEnv), mypRecorder(pRec),
      mypStateTimeClassifier(pSTC), mypSuccessorGenerator(pSG)
    { }

  virtual ~PathGenerator( )
    { }

    // standard actions
  virtual void startSession( );

  virtual void startCase( );
  virtual void startTree(const StatePoint& sp, double probability) throw
      (std::runtime_error, std::logic_error);
  virtual void finishCase( );

  virtual void finishSession( );

protected:    // accessors
  Recorder& recorder( )
    { return *mypRecorder; }

  StateTimeClassifier& stateTimeClassifier( )
    { return *mypStateTimeClassifier; }

  SuccessorGenerator& successorGenerator( )
    { return *mypSuccessorGenerator; }

    // helpers
  void nextBranch( );

    // data – none of these are ownership
  Recorder *mypRecorder;
  StateTimeClassifier *mypStateTimeClassifier;
  SuccessorGenerator *mypSuccessorGenerator;

  } ;   // end class PathGenerator
```

This code is used in section 4.2.

4.2.3 RandomPathGenerator

This creates a single random true path with observations, using the model and data. The probability of generating a particular path and observations is the probability given by the model and data.

The result is a simulated list of observed states, times, and indices in the covariates. Each result is a **RandomPathGenerator**::**SimResult** *and they are collected in a* **RandomPathGenerator**::**Results**. *Note that the indices implicitly give the subject identifier.*

For each time in which there was an observed state there will be a simulated observed state and the original covariates. For each time in which covariates were observed, but not the state, the simulated observation will be the same as the original, i.e., it will indicate state is unobserved.

Generally, the simulated times will match the true observation times. However, if the path enters an absorbing state it must be cut off because data with repeated observations in an absorbing state are considered erroneous by the rest of the system. So the number of simulated observations may be less than the number in the original data.

Further, if the model specifies exact observation time for entry into absorbing states, the simulation must have an observation exactly when it generates an absorbing state. This means the final simulated observation time may not match any of the times in the original data. In this case, covariate values will come from the last available observation at that point. Since times may not match any observed times, **SimResult** *can not use an index to get the time and must preserve it directly.*

Note that the times generated will be exactly on discrete time points used by the simulation.

Each step in the simulation describes a move from discrete point $i-1$ to i. To be consistent with the original model, covariates governing the step are from i, while path-dependent variables are from $i-1$ plus an optional offset.

⟨ *RandomPathGenerator interface 4.2.3* ⟩ ≡

```
class RandomPathGenerator : public AbstractPathGenerator
  {
public:
  typedef AbstractPathGenerator Super;
  ⟨ RandomPathGenerator types 4.2.3.1 ⟩
  RandomPathGenerator(Environment *pEnv, StateTimeClassifier *pSTC,
      Model *pModel, bool isExact,     // entry to terminal state observed exactly?
  Results *pResults) : AbstractPathGenerator(pEnv), myIsExact(isExact),
        mypStateTimeClassifier(pSTC), mypModel(pModel), mypResults(pResults)
    { }

  virtual ~RandomPathGenerator( )
    { }

    // standard actions
  virtual void startSession( );

  virtual void startCase( );
  virtual void startTree(const StatePoint& sp, double probability) throw
      (std::runtime_error, std::logic_error);
```

```
    virtual void finishCase( );

    virtual void finishSession( );


      // accessors (when done)

    Results& results( )
      { return *mypResults; }

  protected:    // accessors
    StateTimeClassifier& stateTimeClassifier( )
      { return *mypStateTimeClassifier; }

    Model& model( )
      { return *mypModel; }

    bool isExact( ) const
      {
      return myIsExact;
      }

    ⟨ RandomPathGenerator internal helpers 4.2.3.2 ⟩

      // data
    bool myIsExact;

      // data – none of these are ownership
    StateTimeClassifier *mypStateTimeClassifier;
    Model *mypModel;
    Results *mypResults;

    } ;
```

This code is used in section 4.2.

These are for the results of the simulation. They hold simulated observations: the state is the simulated observed *state. Each case will typically have several* **SimResult***'s, corresponding to several simulated observations.*

⟨ *RandomPathGenerator types 4.2.3.1* ⟩ ≡

```
class SimResult
  {
public:
  SimResult (ObsState s, Time t, IObservation i) : myState(s), myTime(t), myObsIndex(i)
    { }
  ObsState state( ) const
    {
    return myState;
    }
  Time time( ) const
    {
    return myTime;
    }
  IObservation obsIndex( ) const
    {
    return myObsIndex;
    }
protected:
  ObsState myState;
  Time myTime;
  IObservation myObsIndex;
  };
typedef std:: vector ⟨SimResult⟩ Results;
```

This code is used in section 4.2.3.

See the code section for what these things do.

⟨ *RandomPathGenerator internal helpers 4.2.3.2* ⟩ ≡

```
void nextStep( );
void addLastPoint( );
void recordObservation( );
```

This code is used in section 4.2.3.

4.3 SuccessorGenerator

This class knows what legal successor states are given the current state, given the **Model**. *Currently, all I really need is the current state, but to allow some more generality I include the current time and next time in the interface. I have no dependence on the* **Environment**, *though there might be in the future. The origin point is assumed to be legal.*

PathGenerator *uses this class.* **Manager** *constructs and manipulates it.*

For now, we could almost dispense with this class and use **Model** *directly. But to allow for possible future complexity, as suggested in the previous paragraph, as well as for symmetry with* **StateTimeClassifier**, *we use a separate class.*

⟨ *SuccessorGenerator.h 4.3* ⟩ ≡

```
@o SuccessorGenerator.h
#ifndef SuccessorGenerator_h
#define SuccessorGenerator_h 1
#include "basic.h"

#include <vector>

#include "Model.h"
#include "Node.h"

  namespace mspath
    {
    class SuccessorGenerator
      {

    public:
      SuccessorGenerator(Model *pModel) : myPModel(pModel)
        {}

        // fill in next with the State's that are allowable at time
        // newTime starting at baseNode

      void nextStates(std:: vector ⟨State⟩ & next, const Node& baseNode, Time newTime);

        // the next few calls are an optimized alternative to nextStates

      State firstState( ) const
        {
        return 0U;
        }

      State lastState( ) const
        {
        return myPModel→nStates( ) − 1U;
        }

      bool isPossibleTransition(State start, State end) const
        {
        return myPModel→isPossibleTransition(start, end);
        }

        // only for knowledgeable clients, i.e., Manager::setModel( )
        // no use of friend to keep header dependencies simple
```

```
    // New model should be same as old in structure; may differ in params.
  void setModel(Model *pModel)
    {
    myPModel = pModel;
    }
protected:
  Model *model( )
    { return (myPModel); }
  Model *myPModel;    // reference, not ownership
  };    // end class SuccessorGenerator
}    // end namespace mspath
#endif    // SuccessorGenerator_h
```

This code is used in section 9.

4.4 StateTimeClassifiers

These classes know which states and times are legal, but know nothing about transitions. They classify true (unobserved) states and times by whether or not they are possible given the data and the measurement model. They also decide if a state and time are terminal, i.e., represents the end of a path. Absorbing states are always terminal. Any combination of these two (terminal and legal) attibutes is possible.

PathGenerator *uses these classes.*

Sometimes the time of observed entry into a state is informative. The **PickyStateTimeClassifier**, *described further below, handles that situation.*

The main method, isOK( ) returns `false` *if the* **StateTimeClassifier** *knows the associated state and time are impossible. The only guaranteed* `false` *responses are for times that match actual observations and states that are inconsistent with those observations and the measurement model. So a* `true` *response does not guarantee that a* **StatePoint** *will ultimately be a part of any successful path.*

In a pure world, queries would only hand in **StatePoint***'s as arguments. However, that would necessitate isOK looking up the associated* **TimePoint**. *As a shortcut, we pass in an iterator for the* **TimePoint**. *This has drawbacks: it complicates the interface, introduces dependencies on other files (though this is more apparent than real, since* **Environment** *already implies dependence of* **TimeSteps***), and allows non-sensical or inconsistent information to go in.*

This class provides two kinds of information. First, it is responsible for judging whether a node is terminal and for eliminating paths that are inconsistent with the measurement model and the data. Second, it may indicate other nodes are not OK as an optimization. For now, it doesn't.

This class is thread-specific, used by a thread-specific **PathGenerator**.

Warning: only a few methods are virtual because only a few need to be. Good design would make them all virtual, but there would be a performance penalty. So add virtual as necessary in further development.

⟨ *StateTimeClassifier.h 4.4* ⟩ ≡

```
@o StateTimeClassifier.h
#ifndef StateTimeClassifier_h
#define StateTimeClassifier_h 1
#include "basic.h"

#include "Environment.h"
#include "Model.h"
#include "TimeSteps.h"
#include "MSPathError.h"

  namespace mspath {⟨ StateTimeClassifier Interface 4.4.1 ⟩
  ⟨ PickyStateTimeClassifier Interface 4.4.2 ⟩}    // end namespace mspath
#endif    // StateTimeClassifier_h
```

This code is used in section 9.

4.4.1 StateTimeClassifier

⟨ *StateTimeClassifier Interface 4.4.1* ⟩ ≡

```
class StateTimeClassifier
  { public:   // constructors
  StateTimeClassifier(Model *pModel) throw (InconsistentModel);

  virtual ~StateTimeClassifier( )
    { }

    // state changes
  virtual void startTree(Environment *pEnv);    // call after base node is on path

  void finishTree(Environment *pEnv)
    { }

    // Queries

    // true if sp is possible for this case
    // Depending on the implementation, may be somewhat sloppy.
    // That is, may return true even if the state isn't really possible.
    // But false always indicates impossible.
    // Pure, simple interface is
    // bool isOK(const StatePoint& sp);
    // but we use this optimization, which assumes that
    // sp.time( ) ≡ time of TimePoint.
  bool isOK(const StatePoint& sp, const TimePoint& tp, Environment *pEnv) const;

    // true if sp is at the end of a path
    // E.g., out of time, out of bounds, dead.
  bool isTerminal(const StatePoint& sp, Environment *pEnv) const;

    // accessors—mostly for internal use

  const Model& model( ) const
    { return *mypModel; }

  Model& model( )
    { return *mypModel; }

    // only for knowledgeable clients, i.e., Manager::setModel( )
    // no use of friend to keep header dependencies simple
    // This should only change model parameters, not structure.

  void setModel(Model *pModel)
    {
    mypModel = pModel;
    }

protected:
  Model *mypModel;    // reference, not ownership
  Time myTerminalTime;

    // myFirstTime[s] is first time I may be in State s.
    // Only used for absorbing states.
  std:: vector ⟨Time⟩myFirstTime;

  } ;    // end class StateTimeClassifier
```

This code is used in section 4.4.

4.4.2 PickyStateTimeClassifier

Sometimes the observed time of entry into a state is informative, telling us not only that the person was in that state at the observation time, but that the transition occurred exactly at the observation time. Death is a common example.[27]

It only makes sense to talk about exact entry time into states we can measure exactly. It is an error for one of those states to be observable as a different state, or for a different state to be observable as the "exact state." The time can't be exact if we don't know what state we're in!

This class assumes all absorbing states have transition times measured exactly, so only paths with the exact observed transition time will be permitted.[28]

⟨ *PickyStateTimeClassifier Interface 4.4.2* ⟩ ≡

```
class PickyStateTimeClassifier : public StateTimeClassifier
  { public:    // constructors
  PickyStateTimeClassifier(Model *pModel) throw (InconsistentModel);

  virtual ~PickyStateTimeClassifier( )
    { }

    // state changes
  virtual void startTree(Environment *pEnv);    // call after base node is on path
  } ;
```

This code is used in section 4.4.

[27] **msm** distinguishes the case in which we know the immediately preceding state ("observation type 2") from the case in which we don't ("type 3"). We only consider the latter.

[28] Because of rounding the time may not be completely exact. The *TimeStep* which matches the observation of the transition is declared to be the time of the transition.

4.5 Model

A **Model** *is the highest-level analytic object, knowing all about the structure of the model (allowed transitions, measurement errors, constraints, path-dependent variables) and the parameter values. It knows what values to compute for each* **Node**. *This* **Model** *computes likelihoods*—not log-likelihoods—*along paths.*[29]

Much of this information is wrapped in lower-level objects, such as **Specification***'s. The* **Model** *and all the objects it contains can be used in several threads at once. The* **Environment** *argument to some methods holds thread-specific information that the* **Model** *both reads and writes.*

The **Model** *is not responsible for constructing itself from raw inputs, for generating paths, for assuring that prior* **Node***'s in the path are evaluated prior to the current one, or for managing elaborate evaluation schemes in which several sets of computations are performed for each* **Node**. *It is responsible for computing likelihoods and related values at a* **Node**, *and for calculating any necessary path-dependent data.*

Note that **msm** *has the following features that the current* **Model** *lacks:*[30]

- *Multiple possible true initial states.*
- *Very general handling of whether transition times are known exactly and whether the prior state is known in those cases. However a* **Model** *using a* **PickyStateTimeClassifier** *does handle the case where transitions to absorbing states are observed exactly, with the prior state unknown.*

Handling more than one initial state poses the following challenges:

1. *What do we do in the case where one of the initial states is inconsistent with a case's data? Most likely, we need to scale up the remaining initial probabilities.*
2. *We need to code the additional logic, including the answer to the preceding question.*
3. *We need to test it.*

[29] This is to save the cost of computing a log at each **Node** and then exponentiating the result at the end of the path.
[30] Earliear versions of **msm** also allowed specification of a lag for covariates, but new versions lack that feature.

Mostly because of the testing time, I'm deferring implementation of the case with multiple initial states.

In the future, it may be appropriate to separate out the parts of the class not involving misclassification, since the original **msm** *C code takes two different routes depending on whether there is a misclassification model.*

In principle it is desirable to insulate this class from knowledge of the particular data structures used in the rest of this system, including the specific setup of **Path***'s and* **Node***'s, as well as the* **Data***. It would be possible to keep all interfaces to low-level objects by disaggregating information (e.g., don't give it the* **Data***, give it the particular vector of covariates that matter).*

In practice, such isolation would be tedious, would involve guessing exactly which aspects of the **Environment** *were relevant (e.g., do the specific times of the observations matter? do the previous covariates matter as well as the current ones?), and violates the spirit of* **Environment***, which is intended to encapsulate the information about the evaluation environment. So, while conceptually this class corresponds to the analytic form of the model, the actual code relies on details of the interfaces to other classes such as* **Path** *and* **Node***.*

⟨ *Model.h 4.5* ⟩ ≡

```
@o Model.h
#ifndef Model_h
#define Model_h 1

#include <boost/ptr_container/ptr_vector.hpp>
#include <iosfwd>
#include <memory>

#include "basic.h"
#include "HistoryComputer.h"
#include "MSPathError.h"
#include "Node.h"
#include "Specification.h"

  namespace mspath {

    // forward declaration. Only name needed for interface
  class Environment;
  class AbstractCovariates;

  class Model
    { public:
        ⟨ Model Types 4.5.0.1 ⟩
        ⟨ Model Constructors 4.5.0.2 ⟩
        ⟨ Model Accessors 4.5.0.4 ⟩
        ⟨ Model Operation 4.5.0.5 ⟩
        ⟨ Model Queries 4.5.0.6 ⟩
        ⟨ Model Simulation 4.5.0.7 ⟩
        ⟨ Model Printing 4.5.0.8 ⟩
    protected:
        ⟨ Model Data 4.5.0.3 ⟩
    } ; }
  #endif
```

This code is used in section 9.

The first type below is for use by clients.

⟨ *Model Types 4.5.0.1* ⟩ ≡

```
    // Container for HistoryComputers
  typedef boost:: ptr_vector⟨HistoryComputer⟩ TComputerContainer;

    // type of data to store in each Node
  typedef ModelData TModelData;
  typedef Array1D⟨State⟩ TState1D;
```

This code is used in section 4.5.

The **Model** *takes ownership of all the first three arguments, which must be heap-allocated. If the model has no misclassification, the second argument may be zero. If it has no path-dependent covariates, the third argument may be 0.*

See the comments in the data section for more on the meaning of the arguments.

Note this may throw any exception that validate( ) does.

Update the exception list for **ModelBuilder**::*makeModel( ) if the list here changes.*

Call release( ) with appropriate arguments (the **Environment***'s in which the model has been used) before destroying the* **Model***. It would be possible to make the class track the arguments it has been called with so this is automatic at destruction.*

⟨ *Model Constructors 4.5.0.2* ⟩ ≡

```
Model(AbstractSpecification *thepTransition, AbstractSpecification *thepMisclassification,
    // may be 0
  TComputerContainer*thepComputerContainer = 0,    // may be 0
  State theInitialState = 0U)
throw (InconsistentModel, BadInitialProbs, OneInitialState) :
  myInitialState(theInitialState), mypTransition(thepTransition),
      mypMisclassification(thepMisclassification), mypComputerContainer(thepComputerContainer)
  {
  validate( );    // next two are here, rather than in initializers, to
    // avoid unpleasantness if thepTransition == 0 (which it
    // shouldn't be)
  myIsAbsorbing.resize(thepTransition→nStates( ));
  myInitProbs.resize(nStates( ));
  for (size_t i = 0; i < myInitProbs.size( ); ++i)
    {
    if (i ≡ myInitialState)
      myInitProbs[i] = 1.0;
    else
      myInitProbs[i] = 0.0;
    myIsAbsorbing[i] = true;
    for (State j = 0U; j < nStates( ); ++j)
      {
      if (i ≠ j ∧ mypTransition→isPermissible(i, j))
        {
        myIsAbsorbing[i] = false;
        break;
        }
      ;
      }
    ;
    }
  }
  // from up data from associated Environment
void release(ScratchPad *thePad)
  {
  mypTransition→release(thePad);
  if (mypMisclassification.get( ))
```

```
        mypMisclassification→release(thePad);
    }
```

This code is used in section 4.5.

⟨ *Model Data 4.5.0.3* ⟩ ≡

```
    // true state at start of process
  const State myInitialState;

    // transition process between true states
  std::auto_ptr⟨AbstractSpecification⟩mypTransition;

    // measurement error (or 0)
  std::auto_ptr⟨AbstractSpecification⟩mypMisclassification;

    // compute path-dependent history values (or 0)
  std::auto_ptr < TComputerContainer > mypComputerContainer;

    // probability of each starting state (cache)
  Double1D myInitProbs;

    // true for absorbing states
  std:: vector ⟨bool⟩myIsAbsorbing;
```

This code is used in section 4.5.

⟨ *Model Accessors 4.5.0.4* ⟩ ≡

```
    // this is the size ModelData should have
    // it includes room for intermediate as well as final values
    // (e.g., the x for ln(x)).
  size_t nPathDependentVariables( ) const
    {
    if (mypComputerContainer.get( ))
      return mypComputerContainer→size( );
    return 0U;
    }

    // number of discrete states

  size_t nStates( ) const
    {
    return mypTransition→nStates( );
    }
```

This code is used in section 4.5.

We'll see how far we want to press this, versus letting some other class handle things.

⟨ *Model Operation 4.5.0.5* ⟩ ≡

```
    // generally, an excuse to compute history-dependent data
  void fillModelData(Node& theNode, Environment& theEnv);

    // heart of the Model
    // call after fillModelData
  void evaluate(Node& theNode, Environment& theEnv) throw (DataModelInconsistency);

  protected:

    // call when Model is completely set up
    // the checks here are minimal
    // In the current design, the last 2 exceptions can't arise
  void validate( )const  throw (InconsistentModel, BadInitialProbs, OneInitialState);
      // update Model() exceptions if these change
  public:
```

This code is used in section 4.5.

These queries are public, but should only be used by knowledgeable clients.

The definitions are inline to aid optimization.

⟨ *Model Queries 4.5.0.6* ⟩ ≡

```
  // is the indicated transition possible in one step?
bool isPossibleTransition(State theFrom, State theTo) const
  {
  return theFrom ≡ theTo ∨ mypTransition→isPermissible(theFrom, theTo);
  }

  // can true state be observed as indicated?

bool isPossibleObservation(State theTrue, State theObserved) const
  {   // arguably this function should never be called if there is
    // no misclassification
  if (theTrue ≡ theObserved)
    return true;
  return mypMisclassification.get( ) ≠ 0∧mypMisclassification→isPermissible(theTrue, theObserved);
  }

bool isPossibleObservation(State theTrue, ObsState theObserved) const
  {   // arguably this function should never be called if there is
    // no misclassification
  return theObserved < 0 ∨ isPossibleObservation(theTrue, static_cast⟨State⟩ (theObserved));
  }

  // initial probability vector

const Double1D& initialProbabilities( ) const
  {
  return myInitProbs;
  }

  // true if State is absorbing

const bool isAbsorbing(State s) const
  {
  return myIsAbsorbing[s];
  }
```

This code is used in section 4.5.

The model assigns each possible path and observed values a probability, given the data. These methods pick at random from those outcomes, weighting by the probability.

simulatePath( ) will generates a true state of the current step in the environment's path and records it by changing the state in the path.

simulateObservation( ) returns an observed state given the true path in the environment.

Both methods assume that the environment and path are properly set up, and that the current node is the focus of attention.

⟨ *Model Simulation 4.5.0.7* ⟩ ≡

```
void simulatePath(Environment& env);
State simulateObservation(Environment& env);
```

This code is used in section 4.5.

⟨ *Model Printing 4.5.0.8* ⟩ ≡

```
friend std::ostream & operator <<(std::ostream & ostr, const Model& model);
```

This code is used in section 4.5.

4.6 Computation and Caching

The final result comes from a series of transformations of the original data. The following sections take those up from the top down, but this one reviews the generall characteristics of those transformations and some repeated patterns.

4.6.1 Main Computation

At the lowest level, one or more sets of **Coefficients** *and* **Covariates** *are multiplied in a* **LinearProduct**. *A* **Specification** *transforms the results into a* **Double2D** *(a matrix of doubles); the transformation is generally nonlinear and expensive.*

Each class has a main operation to perform computations and extract values: **Covariates**::*values*( ), **Coefficients**::*multiply*( ), **LinearProduct**::*evaluate*( ), *and* **Specification**::*evaluate*( ). *The operations usually take an* **Environment**& *as an argument and return a* **const** *reference as a value. The operators themselves are conceptually* **const**, *though the interface is not always so. This is because many of the classes secretly maintain some state.*[31]

So evaluation takes place within a specific **Environment**, *which includes all the thread-specific information about a computation. Most of the classes above (except* **Covariates**) *are not themselves part of the* **Environment** *and may be used by several environments (threads) at the same time. Note that* **Covariates** *are not the data itself, but something that knows how to extract the data from the* **Environment**.

Secondly, note that the return values are not owned by the client that requests them. The class that returns the values is responsible for managing the lifetime of the results.

The references in the return values are only guaranteed to be good until the next evaluation call in the same environment. In the usage pattern of the program, that presents no problem. Clients who wish to preserve values beyond that must make copies.

[31] In view of the public nature of caching and dependence of results on the pattern of calls, the use of **const** may be a mistake. Also, I could make them all **const** and use casts to secretly make changes. The erratic use of **const** is not terribly desirable. However, the relevant state is actually stored in the environment, not the class, in most cases.

4.6.2 Caching

Given the expense of the computation, avoiding unnecessary recomputations may be a big win. Often, the values of **Covariates** *will not change from evaluation to evaluation. This is so for two reasons: first, the time step of the model is generally much smaller than that of the observations, so many steps necessarily use the same observational data. Also, the covariates may be constant across observations. Second, many parts of the model are often entirely constant, and so need to be evaluated only once.*

All these observations have much less force if the model has path-dependent terms. In that case, at each step at least one path-dependent covariate is likely to change in value, and even one change will force reevaluation of the entire **Specification**. *Since most of the computation is in the* **Specification** *rather than the earlier linear products, it's less clear that the overhead of caching produces gains in this case. However, even in that case the error term is often constant; avoiding recomputing it can save half the work.*

There are also more clever kinds of caching the program could do, though it currently doesn't. A longer-lived cache tied to a particular timesteps might help multiple paths when they cross the same timestep. There may be other patterns of sharing that help too, based more on the covariate patterns than their location in the path tree.

Caching raises several issues. Who owns the cache? How do you know if the cached value is still good? Should objects assume that they are only called to evaluate an environment if the cache has changed?

The cache is owned by the environment and is the responsibility of the associated object, the server, not the client. For example, the **LinearProduct**, *not the* **Specification** *that calls the* **LinearProduct**, *caches the results of evaluating the* **LinearProduct**. *It is the* **Environment***'s responsibility to clean up the cache when the former is destroyed.*

This design implies that caches are stored separately for each thread. The mementos, to be discussed shortly, are also thread-specific.

In the rest of this dicussion I will refer to the focal object (e.g., **LinearProduct***) as a* server, *emphasizing that it provides a service, a computation, to a* client, *the* **Specification** *in the above example. Note that most objects have dual roles; for example,* **LinearProduct** *is also a client of* **Covariates** *and* **Coefficients**.

The alternative of having the client own the cache makes some sense, since it is the client that is interested in using the value. I didn't do that, because such an approach has several drawbacks. First, applied rigorously one would end up with the top-level object owning all the caches, since almost all servers are in turn clients.[32] *Second, the server, not the client, is more likely to know what a good caching strategy is. Third, if several clients call the same server, caching will be more efficient on the server.*[33]

A server that wants to know if its cached value is still good needs to know if the values on which it depends have changed, that is if the values in the servers of which it is a client has changed. For example, a **Specification** *caches the results of its transformation of a* **LinearProduct**. *To know if it needs to recompute, it needs to know if the value of* **LinearProduct** *has changed—more precisely, if it has changed* since the computation of the value in **Specification**'s cache.

To accomplish this, servers provide a memento *of their state after a computation. For the memento to be valid it must be requested immediately after the computation.*[34] *The* client *owns the memento, though its*

[32] In the sense that the **Environment** holds on to all the caches, a top-level object is the owner in the current scheme. But the **Environment**'s role is passive; the work is mostly done by the server.

[33] Though I don't think the program currently uses any server for more than one client.

[34] That is, immediately in the same thread. If the sequence is thread 1 calls *evaluate*, thread 2 calls *evaluate*, thread 1 calls *memento*, the memento is still good. The exact condition is that *memento* must be called before the next evaluation type call in the same thread.

I should double-check that this non-interference is actually true.

contents are opaque to the client and the client usually stores the result in some thread-specific data in the **Environment**. *The server can interpret the memento to tell if things have changed.*

A memento has type **ScratchData** ∗, *and can be obtained two ways.* **ScratchData** ∗*memento*(**Environment**&) *is a method for each server that creates a new memento.* **ScratchData** ∗*memento*(**Environment**&, **ScratchData** ∗∗) *accepts a pointer to a memento as an argument, and updates the memento with the new state. It also returns the memento. As a special case, if the argument to memento is* 0, *the server will generate a new memento, just like the memento*( ) *call.*

Note that although mementos are **ScratchData** *they are not ordinarily held directly by the* **ScratchPad**. *Clients usually store pointers to mementos along with other information in their keyed* **ScratchData**.

To use the memento, clients call **bool** *isChanged*(**Environment**& *env*, **ScratchData** ∗*memento*) *on the server. The call returns* `true` *if the server, evaluated in env, would produce a different answer than when it produced memento.*[35] *If the call returns* `false` *the client knows the value has not changed.*

Since isChanged( ) *from a server means a client must recompute, the first call (indicated by a* 0 *pointer for the memento) to it should always return true. This is true even if a particular server is all constant, and thus never changes.*

I have not defined an abstract interface that includes these three methods; I probably should. There are two reasons not to. First, the interface may vary slightly from class to class; in particular it is different for **Covariates**, *which are part of an* **Environment**, *than for the other classes, which are not. I may not have implemented one-argument memento*( ) *uniformly, and maybe should consider dropping it. Finally, virtual methods are slower than non-virtual, so the current implementation may be faster than alternatives.*

The final issue is whether servers should assume that they are only called to do an evaluation when things have changed. More concretely, should they check if the environment has changed, and if not return their cached values?

Currently, the answer is yes. At first it seemed to be a saving to skip the check; after all, a client that found nothing had changed would not ask the server for a new value. This proved a brittle optimization, and the assumption is not necessarily true. A **SumLinearProducts**, *which sums several LinearProducts will need to recompute if any of its terms change. Thus, if one term changes it may need to get the others, even if they haven't changed, in order to get the sum.*[36]

4.6.3 Related Information

The implementation of caches relies heavily on the **ScratchData** *mechanisms desribed in* *§4.19, page 132.*

The following sections present the concrete classes that implement the protocols described above.

[35] Servers may be a bit sloppy, and may return `true` in some cases where there really is no change. But they must always return true when there is a change.

[36] There was another problem before the addition of mementos. Calling *isChanged* sometimes updated the state, so that a later call to *evaluate* would get a stale cache value because the state of the **Environment** had not changed since the last call to the server, even though it had changed since the client's cache was computed.

The main risk with brittleness is that, as in the example just given, the server will return `false` for *isChanged* incorrectly or, roughly equivalently, return a stale cached value. It's not clear this is really a concern since the introduction of mementos, so some of these decisions might merit revisiting.

4.7 Specification

An **AbstractSpecification** *is a small model transforming a vector into a matrix of probabilities or hazard rates. In practice, the vector is usually a result of a linear product of coefficients times covariates.*

These classes specialize in two things. First, they know how to map from the input vector to a matrix. Second, and logically following the first step, they describe the functional form linking the linear products to the final results.

For the first step, these classes hold a matrix of boolean values. They reads this matrix starting at the upper-left-hand corner, moving along the first row, then the second, and so on. For each `true` *entry they fill in the corresponding matrix with the next entry in the input vector (starting with the first entry at the top).*

For the second, step, these class contain the functional forms linking the lower-level inputs to the higher level outputs, which are in turn used by the larger **Model**. *For example, a relation could be linear, log-linear, logistic, or log-log. It could be binary or multinomial. This is the same territory covered analytically in §2.2.4, page 18. Each of these functional forms has a different concrete class.*

A **Specification** *implements a multinomial logistic model. This is not a simple transformation of the linear value for a single cell: the probability of a given outcome depends on the linear terms computed for each possible outcome. This class does the most intense number-crunching of any in the system.*

A **SimpleSpecification** *implements a simple model in which the probabilities are constant and pre-specified.*

More details about each specific class appear in the appropriate section below.

Typical useage is to create an appropriate **LinearProduct** *and instantiate a* **Specification** *using it. Then evaluate( ) the* **Specification** *in some* **Environment**, *and use the resulting matrix.*

If the result concerns transition probabilities or rates, the $(i,\ j)$ *term applies to the transition from state* i *to state* j. *If the result concerns misclassification, the* $(i,\ j)$ *term is the parameter related to true state* i *being observed as* j.

AbstractSpecification *and descendants can be used on multiple threads; all thread-specific data are passed in and out of the class, which is a* **ScratchDataProducer**.

```
⟨ Specification.h 4.7⟩ ≡
    @o Specification.h
  #ifndef Specification_h
  #define Specification_h 1

  #include <iosfwd>

  #include "basic.h"
  #include "LinearProduct.h"
  #include "MSPathError.h"
  #include "ScratchDataProducer.h"

    namespace mspath {
        class Environment;
        ⟨ AbstractSpecification interface 4.7.1⟩
        ⟨ SimpleSpecification interface 4.7.2⟩
        ⟨ Specification interface 4.7.3⟩}    // end namespace mspath
  #endif    // Specification_h
```

This code is used in section 9.

4.7.1 AbstractSpecification

This defines the interface that all subclassess must obey.

Note that most references in the text to **Specification** *should be taken to mean* **AbstractSpecification***; the references are a historical accident of the fact that the latter was developed after the former.*

You will notice this class has some state. This is a short-cut, because it is shared by all subclasses.

⟨ *AbstractSpecification interface 4.7.1* ⟩ ≡

```
class AbstractSpecification : public ScratchDataProducer
  { public:
      ⟨ AbstractSpecification c'tor 4.7.1.1 ⟩
      ⟨ AbstractSpecification basic behaviors 4.7.1.2 ⟩
      ⟨ AbstractSpecification queries 4.7.1.3 ⟩
  protected: ⟨ AbstractSpecification protected accessors 4.7.1.4 ⟩
      ⟨ AbstractSpecification data 4.7.1.5 ⟩
  } ;
```

This code is used in section 4.7.

The class takes ownership of the argument, which must be heap-allocated.

⟨ *AbstractSpecification c'tor 4.7.1.1* ⟩ ≡

```
AbstractSpecification(Bool2D *thepPermissible) : mypPermissible(thepPermissible)
  {}

virtual ~AbstractSpecification( )
  {
  delete mypPermissible;
  }
```

This code is used in section 4.7.1.

Subclasses must implement the following methods. In addition, they probably need to reimplement **virtual void ScratchDataProducer**::*release*(**ScratchPad** ∗) **const***, being sure to call the method in that class.*

⟨ *AbstractSpecification basic behaviors 4.7.1.2* ⟩ ≡

```
    // compute output values based on observation
virtual const Double2D& evaluate(Environment& theEnv)const  = 0;

virtual bool isChanged(Environment& theEnv, ScratchData *thepMemento) = 0;

virtual void memento(Environment& theEnv, ScratchData **theppMemento)const  = 0;

friend std::ostream & operator <<(std::ostream & ostr,
    const mspath::AbstractSpecification & spec);
```

This code is used in section 4.7.1.

Only knowledgeable clients should need to know internal details, as given by the next set of query functions.

Currently there's no need to make these virtual.

⟨ *AbstractSpecification queries 4.7.1.3* ⟩ ≡

```
// number of states this model concerns
// result of evaluate( ) is nStates x nStates
size_t nStates( ) const
  {
  return permissible( ).ncols( );
  }

// Return true if indicated element is OK.
// Clients should not ask about the diagonals,
// which are implicitly true.

bool isPermissible(State f, State t) const
  {
  return permissible( )(f, t);
  }
```

This code is used in section 4.7.1.

Other methods should use the following method, not direct instance variable access, to get internal information.

⟨ *AbstractSpecification protected accessors 4.7.1.4* ⟩ ≡

```
Bool2D& permissible( ) const
  {
  return *mypPermissible;
  }
```

This code is used in section 4.7.1.

The following information lets me map from a vector of values into a matrix. Entries that are true indicate the corresponding matrix entry should get a value. Handling of the diagonal is generally special, though.

I own the object pointed to.

⟨ *AbstractSpecification data 4.7.1.5* ⟩ ≡

```
Bool2D *mypPermissible;
```

This code is used in section 4.7.1.

4.7.2 SimpleSpecification

A **SimpleSpecification** *is one in which the user wishes to specify fixed probabilities in advance. The values given are interpreted directly as probabilities, with the diagonal being computed as a residual. They will not change during the course of evaluation, and they are the same for all threads.*

If the input values don't have sane properties for probabilities, the constructor will throw an exception. Since the values are used untransformed, they must be valid probabilities (between 0 and 1).

The values themselves may still use the constraints mechanism; thus this class uses a **ConstantLinearProduct** *rather than something simpler (e.g.,* **InterceptCoefficients***) as one of its parts.*

⟨ *SimpleSpecification interface 4.7.2* ⟩ ≡

```
class SimpleSpecification : public AbstractSpecification
  { public:
      ⟨ SimpleSpecification constructors 4.7.2.1 ⟩
      ⟨ SimpleSpecification basic behaviors 4.7.2.2 ⟩
  protected: ⟨ SimpleSpecification protected accessors 4.7.2.3 ⟩
      ⟨ SimpleSpecification mementos 4.7.2.5 ⟩
      ⟨ SimpleSpecification data 4.7.2.4 ⟩ } ;
```

This code is used in section 4.7.

I assume ownership of the arguments, which should be heap-allocated.

⟨ *SimpleSpecification constructors 4.7.2.1* ⟩ ≡

```
SimpleSpecification(ConstantLinearProduct *thepLP, size_t thenStates,
    Bool2D *thepPermissible) throw (InconsistentModel);

virtual ~SimpleSpecification( )
  {
  delete mypLP;
  }

  // this class puts no data on the ScratchPad, so no cleanup needed

virtual void release(ScratchPad *thePad) const
  { }
```

This code is used in section 4.7.2.

The arguments to the following functions are for compatibility with the protocol of the base class; this class doesn't need them.

⟨ *SimpleSpecification basic behaviors 4.7.2.2* ⟩ ≡

```
// compute output values based on observation
virtual const Double2D& evaluate(Environment& theEnv) const
  {
  return myResult;
  }

virtual bool isChanged(Environment& theEnv, ScratchData *thepMemento)
  {
  return thepMemento ≡ 0;
  }

virtual void memento(Environment& theEnv, ScratchData **theppMemento) const
  {
  if (*theppMemento ≡ 0)
    *theppMemento = new Memento( );
  }

friend std::ostream & operator <<(std::ostream & ostr,
    const mspath::SimpleSpecification & spec);
```

This code is used in section 4.7.2.

Use this, not direct access.

⟨ *SimpleSpecification protected accessors 4.7.2.3* ⟩ ≡

```
ConstantLinearProduct& linearProduct( ) const
  {
  return *mypLP;
  }
```

This code is used in section 4.7.2.

Normally the results must be tucked into a thread-specific spot, but since the result is a constant, there's no need for that here.

⟨ *SimpleSpecification data 4.7.2.4* ⟩ ≡

```
ConstantLinearProduct *mypLP;
Double2D myResult;    // final values of the specification
```

This code is used in section 4.7.2.

We don't need to store any state in the memento, since we never change. However, the existence of a memento is important to our clients, and to us for indicating if it's the first time through.

⟨ *SimpleSpecification mementos 4.7.2.5* ⟩ ≡

```
struct Memento : public ScratchData
  { };
```

This code is used in section 4.7.2.

4.7.3 Multinomial Logit Specification

This class is the typical **AbstractSpecification** *to use. Because of this and historical accident it is known as* **Specification**. *It implements a multinomial logistic model.*

Because of the repeated transformations and exponentiation, it seems likely that most of the number-crunching time of the program will be spent in this class.

⟨ *Specification interface 4.7.3* ⟩ ≡

```
class Specification : public AbstractSpecification
  { public:
      ⟨ Specification Constructors 4.7.3.1 ⟩
      ⟨ Specification Basic Behavior 4.7.3.2 ⟩

  protected:
      ⟨ Specification Protected Accessors 4.7.3.3 ⟩
      ⟨ Specification Protected Computation 4.7.3.4 ⟩
      ⟨ Specification Scratch Data 4.7.3.5 ⟩
      ⟨ Specification Data 4.7.3.6 ⟩
  } ;   // end Specification
```

This code is used in section 4.7.

Takes ownership of arguments to constructor; those arguments should be heap-allocated.

⟨ *Specification Constructors 4.7.3.1* ⟩ ≡

```
Specification(AbstractLinearProduct *thepLP, Bool2D *thepPermissible) :
      AbstractSpecification(thepPermissible), mypLP(thepLP)
  {}

~Specification( )
  {
  delete mypLP;
  }

virtual void release(ScratchPad *thePad) const
  {
  ScratchDataProducer::release(thePad);
  linearProduct( ).release(thePad);
  }
```

This code is used in section 4.7.3.

The next methods provide the core functionality of the class.

⟨ *Specification Basic Behavior 4.7.3.2* ⟩ ≡

```
// compute output values based on observation
virtual const Double2D& evaluate(Environment& theEnv) const;

virtual bool isChanged(Environment& theEnv, ScratchData *thepMemento)
  {
  return linearProduct( ).isChanged(theEnv, thepMemento);
  }

virtual void memento(Environment& theEnv, ScratchData **theppMemento) const
  {
  linearProduct( ).memento(theEnv, theppMemento);
  }

friend std::ostream & operator <<(std::ostream & ostr, const mspath::Specification & spec);
```

This code is used in section 4.7.3.

Other methods should use the following method, not direct instance variable access, to get internal information.

⟨ *Specification Protected Accessors 4.7.3.3* ⟩ ≡

```
AbstractLinearProduct& linearProduct( ) const
  {
  return *mypLP;
  }
```

This code is used in section 4.7.3.

The core of the computation.

⟨ *Specification Protected Computation 4.7.3.4* ⟩ ≡

```
void computeResult(Double2D& theResult, const Double1D& theLinearResult) const;
```

This code is used in section 4.7.3.

Below is the type used for thread-specific data.

⟨ *Specification Scratch Data 4.7.3.5* ⟩ ≡

```
struct Scratch : public ScratchData
  {
  Scratch(const Specification& theSpec) : result(0.0, theSpec.permissible( ).nrows( ),
        theSpec.permissible( ).ncols( )), pMemento(0)
    { }
  ~Scratch( )
    {
    delete pMemento;
    }
  Double2D result;    // final values
  ScratchData *pMemento;    // as of last computation
  };
```

This code is used in section 4.7.3.

Here are the data I own.

⟨ *Specification Data 4.7.3.6* ⟩ ≡

```
AbstractLinearProduct *mypLP;
```

This code is used in section 4.7.3.

4.8 LinearProduct

Returns a vector containing the product of coefficients and any relevant covariates. This class is part of the **Model**, *and like it and* **Coefficients** *it is not thread-specific. However, unlike* **Coefficients**, *it knows how to retrieve the appropriate thread-specific* **Covariates**.

All these classes own their coefficients and know the exact protocol for accessing them.

As far as I can tell, non-member functions (e.g, **operator** <<*) don't match the run-time types of their arguments, so I must do an ugly work-around with printOn.*

⟨ *LinearProduct.h 4.8*⟩ ≡

```
@o LinearProduct.h
#ifndef LinearProduct_h
#define LinearProduct_h

#include <iosfwd>

#include "basic.h"

#include "Coefficients.h"
#include "Covariates.h"
#include "Environment.h"
#include "ScratchDataProducer.h"

    // used by SumLinearProducts
#include <boost/ptr_container/ptr_vector.hpp>

  namespace mspath {

  class AbstractLinearProduct : public ScratchDataProducer
    {

  public:
    virtual ~AbstractLinearProduct( )
      { }

      // return true if result of evaluate( ) when state was
      // memento is no longer current.
      // If thepMemento is 0, return true.

    virtual bool isChanged(Environment& theEnv, ScratchData *thepMemento)const  = 0;

      // set memento for state as of last evaluate( )
      // Will create memento if arg is 0
    virtual void memento(Environment& theEnv, ScratchData **theppMemento) = 0;

      // number of elements returned by evaluate( )
    virtual size_t size( )const  = 0;

      // return coefficients time relevant covariates in theEnv
      // Return value only good until next call to this function
      // with the same theEnv
    virtual const Double1D& evaluate(Environment& theEnv)const  = 0;

      // subclasses must re-implement

    virtual std::ostream & printOn(std::ostream & ostr) const
      {
```

```
    ostr << "AbstractLinearProduct";
    return ostr;
    }
    // remember virtual void release(ScratchPad *) const too.
  friend std::ostream & operator <<(std::ostream & ostr, const AbstractLinearProduct& lp)
    {
    lp.printOn(ostr);
    return ostr;
    }
  ;
  };
⟨ ConstantLinearProduct definition 4.8.1 ⟩
⟨ DataLinearProduct definition 4.8.2 ⟩
⟨ PathDependentLinearProduct definition 4.8.3 ⟩
⟨ SumLinearProducts Definition 4.8.4 ⟩
}
#endif
```

This code is used in section 9.

4.8.1 ConstantLinearProduct

This class holds constant coefficients. It doesn't need to take a product, and it has no covariates.

However, whether the result is "new" is thread-specific.

The approach to isChanged( ) and mementos is slightly sloppy, but should work for anticipated useage patterns (namely call isChanged( ), set values if it's `true`*, get memento.*

⟨ *ConstantLinearProduct definition 4.8.1* ⟩ ≡

```
class ConstantLinearProduct : public AbstractLinearProduct
  {
public:   // take ownership of thepCoefficients
  ConstantLinearProduct(InterceptCoefficients *thepCoefficients) :
        mypCoefficients(thepCoefficients)
    { }
  virtual ~ConstantLinearProduct( )
    {
    delete mypCoefficients;
    }
    // I have no thread cache
  virtual void release(ScratchPad *thePad) const
    { }
  virtual bool isChanged(Environment& theEnv, ScratchData *thepMemento) const
    {
    return thepMemento ≡ 0;
    }
  virtual void memento(Environment& theEnv, ScratchData **theppMemento)
    {
    if (*theppMemento ≡ 0)
      *theppMemento = new TScratchData( );
    }
    // number of elements returned by evaluate( )
  virtual size_t size( ) const
    {
    return mypCoefficients→nTotal( );
    }
  virtual const Double1D& evaluate(Environment& theEnv) const
    {
    return mypCoefficients→multiply(theEnv);
    }
    // allow knowledgeable clients to skip the meaningless
    // argument
  const Double1D& evaluate( ) const
    {
    return mypCoefficients→multiply( );
    }
```

```
    virtual std::ostream & printOn(std::ostream & ostr) const;
  protected:
    // const InterceptCoefficients except I delete it
  InterceptCoefficients *const mypCoefficients;
  struct TScratchData : public ScratchData
  {   // that's right: there's nothing here
  TScratchData( )
    {}
  };
};
```

This code is used in section 4.8.

4.8.2 DataLinearProduct

This handles multiplying slopes times conventional, observed data. Note that the **Environment** *tracks exactly which parts of the data are relevant.*

The data in question can be either covariates for the transition rates or for misclassification.

⟨ *DataLinearProduct definition 4.8.2* ⟩ ≡

```
class DataLinearProduct : public AbstractLinearProduct
  {
public:
  DataLinearProduct(const SlopeCoefficients *pSlope,
        bool theUseMisclassificationData = false) : mypCoefficients(pSlope),
        myUseMisclassificationData(theUseMisclassificationData)
    { }
  virtual ~DataLinearProduct( )
    {
    delete mypCoefficients;
    }
  virtual void release(ScratchPad *thePad) const
    {
    ScratchDataProducer::release(thePad);
    mypCoefficients->release(thePad);
    }
  virtual bool isChanged(Environment& theEnv, ScratchData *thepMemento) const;
  virtual void memento(Environment& theEnv, ScratchData **theppMemento);
    // number of elements returned by evaluate( )
  virtual size_t size( ) const
    {
    return mypCoefficients->nTotal( );
    }
  virtual const Double1D& evaluate(Environment& theEnv) const;
  virtual std::ostream & printOn(std::ostream & ostr) const;
protected:
  const SlopeCoefficients *const mypCoefficients;
  const bool myUseMisclassificationData;
  struct ScratchDataLinearProduct : public ScratchData
    {
    ScratchDataLinearProduct(const Environment& theEnv,
          const DataLinearProduct& theLP) : covariates(theLP.data(theEnv)),
          results(theLP.size( )), pMemento(0)
      { }
    ~ScratchDataLinearProduct ( )
      {
      delete pMemento;
      }
```

```
  MatrixCovariates covariates;
  Double1D results;    // results of multiplication
  ScratchData *pMemento;    // state at that time
  };
ScratchDataLinearProduct& scratch(Environment& theEnv) const;
const Double2D& data(const Environment& theEnv) const
  {
  if (myUseMisclassificationData)
    return theEnv.data( ).miscCovs( );
  return theEnv.data( ).covs( );
  }
};
```

This code is used in section 4.8.

4.8.3 PathDependentLinearProduct

This class multiplies slopes times path-dependent variables. It saves the 0-based indices of which variables to pull out of the modelData on a **Node**.

⟨ *PathDependentLinearProduct definition 4.8.3* ⟩ ≡

```
class PathDependentLinearProduct : public AbstractLinearProduct
  {
public:
  PathDependentLinearProduct(const SlopeCoefficients *pSlope, const
        TIndirect1D& theIndices) : mypCoefficients(pSlope), myIndices(theIndices)
    { }
  virtual ~PathDependentLinearProduct( )
    {
    delete mypCoefficients;
    }
  virtual void release(ScratchPad *thePad) const
    {
    ScratchDataProducer::release(thePad);
    mypCoefficients→release(thePad);
    }
  virtual bool isChanged(Environment& theEnv, ScratchData *thepMemento) const;
  virtual void memento(Environment& theEnv, ScratchData **theppMemento);
    // number of elements returned by evaluate( )
  virtual size_t size( ) const
    {
    return mypCoefficients→nTotal( );
    }
  virtual const Double1D& evaluate(Environment& theEnv) const;
  virtual std::ostream & printOn(std::ostream & ostr) const;
protected:
  const SlopeCoefficients *const mypCoefficients;
  const TIndirect1D myIndices;
  struct ScratchPathDependentLinearProduct : public ScratchData
    {
    ScratchPathDependentLinearProduct(const TIndirect1D& theIndices) :
          covariates(theIndices)
      { }
    PathCovariates covariates;
    };
  };
```

This code is used in section 4.8.

4.8.4 Sums of Linear Products

Because the previous classes are relatively primitive, we will usually want to combine them, e.g., a slope and an intercept. This class makes that possible.

It owns all the **LinearProduct***'s that are inserted into it.*

Although this class provides an isChanged( ) interface, it really needs to be clever, since its components are likely to change at different rates.

Typical useage is to instantiate the class, insert( ) a bunch of linear products, and then use it.

Note the 's' at the end of the class name; it seemed more natural.

If you insert the products that vary most rapidly last, performance will be slightly better.

⟨ *SumLinearProducts Definition 4.8.4* ⟩ ≡

```
class SumLinearProducts : public AbstractLinearProduct
  {
public:
  SumLinearProducts( )
    { }
    // make the argument part of the sum
    // This is part of initial object creation.
    // Do NOT call it after calling other methods.
  void insert(AbstractLinearProduct *thepLinearProduct);
  virtual ~SumLinearProducts( )
    { }
  virtual void release(ScratchPad *thePad) const
    {
    ScratchDataProducer::release(thePad);    /* this doesn't compile for (TProducts::iterator i =
        myProducts.begin(); i != myProducts.end(); ++i)  i-¿release(thePad);   */
    for (size_t i = 0; i < myProducts.size( ); ++i)
      myProducts[i].release(thePad);
    }    // return true if result of evaluate( ) when state was
    // memento is no longer current.
    // If thepMemento is 0, return true.
  virtual bool isChanged(Environment& theEnv, ScratchData *thepMemento) const;
    // set memento for state as of last evaluate( )
    // Will create memento if arg is 0
  virtual void memento(Environment& theEnv, ScratchData **theppMemento);
    // number of elements returned by evaluate( )
    // obviously senseless before first insert
  virtual size_t size( ) const
    {
    return myProducts.front( ).size( );
    }
    // return coefficients time relevant covariates in theEnv
```

```
    // Return value only good until next call to this function
    // with the same theEnv
  virtual const Double1D& evaluate(Environment& theEnv) const;

  virtual std::ostream & printOn(std::ostream & ostr) const;

  protected:
    typedef boost:: ptr_vector⟨AbstractLinearProduct⟩ TProducts;
  TProducts myProducts;
    ⟨ SumLinearProducts::Memento Definition 4.8.4.1 ⟩
    ⟨ SumLinearProducts::State Definition 4.8.4.2 ⟩
  } ;
```

This code is used in section 4.8.

We need a memento both for ourself, to minimize computation, and for our clients.

Really there's only one memento at the moment, preserving the state of the **Covariates**. *However, there is no assurance it will always be so. Even now, some* **LinearProduct***'s return phoney mementos, so we'd need to check for that if we were trying to be clever.*

Instead, we go with safe, dumb, and relatively simple. The i'th index is the memento for the corresponding element of myProducts in the main class.

The [ ] *operator returns a reference to a pointer, which I hope will work. Each* **LinearProduct**::*memento*( ) *method requires the address of a pointer.*

```
⟨ SumLinearProducts::Memento Definition 4.8.4.1 ⟩ ≡
  struct Memento : public ScratchData
    { Memento(size_t n) : mementos(n, 0)
      {}

    virtual ~Memento( )
      {
      for (size_t i = 0; i < mementos.size( ); ++i)
        {
        delete mementos[i];
        mementos[i] = 0;    // extra protection
        }
      }

    ScratchData *&operator [](size_t i)
      {
      return mementos[i];
      }

    std:: vector < ScratchData *⟩ mementos;    // for corresponding product
    } ;
```

This code is used in section 4.8.4.

The state records the results and appropriate mementos.

⟨ *SumLinearProducts::State Definition 4.8.4.2* ⟩ ≡

```
class State : public ScratchData
  {
public:
  State(const SumLinearProducts& theProd) : myResult(theProd.size( )),
        myMemento(theProd.myProducts.size( ))
    { }
  ;
  ScratchData *&memento(size_t i)
    {
    return myMemento[i];
    }
  Memento& memento( )
    {
    return myMemento;
    }
  size_t size( ) const
    {
    return myResult.size( );
    }
  Double1D& result( )
    {
    return myResult;
    }
protected:
  Double1D myResult;    // the sum of the linear products
  Memento myMemento;
  };
```

This code is used in section 4.8.4.

4.9 Coefficients

This second-generation implementation of coefficients adds thread-safety, and also can handle cases with only slopes, or only intercepts.

Several threads may use the same **Coefficients** *object. These classes work on thread-specific information (such information is both an input and output of the multiply( ) method), but are not themselves thread-specific. In contrast,* **Covariates** are *thread-specific.*

Both of the concrete classes implement constraints on the coefficients, as described more fully in §6.1, page 226. In the language of that section, they take effective coefficients and constraints on them as inputs, and convert those into total coefficients. However, the constraints here are 0-based. *The general rule is that totalCoefficient[i] = effectiveCoefficient[constraint[i]]. Typically there are more total than effective coefficients.*

These classes deliberately know nothing about where their covariates come from, and they know nothing about the fact that their results are destined for a matrix. The result of multiply( ) is a vector; other classes are responsible for mapping from this vector to the matrices of permissible transitions or misclassifications.

Currently, these classes provide very little of their internal information to clients: the intent is to use them to multiply covariates, not to go peeking at their internal structure.

The classes assume valid inputs and do little error checking.

I'll put several related classes in the one header.

For these classes the inherited release should be OK; the covariates are kept in the thread-specific data that are destroyed by the default release( ).

⟨ *Coefficients.h 4.9* ⟩ ≡

```
@o Coefficients.h
#ifndef Coefficients_h
#define Coefficients_h 1
#include "basic.h"
#include "ScratchDataProducer.h"
#include "ScratchPad.h"

#include <iostream>

namespace mspath {
    class AbstractCovariates;

  // Abstract Interface
  // Since number of arguments may vary, there is no common function.
class AbstractCoefficients : public ScratchDataProducer
  {

public:
  virtual ~AbstractCoefficients( )
    { }
  };

  // concrete classes should also implement <<

  // Concrete Classes
```

⟨ *InterceptCoefficients Definition 4.9.1* ⟩
⟨ *SlopeCoefficients Definition 4.9.2* ⟩

} // end **namespace mspath**
#endif

This code is used in section 9.

4.9.1 InterceptCoefficients

The next class holds the intercepts. Since they don't change, we can use them across all threads, as well as pre-calculating the total coefficients.

⟨ *InterceptCoefficients Definition 4.9.1* ⟩ ≡

```
class InterceptCoefficients : virtual public AbstractCoefficients
{
public:
  InterceptCoefficients(const Double1D& theIntercepts, const TIndirect1D& theConstraints) :
      myEffectiveIntercepts(theIntercepts), myInterceptConstraints(theConstraints),
      myTotalIntercepts(theIntercepts[theConstraints])
    { }
  virtual ~InterceptCoefficients( )
    { }
  // accessors
  // size of result vector
  size_t nTotal( ) const
    {
    return interceptConstraints( ).size( );
    }
  // multiply for consistency with other classes.
  // This one has nothing to multiply by.
  // I don't use theScratchPad, but want to hide that detail.
  const Double1D& multiply(ScratchPad& thePad) const
    {
    return totalIntercepts( );
    }
  // however, some may prefer this simpler,
  // but non-standard interface
  const Double1D& multiply( ) const
    {
    return totalIntercepts( );
    }
  friend std::ostream & operator <<(std::ostream & theStream,
      const InterceptCoefficients& theCoeff );
protected:   // accessors
  double operator [](size_t i) const
    {
    return totalIntercepts( )[i];
    }
  const Double1D& totalIntercepts( ) const
    {
    return myTotalIntercepts;
    }
```

```
const Double1D& effectiveIntercepts( ) const
  {
  return myEffectiveIntercepts;
  }
const TIndirect1D& interceptConstraints( ) const
  {
  return myInterceptConstraints;
  }
  // data
const Double1D myEffectiveIntercepts;
const TIndirect1D myInterceptConstraints;
const Double1D myTotalIntercepts;
}
;
```

This code is used in section 4.9.

4.9.2 SlopeCoefficients

The next class holds all the coefficients except the intercepts.

Coefficients for a single final value are in a column, as are their corresponding data. *These are not the usual conventions, but are consistent with the expected C inputs.*

Note that the constraint structure here allows for abitrary constraints between any pair of coefficients. This is more general than the **R** *interface, at least in* **msm**.

⟨ *SlopeCoefficients Definition 4.9.2* ⟩ ≡

```
class Environment;

class SlopeCoefficients : public virtual AbstractCoefficients
{
public:
  SlopeCoefficients(const Double1D& theCoefficients, const TIndirect2D& theConstraints) :
      myEffectiveSlopes(theCoefficients), mySlopeConstraints(theConstraints),
      myTotalSlopes(theCoefficients[theConstraints])
    { }

  virtual ~SlopeCoefficients( )
    { }

    // accessors
    // number of covariates

  size_t nCovs( ) const
    {
    return totalSlopes( ).nrows( );    // yes, rows
    }

    // number of final terms returned by multiply( )

  size_t nTotal( ) const
    {
    return totalSlopes( ).ncols( );
    }

    // main action

  const Double1D& multiply(Environment& theEnv, AbstractCovariates& theCovs) const;

  friend std::ostream & operator <<(std::ostream & theStream, const SlopeCoefficients& theCoeff );

protected:    // accessors
  double operator ( )(size_t i, size_t j) const
    {
    return totalSlopes( )(i, j);
    }

  const Double2D& totalSlopes( ) const
    {
    return myTotalSlopes;
    }

  const Double1D& effectiveSlopes( ) const
    {
```

```
    return myEffectiveSlopes;
    }
  const TIndirect2D& slopeConstraints( ) const
    {
    return mySlopeConstraints;
    }
    // data
  const Double1D myEffectiveSlopes;
  const TIndirect2D mySlopeConstraints;
  const Double2D myTotalSlopes;
    // secret data store
  class WorkData : public ScratchData
    {
  public:
    WorkData(size_t n) : results(n), pCovariateMemento(0)
      {}
    virtual ~WorkData( )
      {
      delete pCovariateMemento;
      }
    Double1D results;
    ScratchData *pCovariateMemento;    // at time results computed
    };
  }
  ;
```

This code is used in section 4.9.

4.10 Covariates

This family of classes describes model covariates. It has two functions:

- *Mediate between the* **Model** *and its helpers on one side, and the* **Environment** *and* **Data** *on the other. It conceals the details of how the covariates are obtained.*
- *Assist optimization to minimize reevaluations of the expressions based on the covariates. Note that doing so also requires intimate knowledge of the* **Environment**.

Concrete subclasses know how to obtain particular kinds of covariates. They could even combine several sources of data (e.g., from the **Data** *and the* **Path***) into a single vector.*[37]

Currently, when we are on a **Node** *we are computing the probability of the transition* to *that* **Node** *from the previous one. As discussed elsewhere (§2.2.6, page 20), we could use values from anywhere along the interval. This implementation gets data values from the end, and path values from the beginning.*

LinearProduct*'s create and own* **Covariates***;* **Coefficients** *use them.*[38]

Covariates *are thread-specific. They require a knowledge of the* **Environment***; a pointer to the latter could go in the class, but it seems more consistent with the general approach to pass* **Environment** *as an argument. Obviously you must pass in the same* **Environment** *as was used to create the* **Covariates***.*[39]

Various users of **Covariates** *may wish to know if they have changed since last seen. Since different users may have different access patterns,* **Covariates** *provides a memento to describe their state. The client becomes the owner of the memento, but should not attempt to peak inside it.*[40] *The memento preserves the state as of the last call to values( ); I recommend getting the memento just after that call.*

I introduced mementos to deal with the sequencing problems just noted, but it would probably be simpler to make more assumptions about the client useage patterns and reinternalize these issues. Probably values( ) should be performed unconditionally, assuming the client has already determined things have changed.

There may be additional issues if several computations (e.g., misclassification and transitions) share the same covariates. I believe that in the current implementation they actually use distinct **Covariates** *objects, even if they are substantively the same. This is yet another example of the safety of using explicit state in mementos.*

⟨ *Covariates.h 4.10* ⟩ ≡
```
  @o Covariates.h
#ifndef Covariates_h
#define Covariates_h 1

#include "basic.h"
#include "ScratchData.h"

  namespace mspath {
  class Environment;
  ⟨ AbstractCovariates Interface 4.10.1 ⟩
  ⟨ MatrixCovariates Interface 4.10.2 ⟩
  ⟨ PathCovariates Interface 4.10.3 ⟩
  }
#endif
```

This code is used in section 9.

[37] They don't, because different sources of data change with different frequencies. We can minimize computation by treating them separately.

[38] Earlier code had specific named references to them in the **Environment**; now we use the generic **ScratchPad** facilities.

[39] Which is an argument for not passing it in at all.

[40] Originally I attempted to have **Covariates** manage all of this. However, **LinearProduct** called *isChanged( )*, updating the state of the covariates. Then when the *Coefficient* attempted the actual computation, it found the **Covariates** were unchanged—which they were not, with respect to the last computation by the *Coefficient*—and erroneously skipped recomputation.

4.10.1 AbstractCovariates

AbstractCovariates *defines the abstract interface for* **Covariates**.

If c is a particular instance of **Covariates** *and e an instance of* **Environment**, *the value returned by c.values(e) is only guaranteed good until the next call to c.values(e). Calls with different* **Environment** *instances do* not *interfere with each other. Clients that wish to preserve the return values longer should make a copy.*[41]

We may also want to cache the **Covariates** *values, but that decision is for subclasses to make.*

Typical useage:

```
ScratchData *pMemento = 0;
Double1D& v = covariates.values(environment);
covariates.memento(&pMemento);
covariates.isChanged(environment, pMemento);
```

Implementation note: it appears that memory allocation is relatively expensive, at least on some platforms[42]*, so we provide an interface to reuse existing mementos.*

⟨ *AbstractCovariates Interface 4.10.1* ⟩ ≡

```
class AbstractCovariates : public ScratchData
  {
public:
  AbstractCovariates( )
    { }
  virtual ~AbstractCovariates( )
    { }
    // return true if values in theEnv changed since memento
    // The pointer must be to an object returned by this class or 0.
  virtual bool isChanged(Environment& theEnv, ScratchData *thepMemento) = 0;
    // state as of last call to values( )
    // caller owns the return value, which may be 0.
  virtual ScratchData *memento( ) = 0;
    // state as of last call to values( )
    // If *theppMemento == 0, allocate memento and update *theppMemento.
    // Otherwise, *theppMemento must point to a memento generated by
    // the concrete class, and that memento will be updated.
    // Return value is a pointer to the memento.
  virtual ScratchData *memento(ScratchData **theppMemento) = 0;
    // return the actual covariate values
  virtual Double1D& values(Environment& theEnv) = 0;
  };
```

This code is used in section 4.10.

[41] I anticipate little need for that. The return value is a reference to avoid needless copying.
[42] Apple

4.10.2 MatrixCovariates

This class retrieves covariates from a standard two-dimensional matrix. Because the ordinary return value from **Double2D**::*col*( ) *is a temp value, we must store it here to return a reference safely.*

⟨ *MatrixCovariates Interface 4.10.2* ⟩ ≡

```
class MatrixCovariates : public AbstractCovariates
  {
public:
  MatrixCovariates(const Double2D& theData) : AbstractCovariates( ), myData(theData),
      mypMemento(0)
    { }
  virtual ~MatrixCovariates( )
    {
    delete mypMemento;
    }
  virtual bool isChanged(Environment& theEnv, ScratchData *thepMemento);
  virtual ScratchData *memento( );
  virtual ScratchData *memento(ScratchData **theppMemento);
  virtual Double1D& values(Environment& theEnv);
protected:
  virtual Double1D& rawValues(Environment& theEnv);
  struct Memento : public ScratchData
    {
      // c'tor
    Memento(const Double2D& theData, const Environment& theEnv);
      // update with current state
      // after call, object has same state as if
      // newly constructed with the same arguments
    void capture(const Double2D& theData, const Environment& theEnv);
    IObservation iObservation;
    Double1D cache;
    };
    // data
  const Double2D& myData;
  Memento *mypMemento;    // cached values are inside this
  };
```

This code is used in section 4.10.

4.10.3 Path Covariates

The "covariates" are the values of the path-dependent variables. We may not use all such variables, or use them in the original order. The values returned will actually be the values for the previous node. To see why, consider a step on which we've just jumped to a new state. When computing the time in state, for example, we need the time in the prior state. Offset variables are available (outside the scope of this class) to add a constant to the these values so they match anywhere in the interval.

The use of myValues is not thread-safe, but since all **Covariates** *are thread-specific that is not a problem.*

⟨ *PathCovariates Interface 4.10.3* ⟩ ≡

```
class PathCovariates : public AbstractCovariates
  {
public:
  PathCovariates(const TIndirect1D& theIndices) : myIndices(theIndices),
        myValues(theIndices.size( ))
    { }
  virtual ~PathCovariates( )
    { }
  virtual bool isChanged(Environment& theEnv, ScratchData *thepMemento)
    {
    return true;
    }
  virtual ScratchData *memento( )
    {
    return 0;
    }
  virtual ScratchData *memento(ScratchData **theppMemento)
    {
    return 0;
    }
  // note: no check modelData( ) and myIndices are sensible
  virtual Double1D& values(Environment& theEnv);
protected:
  const TIndirect1D myIndices;    // 0-based indices in modelData( ) to use
  Double1D myValues;    // staging area for results
  };
```

This code is used in section 4.10.

4.11 Path

A **Path** *is an ordered set of* **Node***'s, with the first being earliest. It contains a possible evolution of a case through time.*

We only manipulate **Path***'s by adding and removing things from the end, so a* **vector** *is an ideal implementation.*

Because an ordering relation is defined on the **Node***'s that make up the* **Path**, **Path***'s can be sorted, put in Set's, and other such operations. That is helpful for testing; in particular* **TestRecorder** *exploits this fact.*

When a node is pushed onto the path, its pointer to the previous node is set appropriately.

The semantics of copy construction/assignment/cloning are to make a new set of nodes, with the same values and relations as the old path. In particular, the pointers to the previous nodes will be reset.

Since the **Node***'s are allocated and owned by my* **NodeFactory**, *I use a special view_clone_allocator to tell the* **ptr_vector** *that it doesn't own the associated objects, and shouldn't delete them when this object is destroyed. Deleting the* **NodeFactory** *will have that effect.*

⟨ *Path.h 4.11* ⟩ ≡

```
@o Path.h
#ifndef Path_h
#define Path_h

#include <iostream>

#include <boost/ptr_container/ptr_vector.hpp>
#include <memory>
#include "basic.h"
#include "Node.h"
#include "NodeFactory.h"

  namespace mspath {

  class Path : public boost:: ptr_vector⟨Node, boost::view_clone_allocator⟩ {

public:
  Path(size_t nPathVars = 0U) : mypNF(new NodeFactory⟨Node⟩(nPathVars))
    {}

  Path(const Path& p);
  Path& operator =(const Path& p);

  Path *clone( ) const
    { return new Path(*this); }

  Node *pathPush(const StatePoint& theSP, const TimePoint& theTP);
  void pathPop( );

    // you must use a Path pointer to get this method

  void clear( )
    {   // base class method will produce double frees
      // and not honor my memory management
    while(¬empty( ))
```

```
        pathPop( );
      }
  protected:    // as in our naughty base class, push_back is not virtual
      // The argument to the next function must be consistent
      // with the other Node's in the Path. In particular,
      // it's ModelData must have the same size.
    void push_back(Node *);
    Path& innerAssign(const Path& p);
    typedef boost:: ptr_vector<Node, boost::view_clone_allocator> Super;
    std::auto_ptr<NodeFactory<Node>>mypNF;    // I own this. It owns the Node's.
    } ;
    std::ostream & operator <<(std::ostream & s, const Path& p);
    }    // close namespace mspath
  #endif    // Path_h
```

This code is used in section 9.

4.12 NodeFactory

This class manages the creation and destruction of **Node***'s. It has several purposes.*

First, it encapsulates all the information that is specific to the type of **Model** *(***ModelData***) and evaluation strategy (***EvaluationData***).*

Second, it permits optimizations, such as the reuse of deleted **Node***'s. This can save the overhead of object creation. Under gcc 3.3 on Apple OS X 10.3, recycling more than* trebles *the run-time of the canonical profiling case (from 1.38 seconds to .44, net of the overhead of monitoring).*[43]

In this implementation, pointers to **Node***'s are stored in a vector. The size of that vector will be the length of the longest path encountered so far. myNext tracks the current position of the next node to be handed out.*

The assumed usage pattern is that destruction happens in reverse order from creation.

The exact type of **Node** *is a paramter; this is mostly for testing at the moment.*

If you change this class, check whether **NodeFactoryTester** *needs updating as well.*

There are some assertions here that I might want to disable or turn into exceptions eventually.

The name of the template argument below is **Node***, which is also the name of a real class in the system. This is apparently OK as long as the code inside the template does not need access to the "real"* **Node** *except through the parameter. And it works if the instantiation is* **NodeFactory**⟨**Node**⟩*, which is the standard case.*[44]

⟨ *NodeFactory.h 4.12* ⟩ ≡

```
@o NodeFactory.h
#ifndef NodeFactory_h
#define NodeFactory_h 1
#include "basic.h"
#include "Node.h"
#include <vector>
#include <cassert>
  namespace mspath {
  class TimePoint;   // only forward decl is required
  class Path;   // forward decl for friend
  class NodeFactoryTester;   // forward decl for friend
  template ⟨class Node⟩
  class NodeFactory
    { public:
        ⟨ NodeFactory Constructors 4.12.0.1 ⟩
        ⟨ NodeFactory Node Creation 4.12.0.2 ⟩
        ⟨ NodeFactory Node Destruction 4.12.0.3 ⟩
    protected:
        ⟨ NodeFactory friends 4.12.0.4 ⟩
        ⟨ NodeFactory Data 4.12.0.5 ⟩} ; }   // end namespace mspath
  #endif
```

This code is used in section 9.

[43] However, this change was coincident with the discovery I wasn't properly deleting old **Node**'s. Some of the speedup undoubtedly is from that. That fix should also reduce total memory use.

[44] It might have been clearer to use different names; the current setup arises from history. Originally, **NodeFactory** was a regular class that used **Node**. When I changed **NodeFactory** into a class template I wanted to avoid rewriting all of its code, and so called the template parameter **Node**.

In principle, this class constructor might take a **Model** *and* **Evaluator** *as arguments, extracting the relevant information. Since they are higher-level concepts, I leave it to clients to pull out the necessary information.*

⟨ *NodeFactory Constructors 4.12.0.1* ⟩ ≡

```
NodeFactory(size_t nPath = 0U) : myNPath(nPath), myNext(0)
  { }
  // next is mostly for Path which uses me
NodeFactory *clone( ) const
  {
  return new NodeFactory⟨Node⟩ (myNPath);
  }
~NodeFactory( )
  {
  for (size_t i = 0U; i < myNodeps.size( ); i++)
    delete myNodeps[i];
  }
```

This code is used in section 4.12.

All clients that want a **Node** *should use this interface. The client is responsible for the returned object, and should call destroyNode( ) (*not delete*) when it's done.*

In retrospect, I probably should have used placement new instead of recreate below, and abolished the latter method.

⟨ *NodeFactory Node Creation 4.12.0.2* ⟩ ≡

```
Node *createNode(const StatePoint& theSP, const TimePoint& theTP)
  {
  Node *pNode;
  if (myNext + 1 ≤ myNodeps.size( ))
    {
    pNode = myNodeps[myNext];
    pNode→recreate(theSP, theTP);
    }
  else
    {
    assert(myNext ≡ myNodeps.size( ));
    pNode = new Node(theSP, theTP, myNPath);
    myNodeps.push_back(pNode);
    }
  myNext++;
  return pNode;
  }

Node *createNode(const Node& theNode)
  {
  Node *pNode;
  if (myNext + 1 ≤ myNodeps.size( ))
    {
    pNode = myNodeps[myNext];
    (*pNode) = theNode;
    }
  else
    {
    assert(myNext ≡ myNodeps.size( ));
    pNode = new Node(theNode);
    myNodeps.push_back(pNode);
    }
  myNext++;
  return pNode;
  }
```

This code is used in section 4.12.

⟨ *NodeFactory Node Destruction 4.12.0.3* ⟩ ≡

```
void destroyNode(Node *thepNode)
  {
  assert(myNext > 0);
  myNext−−;
  assert(thepNode ≡ myNodeps[myNext]);
  }
```

This code is used in section 4.12.

This class basically exists as a client for **Path**, *which needs some other kinds of access. The method defined here forgets which* **Node***'s have been handed out, reverting the factory to a state almost like new. The difference is that some of the necessary objects are already allocated. The corresponding* **Path** *needs to be reset too.*

⟨ *NodeFactory friends 4.12.0.4* ⟩ ≡

```
friend class Path;
friend class NodeFactoryTester;

void reset( )
  {
  myNext = 0U;
  }
```

This code is used in section 4.12.

Since creation and destruction should be from the end, I use a **std**:: **vector** *. myNext is the index from which the next pointer should come; it is* not *guaranteed the vector has that many valid elements. This slightly odd definition enables valid handling in the initial, empty state.*

⟨ *NodeFactory Data 4.12.0.5* ⟩ ≡

```
size_t myNPath;    // number of path-dependent variables
std:: vector < Node *⟩ myNodeps;    // Node pointers
size_t myNext;    // index where next pointer should come from
```

This code is used in section 4.12.

4.13 Node

A **Node** *is a single point on a path of possible moves through the state space. It includes the state, time, associated data, and a pointer to the previous* **Node**. *It also stores the probability or log-probability of the path so far as a service to the* **Evaluator**. *Future* **Evaluator***'s may evaluate several sets of parameters at once; in that case,* **Node** *will store all the accompanying probabilities (one for each set of parameters).* **Node** *also stores supplemental data for the* **Model**.

Because of our approach, **Node***'s will not know about their successors, but will know about their preceding* **Node**.[45] *We use a raw pointer, not an iterator, for the previous node location. Why? First, some iterators, in particular the ones we are using, can not represent a null value. We need a null value for the "previous" node of the base node in a path. Second, the iterators, but not the pointers, become invalid when the collection resizes (if we were holding the nodes themselves in the path the location of the nodes would change and we'd need to clean up when it did; however, since the path just holds pointers that's not an issue). Third, raw pointers are perhaps a hair simpler and faster.*

For similar reasons, the **Node***'s* **TimePoint** *is referenced directly.*[46] *This class requires the* **TimePoint** *passed as an argument to continue to exist for the duration of* **Node***'s existence. In general that might be a problem; it is not for this program.*

Comparing paths requires comparing nodes, so the necessary operators are defined here. Since we may consider paths equal even if they are not object-identical, we do not compare the pointers to the previous node. We also ignore alreadyCountedasGood for equality comparison.

⟨ *Node.h 4.13* ⟩ ≡
```
@o Node.h
#ifndef Node_h
#define Node_h 1

#include <ostream>

#include "basic.h"
#include "TimePoint.h"

  namespace mspath
    {


    class Path;
    class NodeFactoryTester;

    class Node
      {

    public:
      ⟨ Node Typedefs 4.13.0.1 ⟩
      ⟨ Node Constructors 4.13.0.2 ⟩
      ⟨ Node Accessors 4.13.0.3 ⟩
      ⟨ Node Tests 4.13.0.4 ⟩
```

[45] We dynamically generate paths, so there is never a point when all successor nodes are present if more than one successor is possible.

[46] During path generation, I do want to get the following time point from the current one, so using an iterator or an integer index would ease that. But it's easy enough to get the necessary information elsewhere during path generation; this approach keeps this class relatively simple. If I stored an index integer to **TimeSteps** the dependencies would be even simpler, but the actual use of this class would be more awkward.

```
    ⟨ Node Actions 4.13.0.6 ⟩
    ⟨ Node Operators 4.13.0.7 ⟩
    friend class Path;
    friend class NodeFactoryTester;

  protected:
    ⟨ Node Friend Accessors 4.13.0.8 ⟩
    ⟨ Node Data 4.13.0.5 ⟩

    };

  }   // namespace mspath
#endif   // Node_h
```

This code is used in section 9.

Since **Node** *stores information for* **Evaluator** *and* **Model**, *the relevant definitions come from those classes. We put them in basic.h to avoid complex dependencies.*

The **Evaluator** *knows whether it is evaluating one or multiple sets of parameters, whether it is using likelihoods or log-likelihoods, and possibly other information.*

Model *knows what elements of path-dependence need to be computed and made available for the main* **Evaluator**: *time in current state, time in previous state, highest state, average time in state, ... Note that even if there are multiple parameter sets, these data will be the same for all of them.*

⟨ *Node Typedefs 4.13.0.1* ⟩ ≡

```
  typedef Node TNode;
  typedef ModelData TModelData;
  typedef EvaluationData TEvaluationData;
```

This code is used in section 4.13.

Essentially, you must have all the basic info for the **Node** *before constructing it. Clients should* not *depend on the model or evaluation data having any particular values (e.g., 0).*[47]

Again, remember that the **TimePoint** *argument must continue to exist for the length of life of the* **Node**, *since the latter refers to the former, but does not own it.*

Very knowledgeable clients may use the second method to make a "new" object by recycling an old one. It assumes the dimensions of the model and evaluation data don't change, and it does not *reset those values.*

Knowledgeable clients may also change the state of the **Node**, *but doing so is very dangerous because it may invalidate values computed from the current state, both within the* **Node** *and outside of it. This feature is only used by simulation.*

The implementation cheats a bit by initializing myED, whose type should be opaque. The real responsibility for setting it belongs to **EvaluatorRecorder**. *Setting a value here avoids some uninitialized memory warnings.*

⟨ *Node Constructors 4.13.0.2* ⟩ ≡

```
Node(const StatePoint& sp, const TimePoint& tp, size_t theNModelData = 0U) : mySP(sp),
        myMD(theNModelData), myED(0), myPriorp(0), mypTP(&tp), mySeenGood(false)
  { }
Node& recreate(const StatePoint& sp, const TimePoint& tp)
  {
  mySP = sp;
  mypTP = &tp;
  myPriorp = 0;
  mySeenGood = false;
  return *this;
  }
void setState(State s)
  {
  mySP.setState(s);
  }
```

This code is used in section 4.13.

[47] A review of the code on 9 March 2006 shows they do not.

Node *is pretty dumb in this implementation, being basically a holder for some data. Note I do not use virtual accessors, for speed and simplicity.*

I got bored with **const** *correctness, and so have not implemented every variation of accessor I might want.*

⟨ *Node Accessors 4.13.0.3* ⟩ ≡

```
// accessors
const StatePoint& statePoint( ) const
  { return mySP; }
const Time time( ) const
  { return timePoint( ).time( ); }
const State& state( ) const
  { return mySP.state( ); }
StatePoint& statePoint( )
  { return mySP; }
Time time( )
  { return timePoint( ).time( ); }
State state( )
  { return mySP.state( ); }
bool alreadyCountedAsGood( ) const
  { return mySeenGood; }
TNode * previous( ) const
  { return myPriorp; }    // may be NULL
const TimePoint& timePoint( ) const
  { return *mypTP; }
ModelData& modelData( )
  { return myMD; }

EvaluationData& evaluationData( )
  { return myED; }
// We do not have probability because the Node may manage several.
// The evaluationData holds that.
```

This code is used in section 4.13.

⟨ *Node Tests 4.13.0.4* ⟩ ≡

```
bool isRoot( )
  {
  return myPriorp ≡ 0;
  }
```

This code is used in section 4.13.

A lot of the data are just holders to be managed by other classes. At some point we may want to give them a crack at initializing the data, but at the moment that doesn't seem necessary.

⟨ *Node Data 4.13.0.5* ⟩ ≡
```
StatePoint mySP;

ModelData myMD;

EvaluationData myED;

TNode *myPriorp;

const TimePoint *mypTP;    // reference, not ownership
bool mySeenGood;    // true if already counted as good
```

This code is used in section 4.13.

Actions upon or by the **Node**.

⟨ *Node Actions 4.13.0.6* ⟩ ≡
```
void countAsGood( )
  { mySeenGood = true; }
```

This code is used in section 4.13.

The following are all ultimately in support of testing.

⟨ *Node Operators 4.13.0.7* ⟩ ≡
```
friend bool operator <(const Node& lhs, const Node& rhs)
  { if (lhs.statePoint( ) < rhs.statePoint( ))
    return true; if (¬(lhs.statePoint( ) ≡ rhs.statePoint( )))
    return false; return lhs.timePoint( ) < rhs.timePoint( ); }

friend bool operator ≡(const Node& lhs, const Node& rhs)
  { if (¬(lhs.statePoint( ) ≡ rhs.statePoint( )))
    return false; return lhs.timePoint( ) ≡ rhs.timePoint( ); }

friend std::ostream & operator <<(std::ostream & ostr, const Node& n)
  { ostr << "Node(" << n.state( ) << ", " << n.time( ); if (¬n.alreadyCountedAsGood( ))
    ostr << " not"; ostr << " already counted good, md " << n.myMD << ",\
        ed " << n.myED << ")"; return ostr; }
```

This code is used in section 4.13.

These allow **Path** *to manipulate* **Node***'s as it gets them.*

⟨ *Node Friend Accessors 4.13.0.8* ⟩ ≡

```
void setNoPrevious( )
  { myPriorp = static_cast⟨Node *⟩ (0); }

void setPrevious(Node *const p)
  { myPriorp = p; }
```

This code is used in section 4.13.

4.14 Computing Path-Dependent History

HistoryComputer *is the abstract base class for all objects that can compute path-dependent information for a* **Node**. *They are a part of the* **Model** *and, like it, are not tied to any thread.*

Each concrete class specializes in a particular kind of computation; the computations are based either on the previous values in the path, or the values of other path-dependent variables (e.g., taking log *or squaring a value).*

These classes are involved in the following phases of the program, in order:

Initial Creation *Create all the possible concrete* **HistoryComputer** *objects.*

Equation Parsing *Identify which terms of the equation correspond to which* **HistoryComputer***'s.*

Setup *Identify the number, kind, and sequencing of variables we actually must compute. Note that some variables maybe of internal interest only (e.g., if the user wants to analyze* log *of time in state, he or she doesn't care about time in state but we still must compute it).*

Tree Evaluation *Actually evaluate specific nodes.* Only ask for the evaluation of a **Node** if all previous entries in the **Path** have been evaluated.

Cleanup *Need to be clear on ownership issues.*

To understand some of the instance variables requires understanding how **HistoryComputer***'s are used, and some subtleties of the computation.*

Suppose the user requests computations using `"LN(TSO)"`*. Internally, this is represented by a* **HistoryComputer** *that does a log transformation of the value produced by another* **HistoryComputer***, namely the one for* `"TSO"`*. Further, the value for the latter must be computed before computing the former. This implies that the number and sequence of path-dependent variables computed internally may differ from the original client request. The previous example illustrates a case where an unrequested computer, for* `"TSO"`*, must be used internally. If the request had been* `"LN(TSO)"`*,* `"TSO"` *the internal and external variable lists would have matched, but the internal order of computation would need to put* `"TSO"` *first.*

For a full view of **HistoryComputer** *in action, consult the* **ModelBuilder** *code that constructs the history computers and* **PathCovariates** *that uses the results of* **HistoryComputer**::*evaluate*( ).

To understand the instance variables, you need to understand two different lists. The first is the list of path-dependent variables requested by the user. The **HistoryComputer** *that matches the i'th term requested will have myCovariateIndex = i and myIsCovariate =* `true`*. The second list gives the variables we need to compute, which may include additional terms, and the order in which they need to be computed. That is also the order of entries in a* **Node***'s* **ModelData***. Any* **HistoryComputer** *in this list will have isRequred =* `true` *(this includes all those with myIsCovariate ≡* `true`*), and myDataIndex giving the position in* **ModelData**.[48]

"covariate" here refers to the fact that coefficients may be multiplied by these terms. They are not *conventional data covariates, because they are not observed in the dataset but computed from the particular path taken in the evolution of the process.*

⟨ *HistoryComputer.h 4.14* ⟩ ≡
```
@o HistoryComputer.h
#ifndef HistoryComputer_h
#define HistoryComputer_h 1

#include "basic.h"
#include "MSPathError.h"

#include <iostream>     // not just forward reference, because of inline def
#include <stdexcept>
#include <string>

  namespace mspath { class Environment;
  class Node;

  class HistoryComputer
    { public:
        ⟨ HistoryComputer c'tors 4.14.0.1 ⟩
        ⟨ HistoryComputer Parsing 4.14.0.2 ⟩
        ⟨ HistoryComputer Setup Post-Parsing 4.14.0.3 ⟩
        ⟨ HistoryComputer printing 4.14.0.5 ⟩


      // EVALUATION

      virtual void evaluate(Environment& theEnv, Node& theNode) throw (std::out_of_range) = 0;

      protected:
        ⟨ HistoryComputer Data 4.14.0.4 ⟩
```

[48] It may seem silly to have a member of a list include its position in the list. **HistoryComputer**'s need that information to know where to read and write in **ModelData**.

```
    } ; }
  #endif
```

This code is used in section 9.

⟨ *HistoryComputer c'tors 4.14.0.1* ⟩ ≡

```
  HistoryComputer( ) : myIsRequired(false), myIsCovariate(false), myCovariateIndex(0),
      myDataIndex(0)
    { }
  virtual ~HistoryComputer( )
    { }
```

This code is used in section 4.14.

One method of constructing **HistoryComputer***'s is to parse them from a user-provided string. The following methods support this approach.*

⟨ *HistoryComputer Parsing 4.14.0.2* ⟩ ≡

```
// PARSING

// canonical representation of function
virtual const std::string & name( )const  = 0;

// true if input matches this computer
virtual bool matches(const std::string & term)const  = 0;

// SETUP WHILE PARSING

// model requires this computer

virtual void makeRequired( )
  {
  myIsRequired = true;
  }

// user specified this term at indicated index
virtual void makeCovariate(size_t index) throw (DuplicateTerm)
  {
  if (isCovariate( ))
    throw DuplicateTerm( );
  makeRequired( );
  myIsCovariate = true;
  myCovariateIndex = index;
  }

// corresponding queries

// isCovariate( ) ≡ true implies isRequired( ) ≡ true, but not
// the reverse

bool isRequired( ) const
  {
  return myIsRequired;
  }

bool isCovariate( ) const
  {
  return myIsCovariate;
  }

size_t covariateIndex( ) const
  {
  return myCovariateIndex;
  }

// return pointer to HistoryComputer I depend on, if any
// 0 except for Composites

virtual HistoryComputer *requires( ) const
  {
  return 0;
  }
```

This code is used in section 4.14.

Results of the computation go at a particular index in a data vector (the **ModelData** *of the* **Node***). This is not necessarily the same order as given by the covariateIndex. covariateIndex concerns the order in which values must be computed, while dataIndex gives the order in which they are used. As a convenience to the client (and ourselves) we want them in the order in which the terms were originally given, with intermediate variables placed after all the ones the client cares about.*

⟨ *HistoryComputer Setup Post-Parsing 4.14.0.3* ⟩ ≡

```
// where is my variable in the model data?
void setDataIndex(size_t index)
  {
  myDataIndex = index;
  }
size_t dataIndex( ) const
  {
  return myDataIndex;
  }
```

This code is used in section 4.14.

If myIsCovariate ≡ `false` *then myCovariateIndex is not relevant.*

⟨ *HistoryComputer Data 4.14.0.4* ⟩ ≡

```
bool myIsRequired;    // true if I must be computed
bool myIsCovariate;   // true if I match a model term
size_t myCovariateIndex;   // index of that term
size_t myDataIndex;   // where to store my values
```

This code is used in section 4.14.

⟨ *HistoryComputer printing 4.14.0.5* ⟩ ≡

```
friend std::ostream & operator <<(std::ostream & ostr, const HistoryComputer& history)
  {
  ostr << history.name( ) << " saved at index " << history.dataIndex( );
  return ostr;
  }
```

This code is used in section 4.14.

4.14.1 Path-Dependent History Primitives

The primitives of history computation make computations based on the path to this point. We provide an abstract class and some particular implementations.

The initial time (at the start of the process, on entry to a state) is ordinarily 0. However, a 0 value may produce domain errors or excessive weighting on a particular observation. So we allow specification of an arbitrary value for this first point. See the discussion under §2.2.6, page 20 for more about this.

⟨ *PrimitiveHistoryComputer.h 4.14.1* ⟩ ≡
```
@o PrimitiveHistoryComputer.h
#ifndef PrimitiveHistoryComputer_h
#define PrimitiveHistoryComputer_h 1

#include "HistoryComputer.h"

namespace mspath {
    ⟨ PrimitiveHistoryComputer Interface 4.14.1.1 ⟩
    ⟨ TimeInStateComputer Interface 4.14.1.2 ⟩
    ⟨ TimeInPreviousStatesComputer Interface 4.14.1.3 ⟩
    ⟨ TimeSinceOriginComputer Interface 4.14.1.4 ⟩


}
#endif
```

This code is used in section 9.

The next class is still abstract.

⟨ *PrimitiveHistoryComputer Interface 4.14.1.1* ⟩ ≡

```
class PrimitiveHistoryComputer : public HistoryComputer
  {
public:
  PrimitiveHistoryComputer(const std::string & name, double theInitialTime = 0.0) :
        HistoryComputer( ), myName(name), myInitialTime(theInitialTime)
    { }
  virtual ~PrimitiveHistoryComputer( )
    { }
  // primitive naming
  // canonical representation of function
  virtual const std::string & name( ) const
    {
    return myName;
    }
  // true if input matches this computer
  virtual bool matches(const std::string & term) const
    {
    return term ≡ myName;
    }
protected:
  const std::string myName;
  double myInitialTime;
  };
```

This code is used in section 4.14.1.

⟨ *TimeInStateComputer Interface 4.14.1.2* ⟩ ≡

```
class TimeInStateComputer : public PrimitiveHistoryComputer
  {
public:
  TimeInStateComputer(const std::string & theName, double theInitialTime = 0.0) :
        PrimitiveHistoryComputer(theName, theInitialTime)
    { }
  virtual ~TimeInStateComputer( )
    { }
  virtual void evaluate(Environment& theEnv, Node& theNode) throw (std::out_of_range); } ;
```

This code is used in section 4.14.1.

⟨ *TimeInPreviousStatesComputer Interface 4.14.1.3* ⟩ ≡

```
class TimeInPreviousStatesComputer : public PrimitiveHistoryComputer
  {
public:
  TimeInPreviousStatesComputer(const std::string & theName, double theInitialTime = 0.0)
        : PrimitiveHistoryComputer(theName, theInitialTime)
    { }
  virtual ~TimeInPreviousStatesComputer( )
    { }
  virtual void evaluate(Environment& theEnv, Node& theNode) throw (std::out_of_range); } ;
```

This code is used in section 4.14.1.

⟨ *TimeSinceOriginComputer Interface 4.14.1.4* ⟩ ≡

```
class TimeSinceOriginComputer : public PrimitiveHistoryComputer
  {
public:
  TimeSinceOriginComputer(const std::string & theName, double theInitialTime = 0.0) :
        PrimitiveHistoryComputer(theName, theInitialTime)
    { }
  virtual ~TimeSinceOriginComputer( )
    { }
  virtual void evaluate(Environment& theEnv, Node& theNode) throw (std::out_of_range);
  } ;
```

This code is used in section 4.14.1.

4.14.2 Composite Operators on Path-Dependent History

The next classes do simple functional transformations of values of other **HistoryComputer***'s.*

⟨ *CompositeHistoryComputer.h 4.14.2* ⟩ ≡

```
@o CompositeHistoryComputer.h
#ifndef CompositeHistoryComputer_h
#define CompositeHistoryComputer_h 1
#include "HistoryComputer.h"
namespace mspath {
    ⟨ CompositeHistoryComputer Interface 4.14.2.1 ⟩
    ⟨ LnHistoryComputer Interface 4.14.2.2 ⟩
}
```

This code is used in section 9.

Abstract class.

⟨ *CompositeHistoryComputer Interface 4.14.2.1* ⟩ ≡

```
class CompositeHistoryComputer : public HistoryComputer
  {
public:
  CompositeHistoryComputer(const std::string & theFunctionName,
        HistoryComputer& theTarget) : HistoryComputer( ),
        myName(theFunctionName + "(" + theTarget.name( ) + ")"), myTarget(theTarget)
    { }
  virtual ~CompositeHistoryComputer( )
    { }
  // primitive naming
  // canonical representation of function
  virtual const std::string & name( ) const
    {
    return myName;
    }
  // true if input matches this computer
  virtual bool matches(const std::string & term) const
    {
    return term ≡ myName;
    }
  // model requires this computer
  virtual void makeRequired( )
    {
    HistoryComputer::makeRequired( );
    target( ).makeRequired( );
    }
  HistoryComputer& target( )
    {
    return myTarget;
    }
  virtual HistoryComputer *requires( ) const
    {
    return &myTarget;
    }
protected:
  const std::string myName;
  HistoryComputer& myTarget;    // reference, not ownership
  };
```

This code is used in section 4.14.2.

Our only concrete transformation at the moment is taking the natural logarithm.

⟨ *LnHistoryComputer Interface 4.14.2.2* ⟩ ≡

```
class LnHistoryComputer : public CompositeHistoryComputer
  {
public:
  LnHistoryComputer(const std::string & theFunctionName, HistoryComputer& theTarget) :
        CompositeHistoryComputer(theFunctionName, theTarget)
    { }
  virtual ~LnHistoryComputer( )
    { }
  virtual void evaluate(Environment& theEnv, Node& theNode) throw (std::out_of_range); } ;
#endif
```

This code is used in section 4.14.2.

4.15 TimeSteps Generators

All these classes generate **TimeSteps** *for later analysis, based on the observed data and desired spacing parameters. They differ in exactly how they space the grid and associate it with observed data.*

Most clients should use only the abstract interface. It assumes the data in the environment have been properly initialized to refer to a particular case, and it generates appropriate **TimeSteps** *in the* **Environment**.

⟨ *AbstractTimeStepsGenerator.h 4.15* ⟩ ≡

```
  @o AbstractTimeStepsGenerator.h
#ifndef AbstractTimeStepsGenerator_h
#define AbstractTimeStepsGenerator_h 1

#include "basic.h"
#include "Environment.h"
#include "TimePoint.h"

  namespace mspath
    {

    class AbstractTimeStepsGenerator
      {

    public:
      virtual void makeStepsFor(Environment *pEnvironment) = 0;

      virtual ~AbstractTimeStepsGenerator( )
        { }
      ;

    protected:    // for use by subclasses, who probably won't need to override
      virtual void makeStep(Time t, bool trueObs, TimePoint::TIObservation i,
          Environment *pEnv);
      };

    }    // end namespace mspath
#endif
```

This code is used in section 9.

4.15.1 FixedTimeStepsGenerator

This class generates a uniform, fixed-step size grid for all cases. It rounds true observation times to match the grid, and associates the latest possible observation with each **TimePoint**. *Note that the* **R** *code has more elaborate handling of duplicates, and should hand the C code data without duplicates. If two consecutive observations land on the same rounded time, makeStepsFor throws* **FixedTimeStepsGenerator**::**ExtraDataError**.

To avoid funny rounding errors if the desired step size is, for example, 1/3, *this generator represents its step size as a rational number. It relies on the* **Boost** *library for implementation of rational numbers.*

⟨ *FixedTimeStepsGenerator.h 4.15.1* ⟩ ≡
```
@o FixedTimeStepsGenerator.h
#ifndef FixedTimeStepsGenerator_h
#define FixedTimeStepsGenerator_h 1

#include <boost/rational.hpp>
#include <stdexcept>

#include "AbstractTimeStepsGenerator.h"

namespace mspath { class FixedTimeStepsGenerator : public AbstractTimeStepsGenerator
  { public:    // types
  typedef Data::TIObs TIObs; typedef std:: vector ⟨TIObs⟩ TDuplicates;
  typedef int TInt; typedef boost:: rational⟨TInt⟩ TRational;

    // c'tor

  FixedTimeStepsGenerator (TInt stepNumerator = 1, TInt stepDenominator = 1) :
        myStepSize(TRational(stepNumerator, stepDenominator)),
        myStepAsDouble(boost::rational_cast⟨double⟩(myStepSize))
    { }

  virtual ~FixedTimeStepsGenerator( )
    { }
  ;

    // using data in environment, fill in
    // timepoints for environment.
    // May throw ExtraDataError.
  virtual void makeStepsFor(Environment *pEnvironment);

    // error class
  class ExtraDataError : public std ::invalid_argument
    {

  public:
    ExtraDataError (const std::string & msg) : std::invalid_argument (msg)
      { }

    ;
    };

protected:
  TRational myStepSize;
  double myStepAsDouble;    // same as preceding

    // maps times to integers using stepSize
  TInt integerTime(Time t);    // and the reverse
  Time timeFromInteger(TInt i);
```

```
    } ;   // end class FixedTimeStepsGenerator
    }   // end namespace mspath
#endif   // FixedTimeStepsGenerator_h
```

This code is used in section 9.

4.15.2 CompressedTimeStepsGenerator

This class generates a uniform, fixed-step size grid for all cases. It rounds true observation times to match the grid. If more than one observation matches a single rounded time, it uses the first observation for the first step and the last observation for all later steps.

This class does not throw exceptions for duplicate data.

⟨ *CompressedTimeStepsGenerator.h 4.15.2* ⟩ ≡

```
@o CompressedTimeStepsGenerator.h
#ifndef CompressedTimeStepsGenerator_h
#define CompressedTimeStepsGenerator_h 1

#include <vector>

#include "FixedTimeStepsGenerator.h"

namespace mspath { class CompressedTimeStepsGenerator : public
      FixedTimeStepsGenerator
  { public:   // types
  typedef std:: vector ⟨TIObs⟩ TDuplicates;

    // c'tor

  CompressedTimeStepsGenerator (TInt stepNumerator = 1, TInt stepDenominator = 1) :
        FixedTimeStepsGenerator(stepNumerator, stepDenominator)
    { }
  ;

  virtual ~CompressedTimeStepsGenerator( )
    { }
  ;

    // using data in environment, fill in
    // timepoints for environment
  virtual void makeStepsFor(Environment *pEnvironment);

  virtual void clear( )
    {
    myDuplicates.clear( );
    }

    // indices of observations that have same rounded time as
    // previous observation

  virtual const TDuplicates& duplicates( ) const
    {
    return myDuplicates;
    }

protected:
  TDuplicates myDuplicates;

  ⟨ CompressedTimeStepsGenerator helpers 4.15.2.1 ⟩} ;
      // end class CompressedTimeStepsGenerator
  }   // end namespace mspath
#endif   // CompressedTimeStepsGenerator_h
```

This code is used in section 9.

The following methods and data support the operation of makeStepsFor, and are only for its use. See the code section for more information.

⟨ *CompressedTimeStepsGenerator helpers 4.15.2.1* ⟩ ≡

```
TIObs myoblast, myobnext, myobend;
TInt myitlast, myitnext;

void lastObs(const Data&);
```

This code is used in section 4.15.2.

4.15.3 TimeStepsGenerator

This generates **TimePoint***'s that have all the observed times and any times necessary to ensure that the largest step size is myStepSize. It does this by dividing the interval between each two observations into the smallest number of equal-sized steps such that the step size is under myStepSize. Note several implications of this: first, the step size is not necessarily uniform across an individual's entire history; second, steps may be closer than myStepSize.*

⟨ *TimeStepsGenerator.h 4.15.3* ⟩ ≡

```
@o TimeStepsGenerator.h
#ifndef TimeStepsGenerator_h
#define TimeStepsGenerator_h 1

#include "AbstractTimeStepsGenerator.h"

  namespace mspath
    {
    class TimeStepsGenerator : public AbstractTimeStepsGenerator
      {

    public:   // c'tor
      TimeStepsGenerator(double stepSize) : myStepSize(stepSize)
        {}

      ;

        // using data in environment, fill in
        // timepoints for environment
      virtual void makeStepsFor(Environment *pEnvironment);

    protected:
      double myStepSize;
      };   // end class TimeStepsGenerator
    }   // end namespace mspath
#endif   // TimeStepsGenerator_h
```

This code is used in section 9.

4.16 TimeSteps

Conceptually, **TimeSteps** *are containers of* **TimePoint***'s. However, we implement this as a container of pointers to* **TimePoint***. This is an implementation detail. front*, *back*, ∗**iterator** *all yield* **TimePoint**&*. However, when adding, it's best to add a pointer.*

⟨ *TimeSteps.h 4.16* ⟩ ≡
```
@o TimeSteps.h
#ifndef TimeSteps_h
#define TimeSteps_h 1

#include <boost/ptr_container/ptr_vector.hpp>
#include "basic.h"
#include "TimePoint.h"

  namespace mspath {

  class TimeSteps : public boost:: ptr_vector⟨TimePoint⟩
  {

public:
  typedef TimePoint TTimePoint;
  typedef iterator TiTimePoint;

  static const TimePoint& timePoint(const TiTimePoint& i)
    {return ∗i; }

  }
  ; }

#endif  // TimeSteps_h
```

This code is used in section 9.

4.17 TimePoint

TimePoint*'s reflect the structure of the problem and the data. A* **TimePoint** *is a time we have chosen for analysis, and includes pointers to associated observational data. It also declares whether the observation matches the* **TimePoint** *or comes from earlier. This "matching" refers to whether the time of the* **TimePoint** *matches the time of an observation. A match in this sense does not necessarily mean that state was observed at that time. It does imply there is a new set of covariates observed.*

To determine if the state was observed, check the state of the corresponding data.

IObservation *is an iterator to an observation; dereference it through the environment.*

TimePoint *may also store other information that seems natural to associate with it. Currently, there is no such information.*

⟨ *TimePoint.h 4.17* ⟩ ≡
```
@o TimePoint.h
#ifndef TimePoint_h
#define TimePoint_h 1

#include "basic.h"
#include <ostream>

namespace mspath
  {
  class TimePoint
    {

  public:
    ⟨ TimePoint Typedefs 4.17.0.1 ⟩
    ⟨ TimePoint Constructors 4.17.0.2 ⟩
    ⟨ TimePoint Accessors 4.17.0.3 ⟩
    ⟨ TimePoint binary ops 4.17.0.5 ⟩

  protected:
    ⟨ TimePoint data 4.17.0.4 ⟩
    };
  ⟨ TimePoint streaming 4.17.0.6 ⟩

  }
#endif  // TimePoint_h
```

This code is used in section 9.

⟨ *TimePoint Typedefs 4.17.0.1* ⟩ ≡
```
typedef IObservation TIObservation;
```

This code is used in section 4.17.

Since I may want arrays of these, I need a default constructor. My explicit declaration of other constructors implies I must explicitly declare the default constructor. I still get the default copy constructor, which is fine.

⟨ *TimePoint Constructors 4.17.0.2* ⟩ ≡

```
TimePoint( )
  { }

TimePoint(Time t, bool isObservationTime, const IObservation& iobs) :
      myTime(t), myMatchObservation(isObservationTime), myIObservation(iobs)
  { }
```

This code is used in section 4.17.

⟨ *TimePoint Accessors 4.17.0.3* ⟩ ≡

```
Time time( ) const
  {return myTime;}

bool matchesObservation( ) const
  {return myMatchObservation;}

IObservation iObservation( ) const
  {return myIObservation;}
```

This code is used in section 4.17.

If the time (myTime) is considered to match an actual observation time, myMatchObservation will be true, otherwise false. Even if it is false, we may still point to the closest relevant observation—presumably the immediately preceding one. Even if myMatchObservation is true, the observation's time may not match myTime because of rounding or other, as yet unanticipated, reasons. Always consider *myTime* the true time of this **TimePoint**.

⟨ *TimePoint data 4.17.0.4* ⟩ ≡

```
Time myTime;
bool myMatchObservation;
IObservation myIObservation;
```

This code is used in section 4.17.

These operators are a bit tricky, since ≡ is based on whole object identity while < only looks at time. Also, only the bare minimum of operators needed to define sorting is defined.

⟨ *TimePoint binary ops 4.17.0.5* ⟩ ≡

```
friend bool operator <(const TimePoint& lhs, const TimePoint& rhs)
  {
  return lhs.myTime < rhs.myTime;
  }

  // not sure why I'm not getting this by default.

friend bool operator ≡(const TimePoint& lhs, const TimePoint& rhs)
  { return lhs.myTime ≡ rhs.myTime ∧ lhs.myMatchObservation ≡
      rhs.myMatchObservation ∧ lhs.myIObservation ≡ rhs.myIObservation; }
```

This code is used in section 4.17.

I added this because BOOST test suite requires it.

⟨ *TimePoint streaming 4.17.0.6* ⟩ ≡

```
std::ostream & operator <<(std::ostream & ostr, const TimePoint& tp);
```

This code is used in section 4.17.

4.18 Environment

I will first state the aspirations for this class, and then note some current limitations.

The **Environment** *holds all dynamically changing information required to evaluate the model (not just the* **Model** *class, but all the classes that relate to the theoretical model). It provides convenient access to that information, shielding clients from having to know the internals of data representation in other clients.*

It also holds a source of randomness for simulation.

Environment *contains all thread-specific information, and is associated with a single thread of execution. Each thread has exactly one* **Environment**. *Because of this,* **Environment** *should be passed as an argument to calls, rather than held as data (except for the outer level* **Manager** *that owns it).*

This object was not exactly anticipated in the original design, though it is the start of what was desribed as a "Search Strategy" object.

Manager *creates and holds onto* **Environment**.

Environment *holds the current* **Path** *and* **TimeSteps**, *and knows how to access related* **Data**. *It holds an iterator giving the current location in the data.*

PathGenerator *and* **AbstractTimeStepsGenerator***'s create and manipulate the* **Path** *and* **TimeSteps** *in the* **Environment**. *The generators and the* **Manager** *have primary responsibility for keeping* **Environment** *in a valid state.*

AbstractTimeStepsGenerator, **Manager**, **PathGenerator**, **SimpleRecorder**, **StateTimeClassifier**, *and some test classes include* **Environment** *in their headers.* **Covariates** *references* **Environment** *in its function signatures, and includes the headers in implementation. No other classes use* **Environment** *(I think–check on it).*

In the future, **Environment** *may mediate between* **Evaluator***'s, which may deal with several sets of parameters at once, and* **Model***'s, which only deal with one set. The* **Evaluator** *will set the current context in the environment, and the* **Model** *(and related classes) will access "the" parameters and results through the* **Environment**.

In addition to thread-specific information held by name, such as the **Path**, **Environment** *provides a general facility for objects to cache thread-specific information. The information itself is opaque to the environment, but any client may get and access information in a whiteboard dictionary. The keys are the addresses of the owning object; the values are* **void** *pointers to the state data.*

Thread-specific information currently lives in the following classes:

DataIterator *knows position in the current data.*

TimePoint *is not only specific to the data under analysis, but holds state information used by* **StateTimeClassifier**.
TimePoint *is in* **TimeSteps**, *and the latter is already in the* **Environment**.

Node *records the results of model evaluations.* **Node** *is in* **Path**, *which is in the* **Environment**.

Recorder *Holds results of evaluations.*

That's the theory. Practice is uglier.

In the current implementation (hmm, this may already have been fixed) **Environment** *does not know accurately what the current* **Node** *is. I should probably set that explicitly. At the moment, clients make calls with the appropriate* **Node** *as arguments. The "default"* **Node** *is the last one in the* **Path**. *This may not be the one currently being evaluated because of the way recursion works. I think I've corrected this problem.*

The encapsulation of how to get to information, and the search strategy for obtaining it, is imperfect.

Finally, quite a few classes currently hold the **Environment** *as data.*

```
⟨ Environment.h 4.18 ⟩ ≡
    @o Environment.h
  #ifndef Environment_h
  #define Environment_h 1

  #include <memory>
  #include <valarray>     // needed for simulation

  #include "basic.h"
  #include "Covariates.h"
  #include "Data.h"
  #include "MSPathError.h"
  #include "Path.h"
  #include "ScratchPad.h"
  #include "TimeSteps.h"

    namespace mspath
      {
      class Environment : public ScratchPad
        {

      public:
        ⟨ Environment Typedefs 4.18.0.1 ⟩
        ⟨ Environment Constructors 4.18.0.2 ⟩
        ⟨ Environment Accessors 4.18.0.3 ⟩
        ⟨ Environment Setters 4.18.0.7 ⟩
        ⟨ Environment Actions 4.18.0.8 ⟩
        ⟨ Environment Randomness 4.18.0.10 ⟩

      protected:
        ⟨ Environment Data 4.18.0.9 ⟩
        };
      }    // close namespace mspath
  #endif    // Environment_h
```

This code is used in section 9.

⟨ *Environment Typedefs 4.18.0.1* ⟩ ≡

```
typedef Node TNode;

typedef TimePoint TTimePoint;
```

This code is used in section 4.18.

Environment takes ownership of the path, not the data.

⟨ *Environment Constructors 4.18.0.2* ⟩ ≡

```
Environment(Data *pData, Path *pPath = new Path (static_cast⟨size_t⟩ (0U))) :
        mypPath(pPath), mypData(pData), mypDataIterator(new DataIterator(pData))
  { }
virtual ~Environment( )
  {
  delete mypPath;
  }
```

This code is used in section 4.18.

The trickiest thing below is the way the **Environment** *mediates the relation between the* **Evaluator** *and the* **Model**. *An* **Evaluator** *may deal with multiple sets of parameters, but* **Model***'s only deal with one at a time. The* **Evaluator** *notifies the* **Environment** *of the currently active set, and the* **Model** *then asks for that data.*

Currently we have only one set, so the implementation is simple. In the future, the **Environment** *might want to delegate some of this to the* **Evaluator**. *Doing so creates an unpleasant circular dependency between the definitions of* **Evaluator** *and* **Environment**, *so I avoid it for now.*

⟨ *Environment Accessors 4.18.0.3* ⟩ ≡

```
Id id( ) const
  { return dataIterator( ).subject( ); }

Path& path( )
  { return *mypPath; }

TimeSteps& timeSteps( )
  { return myTimeSteps; }

Data& data( )
  { return *mypData; }

  // const versions

const Path& path( ) const
  { return *mypPath; }

const TimeSteps& timeSteps( ) const
  { return myTimeSteps; }

const Data& data( ) const
  { return *mypData; }

const TNode& currentNode( ) const
  { return *mypCurrentNode; }    // first and last+1 of observations for current case

Data::TIObsbegin( ) const
  { return dataIterator( ).begin( ); }

Data::TIObsend( ) const
  { return dataIterator( ).end( ); }
```

⟨ *Environment Node-Specific Accessors (implicit) 4.18.0.4* ⟩
⟨ *Environment Node-Specific Accessors (explicit) 4.18.0.5* ⟩
⟨ *Environment Iterator Accessors 4.18.0.6* ⟩

```
  // the last (latest) Node on the Path.
  // This is not always the current node.
TNode& lastNode( )
  { return path( ).back( ); }

const TNode& lastNode( ) const
  { return path( ).back( ); }
```

This code is used in section 4.18.

The next methods access information that varies with the specific, current **Node**. *They return information relative to the currentNode( ); naturally that must be set properly.*

Most clients will use these interfaces, rather than the ones with explicit **Node** *arguments, most of the time.*

⟨ *Environment Node-Specific Accessors (implicit) 4.18.0.4* ⟩ ≡

```
TNode& currentNode( )
  {
  return *mypCurrentNode;
  }
const TimePoint& timePoint( ) const
  {
  return timePoint(currentNode( ));
  }
  // Careful: iObservation returns the best available
  // index, even if matchesObservation( ) is false.
IObservation iObservation( ) const
  {
  return iObservation(currentNode( ));
  }
  // Careful: matchesObservation can return true even if there
  // is no observed state.
bool matchesObservation( ) const
  {
  return matchesObservation(currentNode( ));
  }
  // true only if this exact time point has an observed state
bool hasObservedState( ) const
  {
  return hasObservedState(currentNode( ));
  }
  // state as of the most recent observation.
  // if the observation lacked a state observation, the value is negative.
ObsState observedState( ) const
  {
  return observedState(currentNode( ));
  }
```

This code is used in section 4.18.0.3.

These are like the previous accessors, except that they take an explicit **Node** *argument. That argument must be in the* **Environment**.

⟨ *Environment Node-Specific Accessors (explicit) 4.18.0.5* ⟩ ≡

```
const TimePoint& timePoint(const Node& node) const
  {
  return node.timePoint( );
  }

  // Careful: iObservation returns the best available
  // index, even if matchesObservation( ) is false.

IObservation iObservation(const Node& node) const
  {
  return timePoint(node).iObservation( );
  }

  // Careful: matchesObservation can return true even if there
  // is no observed state.

bool matchesObservation(const Node& node) const
  {
  return timePoint(node).matchesObservation( );
  }

  // true only if this exact time point has an observed state

bool hasObservedState(const Node& node) const
  {
  return matchesObservation(node) ∧ data( ).hasObservedState(iObservation(node));
  }

  // state as of the most recent observation.
  // if the observation lacked a state observation, the value is negative.

ObsState observedState(const Node& node) const
  {
  return data( ).state(iObservation(node));
  }

  // Evaluator context

ModelData& activeModelData(Node& node)
  { return node.modelData( ); }

  // type double because EvaluationData may be a set

double& activeEvaluationData(Node& node)
  { return node.evaluationData( ); }
```

This code is used in section 4.18.0.3.

The following shields clients from needing to know exactly how to dereference iterator-like objects in the **Environment**.

⟨ *Environment Iterator Accessors 4.18.0.6* ⟩ ≡

```
// May want currentObservation, currentTimePoint as well.
// Let's wait and see.

// iterator dereferencing
// I don't care enough about reverse iterators to put them here
// and for now I'll skip the const iterators too.
static TimePoint& timePoint(const TimeSteps::iterator & i)
  { return *i; }

static TNode& node(const Path::iterator & i)
  { return *i; }
```

This code is used in section 4.18.0.3.

When updating the **Path**, *do it through this interface to ensure the proper* **Node** *is current.*

Clients must call setCurrentNode( ) whenever they change the focus of attention.

⟨ *Environment Setters 4.18.0.7* ⟩ ≡

```
void pathPush(const StatePoint& theSP, const TimePoint& theTP)
  {
  Node *pn = path( ).pathPush(theSP, theTP);
  setCurrentNode(pn);
  }

void pathPop( )
  {
  Node *p = lastNode( ).previous( );
  path( ).pathPop( );
  setCurrentNode(p);
  }

void pathClear( )
  {
  path( ).clear( );
  setCurrentNode(0);
  }

void setCurrentNode(Node& theNode)
  {
  mypCurrentNode = &theNode;
  }

// next interface permits null pointers
// of course, you must then avoid accessing the current node

void setCurrentNode(Node *thepNode)
  {
  mypCurrentNode = thepNode;
  }
```

This code is used in section 4.18.

Commands that act upon the environment.

The environment is fairly passive, so that if you next( ) it will not automatically recompute or clear the timesteps. It is up to the client to keep things in a consistent state.

For setSubset( ) the argument is a list of **Id***'s, which must be in the same order as the* **Id***'s in data( ). The* **Environment** *will take ownership of the list's memory; clients should* not *attempt to free that memory or use pSubset after the call.*

Internal or external clients that change the basic computation being done, whether by altering parameters or the subset of cases being computed, should call newModel( ).

⟨ *Environment Actions 4.18.0.8* ⟩ ≡

```
void clearTimeSteps( )
  { timeSteps( ).clear( ); }

  // advance to next individual/case
  // return false if none left
bool next( ) throw (DataIteratorError)
  { return dataIterator( ).next( ); }

  // reset

void startSession( )
  { dataIterator( ).startSession( ); }

  // change the active subset of cases. Implicitly restarts the session.

  // future computations will only use this subset.

void setSubset(SubsetDataIterator::IDList & pSubset)
  {
  mypDataIterator.reset(new SubsetDataIterator(&(data( )), pSubset));
  newModel( );
  }

  // future computations will use all possible cases

void setAll(void)
  {
  mypDataIterator.reset(new DataIterator(&(data( ))));
  newModel( );
  }

  // respond to change in the model parameters,
  // including change in subset
  // an excuse to invalidate caches

void newModel(void)
  {
  clear( );
  }
```

This code is used in section 4.18.

Except for ∗mypData, the variables are wholly owned by me. However, ownership of the current node is not through mypCurrentNode but through mypPath. Never *delete through mypCurrentNode.*

The dataIterator is done in a different style because I decided this is the way I want to handle such things in the future.

⟨ *Environment Data 4.18.0.9* ⟩ ≡

```
AbstractDataIterator& dataIterator( )
  {
  return ∗mypDataIterator;
  }
;
const AbstractDataIterator& dataIterator( ) const
  {
  return ∗mypDataIterator;
  }
;
Path ∗mypPath;
TimeSteps myTimeSteps;
Data ∗mypData;
TNode ∗mypCurrentNode;
std::auto_ptr⟨AbstractDataIterator⟩mypDataIterator;
```

This code is used in section 4.18.

The environment supports simulation by providing facilities to pick a random state. The main method, randomDraw, takes an array giving the probability of each state. The client must ensure that these are sane, i.e., that they sum to 1 and that each value is in $[0, 1]$. *A 0-based state is returned.*

The current implementation simply uses **R***'s random number generator. You should manage that generator in* **R***, including choosing an appropriate generator (such as* `rsprng` *for parallel work) and setting the seed.* **R** *is not thread-safe, I believe, and so it is likely random number use is* not thread-safe.

The purpose of the **Environment***'s random number interface is to confine the details of the random number generation to the* **Environment** *class, and to hide that choice from C++ clients.*

It is neither possible nor desirable to completely insulate users of **mspath** *from the details of random number generation. It is not desirable because users should be able to control the random number strategy; it is not possible because setting up a random generator, particularly setting its seed, does not have a uniform interface.*

Particular generators may differ in their requirements for a seed: an integer, two doubles, or some other set of information. For example, parallel random number generators need not only a seed but information on the total number of streams and the index of the current process.

Even within **R** *there is not a uniform interface. Most generators respond to the* **R** *function set.seed( ), but* `rsprng` *does not,*[49] *relying instead on its own functions.*

So **mspath** *provides no functions in either C++ or* **R** *to pick the type of random number generator or to set its seed.*

See §5.17, page 221 for more information on random number generation and alternative implementation strategies.

⟨ *Environment Randomness 4.18.0.10* ⟩ ≡

```
// anticipated use is with double or float types
// the argument is likely to be a slice in practice
template ⟨typename Prob⟩
State randomDraw(const std::valarray < Prob > &theDensity);
```

This code is used in section 4.18.

[49] As of version 0.4, 2009-08-01.

4.19 Thread-Specific Scratch Data

A **ScratchDataProducer** *makes* **ScratchData** *for a particular thread and stores it in a* **ScratchPad**. *The producer and other clients using the same thread may later access the* **ScratchData**.

Narrowly, this framework is basically a dictionary. The intended use, however, is to hold thread-specific information. The class itself has no concurrency protection; don't use it like a "blackboard" to share data between threads.

Typically, **ScratchDataProducer** *will be an object shared between threads; it uses the* **ScratchPad** *to hold its thread-specific results, caches, etc.*

See §4.6, page 58 for more about caching. Most concrete classes that implement **ScratchDataProducer**, *and some the implement* **ScratchData**, *also implement the caching interface described there.*

4.19.1 ScratchPad

ScratchPad *is a mix-in for* **Environment**. *By defining* **ScratchPad** *separately, we allow some clients to use it without pulling in the whole* **Environment** *definition.*

This class permits storing arbitrary thread-specific information. **ScratchPad** *(in practice,* **Environment***) owns the data and is responsible for its cleanup. The key is the address of the* **ScratchDataProducer** *that created that information; the values, subclasses of* **ScratchData**, *hold the state.*

⟨ *ScratchPad.h 4.19.1* ⟩ ≡
```
@o ScratchPad.h
#ifndef ScratchPad_h
#define ScratchPad_h 1

#include "ScratchData.h"

#include <boost/ptr_container/ptr_map.hpp>

namespace mspath { class ScratchDataProducer; class ScratchPad
  {

public:
  virtual ~ScratchPad( )
    { }

    // throws an error if key not present
  ScratchData& getScratchData(const ScratchDataProducer *key) throw
        (boost::bad_ptr_container_operation)
    {
    return myState.at(key);
    }

    // assumes key is not already present

  void setScratchData(const ScratchDataProducer *key, ScratchData *pState)
    {
    myState.insert(key, pState);
    }

  ;

  bool hasScratchData(const ScratchDataProducer *key) const
    {
    return myState.find(key) ≠ myState.end( );
    }

    // delete associated value and return count of items deleted
  void release(const ScratchDataProducer *key) {
      // the inner find works around a bug in Boost 1.33
  TStore:: iterator i(myState.find(key));
  if (i ≠ myState.end( ))
    myState.erase(i); }

    // delete all contents
  virtual void clear( )
    {
    myState.clear( );
```

```
    }
  protected:
  typedef boost:: ptr_map < const ScratchDataProducer *, ScratchData > TStore;
  TStore myState; } ; }
#endif
```

This code is used in section 9.

4.19.2 Scratch Data

Abstract base class for thread-specific data, to be stored in **ScratchPad** *and produced by* **ScratchDataProducer**.

```
⟨ ScratchData.h 4.19.2 ⟩ ≡
  @o ScratchData.h
#ifndef ScratchData_h
#define ScratchData_h 1
  namespace mspath
    {
    class ScratchData
      {
    public:
      virtual ~ScratchData( )
        { }
      };
    }
#endif
```

This code is used in section 9.

4.19.3 Scratch Data Producers

Base class for any object that produced **ScratchData**. *Note the producer is not necessarily thread-specific.*[50]

For a unique key[51] *these objects use their own address. Thanks to dark corners of C++, & does not always yield the same value, depending (at least) upon the exact class of pointer or reference being used. To avoid that problem*[52] *always use the scratchKey( ) function to get keys for the* **ScratchPad**. *scratchKey( ) is deliberately not virtual to try to assure the same value with every call.*

Subclasses should get their data like this: aScratchPad.getScratchData < MyDataClass > (aProducer.scratchKey( )). MyDataClass must be a kind of **ScratchData**. *The call will find, or, if necessary, create scratch data of class MyDataClass. There is also a getScratchData( ) call with two arguments; the second is passed to the MyDataClass constructor if necessary.*

I may want to define a type, or rather tell subclasses they must define a type, TScratchData, to hold the scratch data and further automate the use of templates.

⟨ *ScratchDataProducer.h 4.19.3* ⟩ ≡
```
@o ScratchDataProducer.h
#ifndef ScratchDataProducer_h
#define ScratchDataProducer_h
#include "ScratchData.h"
#include "ScratchPad.h"
  namespace mspath {
  class ScratchDataProducer
    {
  public:
      // note my destruction does not affect my ScratchData
    virtual ~ScratchDataProducer( )
      { }
    ;
    const ScratchDataProducer *scratchKey( ) const
      {
      return this;
      }
      // release associated ScratchData from given ScratchPad
    virtual void release(ScratchPad *thePad) const
      {
      thePad→release(scratchKey( ));
      }
  protected:
    template ⟨class TD⟩
    TD& getScratchData(ScratchPad& theTS) const
      {
      if (¬theTS.hasScratchData(scratchKey( )))
```

[50] Thread-specific objects can store data directly in themselves without resorting to the **ScratchPad**.
[51] Unique within a given thread
[52] I hope

```
      theTS.setScratchData(scratchKey( ), makeScratchData⟨TD⟩( ));
    return static_cast⟨TD&⟩(theTS.getScratchData(scratchKey( )));
    }

  template ⟨class TD⟩
  TD *makeScratchData( ) const
    {
    return new TD;
    }

    // if we need to pass arg to c'tor, use next two
    // Note that only the first template argument is required
    // on invocation; the other can be inferred.
    // even if calling type of Arg is const &, that info
    // is stripped out. So I need to put it in explicitly
  template ⟨ class TD, class Arg> TD& getScratchData(ScratchPad& theTS, const
        Arg& theArg) const
    {
    if (¬theTS.hasScratchData(scratchKey( )))
      theTS.setScratchData(scratchKey( ), makeScratchData⟨TD⟩(theArg));
    return static_cast⟨TD&⟩(theTS.getScratchData(scratchKey( )));
    }

  template ⟨ class TD, class Arg> TD *makeScratchData(const Arg& theArg) const
    {
    return new TD(theArg);
    }

  } ; }
#endif
```

This code is used in section 9.

4.20 Recorder

A recorder is an abstract type that records traversal through the paths. It usually records the number of nodes and paths generated, and also accumulates the likelihood.

Actual recorders should subclass and add appropriate state and implementation. They must implement the protocol defined here.

Most recorders will use a **Model** *to accumulate likelihoods, but that is not required, so is not in the base class below. In addition to handling different types of models and evaluators, recorders may track different statistics, such tracking the total number of paths vs. tracking detailed path and node statistics for each case.*

From some perspectives, the central action of the program happens in the recorder.

⟨ *Recorder.h 4.20* ⟩ ≡

```
@o Recorder.h
#ifndef Recorder_h
#define Recorder_h 1

#include "Path.h"

namespace mspath
  {

  class Recorder
    {

  public:
    ⟨ Recorder Typedefs 4.20.0.1 ⟩
    ⟨ Recorder Constructors 4.20.0.2 ⟩
    ⟨ Recorder Accessors 4.20.0.3 ⟩
    ⟨ Recorder Actions 4.20.0.4 ⟩

  private:
    ⟨ Recorder Data 4.20.0.5 ⟩
    };

  }   // close namespace mspath

#endif   // Recorder_h
```

This code is used in section 9.

Use **TCount** *for recording all the counts.*

⟨ *Recorder Typedefs 4.20.0.1* ⟩ ≡

```
typedef Path TPath;
typedef unsigned long int TCount;
```

This code is used in section 4.20.

⟨ *Recorder Constructors 4.20.0.2* ⟩ ≡

```
virtual ~Recorder( )
  { }
```

This code is used in section 4.20.

⟨ *Recorder Accessors 4.20.0.3* ⟩ ≡

```
#if 0
  Id id( ) const
    { return myCurrentId; }    // always return copy
#endif
```

This code is used in section 4.20.

Subclasses must *implement the protocols defined here. Comments indicate some possible future protocols, currently unneeded.*

When using this interface, clients should assume that

- *They must call the indicated actions in the indicated order, though looping is allowed. You may not call a function later in this list unless all earlier functions have been invoked.*
- *The recorder will not have reliable information to report until finishSession( ) has been invoked.*
- *After calling startTree you must call goodNode. Call goodNode on the terminal node before calling goodPath. That is, although startTree and goodPath imply a good node, it is not called automatically for you. This may change.*
- *Call startTree after you have pushed the base node onto the path.*
- *impermissible( ) does not examine the current path; there is no need to push the impermissible node onto it, or even to create such a node.*

Revist and reconsider the preceding requirements.

Particular subclasses may be less restrictive, but adherence to these conventions will assure maximum safety.

⟨ *Recorder Actions 4.20.0.4* ⟩ ≡

```
    // call at beginning of processing
    // likely no-op, since it can be done at object initialization.
  virtual void startSession( )
    { }

    // call at the start of processing a given case

  virtual void startCase( ) = 0;

    // call when starting a tree of paths

  virtual void startTree(double initialProbability)
    { }

    // call every time we've encountered a (seemingly) good node
    // of course, later, it may be revealed to be part of a bad path
  virtual void goodNode( ) = 0;

    // When client generates a node at the end of a good path, it
    // should call goodNode( ) and goodPath( ).

    // call every time we have a complete, valid path
    // here's where to accumulate the liklihood.
    // Since we may write into the path, it is not const
  virtual void goodPath(TPath& path) = 0;

    // call every time we discover we have wandered into an invalid or
    // impossible path.
  virtual void impermissible( ) = 0;

    // finished processing tree. Likely no-op.

  virtual void finishTree( )
    { }

    // finished processing case
  virtual void finishCase( ) = 0;

    // completely done. Might clean up, output results, etc.
  virtual void finishSession( ) = 0;
```

This code is used in section 4.20.

Mostly left for subclasses.

⟨ *Recorder Data 4.20.0.5* ⟩ ≡

```
    // Id myCurrentId;
```

This code is used in section 4.20.

4.20.1 SimpleRecorder

This is probably the simplest possible recorder; it just counts paths and related information. Useful in its own right, it is also allows us to test the overall system before adding the fancier stuff.

The same node may appear in several paths. Such a node will contribute once to goodNodes but many times to goodPathNodes.

I anticipate that run-time will mostly be a function of goodNodes. A naive implementation that generated all paths would have run-time proportional to goodPathNodes.

⟨ *SimpleRecorder.h 4.20.1* ⟩ ≡
```
@o SimpleRecorder.h
#ifndef SimpleRecorder_h
#define SimpleRecorder_h 1
#include "Recorder.h"
#include <ostream>
#include "Environment.h"
#include "MSPathError.h"
#include "Path.h"
  namespace mspath
    {
    class SimpleRecorder : public Recorder
      {
    public:
      ⟨ SimpleRecorder Constructors 4.20.1.1 ⟩
      ⟨ SimpleRecorder Accessors 4.20.1.2 ⟩
      ⟨ SimpleRecorder Actions 4.20.1.3 ⟩
    protected:
      ⟨ SimpleRecorder data 4.20.1.5 ⟩
      };
    ⟨ SimpleRecorder Output 4.20.1.6 ⟩
    }
#endif
```

This code is used in section 9.

Just be sure all counts are 0. May be unnecessary to do explicitly, but can't hurt.

If I later implement startSession( ) the initialization should be done there.

⟨ *SimpleRecorder Constructors 4.20.1.1* ⟩ ≡
```
SimpleRecorder(Environment *const thepEnv) : myNCases(0), myNPaths(0), myNNodes(0),
      myNBads(0), myNPathNodes(0), mypEnv(thepEnv)
  { }
```

This code is used in section 4.20.1.

⟨ *SimpleRecorder Accessors 4.20.1.2* ⟩ ≡

```
TCount goodNodes( ) const
  { return myNNodes; }

TCount goodPaths( ) const
  { return myNPaths; }

TCount cases( ) const
  { return myNCases; }


TCount badNodes( ) const
  { return myNBads; }     // bad nodes or bad paths

double averagePathLength( ) const
  { return static_cast⟨double⟩ (myNPathNodes) / myNPaths; }

TCount goodPathNodes( ) const
  { return myNPathNodes; }

Environment& environment( )
  { return *mypEnv; }
```

This code is used in section 4.20.1.

We just get the overall counts here. Since many of the functions are virtual, they may not be in-lineable.

Some goodNode( )'s turn out not to be on goodPath( )'s. So we can save some computation by doing nothing when a goodNode( ) is detected, and only evaluating **Node***'s after goodPath( ), backtracking as necessary. In practice, the difference is small. In one test case, there were 4,609,622 calls to goodNode( ) but only 4,594,838 unique nodes on good paths. So the naive approach would only result in* 0.32% *more calculations.*

⟨ *SimpleRecorder Actions 4.20.1.3* ⟩ ≡

```
virtual void startSession( );
  // call at the start of processing a given case
virtual void startCase( )
  {
  ++myNCases;
  myNCasePaths = 0;
  }
virtual void startTree(double p)
  {    // No action needed.
    // More complex recorders could gather per-tree statistics
    // or delegate to an evaluator which would do something like
    // pNode→evaluationData( ) = std::log(p);
  }
  // call every time we've encountered a (seemingly) good node
  // of course, later, it may be revealed to be part of a bad path
virtual void goodNode( )
  { }
  // call every time we have a complete, valid path
virtual void goodPath(TPath& path)
  { evaluateNode(environment( ).currentNode( )); ++myNCasePaths; myNPathNodes +=
      path.size( ); }
⟨ SimpleRecorder Action Helpers 4.20.1.4 ⟩
  // call every time we discover we have wandered into an invalid or
  // impossible path.
virtual void impermissible( )
  { ++myNBads; }
virtual void finishCase( ) throw (DataModelInconsistency)
  { if (myNCasePaths ≤ 0)    // No legal paths for this case
    throw DataModelInconsistency(environment( ).id( ), "␣Observed␣path␣can't␣come\
      ␣from␣Model."); myNPaths += myNCasePaths; }
virtual void finishSession( )
  { }
virtual void finishTree( )
  { }
```

This code is used in section 4.20.1.

The following methods are not part of the public interface of the class, but are used to help implement goodPath( ). Subclasses may wish to override or augment them.

⟨ *SimpleRecorder Action Helpers 4.20.1.4* ⟩ ≡

```
protected:
    // evaluate the Node after assuring prior Node's have
    // been evaluated
  virtual void evaluateNode(Node& node)
    {
    if (node.alreadyCountedAsGood( ))
      return;
    if (¬node.isRoot( ))
      {
      Node *pn = node.previous( );
      environment( ).setCurrentNode(pn);
      evaluateNode(*pn);    // recurse
      environment( ).setCurrentNode(node);
      }
    evaluateAnyNode(node);
    }
    // The next method can assume that all previous nodes
    // in the Path have already been evaluated.
    // do processing valid for any node, root or non-root
  virtual void evaluateAnyNode(Node& node)
    {
    myNNodes++;
    node.countAsGood( );
    }
  public:
```

This code is used in section 4.20.1.3.

Just the basics. These statistics apply to the entire run, summing over all cases.

myNNodes counts the number of unique nodes, while myNPathNodes may count the same node several times if it is in several good paths.

A good node (one whose creation is followed by a call to goodNode( )) may not be counted at all if it turns out not to lie on any good paths.

⟨ *SimpleRecorder data 4.20.1.5* ⟩ ≡

```
  TCount myNCases;
  TCount myNPaths;    // good, complete paths
  TCount myNNodes;    // unique good nodes created
  TCount myNBads;     // times hit impermissible node/path
  TCount myNPathNodes;    // total nodes in good paths
  Environment *mypEnv;    // just a reference, not ownership
  TCount myNCasePaths;    // good path count for current case
```

This code is used in section 4.20.1.

Dump all my statistics.

⟨ *SimpleRecorder Output 4.20.1.6* ⟩ ≡
std::*ostream* & **operator** << (**std**::*ostream* & , **const SimpleRecorder**&) ;

This code is used in section 4.20.1.

4.20.2 EvaluatorRecorder

This subclass of **SimpleRecorder** *adds the ability to evaluate the* **Node***'s on the* **Path** *and record the results. Though this concept is fairly general, this first cut is specialized for the case where we are computing a single log-likelihood. The evaluator computes the likelihood of each path. We sum them and then get the log-likelihood for the case, summing the log-likelihoods across cases.*

Eventually, a lot of this action should probably be delegated to the **Evaluator***. Then this class will be more general.*

Originally, I thought the **Evaluator** *would be responsible for assuring that all previous nodes are evaluated. However, the* **Recorder** *needs to collect some statistics on the number of* **Node***'s being evaluated, so instead the recursion happens here.*

⟨ *EvaluatorRecorder.h 4.20.2* ⟩ ≡

```
@o EvaluatorRecorder.h
#ifndef EvaluatorRecorder_h
#define EvaluatorRecorder_h 1

#include <cmath>

#include "SimpleRecorder.h"
#include "Evaluator.h"
#include "Environment.h"

#if 0
    // debugging
#include "test/main_test.h"
#endif

  namespace mspath {

  class EvaluatorRecorder : public SimpleRecorder
    {

  public:
    EvaluatorRecorder(Evaluator *thepEvaluator, Environment *const thepEnv) :
          SimpleRecorder(thepEnv), mypEvaluator(thepEvaluator)
      {}

    typedef SimpleRecorder Super;

    ⟨ Evaluator Accessors 4.20.2.1 ⟩
    ⟨ Evaluator Actions 4.20.2.2 ⟩
    protected:
        ⟨ Evaluator Data 4.20.2.3 ⟩
        ⟨ Evaluator Protected Accessors 4.20.2.4 ⟩
    }; }    // end namespace mspath
  #endif    // EvaluatorRecorder_h
```

This code is used in section 9.

⟨ *Evaluator Accessors 4.20.2.1* ⟩ ≡

```
virtual const EvaluationData& logLikelihood( ) const
  {
  return myTotalLogLikelihood;
  }
```

This code is used in section 4.20.2.

Note that startTree will need to do more work as soon as multiple initial states are possible. And here is where a lot of the operations would more properly be delegated to the **Model***, which is the one that should be worrying about log-transforms.*

⟨ *Evaluator Actions 4.20.2.2* ⟩ ≡

```
virtual void startSession( )
  {
  Super::startSession( );
  myTotalLogLikelihood = 0.0;
  }
virtual void startCase( )
  {
  Super::startCase( );
  myTotalPathProbability = 0.0;
  }
virtual void startTree(double initialProbability)
  {
  Super::startTree(initialProbability);    // probability of first node is 1.
  environment( ).currentNode( ).evaluationData( ) = 1.0;
  }
virtual void goodPath(TPath& path)
  {
  Super::goodPath(path);
#if 0
    // debug
  std::cout << environment( ).id( ) << "␣" << environment( ).path( ) <<
      environment( ).currentNode( ).evaluationData( ) << std::endl;    // end debug
#endif
  myTotalPathProbability += environment( ).currentNode( ).evaluationData( );
  }
virtual void finishCase( ) throw (DataModelInconsistency)
  {
  Super::finishCase( );
  myTotalLogLikelihood += std::log(myTotalPathProbability);
  }
protected:    // The next method can assume that all previous nodes
    // in the Path have already been evaluated.
    // do processing valid for any node, root or non-root
  virtual void evaluateAnyNode(Node& node)
    {
    Super::evaluateAnyNode(node);
    evaluator( ).evaluate(node, environment( ));
    } public:
```

This code is used in section 4.20.2.

The use of **EvaluationData** *below gives the appearance of generality. But I've only written this to handle the case where it is a* **double**.

⟨ *Evaluator Data 4.20.2.3* ⟩ ≡
```
Evaluator *mypEvaluator;
EvaluationData myTotalLogLikelihood;
EvaluationData myTotalPathProbability;
```

This code is used in section 4.20.2.

⟨ *Evaluator Protected Accessors 4.20.2.4* ⟩ ≡
```
Evaluator& evaluator( )
  {
  return *mypEvaluator;
  }
EvaluationData& totalPathProbability( )
  {
  return myTotalPathProbability;
  }
```

This code is used in section 4.20.2.

4.21 Evaluator

Once the **PathGenerator** *creates a good node or path, it notifies the* **Recorder**. *Unless the latter is simply counting paths, it asks the* **Evaluator** *to evaluate the node. "Evaluate" means compute all quantities of interest at this node. Evaluation involves three distinct responsibilities.*[53] *Except for special cases, they must be peformed in the following sequence because each step depends on prior ones:*

1. *Evaluation may require information about prior* **Node***'s in the path.* Those nodes may not have been evaluated. *The client (usually a* **Recorder***) is responsible for assuring all prior nodes are evaluated first.*
2. *Populate the* **Node***'s modelData. Ordinarily, this is history dependent-data that depends on the type of* **Model***, but not on particular parameter values. For example, this could be time in current state. For a Markov model it would be null. Ordinarily this task is delegated to the* **Model**.
3. *Populate the* **Node***'s evaluationData. Ordinarily, that contains the results of one or more models. For example, this could be the log-likelihood of the path to this point.*

[53] In the future, these might be delegated to distinct classes owned by this one.

The simplest **Evaluator**, *implemented here, evaluates one set of parameters and records the result in a single number. More complicated ones could evaluate sets of parameters and record corresponding sets of results to raise efficiency. These sets might be in vector, matrix, or some other form.*

If the form of the *evaluationData* changes, update basic.h so that the type **EvaluationData** is correct. *The information is there rather than in this header file to reduce the strength of links between header files, and loops among them.*[54]

Model*'s are only responsible for evaluating one set of parameters and producing one result.* **Evaluator** *is responsible for chopping things up for* **Model***'s digestion. The typical result of the* **Model**, *and thus the* **Evaluator**, *is the log-likelihood up to this point on the* **Path**. *Because evaluating a model requires the* **Environment**, *this class is an indirect client of* **Environment** *and thus the* **Data** *on which the model evaluation depends.*

There are currently no unit tests of this class. Such tests would largely be tests of the underlying **Model** *and HistoryRecorder 's. However, higher-level tests test this class indirectly. If separate testing seems appropriate later, it would likely make sense to use a stub* **Model** *class. That would require a class-hierarchy of* **Model***'s, which might impose some performance penalty. It does not seem worthwhie to make such a hierarchy purely for testing.*

This class suggests a possible refactoring of **Model**. *If we evaluate multiple sets of parameters, it only makes sense for some of the parameters to vary. The ones responsible for generating paths must ordinarily be the same for all sets, as must the structural form of the model (the specific forms of prior history we care about). So it would make sense to have a single model, with the varying parameters stored outside of it. In principle the models could vary in structural ways as well, but there would likely be little optimization to be gained by combining such models. I defer worrying about such things until and unless they become real issues.*

⟨ *Evaluator.h 4.21* ⟩ ≡

```
  @o Evaluator.h
#ifndef Evaluator_h
#define Evaluator_h 1

#include "Model.h"
#include "Node.h"

  namespace mspath
    {

    class Environment;

    class Evaluator
      {

    public:    // constructor
      Evaluator(Model *pModel) : mypModel(pModel)
        {}

        // action

      void evaluate(Node& theNode, Environment& theEnv);

    protected:
      Model *mypModel;

      Model& model( )
```

[54] **Evaluator** has a type of **Node** in its definition, but **Node** uses a type from **Evaluator** in its data definition.

```
        {
        return *mypModel;
        }
    };    // end class Evaluator
  }    // end namespace mspath
#endif    // Evaluator_h
```

This code is used in section 9.

4.22 Data

This class holds the observational data passed in from **R**. *It does not hold model parameters or structure.*

A peculiarity of the covariate data is that the each column contains the covariates for a particular observation, the transpose of the usual arrangement.

Perhaps I'll use different classes for different ways of storing data (e.g., from/to) or presence or absence of misclassification.

This implementation copies the data into C++ **valarray***'s. It would be more efficient to use the data in place.*

The data often include more than one observation on an individual (or, more generally, "case"). We require that the input data be grouped by case, ordered by ascending time within each group.

The **Data** *may be shared between threads; iterators over the data are thread-specific.*

```
⟨ Data.h 4.22 ⟩ ≡
    @o Data.h
  #ifndef Data_h
  #define Data_h 1
  #include "basic.h"
  #include <valarray>
  #include "MSPathError.h"
  #include <R.h>
      // < memory > is for the data iteration definitions
      // It must be defined at global scope to avoid problems.
  #include <memory>
    namespace mspath { class TimeStepsGenerator;
    class FixedTimeStepsGenerator;    // forward decls for friend
    class CompressedTimeStepsGenerator;
    class AbstractDataIterator;
    class Data
      {
    public:
      friend class FixedTimeStepsGenerator;
      friend class TimeStepsGenerator;
      friend class CompressedTimeStepsGenerator;
      friend class DataIterator;
      typedef Double2D::TCol TCovariates;
      typedef size_t TIObs;    // index to observation
      typedef std::valarray⟨double⟩ TTimes;    // observation times
      ⟨ Data Constructors 4.22.1 ⟩
      ⟨ Data Accessors 4.22.1.1 ⟩
    protected:
      ⟨ Data Data 4.22.1.2 ⟩} ;
      ⟨ Data Iteration 4.22.2 ⟩
      } ;
    #endif
```

This code is used in section 9.

4.22.1 Main Data class

Because I copy data, I do not own the pointers when this is done.

⟨ *Data Constructors 4.22.1* ⟩ ≡

```
Data(int *pSubject, size_t nobs, size_t theNpts, double *pTime, ObsState *pState,
    double *pCov, size_t ncovs, double *pMiscCov, size_t nmisccovs) :
  mySubject(pSubject, nobs), myTime(pTime, nobs), myState(pState, nobs), myCov(pCov, ncovs,
      nobs), myMisccov(pMiscCov, nmisccovs, nobs), myNpts(theNpts)
  { }
  // The following is just a convenience for testing
Data( )
  { }
```

This code is used in section 4.22.

I will need to build accessors as needed.

⟨ *Data Accessors 4.22.1.1* ⟩ ≡

```
// Data about a particular observation

Id subject(TIObs i) const
  { return mySubject[i]; }

ObsState state(TIObs i) const
  { return myState[i]; }

double time(TIObs i) const
  { return myTime[i]; }

bool hasObservedState(TIObs i) const
  { return state(i) ≥ 0; }

// I'm not sure how much object creation the next two
// involve, or if they can be reduced, for const or non-const.

TCovariates covs(TIObs i) const
  { return myCov.col(i); }

TCovariates miscCovs(TIObs i) const
  { return myMisccov.col(i); }

// Only knowledgeable clients should use the next four

Double2D& covs( )
  {
  return myCov;
  }

Double2D& miscCovs( )
  {
  return myMisccov;
  }

const Double2D& covs( ) const
  {
  return myCov;
  }

const Double2D& miscCovs( ) const
  {
  return myMisccov;
  }

// Counts

// Number of observations

size_t nObs( ) const
  { return mySubject.size( ); }

//Number of individuals

size_t nPersons( ) const
  { return myNpts; }

// Number of covariates

size_t nCovs( ) const
```

```
    { return myCov.nrows( ); }
    // Number of misclassification covariates
  size_t nMiscCovs( ) const
    { return myMisccov.nrows( ); }
    // Histories
  TTimes observationTimes(const AbstractDataIterator& iterator);
```

This code is used in section 4.22.

⟨ *Data Data 4.22.1.2* ⟩ ≡

```
  Int1D mySubject;
  Double1D myTime;
  Array1D⟨ObsState⟩ myState;
  Double2D myCov;
  Double2D myMisccov;
  size_t myNpts;
```

This code is used in section 4.22.

4.22.2 Data iterators

Iterator moves over cases, and hence entire groups of observations rather than individual observations. They also provide the information necessary to pick out particular observations associated with the case.

For this and other reasons, these iterators are quite unlike the standard library's iterators; their interface is much different too. For example, they are initially (and after startSession( )) before *the first case, so clients must call next( ) before making other queries of the iterator. next( ) returns true if it has moved to new data, and false if it moves past the end of data.*

Iterators are thread-specific, while the **Data** *are shared between threads.*

With a newly initialized class, startSession( ) should be unnecessary.

I used to **#include** `<memory>` *here, but doing so nests it inside the* **mspath** *namespace. Under gcc 4.3 (but not all earlier versions) doing so produces the error (each original line is split across two lines below)*

```
/home/ross/UCSF/peter/R/mspath/src/trueSrc/DataT.cc:
   In member function std::valarray<double>
mspath::Data::observationTimes():
/home/ross/UCSF/peter/R/mspath/src/trueSrc/DataT.cc:8:
   error: slice is not a member of mspath::std
```

Presumably including a standard header inside the **mspath** *namespace screws up some of the original definitions.*

⟨ *Data Iteration 4.22.2* ⟩ ≡

⟨ *AbstractDataIterator 4.22.2.1* ⟩
⟨ *DataIterators 4.22.2.2* ⟩
⟨ *SubsetDataIterator 4.22.2.3* ⟩

This code is used in section 4.22.

This class gives the common protocol for Data Iterators. They refer to **Data** *but do not own it or change it.*

In principle these might throw exceptions at construction time; at the moment, in the interests of speed, they only do so later. It's easier to check validity on the fly.

⟨ *AbstractDataIterator 4.22.2.1* ⟩ ≡

```
class AbstractDataIterator
  { public:    // I refer to Data; I don't own it
  AbstractDataIterator(Data *pData)

throw (DataIteratorError) :
  mypData(pData)
    {}

  ;

  virtual ~AbstractDataIterator( )
    {}
  ;

    // Iterate over cases
  virtual void startSession( ) = 0;    // position before first case
  virtual bool next( ) throw (DataIteratorError) = 0;
      // position at next case. return true if there is one

    // info about the current case
  virtual Data::TIObsbegin( )const  = 0;    // first observation
  virtual Data::TIObsend( )const  = 0;    // 1 past last observation
  virtual Id subject( )const  = 0;    // Id shared by all observations
  virtual size_t size( )const  = 0;    // number of observations in this case

protected:

  const Data& data( ) const
    {
    return *mypData;
    }
  ;

  const Data *const mypData;    // ownership, not reference
  } ;
```

This code is used in section 4.22.2.

This is the standard iterator that moves over all cases.

myBegunIterating could be dispensed with by putting, for example, myCurrentFirst > myCurrentLast, but it seems clearer to be explicit. Don't try to be clever by removing it and stuffing negative values in somewhere; these are unsigned types.

⟨ *DataIterators 4.22.2.2* ⟩ ≡

```
class DataIterator : public AbstractDataIterator
  { public: DataIterator(Data *pData)

throw (DataIteratorError) :
  AbstractDataIterator(pData), myBegunIterating(false)
    {}

  ;

  virtual ~DataIterator( )
    {}
  ;

    // Iterate over cases

  virtual void startSession( )
    { myBegunIterating = false; }    // position before first case
  virtual bool next( ) throw (DataIteratorError);
      // position at next case. return true if there is one

  // info about the current case
  // These might also include assert(myBegunIterating)

  virtual Data::TIObsbegin( ) const
    {return myCurrentFirst;}

  virtual Data::TIObsend( ) const
    {return myCurrentLast + 1; }

  virtual Id subject( ) const
    {return myCurrentId;}

  virtual size_t size( ) const
    {return myCurrentSize;}

  protected:    // to start of next case, which begins at i
  bool advanceToNext(Data::TIObsi) throw (DataIteratorError);

    // data
  Id myCurrentId;    // identifier of current case
  Data::TIObsmyCurrentFirst;    // first observation for current case
  Data::TIObsmyCurrentLast;    // last observation for current case
  size_t myCurrentSize;    // number of observations for current case
  bool myBegunIterating;    // true if we have started the iterator

  } ;
```

This code is used in section 4.22.2.

The next iterator evaluates only a subset of the cases in the **Data**. *It takes ownership of a pointer to the list of Id's it cares about; this list should be in the same order as those in the data, and should contain only valid values. No checking is done on those preconditions.*

Note this class provides the type **IDList** *for the list of* **Id***'s. Clients should use it, most likely like this:* **IDList** *pIDs* (**new Int1D** (**int** ∗*ids*, **size_t** *n*), *where ids is an array with n* **int***'s.*

The current implementation is naive; optimization is possible, but seems unlikely to affect performance. The code also depends on the type of **Id** *(really the subject element of* **Data***) in a brittle way. It cheats with the knowledge that* **Id** *is* **int** *to use a vector of type* **Int1D** *to hold the* **Id***'s.*

Originally I thought of passing a boolean mask of observations (not ID's) to this iterator. That would offload the work of figuring out which observations matched which ID's, as well as concerns about getting the ID's in the correct sequence. However, such an interface permits holes within the observations for a single case. The holes make the simple assumption that a beginning and ending observation index suffice to describe the current case incorrect, and would require reworking the interface of **AbstractDataIterator** *and the clients.* **AbstractDataIterator***'s would need to return some other iterators over observations of a single case, and one would then need to translate from those iterators to particular observations.*

Since I anticipate no need for such facilities, I took the simpler approach here.

⟨ *SubsetDataIterator 4.22.2.3* ⟩ ≡

```
class SubsetDataIterator : public AbstractDataIterator
  { public: SubsetDataIterator(Data *pData, std::auto_ptr<Int1D>pIDs)

throw (DataIteratorError) :
  AbstractDataIterator(pData), mypIDs(pIDs), myNextInSubset(0U),
        myEndSubset(mypIDs→size( ))
    { }

  virtual ~SubsetDataIterator( )
    { }

  typedef std::auto_ptr<Int1D> IDList;

    // Iterate over cases

  virtual void startSession( )
    { myNextInSubset = 0U; }    // position before first case
  virtual bool next( ) throw (DataIteratorError);
      // position at next case. return true if there is one

    // info about the current case
    // These might also include assert(myBegunIterating)

  virtual Data::TIObsbegin( ) const
    { return myCurrentFirst; }

  virtual Data::TIObsend( ) const
    { return myCurrentLast + 1; }

  virtual Id subject( ) const
    { return myCurrentId; }

  virtual size_t size( ) const
    { return myCurrentSize; }
  ;

  protected:    // to start of next case, beginning search at i
```

```
bool advanceToNext(Data::TIObsi) throw (DataIteratorError);

const Id& subsetID(size_t i) const
  {
  return (*mypIDs)[i];
  }

  // data
Id myCurrentId;    // identifier of current case
Data::TIObsmyCurrentFirst;    // first observation for current case
Data::TIObsmyCurrentLast;    // last observation for current case
size_t myCurrentSize;    // number of observations for current case

IDList mypIDs;    // vector of ID's to iterate over
size_t myNextInSubset;    // next entry in mypIDs to look for
size_t myEndSubset;    // stop when you get here

} ;
```

This code is used in section 4.22.2.

4.23 Errors

Implementation note: what( ) is not supposed to throw anything in any class inheriting from **std**::*exception. Many operations require additional memory, and so might violate this requirement. I try to do the allocating in the constructor, though I may not have succeeded completely.*

*My use of exception specifications (***void** *fn( )* **throw** *( )) is spotty at best.*

⟨ *MSPathError.h 4.23* ⟩ ≡
```
  @o MSPathError.h
#ifndef MSPathError_h
#define MSPathError_h 1

#include <sstream>
#include <stdexcept>
#include <string>

#include "basic.h"

  namespace mspath {

  ⟨ DataModelInconsistency 4.23.0.1 ⟩
  ⟨ OneInitialState 4.23.0.3 ⟩
  ⟨ BadInitialProbs 4.23.0.4 ⟩
  ⟨ InconsistentModel 4.23.0.2 ⟩
  ⟨ DuplicateTerm 4.23.0.5 ⟩
  ⟨ UnknownTerm 4.23.0.6 ⟩
  ⟨ TangledDependencies 4.23.0.7 ⟩
  ⟨ DataIteratorErrors 4.23.0.8 ⟩


  }   // end namespace mspath
#endif
```

This code is used in section 9.

Use this error if the data and model are inconsistent, in other words if the data show paths that the model says are impossible.

I tried this with a stringstream as member data, but it is not copyable.

⟨ *DataModelInconsistency 4.23.0.1* ⟩ ≡

```
struct DataModelInconsistency : public std ::domain_error
  { DataModelInconsistency (Id theId, const std::string&msg) : std::domain_error (msg),
        myId(theId)
    {
    std::stringstream myStream;
    myStream << "Model␣is␣inconsistent␣with␣data␣at␣case␣" << theId << ":␣" << msg;
    myMsg = myStream.str( );
    }
  virtual ~DataModelInconsistency( ) throw ( )
    { } virtual const char *what( )const  throw ( )
    {
    return myMsg.c_str( );
    }
  Id id( ) const
    {
    return myId;
    }
protected:
  Id myId;
  std::string myMsg; } ;
```

This code is used in section 4.23.

This error applies to models (not necessarily only of the **Model** *class) that are internally inconsistent. Use this when the user gives an unreasonable model. Somewhat confusingly, this error is ultimately subclass of* **std**::*logic_error which the standard says is for "errors in the internal logic of the program."*[55]

⟨ *InconsistentModel 4.23.0.2* ⟩ ≡

```
struct InconsistentModel : public std ::domain_error { InconsistentModel (const
      std::string & msg) : std::domain_error (msg)
  { } virtual ~InconsistentModel( ) throw ( )
  { } } ;
```

This code is used in section 4.23.

[55] Which doesn't seem how they are used. They seem to be used as I am doing here. Perhaps the argument is that anyone seeing the inputs could tell there was a problem, vs. a resource exhaustion issue, which is a *runtime_error*.

Currently, though not permanently, we only handle a single initial state. Throw this if someone tries for more.

⟨ *OneInitialState 4.23.0.3* ⟩ ≡

```
struct OneInitialState : public std ::invalid_argument { OneInitialState( ) :
        std::invalid_argument("Sorry, I can only handle a single initial state for now")
  { }
virtual ~OneInitialState( ) throw ( )
  { } } ;
```

This code is used in section 4.23.

Throw this exception if the vector of initial probabilities fails basic tests of how probabilities behave. While **OneInitialState** *reflects a temporary limitation of this program, the error below signals that the user has given nonsense inputs.*

⟨ *BadInitialProbs 4.23.0.4* ⟩ ≡

```
struct BadInitialProbs : public std ::invalid_argument { BadInitialProbs(const
        std::string & msg) : std::invalid_argument(std::string("Impossible initial probab\
        ilities: ") + msg)
  { }
virtual ~BadInitialProbs( ) throw ( )
  { }
; } ;
```

This code is used in section 4.23.

Throw this exception if the user specifies a model with the same term appearing twice.

Someday it would be nice to provide more of a clue about what term is duplicated. That will require help from the code that throws the exception.

⟨ *DuplicateTerm 4.23.0.5* ⟩ ≡

```
struct DuplicateTerm : public std ::invalid_argument { DuplicateTerm( ) :
        std::invalid_argument(std::string("Duplicate terms specified: "))
  { }
virtual ~DuplicateTerm( ) throw ( )
  { }
; } ;
```

This code is used in section 4.23.

User has specified an unknown term. Currently, only applicable to path-dependent history terms.

⟨ *UnknownTerm 4.23.0.6* ⟩ ≡

```
struct UnknownTerm : public std ::invalid_argument
{
UnknownTerm(const std::string & term) :
      std::invalid_argument(std::string("Unrecognized␣term:␣") + term)
  { }
}
;
```

This code is used in section 4.23.

Somehow dependency analysis has been unable to get any clear ordering of terms. This is probably a program error, rather than a user input error.

⟨ *TangledDependencies 4.23.0.7* ⟩ ≡

```
struct TangledDependencies : public std ::logic_error
  {
  TangledDependencies(const std::string & msg) : std::logic_error(msg)
    { }
  };
```

This code is used in section 4.23.

These errors are thrown by subclasses of **AbstractDataIterator** *when they run into trouble. The only case currently possible is with the* **SubsetDataIterator**, *but for purposes of providing an exception specification I need a more abstract class.*

This class uses my new, maybe cleverer, way to build up strings in the constructor by instantiating a string stream, streaming to it, and getting the result all in one step. This allows me to use the base exceptions, which require a fixed string.

⟨ *DataIteratorErrors 4.23.0.8* ⟩ ≡

```
struct DataIteratorError : public std ::logic_error
{
DataIteratorError(const std::string & msg) : std::logic_error(msg)
  { }
}
;

struct SubsetDataIteratorError : public DataIteratorError
  {   // argument is the bad id
  SubsetDataIteratorError(int id) : DataIteratorError ( static_cast ⟨
        std::ostringstream &> ((std::ostringstream( ) << "SubsetDataIterator␣failed\
        ␣at␣subset␣case␣id␣" << id << ".␣␣␣Likely␣it␣is␣out␣of␣order,␣duplicate,\
        ␣or␣invalid.")).str( ) ) , badid(id)
    { }

  int badid; } ;
```

This code is used in section 4.23.

4.24 Builders

Builders do most of the work constructing objects out of C inputs.

4.24.1 ModelBuilder

Builds an appropriate model, and all the objects it contains.

I try to guard against leaks for errors being thrown.

⟨ *ModelBuilder.h 4.24.1* ⟩ ≡
```
@o ModelBuilder.h
#ifndef ModelBuilder_h
#define ModelBuiler_h 1

#include <memory>
#include <string>
#include <vector>

#include <boost/ptr_container/ptr_list.hpp>
#include <boost/bind.hpp>

#include "basic.h"
#include "Coefficients.h"
#include "HistoryComputer.h"
#include "LinearProduct.h"
#include "Model.h"
#include "MSPathError.h"
#include "Specification.h"

  namespace mspath {

  class ModelBuilder
    { public:
        ⟨ ModelBuilder c'tors 4.24.1.7 ⟩
        ⟨ ModelBuilder::makeModel interface 4.24.1.1 ⟩
    protected:
        ⟨ ModelBuilder::makeInitial interface 4.24.1.11 ⟩
        ⟨ ModelBuilder::makeHistory interfaces 4.24.1.8 ⟩
        ⟨ ModelBuilder::makeHistoryIndirection interface 4.24.1.10 ⟩
        ⟨ ModelBuilder::makeSpecification interface 4.24.1.3 ⟩
        ⟨ ModelBuilder::makeSimpleSpecification interface 4.24.1.4 ⟩
        ⟨ ModelBuilder::fillparvec interface 4.24.1.5 ⟩
        ⟨ ModelBuilder Data 4.24.1.6 ⟩


    } ;

    }   // end mspath
  #endif
```

This code is used in section 9.

The one public interface to this class. This can throw anything the **Model** *constructor can throw.*

⟨ *ModelBuilder::makeModel interface 4.24.1.1* ⟩ ≡

```
std::auto_ptr<Model> makeModel(⟨ ModelBuilder::makeModel arguments 4.24.1.2 ⟩) throw
    (InconsistentModel, BadInitialProbs, OneInitialState, UnknownTerm);
```

This code is used in section 4.24.1.

These are here so I can reuse them in the implementation. Allowed values for ∗history are "TIS" (Time in State), "TIP" (Time in Previous States), "TSO" (Time Since Origin), "LN(TIS)," "LN(TIP)" and "LN(TSO)" for natural logs of the same.

initprobs is ignored currently, and the logic can only handle a single initial state.

The handling of path variables probably will need work if I ever really use path effects on misclassification. Currently path effects on transitions and misclassification share the same path/history variables. This is likely to be inefficient.

∗*npathmisceffs* ≡ 0*, no effect of path on misclassification, is the only case I'm worrying about now.*

⟨ *ModelBuilder::makeModel arguments 4.24.1.2* ⟩ ≡

```
int *misc ,     /* 0 = no misclassification 1 = full misclassification 2 = simple misclassification */
int *qvector ,     /* vectorised matrix of allowed transition indicators */
int *evector ,     /* vectorised matrix of allowed misclassification indicators */
int *constraint ,     /* list of constraints for each covariate */
int *miscconstraint ,     /* list of constraints for each misclassification covariate */
int *baseconstraint ,     /* constraints on baseline transition intensities */
int *basemiscconstraint ,     /* constraints on baseline misclassification probabilities */
int *pathconstraint ,     /* constraints on path effects on intensities */
int *pathmiscconstraint ,     /* constraints on path effects on misclassification */
double *initprobs ,     /* initial state occupancy probabilities */
int *nms ,     /* number of underlying states which can be misclassified */
int *nintens ,     /* number of intensity parameters */
int *nintenseffs ,     /* number of distinct intensity parameters */
int *nmisc ,     /* number of misclassification rates */
int *nmisceffs ,     /* number of distinct misclassification rates */
int *ncovs ,     /* number of covariates on transition rates */
int *ncoveffs ,     /* number of distinct covariate effect parameters */
int *nmisccovs ,     /* number of covariates on misclassification probabilities */
int *nmisccoveffs ,     /* number of distinct misclassification covariate effect parameters */
int *nhistory,     /* number of history dependent variables */
const char **history,     /* names of history-dependent variables */
double *initialOffset,     /* add this to every 0 time in paths */
int *npatheffs,     /* number of distinct path effects on transitions */
int *npathmisceffs     /* number of distinct path effects on misclassification */
```

This code is used in sections 4.24.1.1 and 6.5.1.

Return an appropriate, heap-allocated **Specification**. *Note the order in which this is called is important, because the class keeps other variables updated behind the scenes.*

The constraints should be 1-based and refer only to coefficients within their appropriate sets of coefficients. Both interceptConstraints and covConstraints values should start at 1.

⟨ *ModelBuilder::makeSpecification interface 4.24.1.3* ⟩ ≡

```
std::auto_ptr<Specification> makeSpecification(int ninterceptEff, int nintercept,
    int *interceptConstraints, int ncovEff, int ncov,
    int *covConstraints, int npathEff, int npath, int *pathConstraints, int *permissible, const
    TIndirect1D *pathIndirect, bool useMisclassification = false);

  // helper
std::auto_ptr<SlopeCoefficients> makeSlope(Double1D& effective, int neffective, int ncov,
    int nterms, int *constraints);
```

This code is used in section 4.24.1.

Here's a simpler version of the previous method, for those cases in which the fixed probabilities are specified a priori.

⟨ *ModelBuilder::makeSimpleSpecification interface 4.24.1.4* ⟩ ≡

```
std::auto_ptr<SimpleSpecification> makeSimpleSpecification(int ninterceptEff, int nintercept,
    int *interceptConstraints, int *permissible);
```

This code is used in section 4.24.1.

Fill a parameter vector with either the current values from the optimisation or the fixed initial values.

⟨ *ModelBuilder::fillparvec interface 4.24.1.5* ⟩ ≡

```
void fillparvec(Double1D& parvec,    /* named vector to fill (e.g. intens = baseline intensities) */
 int ni    /* length of parvec */
);
```

This code is used in section 4.24.1.

This includes data from the original C call and working data. This class does not own the pointed to data.

⟨ *ModelBuilder Data 4.24.1.6* ⟩ ≡

```
const double *myparams;    /* full parameter vector—free parameters only */
const double *myallinits;    /* all initial values */
const size_t myp;    /* number of parameters optimised over */
const size_t mynst;    /* number of Markov states */
const size_t mynfix;    /* number of fixed parameters */
int *myfixedpars;    /* which parameters to fix */

  // track where we are in eating inputs
size_t myifix;    /* current index into fixedpars */
size_t myiopt;    /* current index into params */
size_t myiall;    /* current index into allinits */
```

This code is used in section 4.24.1.

The constructor captures many of the initial arguments, so we don't have to keep passing them around.

⟨ *ModelBuilder c'tors 4.24.1.7* ⟩ ≡

```
ModelBuilder(double *params,      /* full parameter vector—free parameters only */
double *allinits,      /* all initial values */
int np,      /* number of parameters optimised over */
int nst,      /* number of Markov states */
int nfix,      /* number of fixed parameters */
 int *fixedpars      /* which parameters to fix */
) : myparams(params), myallinits(allinits), myp(np), mynst(nst), mynfix(nfix),
        myfixedpars(fixedpars), myifix(0), myiopt(0), myiall(0)
  {}
```

This code is used in section 4.24.1.

This parses the input specification for path-dependent covariates and returns a pointer to an object to add to the model. The pointer may be 0. The primary method is first; it calls the others, in the sequence shown.

Some of the helper classes, defined for the benefit of the standard library algorithms, would be natural to put inside of functions. However, the templates won't let local classes be used, so they have to go in the class itself.

THistoryVector can not be a vector because we require that iterators remain valid after erasing elements.

⟨ *ModelBuilder::makeHistory interfaces 4.24.1.8* ⟩ ≡

```
    // types
  typedef boost:: ptr_list⟨HistoryComputer⟩ THistoryVector; typedef std:: vector < std::string
      > TStringVector ;
    // main method
  std::auto_ptr < Model::TComputerContainer > makeHistory(const char **history,
      /* names of history-dependent variables */
  int nhistory,      /* number of history dependent variables */
   double initialOffset      /* offset times by this amount */
  ) throw (UnknownTerm, TangledDependencies);
    // massage the C arguments into C++
  TStringVector makeRequests(const char **history,      /* names of history-dependent variables */
   int nhistory      /* number of history dependent variables */
  );
    // create all possible instances of HistoryComputer
  std::auto_ptr < THistoryVector > allComputers(double offset = 0.0);
    // fill in first argument according to 2nd
  void makeHistoryStage1(THistoryVector & theComputers, const TStringVector& theRequests)
      throw (UnknownTerm);
```

⟨ *makeHistoryStage1 helper class 4.24.1.9* ⟩

```
    // emit the desired TComputerContainer from stage1 results
  std::auto_ptr < Model::TComputerContainer > makeHistoryStage2(THistoryVector & theComputers)
      throw (TangledDependencies);
```

This code is used in section 4.24.1.

This functional marks all **HistoryComputer***'s that have been requested.*

I use boost::*bind because the corresponding* **std**::*bind2nd*(**std**::*mem_fun_ref*(&**HistoryComputer**::*matches*), *theRequest*) *fails with a "forming a reference to a reference type" error. The problem is theRequest is already a reference, and the standard library tries to make a reference to it, which it can't.*

⟨ *makeHistoryStage1 helper class 4.24.1.9* ⟩ ≡

```
// not sure how portable class in function is
struct Mark
  { Mark(THistoryVector & theComputers) : myComputers(theComputers), myIndex(0)
    { }
    // predicate: search for and mark appropriate history computer
  void operator ()(const std::string & theRequest) { THistoryVector::
      iterator i = std::find_if(myComputers.begin( ), myComputers.end( ),
      boost::bind(&HistoryComputer::matches, _1, theRequest));
  if (i ≡ myComputers.end( ))
    throw (UnknownTerm(theRequest));
  i→makeCovariate(myIndex++); }    // state
  THistoryVector & myComputers;
  size_t myIndex; } ;
```

This code is used in section 4.24.1.8.

Once we've computed the vector of path-dependent history computers, we need to supply proper indirection for the covariates.

⟨ *ModelBuilder::makeHistoryIndirection interface 4.24.1.10* ⟩ ≡

```
std::auto_ptr⟨TIndirect1D⟩makeHistoryIndirection(const
    Model::TComputerContainer & theComputers) const;
```

This code is used in section 4.24.1.

This processes the initial probability vector and checks it. makeModel calls this method.

⟨ *ModelBuilder::makeInitial interface 4.24.1.11* ⟩ ≡

```
State makeInitial(size_t n, double *p)const  throw (OneInitialState, BadInitialProbs);
```

This code is used in section 4.24.1.

5 Meaty Code

I have a feeling I'm going to want to keep the headers and bodies closer together, but for now I'll put some here. This is where the .cc files go, as opposed to the .h in the previous section.

I imagine the interaction of components to path generation as passing information that certain events have happened. Here is an attempt to list the events, in sequence:

- *Got Data.*
- *Got Model with Parameters (likely to happen many times for given data).*
- *Start Path Generation, overall (startSession( )).*
- *Start Processing one case.*
- *Generated Time Steps (startCase( ) is done here so we know time steps are done).*
- *Start Processing a particular path/tree (i.e., from the root) (startTree( )).*
- *Created Node.*
- *Found Node was good (goodNode( )).*
- *Invalid Node (impermissible( )).*
- *Terminal Node (goodPath( )). Note that many individual paths may be generated from a single root; there will generally be many calls to goodPath (and impermissible) between startTree and finishTree.*
- *Finish Processing a particular path/tree (finishTree( )).*
- *Finish processing case (all paths) (finishCase( )).*
- *Finish all Path Generation (finishSession( )).*

Note the use of start/finish rather than begin/end to avoid confusion with the STL container use of begin/end.

5.1 Manager

The **Manager** *delegates almost all action to its components. In particular, it leaves it to the* **PathGenerator** *to manage the* **Recorder**.

⟨ *Manager.cc 5.1* ⟩ ≡
```
@o Manager.cc
#include "Manager.h"

#include <R.h>
#include <sstream>

namespace mspath {
    ⟨ Manager::go 5.1.0.1 ⟩
    ⟨ Manager::setupCount 5.1.0.2 ⟩
    ⟨ Manager::setupLikelihood 5.1.0.3 ⟩
    ⟨ Manager::mainOperation 5.1.0.4 ⟩
    ⟨ Manager::getResults 5.1.0.5 ⟩
    ⟨ Manager::simulate 5.1.0.6 ⟩
}
```

This code is used in section 9.

This is the outer-level, public function.

⟨ *Manager::go 5.1.0.1* ⟩ ≡
```
void Manager::go(double *results, int do_what)
  {    // setup
  if (do_what ≡ 0)
    setupCount( );
  else
    setupLikelihood( );

    // common setup for both
  mypPathGenerator.reset(new PathGenerator(&environment( ), &recorder( ),
      &stateTimeClassifier( ), &successorGenerator( )));

  mainOperation( );
  getResults(results);
  }
```

This code is used in section 5.1.

Setup proper objects for simply counting paths.

⟨ *Manager::setupCount 5.1.0.2* ⟩ ≡
```
void Manager::setupCount( )
  {
  mypRecorder.reset(new SimpleRecorder(&(environment( ))));
  }
```

This code is used in section 5.1.

Setup for computing likelihood.

⟨ *Manager::setupLikelihood 5.1.0.3* ⟩ ≡

```
void Manager::setupLikelihood( )
  {
  mypEvaluator.reset(new Evaluator(&(model( ))));
  mypRecorder.reset(new EvaluatorRecorder(mypEvaluator.get( ), &(environment( ))));
  }
```

This code is used in section 5.1.

The main processing doesn't care what type of recorder we are using.

⟨ *Manager::mainOperation 5.1.0.4* ⟩ ≡

```
void Manager::mainOperation( )
  {
  environment( ).startSession( );
  pathGenerator( ).startSession( );
  while (environment( ).next( ))
    {    // environment is now set to a single person
    timeStepsGenerator( ).makeStepsFor(&(environment( )));
    pathGenerator( ).startCase( );

      // loop over initial true states
    for (State s = 0; s < model( ).nStates( ); s++)
      {
      double p = model( ).initialProbabilities( )[s];
      if (p ≤ 0.0)
        continue;
      StatePoint sp0(s, environment( ).timeSteps( ).front( ).time( ));
      pathGenerator( ).startTree(sp0, p);
      }
    pathGenerator( ).finishCase( );
    }
  pathGenerator( ).finishSession( );

    // debug
    // std::cout << recorder( ) << std::endl;
  }
```

This code is used in section 5.1.

Finally, we return the results.

```
⟨ Manager::getResults 5.1.0.5 ⟩ ≡
    void Manager::getResults(double *results)
      {
      if (hasLikelihood( ))
        *results++ = dynamic_cast⟨EvaluatorRecorder *⟩ (mypRecorder.get( ))→logLikelihood( );
      else
        *results++ = 0.0;
      *results++ = recorder( ).cases( );
      *results++ = recorder( ).goodPaths( );
      *results++ = recorder( ).goodNodes( );
      *results++ = recorder( ).badNodes( );
      *results++ = recorder( ).goodPathNodes( );
      }
```

This code is used in section 5.1.

Perform a simulation.

```
⟨ Manager::simulate 5.1.0.6 ⟩ ≡
    std::auto_ptr < RandomPathGenerator::Results > Manager::simulate( )
      {
      std::auto_ptr < RandomPathGenerator::Results > pResults(new
          RandomPathGenerator::Results);
      mypPathGenerator.reset(new RandomPathGenerator (&environment( ),
          &stateTimeClassifier( ), &model( ), myIsExactTimeAbsorb, pResults.get( )));
      environment( ).startSession( );
      pathGenerator( ).startSession( );
      ⟨ get initial state 5.1.0.7 ⟩
      ⟨ simulation loop 5.1.0.8 ⟩
      pathGenerator( ).finishSession( );
      return pResults;
      }
```

This code is used in section 5.1.

Right now we know there is only one initial state, but this should work even if there are more, as long as the initial probabilities are valid. If the initial state is impossible given the data—which is particularly likely if we have multiple possible initial states, we have a problem!

```
⟨ get initial state 5.1.0.7 ⟩ ≡
    State initialState = environment( ).randomDraw(model( ).initialProbabilities( ));
```

This code is used in section 5.1.0.6.

⟨ *simulation loop* 5.1.0.8 ⟩ ≡
```
while (environment( ).next( ))
  {   // environment is now set to a single person
  timeStepsGenerator( ).makeStepsFor(&(environment( )));
  pathGenerator( ).startCase( );
  StatePoint sp0(initialState, environment( ).timeSteps( ).front( ).time( ));
  pathGenerator( ).startTree(sp0, 1.0);
  pathGenerator( ).finishCase( );
  }
```

This code is used in section 5.1.0.6.

5.2 Path Generation

This is perhaps the core operation of the program. Currently, the timeSteps in the environment do not need to be set up until you call startTree, though that might change down the line to a requirement they be set up before startCase.

It does not matter what is in the environment's path; the generator will blow it away.

We group the **AbstractPathGenerator** *and corresponding* **PathGenerator** *code together in what follows, though some of the section names suggest they are only for* **PathGenerator**.

⟨ *PathGenerator.cc* 5.2 ⟩ ≡
```
  @o PathGenerator.cc
#include <stdexcept>

#include "PathGenerator.h"
  namespace mspath {
      ⟨ PathGenerator state changes 5.2.0.4 ⟩
  ⟨ PathGenerator::startTree 5.2.0.1 ⟩
  ⟨ PathGenerator::nextBranch 5.2.0.2 ⟩
  ⟨ RandomPathGenerator implementation 5.2.0.5 ⟩
      }   // end namepace mspath
```

This code is used in section 9.

Do overall generation of a tree of paths from a single root. We assume the initial point is valid.

⟨ *PathGenerator::startTree 5.2.0.1* ⟩ ≡

```
void AbstractPathGenerator::startTree(const StatePoint& sp0, double p) throw
    (std::runtime_error, std::logic_error)
  {
  environment( ).pathClear( );    // path must be clear for nextTimePoint to work right
  environment( ).pathPush(sp0, nextTimePoint( ));
    // now that it's in the environment, others can see it
  }

void PathGenerator::startTree(const StatePoint& sp0, double p) throw (std::runtime_error,
    std::logic_error)
  {
  Super::startTree(sp0, p);    // to finish the setup.
  stateTimeClassifier( ).startTree(&(environment( )));

    // could check validity of initial point here

  recorder( ).startTree(p);

    // main action
  if (¬stateTimeClassifier( ).isTerminal(sp0, &(environment( ))))
    nextBranch( );
  else
    {    // degenerate case with no steps
    recorder( ).goodPath(environment( ).path( ));
    environment( ).pathPop( );
    }

    // done
  recorder( ).finishTree( );
  stateTimeClassifier( ).finishTree(&(environment( )));
  }
```

This code is used in section 5.2.

Well, we're just too excited to deal with any more preliminaries. Here's the central algorithm for recursing over all possible paths.

This has no arguments. Current state and recursion are managed via myPath and the stack. Only a single thread can run against a single instance of a class, though with multiple class instances multiple threads are possible.

The function assumes, as a precondition, that a valid path is present up to the end of the path. It also assumes the end of the path is not already in a terminal state, i.e., there's room to grow. Finally, it assumes that something has already generated a completed set of timeSteps.

nextBranch generates all possible successor paths, in turn. It goes depth-first, where time is depth. While generating paths, **StateTimeClassifier** *may indicate a path has become impossible; it is thrown out and iteration proceeds to the next. Or,* **StateTimeClassifier** *may indicate a path is complete (when it discovers a node is "terminal"). In that case, the completion is noted in myRecorder, which also notes other information as relevant. TransitionGenerator is used to get the list of possible successor states given the current end-state of the path.*

Processing of the generated paths occurs in myRecorder, though that is a side-effect from the standpoint of this class.

⟨ *PathGenerator::nextBranch 5.2.0.2* ⟩ ≡

```
void PathGenerator::nextBranch( )
  {
    // get references to last time
  const Node& baseNode = environment( ).currentNode( );
      // not needed const TimePoint& baseTimePoint = baseNode.timePoint( );
    // and info on the new time
  const TimePoint& newTimePoint = nextTimePoint( );
  Time newTime = newTimePoint.time( );
  const State n= successorGenerator( ).lastState( );
  const State startState= baseNode.state( );
  for (State endState = successorGenerator( ).firstState( ); endState ≤ n; endState++)
    {
    if (¬successorGenerator( ).isPossibleTransition(startState, endState))
      continue;
    ⟨ Path Generator Inner Loop Body 5.2.0.3 ⟩
    }
  }
```

This code is used in section 5.2.

We now have a new state and time. We need to check for various exceptional conditions, and otherwise create a new node and recurse.

⟨ *Path Generator Inner Loop Body* 5.2.0.3 ⟩ ≡

```
StatePoint nextSP(endState, newTime);
if (stateTimeClassifier( ).isOK(nextSP, newTimePoint, &(environment( ))))
  {    // ready to go
  environment( ).pathPush(nextSP, newTimePoint);
  recorder( ).goodNode( );
  if (stateTimeClassifier( ).isTerminal(nextSP, &(environment( ))))
    recorder( ).goodPath(environment( ).path( ));    // we have a winner!
  else
    nextBranch( );    // recurse
    // clean up in any case
  environment( ).pathPop( );
  }
else
  {
  recorder( ).impermissible( );
  }
```

This code is used in section 5.2.0.2.

Here we deal with the standard state change notifications.

⟨ *PathGenerator state changes 5.2.0.4* ⟩ ≡

```
void AbstractPathGenerator::startSession( )
  { }
void PathGenerator::startSession( )
  {
  Super::startSession( );
  mypRecorder→startSession( );
  }
void AbstractPathGenerator::finishSession( )
  { }
void PathGenerator::finishSession( )
  {
  Super::finishSession( );
  mypRecorder→finishSession( );
  }
void AbstractPathGenerator::startCase( )
  {
  environment( ).path( ).reserve(environment( ).timeSteps( ).size( ));
  }
void PathGenerator::startCase( )
  {
  Super::startCase( );
  recorder( ).startCase( );
  }
void AbstractPathGenerator::finishCase( )
  { }
void PathGenerator::finishCase( )
  {
  Super::finishCase( );
  recorder( ).finishCase( );
  }
```

This code is used in section 5.2.

This code generates a single random path for a case.

⟨ *RandomPathGenerator implementation 5.2.0.5* ⟩ ≡

⟨ *RandomPathGenerator state changes 5.2.0.6* ⟩
⟨ *RandomPathGenerator::startTree 5.2.0.7* ⟩
⟨ *RandomPathGenerator::nextStep 5.2.0.8* ⟩
⟨ *RandomPathGenerator::addLastPoint 5.2.0.9* ⟩
⟨ *RandomPathGenerator::recordObservation 5.2.0.10* ⟩

This code is used in section 5.2.

Here are the trivial state changes.

⟨ *RandomPathGenerator state changes 5.2.0.6* ⟩ ≡

```
void RandomPathGenerator::startSession( )
  {
  Super::startSession( );
  results( ).clear( );
  }
void RandomPathGenerator::startCase( )
  {
  Super::startCase( );
  }
void RandomPathGenerator::finishCase( )
  {
  Super::finishCase( );
  }
void RandomPathGenerator::finishSession( )
  {
  Super::finishSession( );
  }
```

This code is used in section 5.2.0.5.

Here we kick off the main generation of the path. It doesn't make sense to call this more than once per case, but we don't check for that since we only allow a single starting state at the moment.

⟨ *RandomPathGenerator::startTree* 5.2.0.7 ⟩ ≡

```
void RandomPathGenerator::startTree(const StatePoint& sp0, double p) throw
        (std::runtime_error, std::logic_error)
  {
  Super::startTree(sp0, p);
    // initial point is now on the path
    // to finish the setup.
  stateTimeClassifier().startTree(&(environment()));
  model().fillModelData(environment().currentNode(), environment());

    // could check validity of initial point here

  if (environment().matchesObservation())
    recordObservation();

    // main action
  while (¬stateTimeClassifier().isTerminal(environment().currentNode().statePoint(),
        &(environment())))
    nextStep();

    // Put final point in results if necessary.
    // If it matched an observation, it's already recorded.
  if (¬environment().matchesObservation())
    addLastPoint();

    // done
  environment().pathClear();
  stateTimeClassifier().finishTree(&(environment()));
  }
```

This code is used in section 5.2.0.5.

We now have a path with a non-terminal node at the end. This step generates the next point on the path and handles the observation, if any. The current node is always the last node in this algorithm.

For this to work, it is important that we only use the history through the previous step to calculate probabilities for the next one. That is the way the **Model** *and in particular the* **PathCovariates** *behave.*

⟨ *RandomPathGenerator::nextStep 5.2.0.8* ⟩ ≡

```
void RandomPathGenerator::nextStep( )
  {
  const State lastState= environment( ).currentNode( ).state( );
  const TimePoint& nextTP = nextTimePoint( );
  Time nextTime = nextTP.time( );    // we push a no-change value onto path
  StatePoint nextSP = StatePoint (lastState, nextTime);
  environment( ).pathPush(nextSP, nextTP);

    // and then let model possibly modify it
  model( ).simulatePath(environment( ));

    // based on the simulated state, compute path-dependent vars
  model( ).fillModelData(environment( ).currentNode( ), environment( ));

    // create a an observation if appropriate
  if (environment( ).matchesObservation( ))
    recordObservation( );

  }
```

This code is used in section 5.2.0.5.

We've now encountered a terminal but non-observed node, so we must be in an absorbing state. Figure out when it is observed in the simulation and the accompanying values. Because the observation model could depend on the complete path history (though that's unlikely in practice), we generate a path out to through the next observation point.

⟨ *RandomPathGenerator::addLastPoint 5.2.0.9* ⟩ ≡

```
void RandomPathGenerator::addLastPoint( )
  {

  if (¬isExact( ))
    {
    const State finalState= environment( ).currentNode( ).state( );
    do
      {
      const TimePoint& tp = nextTimePoint( );
      environment( ).pathPush(StatePoint (finalState, tp.time( )), tp);
      model( ).fillModelData(environment( ).currentNode( ), environment( ));
      }
    while (¬environment( ).matchesObservation( ));
        // there should always be an observation at the end
    }
  recordObservation( );
  }
```

This code is used in section 5.2.0.5.

Records one observation in the output dataset, using the current node in the environment. The current timePoint may not be an observed time in the original data, but it will be one in the simulated data.

⟨ *RandomPathGenerator::recordObservation* 5.2.0.10 ⟩ ≡

```
void RandomPathGenerator::recordObservation( )
  {
  ObsState s = −1;    // i.e., unobserved state by default
  if (environment( ).data( ).hasObservedState(environment( ).iObservation( )))
    s = static_cast⟨ObsState⟩ (model( ).simulateObservation(environment( )));
  results( ).push_back(SimResult (s, environment( ).timePoint( ).time( ),
      environment( ).iObservation( )));
  }
```

This code is used in section 5.2.0.5.

5.3 SuccessorGenerator

The implementation obviously uses knowledge about exactly what kind of type **State** *is. The mixing of it and* **size_t***'s may not be entirely a happy one.*

Caching is a potential optimization here, since for the entire session the answer depends only on the baseNode.state( ).

⟨ *SuccessorGenerator.cc* 5.3 ⟩ ≡

```
@o SuccessorGenerator.cc
#include "SuccessorGenerator.h"
  namespace mspath
    {
      // For the current model, time is irrelevant
    void SuccessorGenerator::nextStates(std:: vector ⟨State⟩ & next, const
        Node& baseNode, Time newTime)
    {
    next.clear( );
    size_t n = model( )→nStates( );
    const State startState= baseNode.state( );
    State endState;
    for (endState = 0U; endState < static_cast⟨State⟩ (n); endState++)
      if (startState ≡ endState ∨ model( )→isPossibleTransition(startState, endState))
        next.push_back(endState);
    }
  }    // end namespace mspath
```

This code is used in section 9.

5.4 StateTimeClassifiers

This could do a lot of clever things to cut off bad paths early, but at the moment it doesn't.

The implementations do not and should not require that the currentNode of the environment be set to the **StatePoint** *under consideration.*

It would perhaps be less prone to rounding error if I used the index of the **TimePoint** *instead of the time.*

⟨ *StateTimeClassifier.cc 5.4* ⟩ ≡

```
@o StateTimeClassifier.cc
#include "StateTimeClassifier.h"

#include <cassert>

#include "TimePoint.h"
#include "TimeSteps.h"

  ⟨ StateTimeClassifier::c'tor 5.4.0.1 ⟩
  ⟨ StateTimeClassifier::startTree 5.4.0.5 ⟩
  ⟨ StateTimeClassifier::isTerminal 5.4.0.3 ⟩
  ⟨ StateTimeClassifier::isOK 5.4.0.4 ⟩


  ⟨ PickyStateTimeClassifier::c'tor 5.4.0.2 ⟩
  ⟨ PickyStateTimeClassifier::startTree 5.4.0.6 ⟩
```

This code is used in section 9.

We need to object if exact observation times are specified for inexactly measured states. Currently, I never throw an exception, though my subclasses may.

⟨ *StateTimeClassifier::c'tor 5.4.0.1* ⟩ ≡

```
mspath::StateTimeClassifier::StateTimeClassifier(Model *pModel)

throw (mspath::InconsistentModel) :
  mypModel(pModel), myFirstTime(pModel→nStates( ))
    { }
```

This code is used in section 5.4.

Complain if exact time for a fuzzy state.

⟨ *PickyStateTimeClassifier::c'tor 5.4.0.2* ⟩ ≡

```
mspath::PickyStateTimeClassifier::PickyStateTimeClassifier(Model *pModel)
  throw (mspath::InconsistentModel) :
    StateTimeClassifier(pModel)
      {
      for (State s = 0U; s < pModel→nStates( ); ++s)
        {
        if (¬pModel→isAbsorbing(s))
          continue;
        for (State j = 0U; j < pModel→nStates( ); ++j)
          {
          if (s ≡ j)
            continue;
          if (pModel→isPossibleObservation(s, j) ∨ pModel→isPossibleObservation(j, s))
            throw InconsistentModel("Absorbing␣States␣with␣exa\
                ct""␣times␣must␣be␣measured␣exactly.");
          }
        }
      }
```

This code is used in section 5.4.

This code used to test for entry into an absorbing state and report it as terminal. While it is terminal substantively, we must continue the path to see if it is consistent with subsequent observations. Even if it is consistent, we need to adjust for the probability of those subsequent observations.

For **PickyStateTimeClassifier** *if the path enters and absorbing state any time other than when it was observed, it will fail because of the isOK( ) test.*

⟨ *StateTimeClassifier::isTerminal 5.4.0.3* ⟩ ≡

```
bool mspath::StateTimeClassifier::isTerminal(const StatePoint& sp,
      Environment *pEnv) const
  {
  return sp.time( ) ≥ myTerminalTime;
  }
```

This code is used in section 5.4.

See the interface comments for the correct interpretation of the response values. For now, returns `true` *in all cases except those on* **TimePoint***'s that match an observation with state inconsistent with observed state and measurement model.*

In the future, it might be more aggressive about knocking states off.

The minimal requirements of this class are

- *It must eliminate paths that are inconsistent with the observed data and the measurement model.*
- *It must return* `true` *for all* **StatePoint***'s that are part of a valid path.*

⟨ *StateTimeClassifier::isOK 5.4.0.4* ⟩ ≡

```
// assumes sp.time( ) ≡ associated TimePoint's time.
bool mspath::StateTimeClassifier::isOK(const StatePoint& sp, const TimePoint& tp,
    Environment *pEnv) const
{
assert(sp.time( ) ≡ tp.time( ));
if (model( ).isAbsorbing(sp.state( )))
  return sp.time( ) ≥ myFirstTime[sp.state( )];
return ¬tp.matchesObservation( ) ∨ model( ).isPossibleObservation(sp.state( ),
    pEnv→data( ).state(tp.iObservation( )));
}
```

This code is used in section 5.4.

Many possible optimizations could go here. There were extensive optimizations before 6/28/05, but they weren't quite right.

```
⟨ StateTimeClassifier::startTree 5.4.0.5 ⟩ ≡
    void mspath::StateTimeClassifier::startTree(Environment *thepEnv)
      {
      const TimeSteps& ts = thepEnv→timeSteps( );
      const Data& data = thepEnv→data( );
      myTerminalTime = ts.back( ).time( );

        // find first time we could be in absorbing states
        // given the observed data
      TimeSteps::const_reverse_iterator rb, re, ri, rlag;
      rb = ts.rbegin( );
      re = ts.rend( );
      for (State s = 0U; s < model( ).nStates( ); ++s)
        {
        if (model( ).isAbsorbing(s))
          {
          myFirstTime[s] = ts.front( ).time( );     // value if fall through loop
          rlag = rb;
          for (ri = rb; ri ≠ re; ++ri)
            {
            if (¬(ri→matchesObservation( ) ∧ data.hasObservedState(ri→iObservation( ))))
              {
              rlag = ri;
              continue;
              }
            if (¬model( ).isPossibleObservation(s, data.state(ri→iObservation( ))))
              {
              if (ri ≠ rb)
                myFirstTime[s] = rlag→time( );
              else
                myFirstTime[s] = ri→time( )+static_cast⟨Time⟩(1);
              break;
              }     // match an observation and it's possible
            rlag = ri;
            }
          }
        }

      }
```

This code is used in section 5.4.

Find first observation in absorbing state. Transition must be at exactly that time.

⟨ *PickyStateTimeClassifier::startTree 5.4.0.6* ⟩ ≡

```
void mspath::PickyStateTimeClassifier::startTree(Environment *thepEnv)
  {
  const TimeSteps& ts = thepEnv→timeSteps( );
  const Data& data = thepEnv→data( );
  myTerminalTime = ts.back( ).time( );

    // find first time we could be in absorbing states
    // given the observed data
  TimeSteps::const_reverse_iterator rb, re, ri, rlag;
  rb = ts.rbegin( );
  re = ts.rend( );
  for (State s = 0U; s < model( ).nStates( ); ++s)
    {
    if (model( ).isAbsorbing(s))
      {
      myFirstTime[s] = ts.front( ).time( );     // value if fall through loop
      rlag = rb;
      for (ri = rb; ri ≠ re; ++ri)
        {
        if (¬data.hasObservedState(ri→iObservation( )))
          {
          continue;
          }
        if (¬model( ).isPossibleObservation(s, data.state(ri→iObservation( ))))
          {
          if (ri ≠ rb)
            myFirstTime[s] = rlag→time( );
          else
            myFirstTime[s] = ri→time( )+static_cast⟨Time⟩(1);
          break;
          }    // match an observation and it's possible
        rlag = ri;
        }
      }
    }

  }
```

This code is used in section 5.4.

5.5 Model

⟨ *Model.cc 5.5* ⟩ ≡
```
@o Model.cc
#include "Model.h"
#include <iostream>
```
⟨ *Model::fillModelData 5.5.0.1* ⟩
⟨ *Model::evaluate 5.5.0.2* ⟩
⟨ *Model::validate 5.5.0.3* ⟩
⟨ *Model::operator¡¡ 5.5.0.4* ⟩
⟨ *Model::simulatePath 5.5.0.5* ⟩
⟨ *Model::simulateObservation 5.5.0.6* ⟩

This code is used in section 9.

Fill in path-dependent data, if any.

Consider changing interface of **HistoryComputer** *to use the* **ModelData** *obtained from theEnv.activeModelData.*

⟨ *Model::fillModelData 5.5.0.1* ⟩ ≡
```
void mspath::Model::fillModelData(Node& theNode, Environment& theEnv)
  {
  if (mypComputerContainer.get( ) ≡ 0)
    return;
  for (size_t i = 0U; i < mypComputerContainer→size( ); ++i)
    {
    (∗mypComputerContainer)[i].evaluate(theEnv, theNode);
    }
  }
```

This code is used in section 5.5.

Evaluate the likelihood (not the log-likelihood). Assumes all prior data, and all path-dependent data, have been filled in.

In something of a special case, we don't consider the measurement model on the first observation. This fits our data, but may not be very general.

Wow! I evaluate the entire specification, but I only need one term. This needs optimization.

⟨ *Model::evaluate 5.5.0.2* ⟩ ≡

```
void mspath::Model::evaluate(Node& theNode, Environment& theEnv) throw
        (DataModelInconsistency)
  {   // compute transition probability
  if (theNode.isRoot( ))
    {
    if (theNode.state( ) ≠ myInitialState)
      throw DataModelInconsistency (theEnv.id( ), "Initial␣Path␣State␣Incons\
          istent␣with""Model");
    theEnv.activeEvaluationData(theNode) = 1.0;
    return;   // no measurement error on root
    }
  else
    {
    Node& lastNode = *(theNode.previous( ));
    double lastLik = theEnv.activeEvaluationData(lastNode);
    const Double2D& trans = mypTransition→evaluate(theEnv);
    theEnv.activeEvaluationData(theNode) = lastLik * trans(lastNode.state( ), theNode.state( ));
    }
  if (mypMisclassification.get( ) ≡ 0 ∨ ¬theEnv.hasObservedState(theNode))
    return;   // adjust for likelihood of observation given error
  const Double2D& misc = mypMisclassification→evaluate(theEnv);
  double obs = misc(theNode.state( ), static_cast⟨State⟩ (theEnv.observedState(theNode)));
  if (obs ≡ 0.0)   // could be user or program error below
    throw DataModelInconsistency (theEnv.id( ),
        "Generated␣path␣has␣an␣impossible␣observation.");
  theEnv.activeEvaluationData(theNode) *= misc(theNode.state( ),
      static_cast⟨State⟩ (theEnv.observedState(theNode)));
  }
```

This code is used in section 5.5.

Check that the model has been setup with sensible values. Note this check assumes the individual objects given to the c'tor are sane, and only checks their consistency with each other.

This is a protected method, invoked from the c'tor.

⟨ *Model::validate* 5.5.0.3 ⟩ ≡

```
void mspath::Model::validate( )const  throw (InconsistentModel, BadInitialProbs,
        OneInitialState)
  {
  if (mypTransition.get( ) ≡ 0)
    throw InconsistentModel ("Model␣requires␣at␣least␣a␣Transition␣Specification");
  if (myInitialState ≥ nStates( ))
    throw InconsistentModel ("Initial␣State␣is␣too␣high");
  if (mypMisclassification.get( ) ≠ 0)
    {
    if (mypMisclassification→nStates( ) ≠ mypTransition→nStates( ))
      throw InconsistentModel ("Model␣transition␣and␣misc\
          lassification""␣specifications␣use␣different␣number␣of␣states");
          // could also check if initialState was feasible in
        // misclassification model. For now, we catch it when we try to
        // evaluate the path
    }
  if (mypComputerContainer.get( ))
    {
    const TComputerContainer & cc = ∗mypComputerContainer;
    const size_t n= cc.size( );
    for (size_t i = 0U; i < n; ++i)
      {
      if (cc[i].isCovariate( ))
        if (cc[i].covariateIndex( ) ≥ n)
          throw InconsistentModel ("Path-Dependent␣Variable␣t\
              hinks␣it""␣lives␣at␣an␣impossibly␣high␣index");
      }
    }

  }
```

This code is used in section 5.5.

Well, really a friend function.

⟨ *Model::operator¡¡ 5.5.0.4* ⟩ ≡

```
std::ostream & mspath::operator <<(std::ostream & ostr, const mspath::Model & model)
  {
  ostr << "A␣model␣with␣" << model.nStates( ) << "␣states,\
      ␣initial␣state␣of␣" << model.myInitialState << std::endl <<
      "Transitions␣governed␣by␣" << *(model.mypTransition);
  if (model.mypMisclassification.get( ) ≡ 0)
    ostr << "No␣misclassification." << std::endl;
  else
    ostr << "Misclassification␣governed␣by␣" << *(model.mypMisclassification);
  if (model.nPathDependentVariables( ) > 0U)
    {
    ostr << model.nPathDependentVariables( ) << "␣path-dependent␣variables:" << std::endl;
    mspath::Model::TComputerContainer::const_iterator i;
    for (i = model.mypComputerContainer→begin( ); i ≠ model.mypComputerContainer→end( );
          ++i)
      ostr << (*i) << std::endl;
    }
  return ostr;
  }
```

This code is used in section 5.5.

Simulate a single step along the path of true states. This assumes the current node in the environment is the one to get a simulated state. That state will be inserted into that node. The time and data are taken as given. Note that fillModelData( ) has not been run at this point.

This assumes the current node's provision state is equal to the state of the last node, i.e., the one out of which we are transitioning.

This should never be called on the root node.

Like the evaluate( ) method, this one does more computation than strictly necessary. In this case it's perhaps more wasteful, since in the former we may get some value out of reusing the cache as we go through all possibilities.

⟨ *Model::simulatePath 5.5.0.5* ⟩ ≡

```
void mspath::Model::simulatePath(Environment& env)
  {
  const Double2D& trans = mypTransition→evaluate(env);
  State s = env.randomDraw(trans.row(env.currentNode( ).state( )));
  env.currentNode( ).setState(s);
  }
```

This code is used in section 5.5.

Here we get a random observation for the current true state. The algorithm ignores whether the environment (i.e., the true data) thinks we are on an observation or not.

⟨ *Model::simulateObservation* 5.5.0.6 ⟩ ≡

```
mspath::Statemspath::Model::simulateObservation(Environment& env)
  {
  State s = env.currentNode( ).state( );
  if (mypMisclassification.get( ) ≡ 0)
    return s;
  const Double2D& misc = mypMisclassification→evaluate(env);
      // breaking out the steps to aid debugging
  s = env.randomDraw(misc.row(s));
  return s;
  }
```

This code is used in section 5.5.

5.6 Specification

⟨ *Specification.cc* 5.6 ⟩ ≡

```
@o Specification.cc
#include <cmath>    // for exp
#include <iostream>
#include "Specification.h"
    // next 2 were forward declared in header to reduce complexity
#include "Environment.h"
#include "LinearProduct.h"
```

⟨ *SimpleSpecification constructor implementation* 5.6.0.1 ⟩ ⟨ *Specification::evaluate* 5.6.0.2 ⟩ ⟨ *Specification::computeResult* 5.6.0.3 ⟩ ⟨ *all Specification streaming operators* 5.6.0.4 ⟩

This code is used in section 9.

Because the values are constant, we do all the setup and checking in the constructor.

The code is borrowed from that of **Specification**::*computeResults below and then simplified. Someday further code factoring could make things even more lovely.*

⟨ *SimpleSpecification constructor implementation 5.6.0.1* ⟩ ≡

```
mspath::SimpleSpecification::SimpleSpecification(ConstantLinearProduct *thepLP,
    size_t thenStates, Bool2D *thepPermissible)
  throw (InconsistentModel) :
    AbstractSpecification(thepPermissible), mypLP(thepLP), myResult(0.0, thenStates, thenStates)
    {
    const Double1D& r0 = linearProduct( ).evaluate( );
    if (r0.max( ) > 1.0)
      throw InconsistentModel("SIMPLE Measurement error"" probabilities can not ex\
          ceed 1.");
    if (r0.min( ) < 0.0)
      throw InconsistentModel("SIMPLE Measurement error"" probabilities can't be u\
          nder 0.");

    size_t k = 0U;
    size_t i, j;
    const size_t n= thenStates;
    double sum;
    for (i = 0U; i < n; i++)
      {
      sum = 1.0;    // for diagonal element
      for (j = 0U; j < n; j++)
        {   // sum row
        if (i ≠ j)
          {
          if (permissible( )(i, j))
            {
            myResult(i, j) = r0[k];
            sum -= r0[k++];
            }
          }
        if (sum ≤ 0.0)
          throw InconsistentModel("SIMPLE Measurement error"" off-diagonal row probabi\
              lities sum to => 1.");
        myResult(i, i) = sum;
        }
      }
    }
```

This code is used in section 5.6.

This is the client-level interface.

⟨ *Specification::evaluate 5.6.0.2* ⟩ ≡

```
const mspath::Double2D & mspath::Specification::evaluate(Environment& theEnv) const
  {
  Scratch& s = getScratchData⟨Scratch⟩(theEnv, *this);
  if (linearProduct( ).isChanged(theEnv, s.pMemento))
    {
    computeResult(s.result, linearProduct( ).evaluate(theEnv));
    linearProduct( ).memento(theEnv, &(s.pMemento));
    }
  return s.result;
  }
```

This code is used in section 5.6.

Unconditionally do the core transformation of this class. This is where the time goes in computation.

⟨ *Specification::computeResult 5.6.0.3* ⟩ ≡

```
void mspath::Specification::computeResult(Double2D& theResult,
        const Double1D& theLinearResult) const
  {
#ifdef DEBUG
  std::cout << "Total Linear Product:" << std::endl << theLinearResult << std::endl;
#endif
  Double1D r0(std::exp(theLinearResult));
  size_t k = 0U;
  size_t i, j;
  size_t n = theResult.nrows( );     // it better be square
  double sum;
  for (i = 0U; i < n; i++)
    {
    sum = 1.0;     // for diagonal element
    for (j = 0U; j < n; j++)
      {     // sum row
      if (i ≠ j)
        {
        if (permissible( )(i, j))
          {
          theResult(i, j) = r0[k];
          sum += r0[k++];
          }
        }
      }

      // compute final values
    for (j = 0; j < n; j++)
      {
      if (i ≠ j)
        {
        if (permissible( )(i, j))
          theResult(i, j) /= sum;     // since theResult initialized to 0
          // no need to set it to 0
        }
      else
        {     // diagonal
        theResult(i, j) = 1.0 / sum;
        }
      }
    }
#ifdef DEBUG
  std::cout << "Final outputs:" << std::endl << theResult << std::endl;
#endif
  }
```

This code is used in section 5.6.

⟨ *all Specification streaming operators 5.6.0.4* ⟩ ≡
⟨ *AbstractSpecification::operator¡¡ 5.6.0.5* ⟩⟨ *SimpleSpecification::operator¡¡ 5.6.0.7* ⟩⟨ *Specification::operator¡¡ 5.6.0.6* ⟩

This code is used in section 5.6.

Not really useful on its own, but can be called by subclasses. That way if I ever get fancy, I only need to do it one place.

⟨ *AbstractSpecification::operator¡¡ 5.6.0.5* ⟩ ≡
```
std::ostream & mspath::operator <<(std::ostream & ostr,
        const mspath::AbstractSpecification & spec)
  {
  ostr << spec.permissible( ) << std::endl;
  return ostr;
  }
```

This code is used in section 5.6.0.4.

⟨ *Specification::operator¡¡ 5.6.0.6* ⟩ ≡
```
std::ostream & mspath::operator <<(std::ostream & ostr, const mspath::Specification & spec)
  {
  ostr << "Multinomial␣Logit␣Specification␣for␣" << std::endl << spec.permissible( ) <<
      std::endl;
  spec.linearProduct( ).printOn(ostr) << std::endl;
  return ostr;
  }
```

This code is used in section 5.6.0.4.

It doesn't seem worthwhile to give the gory details of this one.

⟨ *SimpleSpecification::operator¡¡ 5.6.0.7* ⟩ ≡
```
std::ostream & mspath::operator <<(std::ostream & ostr,
        const mspath::SimpleSpecification & spec)
  {
  ostr << "Simple␣Specification:" << std::endl << spec.myResult << std::endl;
  return ostr;
  }
```

This code is used in section 5.6.0.4.

5.7 LinearProduct

Note I leave it to the **Coefficients** *to be clever about avoiding unnecessary computation. We pull the covariates out of an appropriate thread-specific place.*

⟨ *LinearProduct.cc 5.7* ⟩ ≡
```
@o LinearProduct.cc
#include "LinearProduct.h"
#include <iostream>
```
⟨ *ConstantLinearProduct Implementation 5.7.1* ⟩
⟨ *DataLinearProduct Implementation 5.7.2* ⟩
⟨ *PathDependentLinearProduct Implementation 5.7.3* ⟩
⟨ *SumLinearProducts Implementation 5.7.4* ⟩

This code is used in section 9.

5.7.1 ConstantLinearProduct

⟨ *ConstantLinearProduct Implementation 5.7.1* ⟩ ≡
```
std::ostream & mspath::ConstantLinearProduct::printOn(std::ostream & ostr) const
  {
  ostr << "Constant␣Terms␣" << *mypCoefficients << std::endl;
  return ostr;
  }
```

This code is used in section 5.7.

5.7.2 DataLinearProduct

It's not clear that the caching is actually worth the trouble, but I do it for now.

⟨ *DataLinearProduct Implementation 5.7.2* ⟩ ≡

```
bool mspath::DataLinearProduct::isChanged(Environment& theEnv,
        ScratchData *thepMemento) const
  {
  return scratch(theEnv).covariates.isChanged(theEnv, thepMemento);
  }

void mspath::DataLinearProduct::memento(Environment& theEnv,
        ScratchData **theppMemento)
  {
  scratch(theEnv).covariates.memento(theppMemento);

  }

const mspath::Double1D & mspath::DataLinearProduct::evaluate(Environment& theEnv) const
  {
  ScratchDataLinearProduct& s = scratch(theEnv);
  if (s.covariates.isChanged(theEnv, s.pMemento))
    {
    s.results = mypCoefficients→multiply(theEnv, s.covariates);
    s.covariates.memento(&(s.pMemento));
    }
  return s.results;

  }

mspath::DataLinearProduct::ScratchDataLinearProduct&
        mspath::DataLinearProduct::scratch(Environment& theEnv) const
  {
  if (¬theEnv.hasScratchData(scratchKey( )))
    theEnv.setScratchData(scratchKey( ),
          // because next c'tor has 2 args, we can't use the default methods
      new ScratchDataLinearProduct (theEnv, *this));
  return static_cast⟨ScratchDataLinearProduct&⟩ (theEnv.getScratchData(scratchKey( )));
  }

std::ostream & mspath::DataLinearProduct::printOn(std::ostream & ostr) const
  {
  ostr << "␣Linear␣Product␣of␣";
  if (myUseMisclassificationData)
    ostr << "Misclassification␣Covariates";
  else
    ostr << "Regular␣Covariates";
  ostr << "␣and␣" << *mypCoefficients << std::endl;
  return ostr;
  }
```

This code is used in section 5.7.

5.7.3 PathDependentLinearProduct

⟨ *PathDependentLinearProduct Implementation 5.7.3* ⟩ ≡

```
bool mspath::PathDependentLinearProduct::isChanged(Environment& theEnv,
    ScratchData *thepMemento) const
  {
  ScratchPathDependentLinearProduct& scratch =
      getScratchData⟨ScratchPathDependentLinearProduct⟩(theEnv, myIndices);
  return scratch.covariates.isChanged(theEnv, thepMemento);
  }

void mspath::PathDependentLinearProduct::memento(Environment& theEnv,
    ScratchData **theppMemento)
  {
  ScratchPathDependentLinearProduct& scratch =
      getScratchData⟨ScratchPathDependentLinearProduct⟩(theEnv, myIndices);
  scratch.covariates.memento(theppMemento);
  }

const mspath::Double1D & mspath::PathDependentLinearProduct::evaluate(Environment& theEnv) const
  {
  ScratchPathDependentLinearProduct& scratch =
      getScratchData⟨ScratchPathDependentLinearProduct⟩(theEnv, myIndices);
  return mypCoefficients→multiply(theEnv, scratch.covariates);
  }

std::ostream & mspath::PathDependentLinearProduct::printOn(std::ostream & ostr) const
  {
  ostr << "␣Linear␣Product␣of␣Path-Dependent␣Variables␣at␣ind\
      ices␣" << myIndices << "␣and␣" << *mypCoefficients << std::endl;
  return ostr;
  }
```

This code is used in section 5.7.

5.7.4 SumLinearProducts

⟨ *SumLinearProducts Implementation 5.7.4* ⟩ ≡

```
// make the argument part of the sum
// This is part of initial object creation.
// Do NOT call it after calling other methods.
void mspath::SumLinearProducts::insert(AbstractLinearProduct *thepLinearProduct)
{   // we could, but don't, check size compatibility
myProducts.push_back(thepLinearProduct);
}

// return true if result of evaluate( ) when state was
// memento is no longer current.
// If thepMemento is 0, return true.
bool mspath::SumLinearProducts::isChanged(Environment& theEnv,
      ScratchData *thepMemento) const
{
Memento *pMemento = dynamic_cast<Memento *>(thepMemento);
if (pMemento == 0)
  return true;   // backwards since last is most likely to vary
for (size_t i = myProducts.size( ) - 1; ; --i)
  {
  if (myProducts[i].isChanged(theEnv, (*pMemento)[i]))
    return true;
  if (i == 0U)   // because a test i < 0 is meaningless for size_t
    return false;
  }
}

// set memento for state as of last evaluate( )
// Will create memento if arg is 0
void mspath::SumLinearProducts::memento(Environment& theEnv,
      ScratchData **theppMemento)
{
if (*theppMemento == 0)
  *theppMemento = new Memento(myProducts.size( ));
for (size_t i = 0; i < myProducts.size( ); ++i)
  myProducts[i].memento(theEnv, &(dynamic_cast<Memento&>(**theppMemento)[i]));
}

// return coefficients time relevant covariates in theEnv
// Return value only good until next call to this function
// with the same theEnv
const mspath::Double1D & mspath::SumLinearProducts::evaluate(Environment& theEnv) const
{
State& s = getScratchData<State>(theEnv, *this);
Double1D& r = s.result( );
r = myProducts[0].evaluate(theEnv);
for (size_t i = 1; i < myProducts.size( ); ++i)
  r += myProducts[i].evaluate(theEnv);
return r;
```

```
    }
  std::ostream & mspath::SumLinearProducts::printOn(std::ostream & ostr) const
    {
    const size_t n= myProducts.size( );
    ostr << " Sum of " << n << " Linear Products" << std::endl;
    for (size_t i = 0U; i < n; i++)
      {
      ostr << "Linear Product " << i << " is ";
      myProducts[i].printOn(ostr);
      }
    return ostr;
    }
```

This code is used in section 5.7.

5.8 Coefficients

⟨ *Coefficients.cc 5.8* ⟩ ≡

```
  @o Coefficients.cc
#include "Coefficients.h"
#include "Covariates.h"
#include "Environment.h"
```

⟨ *InterceptCoefficients output 5.8.0.2* ⟩
⟨ *SlopeCoefficients output 5.8.0.3* ⟩
⟨ *SlopeCoefficients multiply 5.8.0.1* ⟩

This code is used in section 9.

⟨ *SlopeCoefficients multiply 5.8.0.1* ⟩ ≡

```
const mspath::Double1D & mspath::SlopeCoefficients::multiply(Environment& theEnv,
        AbstractCovariates& theCovs) const
  {
  WorkData& w = getScratchData⟨WorkData⟩(static_cast⟨ScratchPad&⟩ (theEnv), nTotal( ));
  if (¬theCovs.isChanged(theEnv, w.pCovariateMemento))
    return w.results;
  const Double1D& cov = theCovs.values(theEnv);
  theCovs.memento(&(w.pCovariateMemento));

    // this is a matrix vector multiply
    // done somewhat manually
  for (size_t i = 0; i < w.results.size( ); ++i)
    {
    w.results[i] = (cov * (totalSlopes( ).col(i))).sum( );
    }
  return w.results;
  }
```

This code is used in section 5.8.

⟨ *InterceptCoefficients output 5.8.0.2* ⟩ ≡

```
std::ostream & mspath::operator <<(std::ostream & theStream,
        const mspath::InterceptCoefficients & theC)
  {
  theStream << "Intercept␣Only␣Coefficients" << std::endl << "␣" << theC.totalIntercepts( ) <<
        std::endl << "␣from␣effective␣coefficients␣" << theC.effectiveIntercepts( ) <<
        std::endl << "␣and␣constraints␣" << theC.interceptConstraints( ) << std::endl;
  return theStream;
  }
```

This code is used in section 5.8.

⟨ *SlopeCoefficients output 5.8.0.3* ⟩ ≡

```
std::ostream & mspath::operator <<(std::ostream & theStream,
        const mspath::SlopeCoefficients & theC)
  {
  theStream << "Slope␣Only␣Coefficients" << std::endl << "␣" << theC.totalSlopes( ) <<
        std::endl << "␣from␣effective␣coefficients␣" << theC.effectiveSlopes( ) << std::endl <<
        "␣and␣constraints␣" << theC.slopeConstraints( ) << std::endl;
  return theStream;
  }
```

This code is used in section 5.8.

5.9 Path

```
⟨ Path.cc 5.9 ⟩ ≡
    @o Path.cc
  #include <algorithm>
  #include "Path.h"
    namespace mspath
      {
      Path::Path(const Path& p) : mypNF(p.mypNF→clone( ))
        { innerAssign(p); }
      Path& Path::operator =(const Path& p)
        {
        if (this ≠ &p)
          { clear( );    // since this is a view, it shouldn't deallocate
          mypNF→reset( ); innerAssign(p); }
        return ∗this;
        }
      Node ∗Path::pathPush(const StatePoint& theSP, const TimePoint& theTP)
        {
        Node ∗pn = mypNF→createNode(theSP, theTP);
        push_back(pn);
        return pn;
        }
      void Path::pathPop( )
        {
        mypNF→destroyNode(pop_back( ).release( ));
        }
        // protected because it doesn't assure Node comes
        // from my NodeFactory.
      void Path::push_back(Node ∗pn)
        { if (empty( ))
            pn→setNoPrevious( );
          else    // back() returns an object
            pn→setPrevious(&(this→back( ))); Super::push_back(pn); }
      Path& Path::innerAssign(const Path& p)
        {
        Path::const_iterator i = p.begin( );
        while (i ≠ p.end( ))
          {
          push_back(mypNF→createNode(∗i));
          ++i;
          }
        return ∗this;
        }
        // quick summary form likely to be most useful
      std::ostream & operator <<(std::ostream & s, const Path& p)
        {
```

```
    Path::const_iterator i, e(p.end( ));
    for (i = p.begin( ); i ≠ e; ++i)
      { s << i→state( ) << "␣"; }
    ;
    return s;
    }
  }    // end namespace mspath
```

This code is used in section 9.

5.10 Time Step Generation

5.10.1 Utility Functions

⟨ *AbstractTimeStepsGenerator.cc 5.10.1* ⟩ ≡

```
  @o AbstractTimeStepsGenerator.cc
#include "AbstractTimeStepsGenerator.h"
  namespace mspath
    {

      // Insert a single TimePoint into the Environment.
    void AbstractTimeStepsGenerator::makeStep(Time t,
          bool trueObs, TimePoint::TIObservation i, Environment *pEnv)
      {
      TimePoint *p = new TimePoint (t, trueObs, i);
      pEnv→timeSteps( ).push_back(p);
      }

    }
```

This code is used in section 9.

5.10.2 Fixed Time Steps Generation

*We'll use the following variable prefixes: ob for indices to observations (***TIObs***); it for integral time as produced by integerTime( ) (***TInt***).*

integerTime maps continuous time into a set of integers, with each integer corresponding to a step in our grid. Note the first one for a case is not necessarily 0.

Use timeFromInteger to recover the rounded time.

Generally, work in the integer times as long as possible to avoid problems with inexact floating point comparisons.

⟨ *FixedTimeStepsGenerator time manipulation 5.10.2* ⟩ ≡

```
FixedTimeStepsGenerator::TIntFixedTimeStepsGenerator::integerTime(Time t)
  {
  return static_cast⟨TInt⟩(std::floor(t / myStepAsDouble + 0.5));
  }

Time FixedTimeStepsGenerator::timeFromInteger(FixedTimeStepsGenerator::TInt i)
  {   // use rational to retain precision
  return boost::rational_cast⟨Time⟩(i * myStepSize);
  }
```

This code is used in section 5.10.2.1.

Here's the main routine and the outer skeleton of the file.

⟨ *FixedTimeStepsGenerator.cc 5.10.2.1* ⟩ ≡

```
@o FixedTimeStepsGenerator.cc
#include "FixedTimeStepsGenerator.h"

#include <cmath>
#include <sstream>

#include "Data.h"

  namespace mspath {

  void FixedTimeStepsGenerator::makeStepsFor(Environment *pEnvironment)
    {
    Data& data = pEnvironment→data( );
    pEnvironment→clearTimeSteps( );
    TIObs obnext = pEnvironment→begin( );
    TIObs obend = pEnvironment→end( );
    TInt iti;    // loop variable
    if (obnext ≡ obend)    // probably an error, might throw something
      return;
    TInt itnext = integerTime(data.time(obnext));
    makeStep(timeFromInteger(itnext), true, obnext, pEnvironment);
    TInt itlast = itnext;
    obnext++;
    while (obnext ≠ obend)
      {
      itnext = integerTime(data.time(obnext));
      if (itnext ≡ itlast)
        {
        std::stringstream s;
        s << "Observation " << obnext << " (case " << pEnvironment→id( ) <<
            ") overlaps in rounded time" "with previous observation. Aborting.";
        throw ExtraDataError(s.str( ));
        }
      for (iti = itlast + 1; iti < itnext; iti++)
        makeStep(timeFromInteger(iti), false, obnext − 1, pEnvironment);
      makeStep(timeFromInteger(itnext), true, obnext, pEnvironment);
      itlast = itnext;
      obnext++;
      }
    } ⟨ FixedTimeStepsGenerator time manipulation 5.10.2 ⟩

  }
```

This code is used in section 9.

5.10.3 Compressed Time Steps Generation

*We'll use the following variable prefixes: ob for indices to observations (***TIObs***); it for integral time as produced by integerTime( ) (***TInt***).*

The first observation always gets a TimeStep. After that, the last observation in a given time interval is used.

Here's the main routine and the outer skeleton of the file.

```
⟨ CompressedTimeStepsGenerator.cc 5.10.3 ⟩ ≡
    @o CompressedTimeStepsGenerator.cc
  #include "CompressedTimeStepsGenerator.h"
  #include "Data.h"

  #include <cmath>

    namespace mspath {

    void CompressedTimeStepsGenerator::makeStepsFor(Environment *pEnvironment)
      {
      Data& data = pEnvironment→data( );
      pEnvironment→clearTimeSteps( );
      myobnext = pEnvironment→begin( );
      myobend = pEnvironment→end( );
      if (myobnext ≡ myobend)     // probably an error, might throw something
        return;
      myitnext = integerTime(data.time(myobnext));
      makeStep(timeFromInteger(myitnext), true, myobnext, pEnvironment);
      lastObs(data);
      while (myobnext ≠ myobend)
        {
        int itprev = myitlast;
        Data::TIObsobprev = myoblast;
        lastObs(data);
        for (int it = itprev + 1; it < myitlast; it++)
          {
          makeStep(timeFromInteger(it), false, obprev, pEnvironment);
          }
        makeStep(timeFromInteger(myitlast), true, myoblast, pEnvironment);
        }
      } ⟨ CompressedTimeStepsGenerator::lastObs 5.10.3.1 ⟩}
```

This code is used in section 9.

lastObs mostly uses instance variables for input and output. This violation of structured programming is to buy a little speed.

On input, myobnext is the index of the anchor observation, at integral time myitnext. Also, myobend should be one past the end of indices for this case.

On output, myoblast and myitlast have the values of the last observation with the same integral time as the initial myitnext. Generally, these should refer to the original anchor observation. If not, there are several observations that map to the same integral time. The duplicate observation indices go in myDuplicates.

myobnext points to the first observation following the anchor that has a later integral time, and myitnext is that integral time. If there are no later observations, myobnext will be myobend, and myitnext will be garbage.

⟨ *CompressedTimeStepsGenerator::lastObs* 5.10.3.1 ⟩ ≡

```
void CompressedTimeStepsGenerator::lastObs(const Data& data)
  {
  myitlast = myitnext;
  while (++myobnext ≠ myobend)
    {
    myitnext = integerTime(data.time(myobnext));
    if (myitnext > myitlast)
      {
      myoblast = myobnext − 1;
      return;
      }    // we have multiple observations on one integral time
    myDuplicates.push_back(myitnext);
    }    // end of data for this case
  myoblast = myobnext − 1;
  }
```

This code is used in section 5.10.3.

5.10.4 TimeStepGenerator

The algorithm here assures that our TimePoints will include every observed point, and also have enough points in between so that a maximum step size will not be exceeded. Between any two observed times, the generated times are evenly spaced.

The next concern may be moot because of the move to exact rational numbers: It would be good to make this routine robust against rounding errors, so that people don't discover unpleasant things such as $.25 \times 4 < 1$ *so one year intervals divided into 5 parts. How far should this go? For example, should* .33 *be treated as* $1/3$*?*

⟨ *TimeStepsGenerator.cc 5.10.4* ⟩ ≡

```
@o TimeStepsGenerator.cc
#include "TimeStepsGenerator.h"
#include "Data.h"

#include <cmath>

  namespace mspath
    {
    void TimeStepsGenerator::makeStepsFor(Environment *pEnvironment)
      {
      Data& data = pEnvironment→data( );
      pEnvironment→clearTimeSteps( );
      Data::TIObsb, e, i, j;
      Time t0, t1;    // start and end of interval
      Time delta;    // increments
      size_t n;    // number of increments
      size_t ni;    // inner loop
      b = pEnvironment→begin( );
      e = pEnvironment→end( );
      if (b ≡ e)    // perhaps throw an exception?
        return;    // nothing to do
      i = b;
      t0 = data.time(i);
      t1 = t0;    // if we have a single observation
        // we will fall through the loop
        // though perhaps we should throw an error
      for (j = b + 1; j < e; j++)
        {
        t1 = data.time(j);
        delta = t1 − t0;
        n = static_cast⟨size_t⟩ (std::ceil(delta / myStepSize));
        delta /= n;
        for (ni = 0; ni < n; ni++)
          makeStep(t0 + delta * ni, ni ≡ 0, i, pEnvironment);    // prepare for next iteration
        t0 = t1;
        i = j;
        }
      makeStep(t1, true, i, pEnvironment);
      }    // end makeStepsFor
    }    // end namespace mspath
```

This code is used in section 9.

5.11 Computing Path-Dependent History

⟨ *PrimitiveHistoryComputer.cc 5.11* ⟩ ≡

```
@o PrimitiveHistoryComputer.cc
#include "PrimitiveHistoryComputer.h"
#include "Environment.h"

void mspath::TimeInStateComputer::evaluate(Environment& theEnv, Node& theNode)
    throw (std::out_of_range) { Node *pprior = theNode .previous ( );

if (pprior)
  {
  if (pprior→state( ) ≡ theNode.state( ))
    theNode.modelData( )[dataIndex( )] = pprior→modelData( )[dataIndex( )] + (theNode.time( ) −
        pprior→time( ));
  else
    theNode.modelData( )[dataIndex( )] = myInitialTime;
  }
else
  theNode.modelData( )[dataIndex( )] = myInitialTime; }

void mspath::TimeInPreviousStatesComputer::evaluate(Environment& theEnv,
    Node& theNode) throw (std::out_of_range) { Node *pprior = theNode .previous ( );

if (pprior)
  {
  if (pprior→state( ) ≠ theNode.state( ))
    theNode.modelData( )[dataIndex( )] = myInitialTime + theNode.time( ) −
        theEnv.path( ).front( ).time( );
  else    /* no change */
    theNode.modelData( )[dataIndex( )] = pprior→modelData( )[dataIndex( )];
  }
else
  theNode.modelData( )[dataIndex( )] = myInitialTime; }

void mspath::TimeSinceOriginComputer::evaluate(Environment& theEnv, Node& theNode)
    throw (std::out_of_range) { Node *pprior = theNode .previous ( );

if (pprior)
  {
  theNode.modelData( )[dataIndex( )] = theNode.time( ) − theEnv.path( ).front( ).time( ) +
      theEnv.path( ).front( ).modelData( )[dataIndex( )];
      // last term above is currently equivalent to myInitialTime
  }
else
  theNode.modelData( )[dataIndex( )] = myInitialTime; }
```

This code is used in section 9.

The next classes do simple functional transformations of history.

I judge the possibilities that the user would specify out of range values sufficient to merit an explicit check. For one thing, it is easy to get 0 values at the start of an interval.

⟨ *CompositeHistoryComputer.cc 5.11.0.1* ⟩ ≡

```
@o CompositeHistoryComputer.cc
#include "CompositeHistoryComputer.h"
#include <cmath>
#include "Environment.h"

void mspath::LnHistoryComputer::evaluate(Environment& theEnv, Node& theNode) throw
        (std::out_of_range)
  {
  if (theNode.modelData( )[target( ).dataIndex( )] ≤ 0.0)
    throw std::out_of_range(name( ) + " attempts ln of value <= 0");

  theNode.modelData( )[dataIndex( )] = std::log(theNode.modelData( )[target( ).dataIndex( )]);
  }
```

This code is used in section 9.

5.12 SimpleRecorder

```
⟨ SimpleRecorder.cc 5.12 ⟩ ≡
    @o SimpleRecorder.cc
  #include "SimpleRecorder.h"
    namespace mspath
      {
      void SimpleRecorder::startSession( )
        {
        myNCases = 0;
        myNPaths = 0;
        myNNodes = 0;
        myNBads = 0;
        myNPathNodes = 0;
        }
      std::ostream & operator <<(std::ostream & s, const SimpleRecorder& r)
        {
        std::streamsize prec = s.precision( );
        s.precision(3);
        s << r.cases( ) << " cases with " << r.goodPaths( ) << " good paths and " <<
            r.goodNodes( ) << " good nodes." << std::endl << "The paths had " <<
            r.goodPathNodes( ) << " total nodes, so we would calculate on\
            ly " << 100.0 * r.goodNodes( ) / r.goodPathNodes( ) << "%." << std::endl <<
            "Average path length = " << r.averagePathLength( ) << ". " << r.badNodes( ) <<
            " bad nodes." << std::endl;
        s.precision(prec);
        return s;
        }
      }
```

This code is used in section 9.

5.13 Evaluator

```
⟨ Evaluator.cc 5.13 ⟩ ≡
    @o Evaluator.cc
  #include "Evaluator.h"
    namespace mspath
      {
      void Evaluator::evaluate(Node& theNode, Environment& theEnv)
        {
        model( ).fillModelData(theNode, theEnv);
        model( ).evaluate(theNode, theEnv);
        }
      }
```

This code is used in section 9.

5.14 Data

⟨ *Data.cc 5.14* ⟩ ≡
@o Data.cc
#include "Data.h"

#include <sstream>
#include <Rinternals.h>

namespace mspath { **Data**::**TTimesData**::*observationTimes*(**const AbstractDataIterator**& *theI*)
{
return *myTime*[**std**::*slice*(*theI*.*begin*( ), *theI*.*size*( ), 1)];
}

⟨ *Data Iterator Code 5.14.0.1* ⟩
⟨ *Subset Data Iterator Code 5.14.0.2* ⟩}

This code is used in section 9.

Implement Data Iterator functions.

⟨ *Data Iterator Code 5.14.0.1* ⟩ ≡

```
bool DataIterator::next( ) throw (DataIteratorError)
  {
  if (myBegunIterating)
    return advanceToNext(myCurrentLast + 1);
  else
    {
    myBegunIterating = true;
    return advanceToNext(0);
    }
  }

  // prepare next iterator group starting at observation i
bool DataIterator::advanceToNext(Data::TIObsi) throw (DataIteratorError)
  {
  if (i + 1 ≥ data( ).nObs( ))     // OOPS–no more data
    return false;
  myCurrentId = data( ).subject(i);
  myCurrentFirst = i;
  do
    ++i;
  while (i < data( ).nObs( ) ∧ data( ).subject(i) ≡ myCurrentId);
  myCurrentLast = i − 1;
  myCurrentSize = i − myCurrentFirst;

#ifdef DEBUG
  std::ostringstream buf;
  buf << "DataIterator::advanceToNext␣advanced␣to␣" << myCurrentId << std::endl;
  Rprintf("%s", buf.str( ).c_str( ));
#endif

  return true;
  }
```

This code is used in section 5.14.

⟨ *Subset Data Iterator Code 5.14.0.2* ⟩ ≡

```
bool SubsetDataIterator::next( ) throw (DataIteratorError)
  {
  if (myNextInSubset ≥ myEndSubset)
    return false;
  if (myNextInSubset ≡ 0U)
    return advanceToNext(0U);
  else
    return advanceToNext(myCurrentLast + 1U);
  }

  // find next obs in subset, starting at or after i
bool SubsetDataIterator::advanceToNext(Data::TIObsi) throw (DataIteratorError)
  {
  myCurrentId = subsetID(myNextInSubset);

#ifdef DEBUG
  std::ostringstream buf;
  buf << "SubsetDataIterator::advanceToNext␣looking␣for␣id␣" << myCurrentId << std::endl;
  Rprintf("%s", buf.str( ).c_str( ));
#endif    // DEBUG

  size_t nAll = data( ).nObs( );
  while (i < nAll)
    {
    if (data( ).subject(i) ≡ myCurrentId)
      break;
    ++i;
    }
  if (i ≥ nAll)
    throw SubsetDataIteratorError(myCurrentId);

    // we've now found the start of the next in subset
  myCurrentFirst = i;
  ++myNextInSubset;
  do
    ++i;
  while (i < nAll ∧ data( ).subject(i) ≡ myCurrentId);
  myCurrentLast = i − 1;
  myCurrentSize = i − myCurrentFirst;
#ifdef DEBUG
  std::ostringstream buf2;
  buf2 << "SubsetDataIterator::advanceToNext␣moved␣to␣id␣" << myCurrentId << std::endl;
  Rprintf("%s", buf2.str( ).c_str( ));
#endif
  return true;
  }
```

This code is used in section 5.14.

5.15 Covariates

⟨ *Covariates.cc 5.15* ⟩ ≡
```
    @o Covariates.cc
  #include "Covariates.h"
  #include "Environment.h"     // deferred from header file
    namespace mspath {
        ⟨ MatrixCovariates Implementation 5.15.1 ⟩
        ⟨ PathCovariates Implementation 5.15.2 ⟩
    }
```

This code is used in section 9.

5.15.1 MatrixCovariates

We deliberately return isChanged( ) `true` on the first use to force initial computation of results.

This implementation is optimal if it is cheaper to check for changes than re-fetch the values. It's possible that is not so, particularly if there are only a few values. Note that the mementos return the state of this object as of its last operation; this is not necessarily the state it would have in the current environent.

⟨ *MatrixCovariates Implementation 5.15.1* ⟩ ≡

```
bool MatrixCovariates::isChanged(Environment& theEnv, ScratchData *thepMemento)
  {
  Memento *pMemento = dynamic_cast<Memento *> (thepMemento);
  if (¬pMemento)
    return true;
  if (pMemento→iObservation ≡ theEnv.iObservation( ))
    return false;
     // we have a new observation
#if 0
    // we shouldn't be using this class if there are no data
  if (myData.nrows( ) ≡ 0)
    return false;
#endif
  rawValues(theEnv);
  size_t n = mypMemento→cache.size( );    // assume two caches have same size
  for (size_t i = 0; i < n; ++i)
    if (mypMemento→cache[i] ≠ pMemento→cache[i])
      return true;
  return false;    // new observation, but same values
  }

  // fetch the current covariates
Double1D& MatrixCovariates::rawValues(Environment& theEnv)
  {
  if (mypMemento ≡ 0)
    mypMemento = new Memento (myData, theEnv);
  else if (mypMemento→iObservation ≠ theEnv.iObservation( ))
    mypMemento→capture(myData, theEnv);    // fall through if same observation as before
  return mypMemento→cache;
  }

  // separate copy
ScratchData *MatrixCovariates::memento( )
  {
  if (mypMemento ≡ 0)
    return 0;
  return new Memento (*mypMemento);
  }

ScratchData *MatrixCovariates::memento(ScratchData **theppMemento)
  {
  if (*theppMemento ≡ 0)
    {
```

```
    *theppMemento = memento( );
    return *theppMemento;
    }
  ;
  *dynamic_cast⟨Memento *⟩ (*theppMemento) = *mypMemento;
  return *theppMemento;
  }

  // clients should use this interface to get values

Double1D& MatrixCovariates::values(Environment& theEnv)
  {
  return rawValues(theEnv);
  }
```

⟨ *MatrixCovariates::Memento Implementation* 5.15.1.1 ⟩

This code is used in section 5.15.

Although it seems natural to implement the next section in the original declaration, attempting to do so would mean **Environment** *would need to be declared, not forward declared. Attempting to include* `"Environment.h"` *in the header doesn't work because the mutual dependency implies that it will not always be parsed before needed.*

Note the constructor encapsulates exactly what information is extracted.

⟨ *MatrixCovariates::Memento Implementation* 5.15.1.1 ⟩ ≡

```
  // c'tor
MatrixCovariates::Memento::Memento(const Double2D& theData, const
      Environment& theEnv) : iObservation(theEnv.iObservation( )),
      cache(theData.col(iObservation))
  { }

  // update with current state

void MatrixCovariates::Memento::capture(const Double2D& theData, const
      Environment& theEnv)
  {
  iObservation = theEnv.iObservation( );
  cache = theData.col(iObservation);
  }
```

This code is used in section 5.15.1.

5.15.2 PathCovariates

Get the lagged value, unless there is none.

⟨ *PathCovariates Implementation 5.15.2* ⟩ ≡

```
    // note: no check modelData( ) and myIndices are sensible
  Double1D& PathCovariates::values(Environment& theEnv)
    {
    Node *pNode = &(theEnv.currentNode( ));
    Node *prior = pNode→previous( );
    if (prior ≡ 0)
      prior = pNode;
    myValues = prior→modelData( )[myIndices];
    return myValues;
    }
```

This code is used in section 5.15.

5.16 TimePoint

⟨ *TimePoint.cc 5.16* ⟩ ≡

```
  @o TimePoint.cc
#include "TimePoint.h"
  namespace mspath
    {
    std::ostream & operator <<(std::ostream & ostr, const TimePoint& tp)
      {
      ostr << "TimePoint(" << tp.time( ) << ",␣" << tp.matchesObservation( ) << ",\
          ␣" << tp.iObservation( ) << ")";
      return ostr;
      }
    }
```

This code is used in section 9.

5.17 Environment Simulation Support

Good random number generation is tricky, especially in a parallel environment.

In the interest of simplicity, portability, and expediency I use **R***'s random number facilities. Users should select an appropriate generator, and manage its state (e.g., set.seed( )) in* **R***. Note that set.seed( ) has no effect on* `rsprng`*,*[56] *which must be explicitly initialized using functions specific to that package.*

Calls to unif_rand( ), a C function provided by the **R** *API, must be bracketed by GetRNGstate( ) and SetRNGstate( ); however doing that for each random number would be very inefficient. The only way to get here from* **R** *is to call simulate, and so we make the bracketing calls there.*

Some notes on alternative generators and implementations follow.

R*, the system or language libraries, hardware,* **Boost***, and the GNU Scientific Library each provide one or more ways to generate random numbers. Since* **mspath** *is compute intensive, it is natural to think of running it, particularly in simulation, in parallel. Parallel random number generators packaged for* **R** *include* `SPRNG` *via* `rsprng` *and L'Ecuyer's via* `rlecuyer`*.*

Three possible strategies are

1. *rely on* **R***'s random number generator (the present strategy);*
2. *rely on some other random number facility;*
3. *provide a selection of random number generators.*

[56] In version 0.4, current as of 2009-08-01.

For the second approach, bringing in any additional library will complicate the build and run-time environment, resulting in a more brittle, failure-prone system. Picking any single random number strategy will likely leave some users unsatisified. Parallel users may want a parallel generator, while single process users will probably find the extra complexities of parallel generators annoying.

Since **R***'s random number framework allows a variety of random number generators, including parallel and sequential ones, and since it adds no additional complexity, it is among the most flexible of the single library choices. The performance penalty of calling back to* **R***—something that must be done frequently since one can only request a single random number at a time—is a potential drawback.*

The third stategy—provide a selection of random number generators—avoid the restrictiveness of any one. However, if one additional library is a problem, managing a whole set of them is even worse.

For the third strategy, it is tempting to factor out the details of random number generation into a separate class, so that one could choose alternatives. This would introduce a performance penalty because of the extra indirection,[57] *and random number generation calls occur frequently when they are being used. It might be possible to cache a vector of values in the environment to reduce this problem, but that's more complicated. One could also use a class-based traits approach to avoid the run-time indirection, but that would then require making several different kinds of environments for the different kinds of random number generators.*

A final general problem (even for the first strategy) is to determine the relation between what one does with the **R** *random generators and what happens down in the C++ code. For example,* `rsprng` *provide some unpleasant surprises:*

- *Calling set.seed( ) in* **R** *has no effect on the random numbers in* **R** *or in C++ when* `rsprng` *has been initialized.*
- *If C++ code attempted to use* `SPRNG` *directly (rather than calling back to* **R***) it would have no access to the stream used by* `rsprng`*; the stream id is a private variable* `rsprng` *does not share. So the two would be completely independent.*
- `SPRNG` *not only comes in 4 versions, but each version has a number of variants. If* **mspath** *built against something that was not an exact match for what* `rsprng` *uses, the result could be unfortunate.*[58]

[57] The client asks the **Environment** for a random choice and the **Environment** asks some object it contains. The second request uses a virtual function call, which is relatively slow.

[58] Actually, even if the match were perfect, the result might still be unfortunate.

⟨ *Environment.cc 5.17* ⟩ ≡

```
@o Environment.cc
#include "Environment.h"
#include <cstdlib>

extern "C"
  {
#define STRICT_R_HEADERS 1
#include <R.h>
  }

namespace mspath {

template ⟨typename Prob⟩
State Environment::randomDraw(const std::valarray < Prob > &theDensity)
  {
  Prob cum = static_cast⟨Prob⟩ (unif_rand( ));
  Prob sofar = 0.0;     // calculate cdf as we go
  State lastState;     // last state with positive probability
  for (size_t i = 0U; i < theDensity.size( ); i++)
    {
    if (theDensity[i] > 0)
      {
      sofar += theDensity[i];
      if (sofar ≥ cum)
        return i;
      lastState = i;
      }
    }    /* It is unusual to fall through the loop, but it might happen if the sum of the densities is a
        bit under 1 because of numerical noise and we draw a high cum. */
  return lastState;
  }

  // because this is in a .cpp file, we need explicit instantiation
template State Environment::randomDraw⟨double⟩ ( const std::valarray⟨double⟩ & ) ;

}
```

This code is used in section 9.

5.18 Basic Output

These exist mostly to support debugging.

All these routines are templates, so I include them in the .h file to allow the proper instantiations to work. Depending on your compiler technology and logic, you may be able to manage something more elegant (this with with g++ 3.3).

In other words, there is no separate basic.cc file.

⟨ *basic.cc 5.18* ⟩ ≡

```
namespace mspath
  {
  ⟨ Vector Output 5.18.0.1 ⟩
  ⟨ Matrix Output 5.18.0.2 ⟩

  }
```

This code is used in section 3.

⟨ *Vector Output 5.18.0.1* ⟩ ≡

```
template ⟨typename T ⟩
std::ostream & operator <<(std::ostream & s, const Array1D⟨ T ⟩& a)
  {
  for (std::size_t i = 0; i < a.size( ); i++)
    {
    if ((i % 5) ≡ 0)
      s << std::endl << "[" << i << "]";
    s << "␣" << a[i];
    }
  s << std::endl;
  return s;
  }
```

This code is used in section 5.18.

⟨ *Matrix Output 5.18.0.2* ⟩ ≡

```
template ⟨typename T ⟩
std::ostream & operator <<(std::ostream & s, const Array2D⟨ T ⟩& a)
  {
  size_t nr = a.nrows( );
  size_t nc = a.ncols( );
  size_t r, c;
  for (r = 0; r < nr; r++)
    {
    s << std::endl << r << ":";
    for (c = 0; c < nc; c++)
      s << " " << a(r, c);
    }
  s << std::endl;
  return s;
  }
```

This code is used in section 5.18.

6 Interface with R

R *can call C in two ways, with* . **C**( ) *or with* . *Call*( ). *The former is more traditional, simpler, and was implemented first. With it, the C code receives and returns only standard C types.* . *Call*, *in contrast, uses* **R***'s internal C types.*

The second approach became essential for later development, because optimizations require that C++ data remain persistent between calls. So we must return these data to the calling **R** *program; doing so requires the use of R_ExternalPtr types. One can only return such types through* . *Call or* . *External*.

The second approach permits other flexibility in return values that we may later exploit to return richer data (e.g., per-case data) from an analysis.

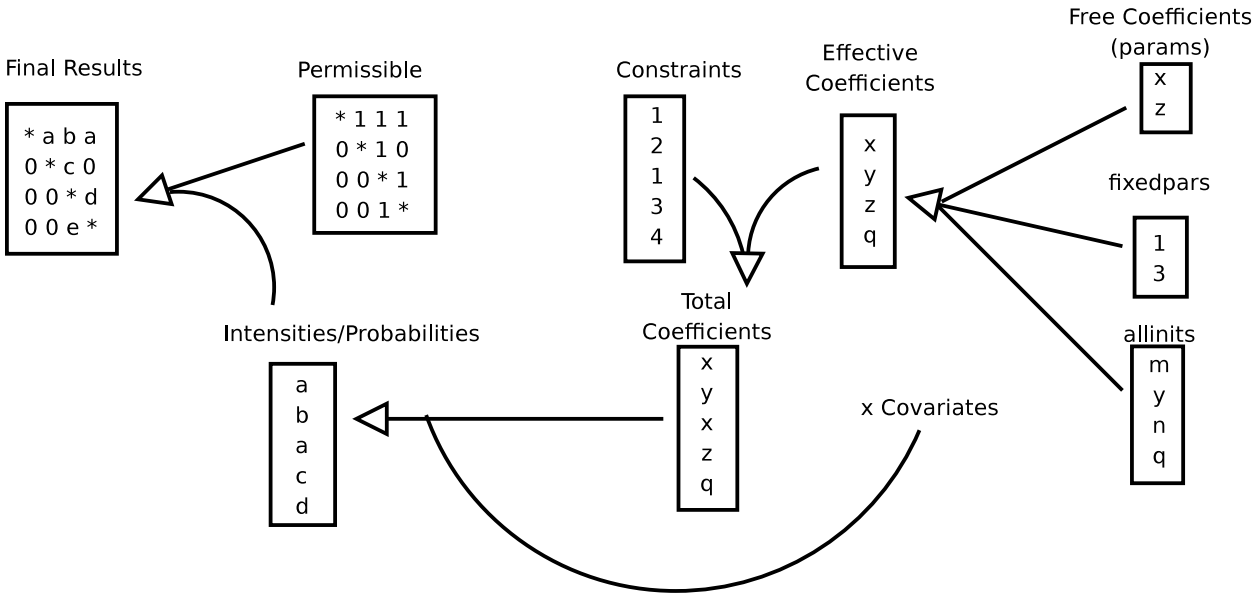

Figure 1: Coefficient Handling

6.1 Meaning and Interpretation of Parameters

We now describe the parameters passed in from **R** *and their interpretation.*

The model for specifying effects is quite rich, and correspondingly complicated. Many of the input parameters concern either the specification of transition intensities or misclassification probabilities (the probability of observing one state, given some underlying true state).

Figure 1 presents a stylized description of where these values come from. It is easiest to grasp by working backwards from the final result, shown on the left of the figure. There are s possible states, so the final result is an $s \times s$ matrix of transition intensities or misclassification probabilities. With i indexing rows and j columns, the (i, j)'th element gives the transition intensity from state i to state j or the probability of observing state j given a true state of i.

Notice the diagonals have asterisks; they are derived from the other values in the same row (currently the code assures each row sums to 0). One input, shown immediately to the right of the "Final Results" is the matrix of "Permissible" values. Only cells with 1's in them can have values (again, the diagonals are special). These values are peeled off from a vector of intensities or probabilities, shown below, where the first value in that vector goes in the first available position in the final results, the second goes in the second position (reading across the row) and so on. This vector in turn is computed from other values; it is not a direct input.

Those values come from multiplying the "Total Coefficients," shown to the right, by the relevant set of covariates. (This glosses over several complications, spelled out in a moment). Various rules may constrain the values of those coefficients. One possible constraint makes one or more coefficients match the value of another coefficient. Another constraint simply pegs a coefficient to a constant value.

As we work backward through the processing, the equality constraints come next. As shown above the "Total Coefficients" vector, a constraint vector is used to expand a smaller set of "Effective Coefficients" into the "Total Coefficients." "Constraints" has one entry for each position in "Total"; the entry gives the index in "Effective" from which the entry should be drawn.[59] *This 1-based indexing scheme differs from the way constraints are specified in* **R**, *which permits any vector of increasing numbers for the Constraints vector. Variables relating to the "Effective Coefficients" often have eff in their names.*

At the very start of the process, shown at the right, the "Effective Coefficients" are constructed from a mix of the free parameters actually being optimized and the initial values. The vector fixedpars give the 0-based indices[60] *in the initial values, allinits, from which to draw fixed coefficients. Of course, there may be no fixed coefficients. The remaining ones are the true free parameters of the model, and come from the input params.*

Now for the promised details about the coefficients and covariates. First, there are actually two sets of constraints and effective coefficients. The first pair give baseline values and constraints for the quantity (transition intensity or misclassification probability) of interest. These have no corresponding covariates. The second pair gives coefficients and constraints on them; these are multiplied by the covariates. The coefficients are stacked so that first all the coefficients of the first covariate appear, then the ones for the second, and so on.

Second, the transformation of these values (including the covariates) to intensities or probabilities involves more than matrix multiplication. The baseline values are simple hazard rates or probabilities, but the coefficients operate linearly on log-odds for probabilities and log-hazards for transition rates.

Tables 1 and 2 give the details of input parameters corresponding to each quantity of interest. Where relevant, the matrix or vector dimensions appear in parentheses, using other input parameters as dimensions. All arrays are passed in with the column parameter varying most rapidly, so all the first row appears in memory, then all the second, and so on.

6.2 . *Call* Interface

These are generally thin wrappers.

[59] The constraints are on the coefficients, rather than the intensities. Though in this example the effect would be the same, this structure allows the creation of two intensities for which only a subset of the relevant coefficients are constrained to be equal.

[60] However, these are 1-based when specified within **R**.

Input	Name	Comments
Optimization Parameters	*params*	Free parameters.
Fixed Parameters	*fixedpars*	1-based indices into *allinits* of those values to fix at the initial level. This is a vector of indices, not the values themselves.
Initial Parameters	*allinits*	Reused for fixed parameters.
Probabilities at $t = 0$	*initprobs*(*nst*)	Assumed probabilities of true states on the first observation for a case.

The first 3 vectors refer to *intens*, *coveffect*, *patheffect*, *miscprobs*, *misccoveffect* and *miscpatheffect*, in that order. All those variables are internal variables, not direct inputs, and are defined in the next table.

Table 1: Some Generally Applicable Input Parameters

Input	Transition Probabilities	Misclassification Probabilities
Baseline	*intens*(*nintenseffs*)	*miscprobs*(*nmisceffs*)
Coefficients	*coveffect*(*ncoveffs*)	*misccoveffects*(*nmisccoveffs*)
Path Coefficients	*patheffect*(*npatheffs*)	*miscpatheffect*(*npathmisceffs*)
Permissible	*qvector*(*nst*, *nst*)	*evector*(*nst*, *nst*)
Baseline Constraint	*baseconstraint*(*nintens*)	*basemiscconstraint*(*nmisc*)
Coefficient Constraint	*constraint*(*ncovs*, *nintens*)	*miscconstraint*(*nmisccovs*, *nmisc*)
Path Coefficient Constraint	*pathconstraint*(*nhistory*, *nintens*)	*pathmiscconstraint*(*nhistory*, *nmisc*)
Covariates	*covvec*(*ncovs*, *nobs*)	*misccovvec*(*nmisccovs*, *nobs*)
Path Variables	*history*(*nhistory*)	

Note the covariates are already transposed, relative to the usual arrangement.
Despite the label "Inputs" some of these variables are only derived from the inputs.
For likelihoods, the expected return values are $-2\ln(\mathcal{L})$.

Table 2: Input Parameters for Intensities and Misclassification Probabilities

Note that we use **R***-specific macros of Rinternals.h, rather than the ones shared with S. The documenation indicates the latter are relatively unused and untested in* **R***.*

Call makeManager to set up the computation, compute to carry it out, and the finalizer will be called implicitly when you are done.

It is not guaranteed the finalizer will be called if you exit the system; the optional argument to R_RegisterCFinalizerEx might assure that, but I don't know what it really does. Since all memory should be released by the OS, it doesn't seem worth worrying about.

simulate( ) *returns an* **R** `list` *of simulated data. The columns are* `state`*, the simulated observed state (integer),* `time`*, the simulated observation time (real), and* `row`*, the row index (integer) in the input data (including the vector of subjects, which can be used to recover the subject id). See §4.2.3, page 43 for the details of how the simulation is carried out. There is no corresponding .* **C** *interface for this function, since its return value is not a simple C type.*

⟨ *mspathR.h 6.2.1* ⟩ ≡
```
@o mspathR.h
#ifndef mspathR_h
#define mspathR_h 1
#define R_NO_REMAP 1
#include <R.h>
```

```
#include <Rinternals.h>
#include "Data.h"    // for SubsetDataIterator
  extern "C"
    {   // returns an ExternalPtr
    SEXP makeManager(⟨ makeManager args 6.2.2 ⟩);
      // first arg is return value of makeManager
      // remaining args should all be same as in call to makeManager
      // except that the free parameters (params) may be different
      // The function does not check for this precondition.
      // Returns the manager.
    SEXP setParams(SEXP ptr, ⟨ makeManager args 6.2.2 ⟩);
      // subset is an integer vector of subject id's, in same order as original
      // returns self
    SEXP selectSubset(SEXP ptr, SEXP subset);
      // analyze all. returns self
    SEXP selectAll(SEXP ptr);
      // returns vector of results
      // in the same style as the . C interface return values (for now).
      // ptr is the value returned by makeManager( )
      // do_what is an integer requesting the kind of operation
    SEXP compute(SEXP ptr, SEXP do_what);
      // Return simulated paths not the manager.
      // The argument remains the manager.
    SEXP simulate(SEXP ptr);
      // user should not need to call
      // cleanup
    void finalizeManager(SEXP ptr);
    }
#endif    // mspathR_h
```

This code is used in section 9.

The arguments defined in this section are almost the same arguments as used in the older interface. The first and last arguments of the old interface are omitted, since irrelevant for setup. All the types become SEXP, with comments indicating the expected true type of the arguments. I don't currently check argument types; the caller must get them right.

This duplication of arguments is annoying, and a potential source of trouble. Updates probably belong in both places. They also belong in the section after this one.

Some of the counts could probably be omitted, since we can get them from the vectors.

```
⟨ makeManager args 6.2.2 ⟩ ≡
    /* double */
  SEXP params,    /* full parameter vector—free parameters only */
    /* double */
  SEXP allinits,    /* all initial values */

    /* int */
  SEXP misc,
      /* 0 = no misclassification 1 = full misclassification model 2 = simple, fixed misclassification */
    /* int */
  SEXP p,    /* number of parameters optimised over (length of params) */
    /* int */
  SEXP subjvec,    /* vector of subject IDs */
    /* double */
  SEXP timevec,    /* vector of observation times */
    /* int */
  SEXP statevec,    /* vector of observed states */
    /* int */
  SEXP qvector,    /* vectorised matrix of allowed transition indicators */
    /* int */
  SEXP evector,    /* vectorised matrix of allowed misclassification indicators */
    /* double */
  SEXP covvec,    /* vectorised matrix of covariate values */
    /* int */
  SEXP constraint,    /* list of constraints for each covariate */
    /* double */
  SEXP misccovvec,    /* vectorised matrix of misclassification covariate values */
    /* int */
  SEXP miscconstraint,    /* list of constraints for each misclassification covariate */
    /* int */
  SEXP baseconstraint,    /* constraints on baseline transition intensities */
    /* int */
  SEXP basemiscconstraint,    /* constraints on baseline misclassification probabilities */

    /* char ** */
  SEXP history,    /* names of history-dependent variables */
    /* double */
  SEXP initialOffset,    /* add this to every 0 time in paths */
    /* int */
  SEXP pathconstraint,    /* constraints on path effects on intensities */
    /* int */
  SEXP pathmiscconstraint,    /* constraints on path effects on misclassification */

    /* double */
  SEXP initprobs,    /* initial state occupancy probabilities */
```

```
  /* int */
SEXP nst,    /* number of Markov states */
  /* int */
SEXP nms,    /* number of underlying states which can be misclassified */
  /* int */
SEXP nintens,    /* number of intensity parameters */
  /* int */
SEXP nintenseffs,    /* number of distinct intensity parameters */
  /* int */
SEXP nmisc,    /* number of misclassification rates */
  /* int */
SEXP nmisceffs,    /* number of distinct misclassification rates */
  /* int */
SEXP nobs,    /* number of observations in data set */
  /* int */
SEXP npts,    /* number of individuals in data set */
  /* int */
SEXP ncovs,    /* number of covariates on transition rates */
  /* int */
SEXP ncoveffs,    /* number of distinct covariate effect parameters */
  /* int */
SEXP nmisccovs,    /* number of covariates on misclassification probabilities */
  /* int */
SEXP nmisccoveffs,    /* number of distinct misclassification covariate effect parameters */

  /* int */
SEXP nhistory,    /* number of history dependent variables */
  /* int */
SEXP npatheffs,    /* number of distinct path effects on transitions */
  /* int */
SEXP npathmisceffs,    /* number distinct path effects on misclassification */

  /* int */
SEXP isexact,    /* non-0 if we observe entry time to absorbing states exactly */
  /* int */
SEXP nfix,    /* number of fixed parameters */
  /* int */
SEXP fixedpars,    /* which parameters to fix */
  /* stepNumerator/stepDenominator gives the maximum size of time step in discrete approximation.
     Note both arguments are integers. */
  /* int */
SEXP stepNumerator,    /* int */
SEXP stepDenominator
```

This code is used in sections 6.2.1, 6.2.5, 6.2.6, and 6.2.10.

Use the next definition when you pass the arguments on to lower level functions. More tedious repitition: this is the same as the previous section, except the type declarations are missing.

⟨ *makeManager params 6.2.3* ⟩ ≡

```
    /* double */
  params,   /* full parameter vector—free parameters only */
    /* double */
  allinits,   /* all initial values */
    /* int */
  misc,
     /* 0 = no misclassification 1 = full misclassification model 2 = simple, fixed misclassification */
    /* int */
  p,   /* number of parameters optimised over (length of params) */
    /* int */
  subjvec,   /* vector of subject IDs */
    /* double */
  timevec,   /* vector of observation times */
    /* int */
  statevec,   /* vector of observed states */
    /* int */
  qvector,   /* vectorised matrix of allowed transition indicators */
    /* int */
  evector,   /* vectorised matrix of allowed misclassification indicators */
    /* double */
  covvec,   /* vectorised matrix of covariate values */
    /* int */
  constraint,   /* list of constraints for each covariate */
    /* double */
  misccovvec,   /* vectorised matrix of misclassification covariate values */
    /* int */
  miscconstraint,   /* list of constraints for each misclassification covariate */
    /* int */
  baseconstraint,   /* constraints on baseline transition intensities */
    /* int */
  basemiscconstraint,   /* constraints on baseline misclassification probabilities */
    /* char ** */
  history,   /* names of history-dependent variables */
    /* double */
  initialOffset,   /* add this to every 0 time in paths */
    /* int */
  pathconstraint,   /* constraints on path effects on intensities */
    /* int */
  pathmiscconstraint,   /* constraints on path effects on misclassification */
    /* double */
  initprobs,   /* initial state occupancy probabilities */
    /* int */
  nst,   /* number of Markov states */
    /* int */
  nms,   /* number of underlying states which can be misclassified */
    /* int */
  nintens,   /* number of intensity parameters */
    /* int */
```

```
nintenseffs,    /* number of distinct intensity parameters */
 /* int */
nmisc,    /* number of misclassification rates */
 /* int */
nmisceffs,    /* number of distinct misclassification rates */
 /* int */
nobs,    /* number of observations in data set */
 /* int */
npts,    /* number of individuals in data set */
 /* int */
ncovs,    /* number of covariates on transition rates */
 /* int */
ncoveffs,    /* number of distinct covariate effect parameters */
 /* int */
nmisccovs,    /* number of covariates on misclassification probabilities */
 /* int */
nmisccoveffs,    /* number of distinct misclassification covariate effect parameters */
 /* int */
nhistory,    /* number of history dependent variables */
 /* int */
npatheffs,    /* number of distinct path effects on transitions */
 /* int */
npathmisceffs,    /* number distinct path effects on misclassification */
 /* int */
isexact,    /* non-0 if we observe entry time to absorbing states exactly */
 /* int */
nfix,    /* number of fixed parameters */
 /* int */
fixedpars,    /* which parameters to fix */
 /* stepNumerator/stepDenominator gives the maximum size of time step in discrete approximation.
    Note both arguments are integers. */
 /* int */
stepNumerator,    /* int */
stepDenominator
```

This code is used in sections 6.2.5 and 6.2.6.

Now we turn to the implementation. Note that in most of this document "interface" refers to header files. In this section, the "interface" with **R** *includes all facilities supporting communication between* **R** *and C, and thus include the file that follows.*

⟨ *mspathR.cc 6.2.4* ⟩ ≡

```
@o mspathR.cc
#include "mspathR.h"

    // used by internal functions
#include <memory>
#include "Manager.h"
#include "ModelBuilder.h"

  extern "C" {
#include <R.h>
  ⟨ R ModelBuilder helper 6.2.10 ⟩
  ⟨ makeManager body 6.2.5 ⟩
  ⟨ R setParams body 6.2.6 ⟩
  ⟨ R compute body 6.2.7 ⟩
  ⟨ R subsetting bodies 6.2.9 ⟩
  ⟨ R simulation body 6.2.12 ⟩
  ⟨ R finalizer body 6.2.8 ⟩
  }
```

This code is used in section 9.

More incredibly tedious and redundant code. Note the absence of safety checks. I've stolen code from mspathCEntry( ).

```
⟨ makeManager body 6.2.5 ⟩ ≡
    // returns an ExternalPtr
  SEXP makeManager(⟨ makeManager args 6.2.2 ⟩)
    {
    using namespace mspath;
    std::auto_ptr⟨Model⟩pm = makeModel(⟨ makeManager params 6.2.3 ⟩);
      // data setup
    Data *pd = new Data(INTEGER(subjvec), *INTEGER(nobs), *INTEGER(npts),
        REAL(timevec), INTEGER(statevec), REAL(covvec), *INTEGER(ncovs), REAL(misccovvec),
        *INTEGER(nmisccovs));
  #if 0
    std::cout << "Data has " << pd->nPersons( ) << " people in " << pd->nObs( ) <<
        " records." << std::endl;
  #endif
    Manager *pmanager = new Manager(pd, pm.release( ), *INTEGER(stepNumerator),
        *INTEGER(stepDenominator), (*INTEGER(isexact)) ≠ 0);
      // one example didn't use PROTECT( )
    SEXP ptr;
    Rf_protect(ptr = R_MakeExternalPtr(pmanager, R_NilValue, R_NilValue));
    R_RegisterCFinalizer(ptr, (R_CFinalizer_t)finalizeManager);
    Rf_unprotect(1);
    return ptr;
    }
```

This code is used in section 6.2.4.

In some sense the ideal interface for the next function, setParams, would include only the new parameter values. The problem with that is that it would require a major overhaul of the code underneath. As an alternative, I could cache the supposedly constant values, but since I already have them in **R** *that seems like a waste.*

```
⟨ R setParams body 6.2.6 ⟩ ≡
  SEXP setParams(SEXP ptr, ⟨ makeManager args 6.2.2 ⟩)
    {
    using namespace mspath;
    std::auto_ptr⟨Model⟩pm = makeModel(⟨ makeManager params 6.2.3 ⟩);
    Manager *pmanager = static_cast⟨Manager *⟩(R_ExternalPtrAddr(ptr));
    pmanager->setModel(pm);
    return ptr;
    }
```

This code is used in section 6.2.4.

⟨ **R** *compute body 6.2.7* ⟩ ≡

```
SEXP compute(SEXP ptr, SEXP do_what)
  {
  using namespace mspath;
  Manager *pmanager = static_cast⟨Manager *⟩(R_ExternalPtrAddr(ptr));
  SEXP newvec;
  Rf_protect(newvec = Rf_allocVector(REALSXP, 6U));
  double *returned = REAL(newvec);
  std::stringstream serror;

  try
    {
    pmanager→go(returned, *INTEGER(do_what));
    *returned *= −2;
    }

  catch(std::exception & exc)
    {
    serror << "Caught␣exception:␣" << exc.what( );
    }

  catch(...)
    {
    serror << "Some␣non-standard␣exception␣was␣thrown" << std::endl;
    }
  if (¬serror.str( ).empty( ))
    {
    finalizeManager(ptr);    // kill manager
    Rf_error("%s", serror.str( ).c_str( ));
    }
  Rf_unprotect(1);
  return newvec;
  }
```

This code is used in section 6.2.4.

The user shouldn't need to call this function.

⟨ **R** *finalizer body 6.2.8* ⟩ ≡

```
void finalizeManager(SEXP ptr)
  {
  using namespace mspath;
  Manager *pmanager = static_cast⟨Manager *⟩(R_ExternalPtrAddr(ptr));
  delete pmanager;
  R_ClearExternalPtr(ptr);
  }
```

This code is used in section 6.2.4.

⟨ **R** *subsetting bodies 6.2.9* ⟩ ≡

```
#include <sstream>
#include <Rinternals.h>
  SEXP selectSubset(SEXP ptr, SEXP subset)
    {
    using namespace mspath;
    Manager *pmanager = static_cast⟨Manager *⟩(R_ExternalPtrAddr(ptr));
    SubsetDataIterator::IDList pSub(new Int1D(INTEGER(subset), LENGTH(subset)));
    std::ostringstream buf;
    std::size_t count = pSub→size( );
    buf << "mspath::selectSubset got " << LENGTH(subset) << " ID's in selectSubset: ";
    for (size_t i = 0; i < count; i++)
      buf << " " << (*pSub)[i] << ";";
    buf << std::endl;     // Rprintf("pmanager→setSubset(pSub);
    return ptr;
    }
  SEXP selectAll(SEXP ptr)
    {
    using namespace mspath;
    Manager *pmanager = static_cast⟨Manager *⟩(R_ExternalPtrAddr(ptr));
    pmanager→setAll( );
    return ptr;
    }
```

This code is used in section 6.2.4.

This code is a utility for some of the external functions. It could go in an unnamed namespace if those worked more reliably.

```
⟨ R ModelBuilder helper 6.2.10 ⟩ ≡
    std::auto_ptr < mspath::Model > makeModel(⟨ makeManager args 6.2.2 ⟩) { using namespace
        mspath;
    ⟨ R string conversion 6.2.11 ⟩ ModelBuilder mb(REAL(params), REAL(allinits), *INTEGER(p),
        *INTEGER(nst), *INTEGER(nfix), INTEGER(fixedpars));
    std::auto_ptr⟨Model⟩pm = mb.makeModel(INTEGER(misc), INTEGER(qvector),
        INTEGER(evector), INTEGER(constraint), INTEGER(miscconstraint),
        INTEGER(baseconstraint), INTEGER(basemiscconstraint), INTEGER(pathconstraint),
        INTEGER(pathmiscconstraint), REAL(initprobs), INTEGER(nms), INTEGER(nintens),
        INTEGER(nintenseffs), INTEGER(nmisc), INTEGER(nmisceffs), INTEGER(ncovs),
        INTEGER(ncoveffs), INTEGER(nmisccovs), INTEGER(nmisccoveffs),
        // The compiler (GCC 4.3) seems to require the next cast. I
      // don't think the cast should be necessary, but it's harmless.
     INTEGER(nhistory), const_cast⟨const char **⟩(cptr), REAL(initialOffset), INTEGER(npatheffs),
        INTEGER(npathmisceffs));
  #if 0
    std::cout << "Model␣intensities:" << std::endl << m.intensity( ) << std::endl <<
        "Model␣misclassifications:␣" << std::endl << m.misclassification( ) << std::endl;
  #endif
    return pm; }
```

This code is used in section 6.2.4.

Convert an **R** string vector to a form suitable for C. This code is based on that in *dotcode.c* in the base **R** distribution; that's the code that is used to convert for . **C**. That code is more careful about encoding schemes.

cptr has the result.

```
⟨ R string conversion 6.2.11 ⟩ ≡
      /* case STRSXP: */
    int n = LENGTH(history);
    char **cptr = (char **) R_alloc(n, sizeof(char *));
    for (int i = 0; i < n; i++)
      {
      size_t l = strlen(CHAR(STRING_ELT(history, i)));
          /* I'm not sure if we need the next, inner copy for read-only access, but I retain it from the
          original for safety. Note the memory will not persist after we return to R. RB */
      cptr[i] = (char *) R_alloc(l + 1, sizeof(char));
      strcpy(cptr[i], CHAR(STRING_ELT(history, i)));
      }
```

This code is used in section 6.2.10.

I tried returning a `data.frame`, but setting it up properly got too involved. Simply creating a `list` and setting its `class` and `row.names` attributes was not adequate.

This package's C code calls back to **R**'s random number generator. The only way to get to our C code from **R** is through *simulate*( ), and so we make the required *GetRNGstate*( ) and *PutRNGstate*( ) calls here for efficiency.

If our C code had other callbacks to **R** it might be necessary to be more careful, because **R** might want to use random numbers during the callback.

⟨ **R** simulation body 6.2.12 ⟩ ≡

```
SEXP simulate(SEXP ptr) {
    using namespace mspath;
    typedef RandomPathGenerator::Results Results;

  Manager *pmanager = static_cast⟨Manager *⟩ (R_ExternalPtrAddr(ptr));
  GetRNGstate( );
  std::auto_ptr⟨Results⟩pResults = pmanager→simulate( );    // main computation

  SEXP rvalue, names, state, time, iobs;

  ⟨ R simulation list setup 6.2.13 ⟩
  ⟨ R simulation get individual results 6.2.14 ⟩
  ⟨ R simulation put results in frame 6.2.15 ⟩

  PutRNGstate( );
  Rf_unprotect(5U);
  return rvalue; }
```

This code is used in section 6.2.4.

Set up the `list`.

⟨ **R** simulation `list` setup 6.2.13 ⟩ ≡

```
const int nCol= 3U;
const char *cnames[ ] = {"state", "time", "row"};

  // VECSXP means an R list.
Rf_protect(rvalue = Rf_allocVector(VECSXP, nCol));

Rf_protect(names = Rf_allocVector(STRSXP, nCol));
for (size_t i = 0U; i < nCol; i++)
  {
  SET_STRING_ELT(names, i, Rf_mkChar(cnames[i]));
  }
Rf_setAttrib(rvalue, R_NamesSymbol, names);
```

This code is used in section 6.2.12.

⟨ **R** simulation get individual results 6.2.14 ⟩ ≡

```
size_t n = pResults→size( );
Rf_protect(state = Rf_allocVector(INTSXP, n));
Rf_protect(time = Rf_allocVector(REALSXP, n));
Rf_protect(iobs = Rf_allocVector(INTSXP, n));
int *cstate = INTEGER(state);
double *ctime = REAL(time);
int *ciobs = INTEGER(iobs);
for (size_t i = 0U; i < n; i++)
  {
  RandomPathGenerator::SimResult & r = (*pResults)[i];    // R states and indices start at 1.
  cstate[i] = r.state( ) + 1U;
  ctime[i] = r.time( );
  ciobs[i] = r.obsIndex( ) + 1U;
  }
```

This code is used in section 6.2.12.

Stick individual results in the `data.frame`.

⟨ **R** simulation put results in frame 6.2.15 ⟩ ≡

```
SET_VECTOR_ELT(rvalue, 0U, state);
SET_VECTOR_ELT(rvalue, 1U, time);
SET_VECTOR_ELT(rvalue, 2U, iobs);
```

This code is used in section 6.2.12.

6.3 Implementation Notes, with Interface Consequences

Many operations from **R** are variations on a theme. The main variation driving the switch to . *Call* from . **C** is that one call may request a function evaluation while another requests the gradient. I anticipate big time savings if the likelihood and gradient are computed at the same time, with the results saved to answer subsequent queries.

This means the results must persist between calls from **R**.[61] The C++ data must be passed opaquely to **R**. As noted earlier, that requires returning an external pointer, which can not be done from . **C**. So the interface must be . *Call* or . *External*.

Other operations from **R** involve partial changes to what is being computed: changes in the parameters or the subset of cases. In this case, caches must be invalidated. In some cases, the invalidation might be partial.

There are several possible approaches in handling variations. I cast the discussion in terms of the **Model**; changing the subset raises similar issues, though with different objects.

Recreate Everything One could simply rebuild the model from scratch and clear the entire cache whenever anything changed.

Pros
- Maximal reuse of existing code.
- Simplicity. There is no need to manage a separate update mechanism.

Cons
- Fat interfaces: one must supply all the model parameters each time, even if only one thing changes. I could save them in C++, as a wrapper around **ModelBuilder**. Or I could pass in the **R** object and pull values from there.
- Consequent possible illogical calls. Most of the model must remain unchanged, but in principle a completely new model could get passed in. Either the inputs must be assumed good, or they need to be validated.
- Bad interfaces: In other words, a good interface would only provide the changed parameters, making it natural to avoid doing stupid or meaningless things.
- More overhead of recreating the entire model and of saving the original parameter values. The overhead affects time and memory.

Have a Specialist Object Update the Model In this scenario, the **ModelBuilder** or a closely allied class would update the appropriate values.

Pros
- Clean Separation of responsibility.
- It's natural to combine information about how to update and how to create. **ModelBuilder** already knows how to map from free to effective parameters (see section6.1 for the meaning of these terms), while **Model** and **Coefficients** don't.
- Less object creation, and thus likely greater speed than previous approach.

Cons
- Complexity.
- Still requires holding on to all the initial model parameters, at least in the most obvious implementation.

[61] Another approach would be to return the all the information at once from Cand then manage this at the **R** level. Maybe I should have thought of that earlier!

Have the Model Update Itself In this category the **Model** itself does the update from new parameters. Perhaps the cleanest variant in this category involves passing the entire new parameter set to each object recursively. Each object takes whatever action is appropriate.

Pros
- Nicely distributes responsibility.
- Avoids having to redo the discovery of the fundamental structure of the model when only parameters have changed.
- Potentially the fastest approach, because it saves object creation, discovery of the model structure, and intermediate calculations. For example, the mapping from free parameters to effective parameters to total parameters could be calculated once, with updates going directly from free parameters to total parameters (again, see §6.1 for the meaning of those terms).

Cons
- Complicated.
- Lots of new code and chances for new bugs.
- The **Model** and its components would, at least in some versions, need to know about the mapping from free to effective parameters. They don't know anything about this now.
- Muddies the waters, with individual objects needing to know not only how to do a computation but how to update themselves. But see subsequent discussion for a way around this.

In the interests of simplicity, development speed, and minimizing the chances for bugs, I went with the first option. Note there is no indication that model construction is taking a significant amount of run-time, and thus little justification for trying to make it faster.

Though I took that approach, here are some thoughts on the other approaches.

Note that **ModelBuilder** knows about fixed/free parameters, while the individual **Model** parts know about constraints. It is also conceivable that the calculation of effective parameters from fixed and free could be done entirely in **R**. That might make more sense, but it is not the way the code from **msm** was written. Note that the presence of constraints, which means one free parameter may affect more than one total parameter, will complicate calculation of derivatives.

One strategy would be for the **ModelBuilder**, possibly in conjunction with the objects it creates, make special "update" objects. These could be put in a list, and the update strategy would be to present each one with the parameters. The update object would know how to update some real part of the **Model**. This might be the fastest and simplest way to do this self-update strategy. Because the update responsibility is isolated in the update objects, this clears the otherwise muddied waters referred to in the final "Con" of the last option above.

Starting with the free parameters, there are 3 steps:

1. free + fixed params → effective params (via fixed spec)
2. effective params → total coefficients (via constraints)
3. total coefficients → matrix (via permissible matrix)

Currently ModelBuilder does 1, the instances it makes does 2 at creation time of **Coefficients**, and **Specification** does 3 at runtime.

I can make the initial objects so that they know how to accept values from the free parameters. This means they must store 2 vectors: first, a mask of which elements in their (total) parameters to update, and, second, the indices from the free parameters from which to pull the values. I must assign from *allinits* at the start. free parameters may be duplicated thanks to constraints. The free parameter space is the *entire* vector of parameters, not just the ones relevant to this particular item. That way, I can simply pass around the free parameter vector and leave it to the components to pull out the right values. This will also collapse the mapping of free parameters and constraints (steps 1 and 2 above) into a single step.

As the builder goes along, it must compute both of those vectors. Well, the objects constructed need to interpret them, depending on their internal storage scheme. The builder needs to provide the relevant part of *allinits*. the indices of the free parameters, and constraint info. Probably needs total range of indices too.

6.4 . C Interface

This is the routine that **R** calls. It is much like the corresponding **msm** routine, except that it provides additional parameters for maximum time step size, and the return value should be a vector to hold all the information in **Recorder**.

I may gradually trim this as I discover what I need and convert to C++. When using it from C++, be sure to wrap in **extern** `"C"`.

⟨ mspath.h 6.4.1 ⟩ ≡

```
@o mspath.h
#include <math.h>
#include <stdlib.h>
#if 0
#include <R.h>
#include <R_ext/Applic.h>
#endif
    /* #include < stdio.h > */
  extern "C"
    {
  #define MI(i, j, ncols) ((int)((i) * (ncols) + (j)))
        /* index to treat a vector as a matrix. Fills rows first */
  #define logit(x) (log((x) / (1 - (x))))
  #define expit(x) (exp(x) / (1 + exp(x)))
    ⟨ C Matrix Typedefs 6.4.2 ⟩
    ⟨ C Entry Point 6.4.3 ⟩
    ⟨ C Entry Helpers 6.4.5 ⟩
    ⟨ Old C I don't think I'll need 6.4.6 ⟩
    }
```

This code is used in section 9.

These should be wrapped in a proper C++ class.

⟨ C Matrix Typedefs 6.4.2 ⟩ ≡
```
typedef double *Matrix;
typedef int *iMatrix;
typedef double * vector ;
typedef int *ivector;
```

This code is used in section 6.4.1.

This is what we call from **R**.

⟨ C Entry Point 6.4.3 ⟩ ≡
```
void mspathCEntry(⟨ mspathCEntry args 6.4.4 ⟩);
```

This code is used in section 6.4.1.

As noted at the top of this section, the arguments differ slightly from the **msm** original. The legal values for the first parameter reflect the different capabilities of this routine. I've added *stepNumerator* and *stepDenominator* just before the end, and expect the return value *returned* to be a vector big enough to hold the counts as well as the likelihoods. See §4.1.0.3, page 37, for a full description of the return values, except that this function return $-2 * \mathcal{L}$ rather then $\mathcal{L}$, the log-likelihood.

⟨ mspathCEntry args 6.4.4 ⟩ ≡

```
int *do_what,      /* 0 = count paths, 1 = eval likelihood */
  /* not supported 2 = Viterbi, 3 = prediction */
double *params,    /* full parameter vector—free parameters only */
 double *allinits,     /* all initial values */

int *misc ,
    /* 0 = no misclassification 1 = full misclassification model 2 = simple, fixed misclassification */
int *p ,     /* number of parameters optimised over (length of params) */
int *subjvec ,     /* vector of subject IDs */
double *timevec ,     /* vector of observation times */
int *statevec ,     /* vector of observed states */
int *qvector ,     /* vectorised matrix of allowed transition indicators */
int *evector ,     /* vectorised matrix of allowed misclassification indicators */
double *covvec ,     /* vectorised matrix of covariate values */
int *constraint ,     /* list of constraints for each covariate */
double *misccovvec ,     /* vectorised matrix of misclassification covariate values */
int *miscconstraint,    /* list of constraints for each misclassification covariate */
 int *baseconstraint,    /* constraints on baseline transition intensities */
 int *basemiscconstraint,     /* constraints on baseline misclassification probabilities */

const char **history ,     /* names of history-dependent variables */
double *initialOffset,    /* add this to every 0 time in paths */
 int *pathconstraint,    /* constraints on path effects on intensities */
 int *pathmiscconstraint,     /* constraints on path effects on misclassification */

double *initprobs ,     /* initial state occupancy probabilities */
int *nst ,     /* number of Markov states */
int *nms ,     /* number of underlying states which can be misclassified */
int *nintens ,     /* number of intensity parameters */
int *nintenseffs ,     /* number of distinct intensity parameters */
int *nmisc ,     /* number of misclassification rates */
int *nmisceffs ,     /* number of distinct misclassification rates */
int *nobs ,     /* number of observations in data set */
int *npts ,     /* number of individuals in data set */
int *ncovs ,     /* number of covariates on transition rates */
int *ncoveffs,    /* number of distinct covariate effect parameters */
 int *nmisccovs,    /* number of covariates on misclassification probabilities */
 int *nmisccoveffs,     /* number of distinct misclassification covariate effect parameters */

int *nhistory,    /* number of history dependent variables */
 int *npatheffs,    /* number of distinct path effects on transitions */
 int *npathmisceffs,     /* number distinct path effects on misclassification */

int *isexact,    /* non-0 if we observe entry time to absorbing states exactly */
 int *nfix,    /* number of fixed parameters */
 int *fixedpars,     /* which parameters to fix */
  /* stepNumerator/stepDenominator gives the maximum size of time step in discrete approximation.
     Note both arguments are integers. */
```

int ∗*stepNumerator*, **int** ∗*stepDenominator*, **double** ∗*returned*
/∗ returned -2 log likelihood and counts ∗/

This code is used in sections 6.4.3 and 6.4.9.

⟨ C Entry Helpers 6.4.5 ⟩ ≡
// move to static methods on **Model**

This code is used in section 6.4.1.

This is a place-holder, in case I need to dig some of this up.

⟨ Old C I don't think I'll need 6.4.6 ⟩ ≡

```
#if 0
  void msmLikelihood(data * d, model * m, int misc, double *returned);
  double likmisc(int pt, data*d, model*m);
  void AddCovs(int obs, data*d, model*m, double *newintens);
  void AddMiscCovs(int obs, data*d, model*m, double *newp);
  double PObsTrue(int obst,    /* observed state */
  int tst,    /* true state */
   double *miscprobs,    /* misclassification probabilities */
  model*m);
  double liksimple(data * d, model * m);
  void Viterbi(data * d, model * m, double *fitted);
  double pijt(int i, int j, double t, vector intens, int *qvector, int nstates, int exacttimes);
  void Pmat(Matrix pmat, double t, vector intens, int *qvector, int nstates, int exacttimes);
  void FillQmatrix(int *qvector, vector intens, Matrix qmat, int nstates);
  void MatrixExp(Matrix mat, int n, Matrix expmat, double t);
  int repeated_entries( vector vec, int n);
  void MatrixExpSeries(Matrix mat, int n, Matrix expmat, double t);
  void MatTranspose(Matrix A, Matrix AT, int n);
  void MatInv(Matrix A, Matrix Ainv, int n);
  void MultMat(Matrix A, Matrix B, int arows, int acols, int bcols, Matrix AB);
  void MultMatDiag(Matrix A, vector diag, int n, Matrix AB);
  void FormIdentity(Matrix A, int n);
#endif
```

This code is used in section 6.4.1.

Here's the implementation of the methods above. Note we do this as C++ code.

⟨ mspath.cc 6.4.8 ⟩ ≡

```
@o mspath.cc

#include <R.h>

#include <string>
#include <sstream>

#include "basic.h"
#include "mspath.h"
#include "Data.h"
#include "Manager.h"
#include "ModelBuilder.h"

namespace mspath
  {
  ⟨ mspathCEntry function 6.4.9 ⟩
  }
```

This code is used in section 9.

Handle the initial call from **R**.

⟨ mspathCEntry function 6.4.9 ⟩ ≡

```
extern "C" void mspathCEntry(⟨ mspathCEntry args 6.4.4 ⟩)
  {
  using namespace mspath;

  ModelBuilder mb(params, allinits, *p, *nst, *nfix, fixedpars);
  std::auto_ptr⟨Model⟩pm = mb.makeModel(misc, qvector, evector, constraint, miscconstraint,
      baseconstraint, basemiscconstraint, pathconstraint, pathmiscconstraint, initprobs, nms,
      nintens, nintenseffs, nmisc, nmisceffs, ncovs, ncoveffs, nmisccovs, nmisccoveffs, nhistory,
      history, initialOffset, npatheffs, npathmisceffs);

#if 0
  std::cout << "Model intensities:" << std::endl << m.intensity( ) << std::endl <<
      "Model misclassifications: " << std::endl << m.misclassification( ) << std::endl;
#endif

    // data setup
  Data *pd = new Data(subjvec, *nobs, *npts, timevec, statevec, covvec, *ncovs, misccovvec,
      *nmisccovs);
#if 0
  std::cout << "Data has " << pd→nPersons( ) << " people in " << pd→nObs( ) <<
      " records." << std::endl;
#endif
    /* If Manager is an automatic variable inside the try block, the spec seems to say it will be
      destroyed when the exception is caught. Though I'm not seeing that behavior, better safe than
      sorry. I allocate it on the heap so I can clean it up if there is an error() exit, which is a kind of
      longjump. */
  Manager *pmanager = new Manager(pd, pm.release( ), *stepNumerator, *stepDenominator,
      (*isexact) ≠ 0);
  std::stringstream serror;

  try
    {
    pmanager→go(returned, *do_what);
    *returned *= −2;
    }

  catch(std::exception & exc)
    {
    serror << "Caught exception: " << exc.what( );
    }

  catch(...)
    {
    serror << "Some non-standard exception was thrown" << std::endl;
    }
  if (¬serror.str( ).empty( ))
    {
    serror << std::endl;
    delete pmanager;
    error("%s", serror.str( ).c_str( ));
    }
  delete pmanager;
  pmanager = 0;    // safety first
```

```
}
```

This code is used in section 6.4.8.

6.5 ModelBuilder

⟨ ModelBuilder.cc 6.5 ⟩ ≡

```
@o ModelBuilder.cc
#include "ModelBuilder.h"

#include <algorithm>
#include <map>
#include <functional>
#include <sstream>

#include "CompositeHistoryComputer.h"
#include "PrimitiveHistoryComputer.h"
#include "Specification.h"

⟨ ModelBuilder::makeModel implementation 6.5.1 ⟩
⟨ ModelBuilder::makeInitial implementation 6.5.2 ⟩
⟨ ModelBuilder::makeSpecification implementation 6.5.3 ⟩
⟨ ModelBuilder::makeSimpleSpecification implementation 6.5.5 ⟩
⟨ ModelBuilder::fillparvec implementation 6.5.6 ⟩
⟨ ModelBuilder History Implementation 6.5.7 ⟩
⟨ ModelBuilder::makeHistoryIndirection implementation 6.5.13 ⟩
```

This code is used in section 9.

There are also various special parameters I'm not looking at and lot of consistency checks to do. Need to consider how to handle errors. But note some of my c'tors will throw errors.

⟨ ModelBuilder::makeModel implementation 6.5.1 ⟩ ≡

```
std::auto_ptr < mspath::Model > mspath::ModelBuilder::makeModel(⟨ ModelBuilder::makeModel
    arguments 4.24.1.2 ⟩) throw (InconsistentModel, BadInitialProbs, OneInitialState,
    UnknownTerm)
{
  std::auto_ptr < Model::TComputerContainer > pHistory = makeHistory(history, *nhistory,
      *initialOffset);
  std::auto_ptr⟨TIndirect1D⟩pHistoryIndirect(0);
  if (pHistory.get( ))
    pHistoryIndirect = makeHistoryIndirection(*pHistory);
  std::auto_ptr⟨Specification⟩pTransitions = makeSpecification(*nintenseffs, *nintens,
      baseconstraint, *ncoveffs, *ncovs, constraint, *npatheffs, *nhistory, pathconstraint, qvector,
      pHistoryIndirect.get( ));
  State initialState = makeInitial(pTransitions→nStates( ), initprobs);    // may throw
  std::auto_ptr⟨AbstractSpecification⟩pError(0);
  if (*misc)
    {
    if (*misc ≡ 2)
      pError.reset(makeSimpleSpecification(*nmisceffs, *nmisc, basemiscconstraint,
          evector).release( ));
    else
      pError.reset(makeSpecification(*nmisceffs, *nmisc, basemiscconstraint, *nmisccoveffs,
          *nmisccovs, miscconstraint, *npathmisceffs, *nhistory, pathmiscconstraint, evector,
          pHistoryIndirect.get( ), true).release( ));
    }
  return std::auto_ptr⟨Model⟩(new Model (pTransitions.release( ), pError.release( ),
      pHistory.release( ), initialState));
}
```

This code is used in section 6.5.

For now, only one state can be picked. Default is that initial state is 0.

```
⟨ ModelBuilder::makeInitial implementation 6.5.2 ⟩ ≡
    mspath::Statemspath::ModelBuilder::makeInitial(size_t n, double *p)const throw
        (OneInitialState, BadInitialProbs)
    {
    if (p ≡ 0)
      return 0U;
    double total = 0;
    State pick = 0U;
    for (size_t i = 0U; i < n; ++i)
      {
      if (p[i] ≡ 0.0)
        continue;
      if (p[i] < 0.0 ∨ p[i] > 1.0)
        {
        std::ostringstream ostr;
        ostr << "State " << i << " has impossible probability " << p[i];
        throw BadInitialProbs(ostr.str( ));
        }
      if (total > 0.0)
        throw OneInitialState( );
      pick = i;
      total += p[i];
      }
    if (total ≠ 1.0)
      throw BadInitialProbs(std::string(" must sum to 1"));
    return pick;
    }
```

This code is used in section 6.5.

The types of many of the arguments are not ideal, but result from the C interface. Because of type mismatches, many objects can't be constructed directly from the pointers.

Warning! *ncov* and *nintercept* are not parallel. There are *ninterceptEff* effective out of *nintercept* coefficients for the intercept, but *ncovEff* out of *ncov* ∗ *nintercept* cofficients for covariates.

The interface and implementation assume there will be constant terms. In principle they could be omitted, but then we'd need another argument to say how many terms overall there are.

Note that **Coefficients** make copies of their arguments, while higher-level objects take ownership of heap-allocated objects given to their constructors.

⟨ ModelBuilder::makeSpecification implementation 6.5.3 ⟩ ≡

```
std::auto_ptr < mspath::Specification >
    mspath::ModelBuilder::makeSpecification(int ninterceptEff, int nintercept,
    int *interceptConstraints, int ncovEff, int ncov,
    int *covConstraints, int npathEff, int npath, int *pathConstraints, int *permissible, const
    TIndirect1D *pathIndirect, bool useMisclassification)
  {
  Double1D effective;
    // intercept
  fillparvec(effective, ninterceptEff);
  TIndirect1D constraints(static_cast⟨size_t⟩(nintercept));
  for (size_t i = 0; i < static_cast⟨size_t⟩(nintercept); ++i)
    constraints[i] = static_cast⟨size_t⟩(interceptConstraints[i] − 1U);
  std::auto_ptr⟨InterceptCoefficients⟩pIntercepts(new InterceptCoefficients(effective,
      constraints));
  std::auto_ptr⟨AbstractLinearProduct⟩pLP(new
      ConstantLinearProduct(pIntercepts.release( )));    // need typed pointer to
      do insert
  if (ncov > 0 ∨ npath > 0)
    {
    std::auto_ptr⟨SumLinearProducts⟩pSum(0);
    pSum.reset(new SumLinearProducts( ));
    pSum→insert(pLP.release( ));    // old intercept LP is now first entry of pSum
      // covariates
    if (ncov > 0)
      {
      std::auto_ptr⟨SlopeCoefficients⟩pSlopes = makeSlope(effective, ncovEff, ncov, nintercept,
          covConstraints);
      std::auto_ptr⟨DataLinearProduct⟩pDLP(new DataLinearProduct(pSlopes.release( ),
          useMisclassification));
      pSum→insert(pDLP.release( ));
      }
      // path-dependent history
    if (npathEff > 0 ∧ npath > 0)
      {
      std::auto_ptr⟨SlopeCoefficients⟩pSlopes = makeSlope(effective, npathEff, npath,
          nintercept, pathConstraints);
```

```
        std::auto_ptr⟨PathDependentLinearProduct⟩pPLP(new
            PathDependentLinearProduct(pSlopes.release( ), *pathIndirect));
        pSum→insert(pPLP.release( ));
        }    // switch sum into pLP
      pLP = pSum;
      }
    std::auto_ptr⟨Bool2D⟩pPermissible(new Bool2D( ));
    pPermissible→setRaw(permissible, mynst, mynst);
    std::auto_ptr⟨Specification⟩pSpec(new Specification(pLP.release( ), pPermissible.release( )));
    return pSpec;
    }
```

⟨ ModelBuilder::makeSlope implementation 6.5.4 ⟩

This code is used in section 6.5.

Helper for *makeSpecification*.

⟨ ModelBuilder::makeSlope implementation 6.5.4 ⟩ ≡

```
std::auto_ptr < mspath::SlopeCoefficients >
    mspath::ModelBuilder::makeSlope(Double1D& effective, int neffective, int ncov,
    int nterms, int *constraints)
{
fillparvec(effective, neffective);
TIndirect2D c2(ncov, nterms);
for (size_t i = 0U; i < static_cast⟨size_t⟩(ncov); ++i)
  for (size_t j = 0U; j < static_cast⟨size_t⟩(nterms); ++j)
    c2(i, j) = static_cast⟨size_t⟩(constraints[i * nterms + j] − 1U);
return std::auto_ptr⟨SlopeCoefficients⟩(new SlopeCoefficients(effective, c2));
}
```

This code is used in section 6.5.3.

This is a stripped-down version of *makeSpecification*.

```
⟨ ModelBuilder::makeSimpleSpecification implementation 6.5.5 ⟩ ≡
    std::auto_ptr < mspath::SimpleSpecification >
        mspath::ModelBuilder::makeSimpleSpecification(int ninterceptEff, int nintercept,
        int *interceptConstraints, int *permissible)
    {
    Double1D effective;

      // intercept
    fillparvec(effective, ninterceptEff);

    TIndirect1D constraints(static_cast⟨size_t⟩(nintercept));
    for (size_t i = 0; i < static_cast⟨size_t⟩(nintercept); ++i)
      constraints[i] = static_cast⟨size_t⟩(interceptConstraints[i] - 1U);

    std::auto_ptr⟨InterceptCoefficients⟩pIntercepts(new InterceptCoefficients(effective,
        constraints));

    std::auto_ptr⟨ConstantLinearProduct⟩pLP(new
        ConstantLinearProduct(pIntercepts.release( )));

    std::auto_ptr⟨Bool2D⟩pPermissible(new Bool2D( ));
    pPermissible→setRaw(permissible, mynst, mynst);
    std::auto_ptr⟨SimpleSpecification⟩pSpec(new SimpleSpecification(pLP.release( ), mynst,
        pPermissible.release( )));
    return pSpec;
    }
```

This code is used in section 6.5.

Fill a parameter vector with either the current values from the optimisation or the fixed initial values.

This started as the **msm** function of the same name, but it's changed a lot. Many variables are now in the class rather than the arguments.

This depends heavily on the state of the object, which it reads and writes.

⟨ ModelBuilder::fillparvec implementation 6.5.6 ⟩ ≡

```
void mspath::ModelBuilder::fillparvec(Double1D& parvec,
    /* named vector to fill (e.g. intens = baseline intensities) */
 int ni     /* length of parvec when done */
)
  {
  int i;
  parvec.resize(ni);
  for (i = 0; i < ni; ++i, ++(myiall))
    {
    if ((myifix < mynfix) ∧ (myiall ≡ myfixedpars[myifix]))
      {
      parvec[i] = myallinits[myiall];
      ++(myifix);
      }
    else if (myiopt < myp)
      {
      parvec[i] = myparams[myiopt];
      ++(myiopt);
      }
    }
  }
```

This code is used in section 6.5.

The main method for building history appears below, and then its supporting methods.

⟨ ModelBuilder History Implementation 6.5.7 ⟩ ≡

```
std::auto_ptr < mspath::Model::TComputerContainer >
    mspath::ModelBuilder::makeHistory(const char **history,    /* names of
    history-dependent variables */
int nhistory,    /* number of history dependent variables */
 double initialOffset    /* offset times by this amount */
) throw (mspath::UnknownTerm, mspath::TangledDependencies)
  {
  if (nhistory ≤ 0)
    return std::auto_ptr < Model::TComputerContainer > (0);
  TStringVector requests = makeRequests(history, nhistory);
  std::auto_ptr < THistoryVector > allHistoryComputers = allComputers(initialOffset);
  makeHistoryStage1(*allHistoryComputers, requests);
  return makeHistoryStage2(*allHistoryComputers);
  }
;
```

⟨ ModelBuilder::makeRequests implementation 6.5.9 ⟩
⟨ ModelBuilder::allComputers implementation 6.5.8 ⟩
⟨ ModelBuilder::makeHistoryStage1 implementation 6.5.10 ⟩
⟨ ModelBuilder::makeHistoryStage2 implementation 6.5.11 ⟩

This code is used in section 6.5.

The next code returns a container of all possible history computers. This is *very* tightly couple to the entire **HistoryComputer** hiearchy, and arguably should be a static method on the root class of that hiearchy. I don't do that for two reasons.

First, the particular instances needed might vary with the client. Certainly, the names, which are picked here, might vary.

Second, doing so would make the base class (**HistoryComputer**) dependent on all its derived classes, which is very ugly.

⟨ ModelBuilder::allComputers implementation 6.5.8 ⟩ ≡

```
std::auto_ptr < mspath::ModelBuilder::THistoryVector >
    mspath::ModelBuilder::allComputers(double offset) { THistoryVector all;
all.push_back(new TimeInStateComputer("TIS", offset));
all.push_back(new TimeInPreviousStatesComputer("TIP", offset));
all.push_back(new TimeSinceOriginComputer("TSO", offset)); THistoryVector::
    iterator p = all.begin( );
all.push_back(new LnHistoryComputer("LN", *p++));
all.push_back(new LnHistoryComputer("LN", *p++));
all.push_back(new LnHistoryComputer("LN", *p++));
return all.release( ); }
```

This code is used in section 6.5.7.

This just transforms the initial C to C++.

```
⟨ ModelBuilder::makeRequests implementation 6.5.9 ⟩ ≡
    mspath::ModelBuilder::TStringVector mspath::ModelBuilder::makeRequests(const
        char **history,    /* names of history-dependent variables */
     int nhistory    /* number of history dependent variables */
    )
      {
      TStringVector requests;
      for (int i = 0; i < nhistory; ++i)
        requests.push_back(std::string(history[i]));
      return requests;
      }
```

This code is used in section 6.5.7.

Set the state of all computers according to the requests. The first argument is both an input (assumed to have all possible computers) and an output.

```
⟨ ModelBuilder::makeHistoryStage1 implementation 6.5.10 ⟩ ≡
    void mspath::ModelBuilder::makeHistoryStage1(THistoryVector & theComputers, const
          TStringVector& theRequests) throw (mspath::UnknownTerm)
      {
      Mark m(theComputers);
      std::for_each(theRequests.begin( ), theRequests.end( ), m);
      }
```

This code is used in section 6.5.7.

This extracts the computers of interest from our internal set and puts them in the output result.

We need to have all the prerequisites computed before things they depend on.

We do this by constructing a set *untransferred* of all the computers that need to be transferred to the final *ComputerContainer* but have not been. Within that set we evaluate dependencies, and move all items that don't depend on anything in the set to the final result. Reset the dependency information and go again, until all done.

As both *theComputers* and the results are pointer container collections, we need to be careful about transferring ownership.

```
⟨ ModelBuilder::makeHistoryStage2 implementation 6.5.11 ⟩ ≡
    std::auto_ptr < mspath::Model::TComputerContainer >
        mspath::ModelBuilder::makeHistoryStage2( THistoryVector & theComputers )
        throw (mspath::TangledDependencies) { Model::TComputerContainer final;

    typedef std::map < HistoryComputer * , THistoryVector:: iterator > TTransfer;
    TTransfer untransferred;

      // get initial set of computers to move
    for ( THistoryVector:: iterator i = theComputers.begin( );
    i ≠ theComputers.end( ); ++i )
    {
    if (i→isRequired( ))
      untransferred[&*i] = i;
    }

    ⟨ ModelBuilder stage2 main loop 6.5.12 ⟩

    return final.release( ); }
```

This code is used in section 6.5.7.

In the main loop, we repeatedly transfer all items that don't depend on anything in untransferred.

This code is tricky to get right because we modify a container while iterating over it. The standard (§23.1.2–8) says that for **std**::**map** all iterators except the one *erase*'d remain valid after an *erase*. Originally, I attempted to increment the iterator after erasing it; this produced invalid memory references.

⟨ ModelBuilder stage2 main loop 6.5.12 ⟩ ≡

```
size_t modelDataIndex = 0U;    // repeatedly mark and transfer elements
size_t n = untransferred.size( );
while (n > 0U) {
    for ( TTransfer:: iterator i = untransferred.begin( );
i ≠ untransferred.end( ); ) { TTransfer::mapped_type & val = i→second;
HistoryComputer *p = val→requires( ); if (p ≡ 0 ∨ untransferred.count(p) ≡ 0) {
  // transfer didn't work with dissimilar types
  // final.transfer(final.end( ), val, theComputers);
val→setDataIndex(modelDataIndex++);
final.push_back(theComputers.release(val).release( ));    // modify untransferred as we iterate over it
TTransfer:: iterator zap(i);
++i;
untransferred.erase(zap); } else
  {
  ++i;
  } } size_t nold = n;
n = untransferred.size( );
if (n ≡ nold)    // we aren't making any progress
  throw (TangledDependencies("There␣seem␣to␣be␣circular""␣dependencies␣in␣path␣com\
      putation")); }
```

This code is used in section 6.5.11.

We have a certain number of computers, with an order that may differ from what the user wants and with terms the user may not want. They will determine the *modelData* computed for each node. Those computers which the user cares about have *covariateIndex* which is a 0-based index of the terms the user wants. We need to return an array which gives, for each term the user is interested in, the index into the *modelData* for its value.

The values returned for *covariateIndex*( ) must cover the range from 0 up to the number of terms, with no gaps or repeats.

⟨ ModelBuilder::makeHistoryIndirection implementation 6.5.13 ⟩ ≡

```
std::auto_ptr < mspath::TIndirect1D > mspath::ModelBuilder::makeHistoryIndirection(const
        Model::TComputerContainer & theComputers) const
  {
  std::auto_ptr⟨TIndirect1D⟩pIndirect(new TIndirect1D (theComputers.size( )));
  size_t i;
  size_t nUser = 0U;     // number of terms user wants
  for (i = 0; i < theComputers.size( ); ++i)
    if (theComputers[i].isCovariate( ))
      {
      pIndirect→operator [](theComputers[i].covariateIndex( )) = i;
      nUser++;
      }
  if (nUser ≡ pIndirect→size( ))
    return pIndirect;     // shrink
  std::auto_ptr⟨TIndirect1D⟩s(new TIndirect1D (nUser));
  for (i = 0; i < nUser; ++i)
    (*s)[i] = (*pIndirect)[i];
  return s;
  }
```

This code is used in section 6.5.

7 Testing

7.1 Strategies

Validation of the model included the following tests:

- Unit tests of C++ and **R** code. Note that there are separate tests at the C++ and **R** levels, many of which run automatically.
- Check computed likelihood for small problems against manually computed results.
- Check computed likelihood for large problems against results computed with a pure **R** implementation of the model. The latter is in the `src/simulate` directory.
- Check that operations characteristic of distributed optimization (repeated partial computations with varying subsets and parameter values) work correctly in single-processor mode.
- Check results of distributed optimization against same optimization run locally.
- Generate simulated datasets under a model and verify that the model parameters are, on average, recovered by the estimation process. The simulation and the estimation share a lot of code.
- Check that **msm** and **mspath** agree when estimating similar models.

The remainder of this subsection goes into greater detail about the last item.

Since **msm** is a Markov model, it can not be used to evaluate the non-Markov models in which **mspath** specializes. So comparisons must use the subset of **mspath** models that have no path-dependent terms.

msm is a continuous-time model, while **mspath** is discrete time. This causes some non-comparability, because the latter permits only a single state change over a time step, while the former permits multiple state changes in the same period. Consider our canonical model, which permits only moves up to the next state. If **msm** and **mspath** agree on the probability of changing state in an interval, **msm**actually will predict more rapid movement up since some of those changes will jump more than one state. So if one fits both models to the same data, **msm** will appear slower when comparing the probabilities of a jump in a unit interval (as we do below). To minimize this problem, we pick small probabilities of moving even one step, which implies very small probabilities of moving more than one step.[62]

The parameters of the two models also undergo much different transformations on the way to becoming probabilities. Each model results in a linear term (intercept plus coefficients times covariates). For **msm**, call that value a; it is the log-hazard rate. The probability of a move up in the unit interval (the step size in our tests) is

$$1 - e^{-e^a}.$$

For **mspath**, call the corresponding term b, an element in the multinomial logistic, with probability of a move up in our binary logistic case of

$$\frac{1}{1 + e^{-b}}.$$

Equating these terms and solving yields

$$a = \ln\left[\ln(1 + e^b)\right], \tag{18}$$

which re-expresses the parameters of **mspath** (b) on the terms of **msm** (a). As noted immediately before this paragraph, it would not be surprising if this transform of b produced somewhat larger values than the actual fitted a.

Since the relation between a and b is non-linear, the two models do not, in general, match up over arbitrary models and covariates. If the models include only intercepts, comparison is not a problem. Another special case arises with a single, binary covariate x:[63]

$$\begin{aligned} a &= a_0 + a_1 x, \\ b &= b_0 + b_1 x. \end{aligned}$$

Using the only two possible values of the covariate, one gets

$$\begin{aligned} a_0 &= \ln\left[\ln(1 + e^{b_0})\right], \\ a_0 + a_1 &= \ln\left[\ln(1 + e^{b_0+b_1})\right], \text{and thus} \\ a_1 &= \ln\left[\ln(1 + e^{b_0+b_1})\right] - \ln\left[\ln(1 + e^{b_0})\right]. \end{aligned}$$

We checked this special case, as well as some intercept only cases. Note that the likelihoods should be the same.

[62] The analysis in this paragraph is not based on formal proofs, and could be wrong! Surprisingly, the two models seemed to agree well even with significant probabilities—like 1/4—of one step moves.

[63] This case also requires the constraints on the covariates to be a refinement of the constraints on the intercepts. As a negative example, if two transitions have the same coefficient a_1 but two different intercepts, it will not be possible to match up the two different models.

7.2 Remarks about the code

The following material exists only to support testing the main code. Note that some of the methods in the main code, particularly output methods, exist mostly to support testing as well.

The first thing elaborate enough to require testing support classes is the **PathGenerator**.

What is a general path generator required to do?

- It is required to make certain calls to the **Recorder** in certain sequences.
- It is required to generate all valid paths and no invalid paths.
- It should call *goodNode* at least once on each node in valid paths. This includes the terminal node.
- It should construct proper paths, e.g., the pointers should be sensible.

On the other hand, there are a number of features of particular methods that are open:

- The order in which paths are generated.
- The sharing of nodes, if any, between paths.
- The number of invalid entries along the way.
- The production, or absence of production, of good nodes that do not end up on good paths.

The log-likelihood calculations require testing as well. There are many choices of what components (classes) to test and what levels of detail to test.

Levels of detail:

1. Likelihood of one step on one path.
2. Likelihood of one path for one case.
3. Likelihood for one case.
4. Likelihood for a complete dataset.

Testing some of those, at least with some components, may require breaking into the calculation. My inclination is to avoid doing so. However, although paths are normally generated dynamically, for testing purposes I can construct one and have it evaluated. However, the evaluation process is designed to work within the path generation process, so it may be tricky to have everything initialized in the proper sequence. To the extent it is tricky, I have not specified interfaces and assumptions of components adequately.

⟨ Testing Files 7.2 ⟩ ≡
⟨ AllocCounter.h 7.3.1 ⟩
⟨ TestManager.h 7.4.1 ⟩
⟨ TestManager.cc 7.5.1 ⟩
⟨ TestRecorder.h 7.4.2 ⟩
⟨ TestRecorder.cc 7.5.2 ⟩
⟨ TestCovariates.h 7.4.3 ⟩
⟨ TestError.h 7.4.5 ⟩
⟨ NodeFactoryTester.h 7.4.4 ⟩
⟨ NodeFactoryTester.cc 7.5.3 ⟩

This code is used in section 9.

7.3 Monitoring Object Creation and Destruction

For some tests it is useful to know the number of objects of a particular class that are created or destroyed over some section of code. Sometimes it is also useful to know which exact instance we are dealing with. **NodeFactory** was the first class that gave rise to these concerns, which have several sources:

1. Check that there are no memory leaks.
2. Check that objects are being recycled effectively. Early testing showed that allocating new **Node**'s each time one was needed took up a huge amount of time under OS-X. Switching to a design (embodied in **NodeFactory**) that recycled ones that were no longer needed, without recreating them, caused a 3x speedup.
3. Check that objects that should be distinct are distinct.

Counting the number of instance creations and deletions per class helps with all three goals. It is natural to label each instance by the new object count when it was created to achieve the last two goals, and this links naturally to the class level counters.[64]

Because of my initial interest in checking recycling, standard leak detection tools did not seem appropriate. Instead I developed a template that could be combined with an existing class to get one that counted calls to the creator and destructor and identified each object. §7.3.1, page 267 gives the details, and the thread ("templates, constructors, and references" starting January 30, 2007 in `news://comp.lang.c++.moderated` gives some background).[65] There is a slightly different proposal around the 5th item in the thread sited below as well)

Using that template required modifying some other code, specifically **NodeFactory**, so that it took the type of **Node** as a template argument. Since the changes were minimal, this worked out OK.

However, testing **Path** showed the limits of this approach; it would require lots of changes to make sure that **Path** (which is not templated) used the appropriate arguments; the many uses of **Path** in the code would also need to be changed.[66]

This, plus the realization that many memory leak detectors could track allocations and frees (at least on the heap, which is good enough for me) without code modifications led me to look into using one of them. This led to the "Tracking Object Creation and Destruction" thread, also in `news://comp.lang.c++.moderated` starting around February 6.[67] However, they tended to have the following weaknesses:

1. Track individual memory allocations and frees, without grouping them by C++ class.

[64] In principle one could use the object's address as an identifier for the last two goals, but that's complicated. First, one would need to track the object addresses in some kind of hash table. Second, the same C++ object may appear to have different addresses depending on the type of pointer used to reference it. Having a counter is more straightforward.

[65] `http://groups.google.com/group/comp.lang.c++.moderated/browse_thread/thread/883b8037b9c6f247/40a485394bfd1886#40a485394bfd1886`

[66] Or else the **Path** template could be given some other name, with **Path** being an appropriate template instantiation. However, this would require more changes to the code of **Path** itself. Or can a symbol refer both to a class and a template?

[67] Feb 7 was the initial date using UTC: `http://groups.google.com/group/comp.lang.c++.moderated/browse_thread/thread/11ecec56c87e015a/43a1f61da76b58a8#43a1f61da76b58a8`. Unfortunately, none of the responses through Feb. 13, 2007 dealt much with the central problem of classifying memory allocations by type programmatically. Several suggested just running a leak detectory, which might work, although not from **Boost**. One gave an alternate approach to tracking creation and destruction by adding a special instance variable to existing classes.

2. Don't provide a way to tell what the class of a particular instance is (which is undoubtedly a reason for the prior problem).
3. Oriented toward an entire run, rather than part of the code.
4. Oriented toward printed reports.

Individual packages overcome some of these weaknesses, but none I've investigated can get the type information automatically.

Here are some tools that seem in the ballpark (of course, working with C++ rather than just C is a requirement; I'm not sure they all meet that requirement):

LeakTracer `http://www.andreasen.org/LeakTracer/`
Captures the call stack (using a GNU specific feature) and provides printed interpretation of it by controlling a gdb session thru perl. Not a very natural base for tracing a part of a program. Moving programatically from the call stack to the class seems challenging. Does C++ only.

mpatrol `http://www.cbmamiga.demon.co.uk/mpatrol/`
This has an extensive set of functions that can be called programmatically, making it promising for using from a test suite and for monitoring changes in a particular section of code.

There are a number of fields that seem to have type information. One holds the type of allocator (e.g., **new** vs. ***malloc***). The other data (e.g., *typestr*) are empty, at least for my test class.

mpatrol also captures information about the call stack, including the file and line number for each caller. See below for limitations of the call stack. A further limitation of mpatrol is that it deals in mangled names only.

libcwd `http://libcwd.sourceforge.net/`
Strongly oriented to producing output on streams, rather than information to be manipulated programmatically. It can retain the class type of objects, but it only does so if you call *AllocTag*( ) or otherwise alter your source (e.g., there is a *NEW* macro, but to use it you must include a header file and say, e.g., $\mathbf{A} *pA = NEW(\mathbf{A}\,())$). The underlying mechanism is that templates are used to match the type of the pointer at compile-time.

This appears to have functions to unmangle names.

The web site says the program dropped support for BFD, which surprised me; BFD sounds like a good basis for portable debugging.

ccmalloc `http://www.inf.ethz.ch/personal/biere/projects/ccmalloc/`
At first blush looks too stripped down; it's not clear it tracks type info, or that it offers any hooks for programmatic testing.

valgrind `http://valgrind.org`
Works by emulating the CPU and putting hooks in the object code. Works on Linux/x86 and some other platforms. It looks as if it tracks allocations from **new**, but doesn't keep C++ type info and probably has the same problems as other tools getting that info. Code runs much slower and bigger under this tool.

Purify I've used Purify and been very impressed with it. But it is not free software, which poses some hurdles:

Price It's particularly pricey for Unix platforms.

Platforms I've used it on MS-Windows and am not sure of the quality of the Linux/Unix implementation. Probably won't work on some platforms at all.

Non-modifiable If it doesn't already support the functions I need, I'm probably out of luck.

Non-redistributable I would like to redistribute the test suite; I can't if it includes Purify.

The biggest challenge seems to be connecting a particular memory allocation to a particular class. The connection may require both translation from a call stack to a class, and translation between different representations of class types (type as entered in the source code, type as stored by the particular tool, C++ standard typeinfo, printed output).

Unfortunately, one can't "translate" from the call stack to the class since the class constructor is not on the call stack when **new** is called. There are at least three alternatives:

Use the source If the program is compiled with symbolic debugging info, one can go up the stack and get the source line that invoked **new** ( ). Though likely to work pretty well in practice, it could run into trouble if there are multiple calls on one line or the call is spread over multiple lines. More seriously, if the source line is a template, it will not be possible to infer the exact type information from the line with the **new**.[68]

Use the object code A call to the relevant allocator must occur somewhere after the call to new, so one could go up the call stack and then look in the object code for later calls to the constructor. Given compiler cleverness, this too is challenging. One can't count on the allocator call being a set distance after the call to **new** ( ); the next constructor called might not even be the one associated with that particular **new**!

Instrument the program This could be done by inserting calls into the source code so that something like libcwd's *AllocTag* gets called, or some template-based scheme is used. Again, writing a 100% accurate insertion routine is difficult, and some programs that depend on the bit patterns (e.g., streaming data to disk) might break. Or, a hook could be added to the compiler to support this functionality. While that is theoretically clean, it is a substantial project and would require anyone who ran the test to make their own, specially instrumented compiler. Not too practical.

Since some constructors are created automatically, there's not necessarily any code that one could point to for a particular constructor.

My main development is on Linux/i386, though the target is OS-X/PowerPC (and we also have some Suns around–some discussion I've seen suggests they have particularly strong tools for this kind of work).

[68] E.g., **template** ⟨**class C**⟩ **class Foo** { might have a line within it that says **new typename C** ( ); the real type of **C** can't be inferred from that line. In fact, the line may be called with several different types for **C**.

7.3.1 AllocCounter

This is a general-purpose template. It has nothing to do with multi-state path models except that it's used by **NodeFactoryTester**.

AllocCounter⟨**Foo**⟩ wraps class **Foo** with methods that track constructions and destructions of class instances. Those counts are available through static methods.

The name is slightly misleading, since this class does not track (just) heap allocations (with **new** and **delete**). The class tracks object creation and destruction. The latter may occur, for example, on the stack with automatic variables.

The only area that might require extension is creating constructors with enough arguments to match the classes being wrapped. Also, you may wish to custumize the way instances are printed.

Because this is a general purpose class I considered not placing it in the **mspath** namespace. But it seems safer to do so, in case someone else has defined a similar class.

Note there are printing methods for both the instance and the class.

```
⟨ AllocCounter.h 7.3.1 ⟩ ≡
  @o AllocCounter.h
#ifndef AllocCounter_h
#define AllocCounter_h 1

#include <cstddef>    // size_t
#include <ostream>
#include <typeinfo>

  using std::size_t;

  namespace mspath
    {

    template ⟨class C⟩
    class AllocCounter : public C
      {
    public:
      ⟨ AllocCounter Constructors 7.3.1.1 ⟩
      ⟨ AllocCounter Count Accessors 7.3.1.2 ⟩
      ⟨ AllocCounter Actions 7.3.1.3 ⟩
      ⟨ AllocCounter Instance Operations 7.3.1.5 ⟩

    protected:
      ⟨ AllocCounter Data 7.3.1.4 ⟩
      };

    ⟨ AllocCounter Printing 7.3.1.6 ⟩
    ⟨ AllocCounter Static Initialization 7.3.1.7 ⟩
    }

#endif
```

This code is used in section 7.2.

Constructors are a problem. We know neither the number nor type of arguments of constructors of the underlying class. This seems like a good place to use templated functions (namely constructors) within the larger template, since the template arguments can match arbitrary types.

The first problem is that we must know how many arguments there are. A work-around is to make templated constructors with 0, 1, 2, etc arguments to whatever limit seems necessary (I stopped at 3 below).

But a nasty issue remains: references. If the base class has a constructor like **C** (**const** *X1* & , *X2*) and the the template is declared with arguments (*A1*, *A2*) then the call to the templatized constructor will *not* use a reference and *will construct a temporary.* This can lead to trouble if the address of the argument is then taken (which I did), or if object creation and destruction has significant effects. On the other hand, declaring a template constructor arguments like (*A1* & , *A2* &) fails to match temporaries, which are **const**. Note that a function declared **A** *f*( ) is returning a temporary. Declaring templates with **A** and either **A**& or **const A**& will produce a template resolution error, but declaring both references without the vanilla type is OK. I have only declared constructors with **const** & types. If some of the underlying classes use & types you will need to add appropriate templates.[69]

⟨ AllocCounter Constructors 7.3.1.1 ⟩ ≡

```
AllocCounter( ) : myAllocID(myAlloc++), C( )
  { }

template ⟨typename A⟩
AllocCounter(const A& a) : C(a), myAllocID(myAlloc++)
  { }

template ⟨ typename A1, typename A2 > AllocCounter (const A1 & a1, const A2 & a2) :
      C (a1, a2), myAllocID(myAlloc++)
  { }

template ⟨ typename A1 , typename A2, typename A3 > AllocCounter (const A1 & a1,
      const A2 & a2, const A3 & a3) : C (a1, a2, a3), myAllocID(myAlloc++)
  { }

  // copy constructor
AllocCounter (const AllocCounter& x) : C (x), myAllocID(myAlloc++)
  { }

virtual ~AllocCounter( )
  {
  myFree++;
  }
```

This code is used in section 7.3.1.

[69] Thanks to patrik.kahari@googlemail.com and Ulrich Eckhardt <eckhardt@satorlaser.com> for help on this issue. See the "templates, constructors, and references" thread, begining January 30, 2007 in the Usenet group comp.lang.c++.moderated.

I also considered defining one constructor **AllocCounter**(**const C**&), but that requires creation of a temporary and breaks the ability to drop the new, counted class in place of the old.

These class methods allow interested parties to track the number of instance constructions and destructions.

⟨ AllocCounter Count Accessors 7.3.1.2 ⟩ ≡

```
static size_t nAlloc( )
  {
  return myAlloc;
  }
static size_t nFree( )
  {
  return myFree;
  }
static size_t nInUse( )
  {
  return myAlloc − myFree;
  }
```

This code is used in section 7.3.1.

⟨ AllocCounter Actions 7.3.1.3 ⟩ ≡

```
static void resetAllocCounts( )
  {
  myAlloc = 0U;
  myFree = 0U;
  }
  // Note there is no std::endl at the end of the output
static std::ostream & statsPrint(std::ostream & str)
  {
  str << typeid (C).name( ) << " has " << nInUse( ) << " instances after " << nAlloc( ) <<
      " constructions and " << nFree( ) << " destructions.";
  return str;
  }
```

This code is used in section 7.3.1.

Each instance gets a unique ID, and the class tracks constructions and destructions.

⟨ AllocCounter Data 7.3.1.4 ⟩ ≡

```
static size_t myAlloc;
static size_t myFree;
size_t myAllocID;
```

This code is used in section 7.3.1.

⟨ AllocCounter Instance Operations 7.3.1.5 ⟩ ≡

```
    // instance copy; do not copy myAllocID
  AllocCounter& operator =(const AllocCounter& rhs)
    {
    this→C::operator =(rhs);
    return ∗this;
    }

  size_t allocID( ) const
    {
    return myAllocID;
    }
```

This code is used in section 7.3.1.

This provides a simple wrapper to output the relevant statistics. Feel free to specialize if something else is desired.

⟨ AllocCounter Printing 7.3.1.6 ⟩ ≡

```
  template ⟨class C⟩
  std::ostream & operator <<(std::ostream & str, const AllocCounter⟨C⟩& c)
    {
    str << "Instance␣" << c.allocID( ) << ":" << std::endl << static_cast⟨const
        C&⟩(c) << std::endl << c.nAlloc( ) << "␣class␣instances␣construc\
        ted␣and␣" << c.nFree( ) << "␣freed." << std::endl;
    return str;
    }
```

This code is used in section 7.3.1.

The program will not link without the next section.

⟨ AllocCounter Static Initialization 7.3.1.7 ⟩ ≡

```
  template ⟨class C⟩
  size_t AllocCounter⟨C⟩::myAlloc = 0U;
  template ⟨class C⟩
  size_t AllocCounter⟨C⟩::myFree = 0U;
```

This code is used in section 7.3.1.

7.4 Testing Interfaces

7.4.1 TestManager

This class tests **PathGenerator**, with the help of **TestRecorder**. To use it, you must meet the following general requirements:

- The **Data**, and **Model** are already set up.
- The expected paths are in a file.
- The order of paths in that file must match the order of cases in the **Data** by **Id**. That is, if i is the **Id** of the first case in the data, then the file should have all expected paths for i first. If j is the next case in **Data**, all expected paths for j should appear next in the file. Usually there will be a 1 to many relation from the cases in **Data** to the expected paths.

Note the following non-requirements:

- The order of expected paths within a given case is irrelevant; they do not need to match the order in which the **PathGenerator** produces paths. The test simply verifies that all expected paths, and only those, are produced by the **PathGenerator**.[70]
- This class reads the expected paths a little bit at a time, so it is not necessary for all the expected paths to fit in memory at once.

In addition, this class specializes to operate only under the following conditions:

- There are 5 states, numbered 0–4.[71]
- All cases start at true state 0 at time 0.
- This initial state is known accurately, so there are not multiple trees per case (as there would be if there were some chance the person was in a true state of 1, observed as 0).
- The true process allows only moves up one state.
- Time is measured in integers, and the grid spacing is one time unit.
- Each record[72] of the input file should begin with the id and then have fail1 cum1 and as many more fail cum pairs as necessary. failn is 1 if the process ever entered state n, otherwise 0. If failn is 1, cumton is the first time (an integer) the process entered state n. If failn is 0, cumton is the last time observed in state $n-1$ and that is the last entry in the record.

[70] The easiest thing is to order both the input data and expected paths by **Id**. If this class were extended to deal with situations in which there was more than one possible starting state for the paths, then the sorting of both should be by **Id** and then initial **State**.

[71] If you want, you can specify an arbitrary maximum state value as an optional argument to the constructor.

[72] ordinarily a line, though any whitespace separators will do

Typical useage is to instantiate the class and call the *go* method. Most errors are likely to throw exceptions, but you should also check that the return value is `true` to verify the test passed.

As a simple utility, this is not a subclass of **Manager**.

⟨ TestManager.h 7.4.1 ⟩ ≡
```
@o TestManager.h
#ifndef TestManager_h
#define TestManager_h 1

#include <fstream>
#include <memory>
#include <string>
#include <ctime>

#include "basic.h"
#include "Data.h"
#include "Environment.h"
#include "Model.h"
#include "PathGenerator.h"
#include "StateTimeClassifier.h"
#include "SuccessorGenerator.h"
#include "TestError.h"    // potential for use in throw specification
#include "TestRecorder.h"
#include "TimeStepsGenerator.h"

namespace mspath {

class TestManager
  { public:
      ⟨ TestManager Constructors 7.4.1.1 ⟩


    // If seconds > 0 print out progress roughly that often
    virtual bool go(std::size_t seconds = 0);    // return true on success
    // but more likely to throw an error first if fail

    ⟨ TestManager Accessors 7.4.1.3 ⟩

    protected:
      ⟨ TestManager States 7.4.1.5 ⟩
      ⟨ TestManager Helpers 7.4.1.4 ⟩
      ⟨ TestManager Data 7.4.1.2 ⟩
      } ;    // end TestManager
  }    // end namespace mspath
#endif    // TestManager_h
```

This code is used in section 7.2.

The class this tests is hardwired. We assume ownership of the pointers passed in.

⟨ TestManager Constructors 7.4.1.1 ⟩ ≡

```
TestManager(Data *pData, Model *pModel, const char *expectedOutputFilename, unsigned
    int maxState= 4) :
mypData(pData), myEnv(pData), mypModel(pModel), myTSGen(1),
      myStream(expectedOutputFilename), myTestRecorder(&myEnv),
      myStateTimeClassifier(pModel), mySuccessorGenerator(pModel),
      myPathGenerator(&myEnv, &myTestRecorder, &myStateTimeClassifier,
      &mySuccessorGenerator), myMaxState(maxState), myState(initial)
  {}
  // because we use auto_ptr no special action needed
virtual ~TestManager( )
  {}
;
```

This code is used in section 7.4.1.

The naming of the file associated with *myStream* is a bit confusing. The file is an input to this class (hence it is an *ifstream*), but it contains the expected *outputs* of the **PathGenerator**.

⟨ TestManager Data 7.4.1.2 ⟩ ≡

```
std::auto_ptr⟨Data⟩mypData;
Environment myEnv;
std::auto_ptr⟨Model⟩mypModel;
TimeStepsGenerator myTSGen;
std::ifstream myStream;    // order of next items matters for proper initialization
TestRecorder myTestRecorder;
StateTimeClassifier myStateTimeClassifier;
SuccessorGenerator mySuccessorGenerator;
PathGenerator myPathGenerator;    // this is what we test
const unsigned int myMaxState;    // highest allowed state number
  // track where we are in processing
TestState myState;
Id myAheadId;    // the Id we have read for the next line of output file
```

This code is used in section 7.4.1.

The accessors are probably only of internal interest to this class, though they are public.

⟨ TestManager Accessors 7.4.1.3 ⟩ ≡

```
Environment& environment( )
  { return myEnv; }

Data& data( )
  { return *mypData; }

Model& model( )
  { return *mypModel; }

PathGenerator& pathGenerator( )
  { return myPathGenerator; }

TimeStepsGenerator& timeStepsGenerator( )
  { return myTSGen; }

TestRecorder& recorder( )
  { return myTestRecorder; }

unsigned int maxState( ) const
  { return myMaxState; }
```

This code is used in section 7.4.1.

These methods assist *go* and are called by it. They are for internal use only.

⟨ TestManager Helpers 7.4.1.4 ⟩ ≡

```
virtual void doInitialProcessing( );
virtual void generateTimeSteps( );
virtual void generateExpectedPaths( );
```

This code is used in section 7.4.1.

Since we are constantly looking ahead, we need to track how much we've seen.

We can't call this **State** or we get a collision with our basic type.

Since we typically read ahead to get the next id, when we first enter the *eof* state we will still have a good line we've just read to process.

⟨ TestManager States 7.4.1.5 ⟩ ≡

```
enum TestState
  {
  initial,    // very start of processing
   readid,    // we have read ahead one id
   eof    // no more data are available
  };
```

This code is used in section 7.4.1.

7.4.2 TestRecorder

This class exists solely to test whether a **PathGenerator** is working properly. It can be used on its own for simple one shot tests, or in conjunction with **TestManager** for more elaborate ones of entire data sets.

Among other things, the inputs to this class include a list of anticipated good paths. Some way of comparing those to the actual paths in a semantically meaningful way is necessary. **Path**::**operator** $\equiv$ was implemented in support of this class.

I subclass from **SimpleRecorder** so that its statistics are also available to the testing program.

There are two typical useage patterns.

The first is a stand-alone test in which you are checking a single run of the **PathGenerator**. To do so, instantiate this class. Then fill in the expected paths by calling *addExpectedPath* repeatedly. Clone the **Path** argument of that function if necessary.[73]).

One you have filled in the expected paths, you call the owning **PathGenerator**, being sure to mimic the full sequence of calls expected.

When you have made the last *finishSession* call, check *isOK* on this class. It's very likely you'll get an exception thrown earlier if something is wrong, however.

The second useage pattern involves a complete set of input data and expected paths. In this case, you need to fill in the expected paths before each call to **PathGenerator**::*startTree*. Call **TestRecorder**::*clear* before each time you fill the expected paths. You probably want to use **TestManager**, or a variant, if you go this route.

Here are the errors this class currently throws. All exceptions except the last are subclasses of **std**::*runtime_error*. All are inside the **mspath**::**test** namespace:

DuplicatePath caused by an attempt to enter the same path twice in any set. Among the possible causes are errors initializing this class and bad problems encountered during the run, such as multiple copies of the same, unanticipated path.

IllegalStateChange This indicates the **PathGenerator** is not making calls to the recorder in the expected sequence.

PathGeneratorError for a range of other run-time errors indicating the test has failed.

TestInternalError may be thrown by **TestRecorder**::**LegalMoves** if it has been programmed incorrectly. This is a type of **std**::**logic_error**.

[73]It would be necessary if successive **Path**'s had some common nodes, that is ones with the same object identity. It would also be necessary if the client wished to retain ownership of the original pointed to object.

"mspath.C" 7.4.2 ≡
@f *Path int*
@f *TInnerComparator int*
@f *TComparator int*
@f *TPPath int*

⟨ TestRecorder.h 7.4.2 ⟩ ≡
@o TestRecorder.h
#ifndef *TestRecorder_h*
#define *TestRecorder_h* 1

#include <iostream>
#include <map>
#include <sstream>
#include <stdexcept>
#include <vector>

#include <boost/ptr_container/ptr_set.hpp>

#include "basic.h"
#include "Environment.h"
#include "SimpleRecorder.h"
#include "TestError.h"

namespace mspath {
class TestRecorder : **public SimpleRecorder**
{

public:
typedef SimpleRecorder Super;
⟨ PathSet Declaration 7.4.2.7 ⟩
⟨ TestRecorder Constructors 7.4.2.1 ⟩
⟨ TestRecorder Setup Declaration 7.4.2.2 ⟩
⟨ TestRecorder Accessors 7.4.2.3 ⟩
⟨ TestRecorder Actions 7.4.2.4 ⟩

protected:
⟨ TestRecorder State 7.4.2.9 ⟩
⟨ TestRecorder Data 7.4.2.5 ⟩} ; // end **TestRecorder**

⟨ PathSet free functions 7.4.2.8 ⟩} // end **namespace mspath**
#endif // *TestRecorder_h*

This code is used in section 7.2.

⟨ TestRecorder Constructors 7.4.2.1 ⟩ ≡
TestRecorder (**Environment** ∗**const** *thepEnv*) : **SimpleRecorder**(*thepEnv*)
{ }

virtual ∼**TestRecorder**( )
{ }

This code is used in section 7.4.2.

Yes, probably I should add a **throw** spec to the next two functions. The problem is, I'm not sure that's the only exception they could throw.

⟨ TestRecorder Setup Declaration 7.4.2.2 ⟩ ≡

```
// takes ownership of the pointer argument
virtual void addExpectedPath(Path *const pp) throw (test::DuplicatePath,
    test::PathGeneratorError);
```

This code is used in section 7.4.2.

⟨ TestRecorder Accessors 7.4.2.3 ⟩ ≡

```
// return expected paths for the current tree
PathSet& expectedPaths( )
  { return myExpectedPaths; }

const PathSet& expectedPaths( ) const
  { return myExpectedPaths; }

// ——for end of run for tree or everything——

// true if run went OK
// most likely an error will be thrown before getting here

virtual bool isOK( ) const
  { return myLegal.isDone( ); }

// counts

virtual TSize countUnmatchedPaths( ) const
  { return expectedPaths( ).size( ); }

virtual TSize countMatchedPaths( ) const
  { return myMatchedPaths.size( ); }

virtual TSize countUnanticipatedPaths( ) const
  { return mySurprisePaths.size( ); }

// sets

virtual const PathSet& unmatchedPaths( ) const
  { return expectedPaths( ); }

virtual const PathSet& matchedPaths( ) const
  { return myMatchedPaths; }

virtual const PathSet& unanticipatedPaths( ) const
  { return mySurprisePaths; }
```

This code is used in section 7.4.2.

The main action comes when a good path is detected.

⟨ TestRecorder Actions 7.4.2.4 ⟩ ≡

```
    // reset counters and paths
  virtual void clear( );

  virtual void goodPath(Path& path);

    // the following are just here to monitor state change
    // call at beginning of processing
  virtual void startSession( );

    // call at the start of processing a given case
  virtual void startCase( );

    // call when starting a tree of paths
  virtual void startTree(double initialProbability);

    // call every time we've encountered a (seemingly) good node
    // of course, later, it may be revealed to be part of a bad path

  virtual void goodNode( )
    { myLegal.setState(sgoodNode); mypLastNode   =
        &(environment( ).currentNode( )); Super::goodNode( ); }

    // When client generates a node at the end of a good path, it
    // should call goodNode( ) and goodPath( ).

    // call every time we discover we have wandered into an invalid or
    // impossible path.

  virtual void impermissible( )
    { myLegal.setState(simpermissible); Super::impermissible( ); }

    // finished tree
  virtual void finishTree( );

    // finished processing case
  virtual void finishCase( ) throw (DataModelInconsistency);

    // completely done. Might clean up, output results, etc.
  virtual void finishSession( );

    // for diagnostic use
  std::ostream & dumpSets(std::ostream & s) const;
```

This code is used in section 7.4.2.

⟨ TestRecorder Data 7.4.2.5 ⟩ ≡

```
PathSet myExpectedPaths;
PathSet myMatchedPaths;
PathSet mySurprisePaths;
LegalMoves myLegal;
Node *mypLastNode;    // holds address of last arg to goodNode
  // cache during run
Id myId;
```

This code is used in section 7.4.2.

PathSet is an internal data structure for **TestRecorder**. It tracks paths that should be seen, have been seen, etc. These classes own their data and destroy it when they are destroyed. Be careful to use the *transfer* function to move pointers from one of these containers to another without getting 0 or 2 copies of the path.

Rather than tracking duplicates I blow up when one is found; in proper use, there should never be duplicates.

⟨ PathSet Declaration 7.4.2.7 ⟩ ≡

```
class PathSet : public boost:: ptr_set⟨Path⟩ {
public:
  virtual ~PathSet( )
    { }
  ;
    // insert, throwing an error if path already present
  virtual iterator insertUnique(Path *const p) throw (test::DuplicatePath); virtual
      void transferUnique(iterator i, PathSet& p) throw (test::DuplicatePath);
  protected: typedef boost:: ptr_set⟨Path⟩ Super; } ;
  typedef PathSet::size_type TSize;
```

This code is used in section 7.4.2.

Unlike the previous material, the following is at the **mspath** namespace scope.

⟨ PathSet free functions 7.4.2.8 ⟩ ≡

```
std::ostream & operator <<(std::ostream & s, const TestRecorder::PathSet & p);
```

This code is used in section 7.4.2.

To determine if we are being called in the proper sequence, we track the state and legal transitions.

⟨ TestRecorder State 7.4.2.9 ⟩ ≡
```
    // the ordering of the next few items is critical
  ⟨ TestRecorder state enum 7.4.2.10 ⟩
  ⟨ TestRecorder::LegalMoves 7.4.2.11 ⟩
```

This code is used in section 7.4.2.

The following enum lists all possible state-changing messages. Some, but not all of them represent legal states, since some do not cause a state change.

To avoid collision with function names, I preface each state with s for state.

⟨ TestRecorder state enum 7.4.2.10 ⟩ ≡
```
  enum GeneratorState
    { sinitial, sstartSession, sstartCase, sstartTree, sgoodNode, simpermissible, sgoodPath,
        sfinishTree, sfinishCase, sfinishSession };
```

This code is used in section 7.4.2.9.

LegalMoves records the current state and throws an exception if an illegal state change is attempted.

⟨ TestRecorder::LegalMoves 7.4.2.11 ⟩ ≡
```
  class LegalMoves
    {
  public:
    LegalMoves( ); void setState(GeneratorState s) throw (test::IllegalStateChange,
        test::TestInternalError);
    GeneratorState currentState( ) const
      { return myCurrentState; }
    bool isDone( ) const
      { return myCurrentState ≡ sfinishSession; }
    protected:
        typedef std:: vector ⟨GeneratorState⟩ TAllowed;  typedef
        std::map < GeneratorState, TAllowed⟩ TStateMap;
    TStateMap myGoodMoves;    // could be static const
    GeneratorState myCurrentState;
    } ;
```

This code is used in section 7.4.2.9.

7.4.3 TestCovariates

This is a simple wrapper so we can test things that use covariates. It makes no reference to the **Environment**.

```
⟨ TestCovariates.h 7.4.3 ⟩ ≡
    @o TestCovariates.h
  #ifndef TestCovariates_h
  #define TestCovariates_h 1
  #include "Covariates.h"
    namespace mspath
      {
      class TestCovariates : public AbstractCovariates
        {
      public:
        TestCovariates(Double2D& theMatrix, size_t theCol = 0) : myMatrix(theMatrix),
              myc(theCol), myCache(theMatrix.nrows( ))
          { }
        virtual ~TestCovariates( )
          { }
        ;
        virtual bool isChanged(Environment& theEnv)
          {
          return true;
          }
        virtual Double1D& values(Environment& theEnv)
          {
          myCache = myMatrix.col(myc);
          return myCache;
          }
        void setColumn(size_t i)
          {
          myc = i;
          }
      protected:
        Double2D& myMatrix;
        size_t myc;    // column to pull out
        Double1D myCache;
        };
      }
  #endif
```

This code is used in section 7.2.

7.4.4 NodeFactoryTester

This class exercises a **NodeFactory** by generating and deleting **Node**'s. To track the underlying allocations, **NodeFactoryTester** defines and uses **TestNodeFactory**, which is simply a regular **NodeFactory** that generates **TestNode**'s instead of **Node**'s.

The class not only asks the **NodeFactory** to generate and delete **Node**'s, it uses **Boost** assertions to check that everything is operating properly. Thus the expected useage pattern is that within a **Boost** unit test you will create a **NodeFactoryTester**::**TestNodeFactory** and then use that to create the **NodeFactoryTester**. Then make repeated calls to **NodeFactoryTester**::**test** to perform the tests.

For proper cleanup you should delete the **NodeFactoryTester** and the **TestNodeFactory**, in that order.

This is a "white box" test that relies on knowledge of the internals of **NodeFactory** and makes checks beyond those that could be done with the public interfaces to **NodeFactory**⟨**Node**⟩. In particular, it would be perfectly legitimate (in terms of the public interface) if a **NodeFactory** created and destroyed a **Node** every time it received *createNode*( ) and *destroyNode*( ), respectively. However, a major motivation for creating **NodeFactory** was to reuse to objects to cut down on the overhead of object creation. This overhead was substantial on some platforms. These tests look behind the scenes to verify that the expected recycling is going on.

The class here, unlike other Test classes, is not a testing version of the class being tested; it is a tester of that class. It uses **NodeFactory** but is not a **NodeFactory**.

This test generates a huge number of **Boost** assertions.

⟨ NodeFactoryTester.h 7.4.4 ⟩ ≡

```
@o NodeFactoryTester.h
#ifndef NodeFactoryTester_h
#define NodeFactoryTester_h 1

#include "NodeFactory.h"
#include "AllocCounter.h"

#include <vector>

  namespace mspath
    {
    class NodeFactoryTester
      {

    public:
      ⟨ NodeFactoryTester Public Types 7.4.4.1 ⟩
      ⟨ NodeFactoryTester Constructors 7.4.4.2 ⟩
      ⟨ NodeFactoryTester Actions 7.4.4.3 ⟩

    protected:


      ⟨ NodeFactoryTester Internal Types 7.4.4.4 ⟩
      ⟨ NodeFactoryTester Internal Accessors 7.4.4.6 ⟩
      ⟨ NodeFactoryTester Expected Values 7.4.4.5 ⟩
      ⟨ NodeFactoryTester Support Interface 7.4.4.8 ⟩
      ⟨ NodeFactoryTester Data 7.4.4.7 ⟩
      };
    }
#endif
```

This code is used in section 7.2.

The class performs several passes; each pass creates and then destroys a number of steps.

⟨ NodeFactoryTester Public Types 7.4.4.1 ⟩ ≡

```
typedef AllocCounter⟨Node⟩ TestNode;
typedef NodeFactory⟨TestNode⟩ TestNodeFactory;

typedef size_t TPass;
typedef size_t TStep;
```

This code is used in section 7.4.4.

You construct a tester by giving it a pointer to the **TestNodeFactory**, but you retain ownership of that object and are responsible for destroying it. The **TestNodeFactory** determines the number of history variables that are kept at each **Node**.

biggestStep must exceed the maximum number of steps you will use during testing.

Finally, you may specify the initial pass, if you wish to consider the tester to be starting at some pass after the initial pass 0.

⟨ NodeFactoryTester Constructors 7.4.4.2 ⟩ ≡

```
NodeFactoryTester(TestNodeFactory *pNF, TPass biggestStep = 1000, TPass initialPass = 0);

~NodeFactoryTester( );
```

This code is used in section 7.4.4.

test( ) triggers the main action of the tester. It generates **Node**'s from the current step up to step *up*. Then it destroys nodes down to step *down*. It checks the validity of the data after both the up and down phases. In general, these actions will trigger object allocation and destruction, and so may change counts in classes built from **AllocCounter**.

Prerequisites: $up > down$, $up \geq step$, $up \leq biggestStep$, where *step* indicates the current last step (i.e., 0 at the start, or the value of the *down* in the prior call later) and *biggestStep* is the argument given to the constructor.

Repeated calls to this method induce a useage pattern similar to that of the **PathGenerator**, in which a path is evaluated out to the end, then the path is backed up to an ancestor step, and a new path is regenerated from that point. Each call to **test** is considered a *pass*.

⟨ NodeFactoryTester Actions 7.4.4.3 ⟩ ≡

```
// create path to step up, then destroy backwards through down
void test(TStep up, TStep down);
```

This code is used in section 7.4.4.

⟨ NodeFactoryTester Internal Types 7.4.4.4 ⟩ ≡

```
typedef AllocCounter⟨StatePoint⟩ TestStatePoint;
typedef AllocCounter⟨TimePoint⟩ TestTimePoint;
```

This code is used in section 7.4.4.

The following functions generate distinctive values for each piece of data associated with a **Node**. The idea is to make it easy to tell if those data have become corrupt.

These have various magic numbers hardcoded in; since the factor needed to separate the pass from the step depends on the number of steps, that is set in the constructor.

⟨ NodeFactoryTester Expected Values 7.4.4.5 ⟩ ≡

```
State expectedState(TPass pass, TStep step) const
  {
  return pass * PASS_FACTOR + step;
  }

  // this is the Time in the StatePoint

Time expectedSPTime(TPass pass, TStep step) const
  {
  return step + pass / 10.0;
  }

  // This is the Time in the TimePoint, and thus the Node::time( ).
  // Since TimePoint's are shared by passes,there is no pass argument.

Time expectedTPTime(TStep step) const
  {
  return step + 0.5;
  }

  // whether the TimeStep is for an observation time

bool expectedMatchesObservation(TStep step) const
  {
  return step % 3U ≡ 0U;
  }

  // whether Node has been counted as good

bool expectedCountedAsGood(TPass pass, TStep step) const
  {
  return step % 5U ≡ pass % 5U;
  }

  // main results for Node

EvaluationData expectedEvaluationData(TPass pass, TStep step) const
  {
  return expectedState(pass, step) * 10.0;
  }

  // auxiliary data, only relevant if history is used

ModelData expectedModelData(TPass pass, TStep step) const
  {
  return
      static_cast⟨ModelData⟩ (static_cast⟨ModelData::value_type⟩ (expectedEvaluationData(pass,
      step)) + myHistoryOffsets);
  }
```

This code is used in section 7.4.4.

⟨ NodeFactoryTester Internal Accessors 7.4.4.6 ⟩ ≡

```
TestNodeFactory& nodeFactory( )
  {
  return *mypNodeFactory;
  }
const TestNodeFactory& nodeFactory( ) const
  {
  return *mypNodeFactory;
  }
// number of path variables in use
size_t nPath( ) const
  {
  return nodeFactory( ).myNPath;
  }
```

This code is used in section 7.4.4.

For ease of coding, I don't prefix the instance variables with *my*.

⟨ NodeFactoryTester Data 7.4.4.7 ⟩ ≡

```
const TPass PASS_FACTOR;    // multiply pass by this value for
  // unique values
  // Next variable is effectively const after construction
ModelData myHistoryOffsets;    // used to create unique history values
TestNodeFactory *mypNodeFactory;    // reference, not ownership
TPass pass;
TStep step;
TStep highWater;    //1+largest step index so far
  // expected counts;
size_t SPnAlloc;    // TestStatePoint creations
size_t SPnFree;    // and deletes
size_t TPnAlloc;    // TimePoint
size_t TPnFree;
size_t NDnAlloc;    // TestNode
size_t NDnAllocBase;    // records value when this tester made
size_t NDnFree;
  // The following vectors are all indexed by step
std:: vector ⟨TPass⟩vPass;    // Step s created by pass vPass[s]
std:: vector < TestNode *⟩ vNode;    // reference only; *mypNodeFactory owns
std:: vector < TestStatePoint *⟩ vSP;    // ownership of pointed to objects
std:: vector < TestTimePoint *⟩ vTP;    // ownership of pointed to objects
```

This code is used in section 7.4.4.

These are internal utility functions. The first two set the internal counts to match the values held in the global construction/destruction counts and verify that those two sets of values match. The setter is only for unusual circumstances; prefer updating the counts manually as you go to using *snapshotCounts*( ).

The last one validates all supposedly valid nodes.

⟨ NodeFactoryTester Support Interface 7.4.4.8 ⟩ ≡
```
void snapshotCounts( );
void checkCounts( ) const;

void validate( );
```

This code is used in section 7.4.4.

7.4.5 Exceptions for Testing

These are to be included by other files at global namespace. Because both **TestRecorder** and **TestManager** can throw errors, and the errors are substantively similar, I group them in the **mspath**::**test** namespace.[74]

⟨ TestError.h 7.4.5 ⟩ ≡
```
@o TestError.h
#ifndef TestError_h
#define TestError_h 1

#include <stdexcept>
#include <sstream>
#include <string>

#include "basic.h"
#include "Path.h"

  namespace mspath {
      namespace test {
      ⟨ Top-Level Testing Errors 7.4.5.1 ⟩
      ⟨ DuplicatePath Error 7.4.5.2 ⟩
      ⟨ Illegal State Change Error 7.4.5.3 ⟩


  }   // end namespace test
  }   // end namespace mspath

#endif    // TestError_h
```

This code is used in section 7.2.

[74] The alternative would be to make the errors part of **TestManager** or **TestRecorder**.

You can use the following errors directly or inherit from them.

⟨ Top-Level Testing Errors 7.4.5.1 ⟩ ≡

```
// when the data or behavior under test are not as expected
struct TestError : public std ::runtime_error
  { TestError (const std::string & msg) : std::runtime_error (msg)
    { } virtual ~TestError( ) throw ( )
    { } } ;

  // when the test system itself is messed up
  struct TestInternalError : public std ::logic_error { TestInternalError (const
      std::string & msg) : std::logic_error (msg)
    { } virtual ~TestInternalError( ) throw ( )
    { } } ;

  // Specific ones for this case
  struct PathGeneratorError : public TestError
    { PathGeneratorError (const std::string & msg) : TestError(msg)
      { } virtual ~PathGeneratorError( ) throw ( )
      { } } ;
```

This code is used in section 7.4.5.

⟨ DuplicatePath Error 7.4.5.2 ⟩ ≡

```
// clients should include "Path.h" if they care
struct DuplicatePath : public PathGeneratorError
  {
  DuplicatePath (const Path *const p) : PathGeneratorError("Duplicate␣Path␣insertion␣\
      attempted"), path(p)
    { }

  const Path *path;
  };
```

This code is used in section 7.4.5.

We throw the next error if state problems arise, i.e., if calls are made in the wrong sequence.

⟨ Illegal State Change Error 7.4.5.3 ⟩ ≡

```
struct IllegalStateChange : public PathGeneratorError
  { IllegalStateChange (State theFrom,
      State theTo) : PathGeneratorError("Illegal␣State␣Transition␣attempted"),
      from(theFrom), to(theTo)
    { }

  virtual const char *what( )const  throw ( )
    { std::ostringstream s; s  <<  "Illegal␣transition␣attemp\
      ted␣from␣" << from << "␣to␣" << to; return s.str( ).c_str( ); }

  State from;
  State to; } ;
```

This code is used in section 7.4.5.

7.5 Testing Implementation

7.5.1 TestManager

⟨ TestManager.cc 7.5.1 ⟩ ≡

```
    @o TestManager.cc
  #include "TestManager.h"
  #include <iostream>

  #include <sstream>

    namespace mspath {
        ⟨ TestManager::go 7.5.1.1 ⟩
        ⟨ TestManager::doInitialProcessing 7.5.1.2 ⟩
        ⟨ TestManager::generateTimeSteps 7.5.1.3 ⟩
        ⟨ TestManager::generateExpectedPaths 7.5.1.4 ⟩
    }   // end mspath
```

This code is used in section 7.2.

go returns true on success, though throwing an error is more likely if the test fails.

Note that much of the testing for errors occurs in **TestRecorder**.

Since this may take a long time to run, we provide the option to output a progress report (of the number of good paths so far) every *seconds* seconds. If it's 0, we skip the reports. I believe the implementation is reasonably portable, depending only on the C++ standard with one further assumption from POSIX.

```
⟨ TestManager::go 7.5.1.1 ⟩ ≡
    // the next def is supposedly a POSIX standard
  #define CLOCKS_PER_SECOND 1000000
    bool TestManager::go(std::size_t seconds)
      {
      std::clock_t ticks = seconds * CLOCKS_PER_SECOND; std::clock_t nextTick =
          std::clock( ) + ticks; std::clock_t now; pathGenerator( ).startSession( );
      while (environment( ).next( ))
        {
        if (myState ≡ eof)
          { std::stringstream s;
              s << "Ran out of expected output data on input case " <<
              environment( ).id( ); throw test::PathGeneratorError(s.str( )); }
        if (myState ≡ initial)
          doInitialProcessing( );
        if (myAheadId ≠ environment( ).id( ))
          { std::stringstream s;
              s << "Input Data on Case " << environment( ).id( ) << " but expected output on c\
              ase " << myAheadId; throw test::PathGeneratorError(s.str( )); }
        generateTimeSteps( );
        pathGenerator( ).startCase( );
        generateExpectedPaths( );
        pathGenerator( ).startTree(StatePoint (0, 0), 1.0);
        pathGenerator( ).finishCase( );
        if (ticks > 0)
          {
          now = std::clock( );
          if (now > nextTick)
            {
            std::cout << recorder( ).goodPaths( ) << " . . " << std::flush;
            nextTick = now + ticks;
            }
          }
        }   // we have now read all the input data
      if (myState ≠ eof)
        { std::stringstream s;
            s << "All input data done, but still have expected \
            path for case " << myAheadId; throw test::PathGeneratorError(s.str( )); }
      if (ticks > 0)
        std::cout << std::endl;
      pathGenerator( ).finishSession( );
      myStream.close( );
      return recorder( ).isOK( );
      }   // end TestManager::go
```

This code is used in section 7.5.1.

This runs only once, as we first read from the file of expected paths.

⟨ TestManager::doInitialProcessing 7.5.1.2 ⟩ ≡
```
void TestManager::doInitialProcessing( )
  { if (¬myStream.is_open( ))
      throw test::PathGeneratorError("Couldn't␣open␣file␣of␣exp\
          ected␣paths"); myStream >> myAheadId; if (myStream.fail( ))
      throw test::PathGeneratorError("Couldn't␣read␣first␣id␣in\
          ␣expected␣paths␣file"); myState = readid; }
```

This code is used in section 7.5.1.

It seems easiest to use a real *TimeStepGenerator* for the following, though it's possible there will be some rounding error problems. In this case the steps are at every integer to the end.

⟨ TestManager::generateTimeSteps 7.5.1.3 ⟩ ≡
```
void TestManager::generateTimeSteps( )
  { timeStepsGenerator( ).makeStepsFor(&(environment( ))); }
```

This code is used in section 7.5.1.

When this is called, we have already read *myAheadId* from the stream of expected paths. We must read to the end of the line, generate a path, and then attempt to read the next line. We should keep doing this until encountering end of file or a record that doesn't belong in this set (i.e., different **Id**, and in the future perhaps a different base **State**).

On exit, the **TestRecorder** should be loaded with the expected paths for this case only. *myAheadId* will have the next id, and *myState* will be either *readid* or *eof* as appropriate.

```
⟨ TestManager::generateExpectedPaths 7.5.1.4 ⟩ ≡
  void TestManager::generateExpectedPaths( )
    {
    int cumto[maxState( )];    // really Time, but it's integral
    bool fail[maxState( )]; unsigned int i; TimeSteps& ts =
        environment( ).timeSteps( ); recorder( ).clear( );    // preconditions to this call
        guarantee the next test
      // will be true at least the first time through.
    while (environment( ).id( ) ≡ myAheadId ∧ myState ≠ eof)
      {    // read rest of line
      for (i = 0; i < maxState( ); i++)
        { myStream >> fail[i]; myStream >> cumto[i]; if (¬fail[i])
          break; }
      ;
      if (myStream.fail( ))
        { std::stringstream s;
            s << "Unexpected␣failure␣in␣middle␣of␣line␣with␣expected\
            ␣path""␣for␣case␣" << myAheadId; throw test::PathGeneratorError(s.str( )); }
        // create path
        // cheat: since we have exactly one step per time unit,
        // ts[i] is exactly the timestep for time i.
      int lastTime = 0;
      Path *pp = new Path( );    // same head node for all
      pp→pathPush(StatePoint(0, lastTime), ts[lastTime]);
      for (i = 0; i < maxState( ); i++)
        {
        if (¬fail[i])
          { lastTime = cumto[i]; pp→pathPush(StatePoint(i + 1, lastTime), ts[lastTime]); break; }
        else
          {
          if (cumto[i] ≠ lastTime)
            { lastTime = cumto[i]; pp→pathPush(StatePoint(i, lastTime), ts[lastTime]); }
          }
        }
        // and push it on
      recorder( ).addExpectedPath(pp);
        // start on next line
      myStream >> myAheadId;
      if (myStream.eof( ))
        myState = eof;
      }
    }
```

This code is used in section 7.5.1.

7.5.2 TestRecorder

"mspath.C" 7.5.2 ≡
@f *pair int*
@f *iterator int*
@f *bool int*
@f *PathSet int*

⟨ TestRecorder.cc 7.5.2 ⟩ ≡
@o TestRecorder.cc
#include <utility>
#include <algorithm>
#include <vector>

#include "TestRecorder.h"

namespace mspath {⟨ PathSet Implementation 7.5.2.4 ⟩
⟨ TestRecorder Setup Implementation 7.5.2.5 ⟩
⟨ TestRecorder Most State Changes 7.5.2.1 ⟩
⟨ TestRecorder::goodPath 7.5.2.2 ⟩
⟨ TestRecorder::clear 7.5.2.3 ⟩
⟨ TestRecorder::dumpSets 7.5.2.9 ⟩
⟨ LegalMoves Implementation 7.5.2.6 ⟩
}

This code is used in section 7.2.

The following code implements most of the actions (aka state changes) of **TestRecorder**. See the next section for *goodPath*.

Most of these methods throw errors if trouble is detected. This is less complex than attempting to record all the specific error information, and it is more informative than simply recording that some error has occurred somewhere.

⟨ TestRecorder Most State Changes 7.5.2.1 ⟩ ≡

```
  // call at beginning of processing
void TestRecorder::startSession( )
  { myLegal.setState(sstartSession); Super::startSession( ); }

  // call at the start of processing a given case

void TestRecorder::startCase( )
  { myLegal.setState(sstartCase); Super::startCase( ); myId = environment( ).id( ); }

  // call when starting a tree of paths

void TestRecorder::startTree(double initialProbability)
  { myLegal.setState(sstartTree); Super::startTree(initialProbability); }

  // finished tree

void TestRecorder::finishTree( )
  {
  myLegal.setState(sfinishTree); Super::finishTree( );
  if (¬(unmatchedPaths( ).empty( ) ∧ mySurprisePaths.empty( )))
    { std::ostringstream s; s << "Not all expected paths found or some unexpected pa\
        ths found." << std::endl; dumpSets(s);
        // a long message
     test::PathGeneratorError(s.str( )); }
  }

  // finished processing case
void TestRecorder::finishCase( ) throw (DataModelInconsistency)
  { myLegal.setState(sfinishCase); Super::finishCase( ); }

  // completely done. Might clean up, output results, etc.

void TestRecorder::finishSession( )
  { myLegal.setState(sfinishSession); Super::finishSession( ); if (¬myLegal.isDone( ))
    throw test::PathGeneratorError("Got to end, but illegal state changes.");

  }
```

This code is used in section 7.5.2.

Remember that the argument to *goodPath* is not owned by me.

If the client fails to record all the good nodes in a path it is an error. Currently I only check that the last one is mentioned right before *goodPath* is called.

It is an error if a call to *goodNode* was not made immediately before calling *goodPath*. Likely the client thinks *goodPath* calls *goodNode* on the final node. It doesn't. It's also possible the client is making the call, but somewhat out of sequence. That might be OK.

⟨ TestRecorder::goodPath 7.5.2.2 ⟩ ≡

```
void TestRecorder::goodPath(Path& goodPath)
  {
  myLegal.setState(sgoodPath);
  if (&(goodPath.back( )) ≠ mypLastNode)
    throw test::PathGeneratorError("You␣did␣not␣say␣the␣last␣\
        node␣in␣the␣path␣was␣""a␣good␣node␣just␣before␣goodPath.␣␣Probable␣error.");
  PathSet::iterator i = expectedPaths( ).find(goodPath);
  if (i ≠ expectedPaths( ).end( ))
    { myMatchedPaths.transferUnique(i, expectedPaths( )); }
  else
    {   // unique is perhaps unnecessary here
      // we could just throw an exception in all cases
    mySurprisePaths.insertUnique(goodPath.clone( ));
    }
  Super::goodPath(goodPath);
  }
```

This code is used in section 7.5.2.

Clear things for another run.

⟨ TestRecorder::clear 7.5.2.3 ⟩ ≡

```
void TestRecorder::clear( )
  { expectedPaths( ).clear( ); myMatchedPaths.clear( ); mySurprisePaths.clear( );
  }
```

This code is used in section 7.5.2.

Destroy contained objects.

⟨ PathSet Implementation 7.5.2.4 ⟩ ≡

```
    // insert, throwing an error if path already present
  TestRecorder::PathSet::iterator TestRecorder::PathSet::insertUnique(Path *const p) throw
      (test::DuplicatePath)
    { typedef std::pair < PathSet::iterator, bool > TReturn; TReturn r = insert(p); if (¬r.second)
      throw test::DuplicatePath(p); return r.first; }

    // transfer, throwing an error if path already present
    // semantics of transfer in face of duplicates are ambiguous
  void TestRecorder::PathSet::transferUnique ( TestRecorder::PathSet::
      iterator i, TestRecorder::PathSet&p ) throw (test::DuplicatePath)
    { iterator n = find(*i); if (n ≠ end( ))
      throw test::DuplicatePath(&*i); transfer(i, p); }

  std::ostream & operator <<(std::ostream & s, const TestRecorder::PathSet & p)
    {
    TestRecorder::PathSet::const_iterator i,
        e(p.end( )); s << std::endl << "----------------␣PathSet␣\
        with␣" << p.size( ) << "␣entries␣------------------" << std::endl;
    for (i = p.begin( ); i ≠ e; ++i)
      { s << *i << std::endl; }
    s << "^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^" << std::endl;
    return s;
    }
```

This code is used in section 7.5.2.

Note the following counts on the fact that these containers use pointers in place without cloning or copying.

⟨ TestRecorder Setup Implementation 7.5.2.5 ⟩ ≡

```
    // takes ownership of the pointer argument
  void TestRecorder::addExpectedPath(Path *const pp) throw (test::DuplicatePath,
      test::PathGeneratorError)
    { expectedPaths( ).insertUnique(pp); }
```

This code is used in section 7.5.2.

⟨ LegalMoves Implementation 7.5.2.6 ⟩ ≡
⟨ LegalMoves constructor 7.5.2.7 ⟩
⟨ LegalMoves::setState 7.5.2.8 ⟩

This code is used in section 7.5.2.

This initializes the table of legal state changes.

⟨ LegalMoves constructor 7.5.2.7 ⟩ ≡

```
#define makeOK(kname) TAllowed(kname, kname+\
  sizeof(kname) / sizeof(State))

TestRecorder::LegalMoves::LegalMoves( ) : myCurrentState(sinitial)
  { TStateMap::iterator i; myGoodMoves[sinitial] = TAllowed(1, sstartSession);

    GeneratorState kstartSession[ ] = {sstartCase,
        sfinishSession}; myGoodMoves[sstartSession] = makeOK(kstartSession);

    GeneratorState kstartCase[ ] = {sstartTree,
        sfinishCase}; myGoodMoves[sstartCase] = makeOK(kstartCase);

    GeneratorState kstartTree[ ] = {sfinishTree, sgoodNode, simpermissible,
        sgoodPath}; myGoodMoves[sstartTree] = makeOK(kstartTree);

    // goodNode, impermissible, and goodPath do not cause state changes

    GeneratorState kfinishTree[ ] = {sstartTree,
        sfinishCase}; myGoodMoves[sfinishTree] = makeOK(kfinishTree);

    GeneratorState kfinishCase[ ] = {sstartCase,
        sfinishSession}; myGoodMoves[sfinishCase] = makeOK(kfinishCase);

    // final state is absorbing
  myGoodMoves[sfinishSession];    // creates an empty entry
  }
```

This code is used in section 7.5.2.6.

Change state and check for legal moves.

⟨ LegalMoves::setState 7.5.2.8 ⟩ ≡

```
void TestRecorder::LegalMoves::setState(GeneratorState s) throw (test::IllegalStateChange,
      test::TestInternalError)
  { TStateMap::iterator imap,
      emap = myGoodMoves.end( ); imap = myGoodMoves.find(myCurrentState); if (imap ≡ emap)
    //this can only happen if this class has a bug
    throw test::TestInternalError("Invalid state stored in L\
      egalMoves. ""Internal Error."); const TAllowed& legal =
      (*imap).second; TAllowed::const_iterator i, e = legal.end( ); i = std::find(legal.begin( ), e,
      s); if (i ≡ e)    // this is the client's mistake
    throw test::IllegalStateChange(myCurrentState,
      s); if (s ≠ sgoodNode ∧ s ≠ simpermissible ∧ s ≠ sgoodPath)
    myCurrentState = s; }
```

This code is used in section 7.5.2.6.

Quickie debugging aid. Should only be run at end of a run.

⟨ TestRecorder::dumpSets 7.5.2.9 ⟩ ≡
```
std::ostream & TestRecorder::dumpSets(std::ostream & s) const
  { s << "The␣following␣paths␣were␣never␣found:"; s << unmatchedPaths( ); s <<
      "These␣paths␣were␣found␣but␣not␣expected:"; s << unanticipatedPaths( ); s <<
      "These␣paths␣were␣expected␣and␣found:"; s << matchedPaths( ); return s; }
```

This code is used in section 7.5.2.

7.5.3 NodeFactoryTester

The code makes many references to instance variables that are not obviously instance variables, because their names don't start with *my*.

⟨ NodeFactoryTester.cc 7.5.3 ⟩ ≡
```
  @o NodeFactoryTester.cc
#include "NodeFactoryTester.h"
#include <algorithm>    // for max
#include <cassert>
#include <boost/test/test_tools.hpp>

  namespace mspath { ⟨ NodeFactoryTester Constructor Implementation 7.5.3.5 ⟩
  ⟨ NodeFactoryTester Destructor Implementation 7.5.3.6 ⟩
  ⟨ NodeFactoryTester Validate 7.5.3.4 ⟩
  ⟨ NodeFactoryTester Testing 7.5.3.1 ⟩

  ⟨ NodeFactoryTester Support Implementation 7.5.3.7 ⟩}
```

This code is used in section 7.2.

The following code implements the main action of this class.

⟨ NodeFactoryTester Testing 7.5.3.1 ⟩ ≡
```
void NodeFactoryTester::test(TStep up, TStep down)
  {
  assert(up > down);
  assert(PASS_FACTOR ≥ up);
  assert(up ≥ step);
  ⟨ NodeFactoryTester Generate Up 7.5.3.2 ⟩
  validate( );
  ⟨ NodeFactoryTester Destroy Down 7.5.3.3 ⟩
  validate( );
  pass++;
  }
```

This code is used in section 7.5.3.

This code generates **Node**'s as if going out on a branch of the tree. This operation may start in the middle, i.e., not at step 0. This would be the typical operation when we evaluate one branch of the tree and then start work on another one.

⟨ NodeFactoryTester Generate Up 7.5.3.2 ⟩ ≡

```
for (step = step; step < up; step++)
  {
  if (step < highWater)
    {
    delete vSP[step];
    SPnFree++;
    }
  else
    {
    vTP[step] = new TestTimePoint (expectedTPTime(step), expectedMatchesObservation(step), 0);
    TPnAlloc++;
    }
  vSP[step] = new TestStatePoint (expectedState(pass, step), expectedSPTime(pass, step));
  SPnAlloc++;

  vNode[step] = nodeFactory( ).createNode(*(vSP[step]), *(vTP[step]));

    // I must be friend of Node for next section to work
  if (step > 0)
    vNode[step]→setPrevious(vNode[step − 1]);
  else
    vNode[step]→setNoPrevious( );
  if (expectedCountedAsGood(pass, step))
    vNode[step]→countAsGood( );

  vNode[step]→evaluationData( ) = expectedEvaluationData(pass, step);
  vNode[step]→modelData( ) = expectedModelData(pass, step);
  vPass[step] = pass;
  }
highWater = std::max(highWater, up);
NDnAlloc = highWater + NDnAllocBase;
```

This code is used in section 7.5.3.1.

If the factory is working right, this will not destroy the **Node** objects.

```
⟨ NodeFactoryTester Destroy Down 7.5.3.3 ⟩ ≡
    if (down > 0)
      {
      for (step = step − 1U; step ≥ down; step−−)
        {
        nodeFactory( ).destroyNode(vNode[step]);
        }
      step++;
      }
    else
      {
      nodeFactory( ).reset( );    // must be friend
      step = 0U;
      }
```

This code is used in section 7.5.3.1.

Check that every **Node** is in its expected state, and that the number of object creations and destructions match expectations.

```
⟨ NodeFactoryTester Validate 7.5.3.4 ⟩ ≡
    void NodeFactoryTester::validate( )
      {
      checkCounts( );
      for (size_t i = 0U; i < step; i++)
        {
        size_t p = vPass[i];
        BOOST_CHECK_EQUAL(vNode[i]→state( ), expectedState(p, i));
        BOOST_CHECK_EQUAL(vNode[i]→time( ), expectedTPTime(i));
        BOOST_CHECK_EQUAL(vNode[i]→statePoint( ).time( ), expectedSPTime(p, i));
        if (i > 0)
          BOOST_CHECK_EQUAL(vNode[i]→previous( ), vNode[i − 1]);
        else
          BOOST_CHECK(vNode[i]→isRoot( ));
        BOOST_CHECK_EQUAL(vNode[i]→alreadyCountedAsGood( ), expectedCountedAsGood(p, i));
        BOOST_CHECK_EQUAL(vNode[i]→timePoint( ).matchesObservation( ),
            expectedMatchesObservation(i));
        BOOST_CHECK_EQUAL(vNode[i]→evaluationData( ), expectedEvaluationData(p, i));
        for (size_t h = 0; h < nPath( ); h++)
          {
          BOOST_CHECK_EQUAL(vNode[i]→modelData( )[h], expectedModelData(p, i)[h]);
          }
        BOOST_CHECK_EQUAL(vNode[i]→allocID( ), i + NDnAllocBase);
        }
      }
```

This code is used in section 7.5.3.

I'm not sure if the **NodeFactoryTester** :: qualifier is necessary, but it certainly can't hurt.

⟨ NodeFactoryTester Constructor Implementation 7.5.3.5 ⟩ ≡

```
NodeFactoryTester::NodeFactoryTester(NodeFactoryTester::TestNodeFactory * pNF,
    NodeFactoryTester::TPass biggestStep, NodeFactoryTester::TPass initialPass) :
    PASS_FACTOR(biggestStep), myHistoryOffsets(pNF→myNPath), mypNodeFactory(pNF),
    pass(initialPass), step(0), highWater(0), vPass(biggestStep), vNode(biggestStep),
    vSP(biggestStep), vTP(biggestStep)
{
  // Object creation/destruction counts
snapshotCounts( );
  // Setup unique values for history
const ModelData::value_type offset = 0.25;
for (size_t i = 0; i < myHistoryOffsets.size( ); ++i)
  {
  myHistoryOffsets[i] = (i + 1) * offset;
  }
}
```

This code is used in section 7.5.3.

The objects we point to have varying statuses. *vSP* points to **StatePoint**'s that should be irrelevant as soon as they are used. We only kept them to double-check that creating or destroying the **Node** does not touch the original **StatePoint**.

The **TimePoint**'s in *vTP*, on the other hand, are referenced by the **Node**'s that are created. We need to destroy the **TimePoint**'s, but that will mean any remaining **Node**'s are invalid, since they include a pointer to a **TimePoint**.

We can not destroy the **Node**'s through *vNode*, since the **NodeFactory** is the owner. And we can not destroy the latter, since it is the responsibility of the client. What we can and must do is indicate that none of the **Node**'s are in use any longer.

Finally, note that deleting this **NodeFactoryTester** does not zero out the static class allocation counters.

⟨ NodeFactoryTester Destructor Implementation 7.5.3.6 ⟩ ≡

```
NodeFactoryTester::~NodeFactoryTester ( )
  {
  for (size_t i = 0; i < highWater; i++)
    {
    delete vSP[i];
    delete vTP[i];
    }    // Tell NodeFactory that none of the Node's it produced are
    // in use. Need to be friend to reset( ).
  nodeFactory( ).reset( );
  }
```

This code is used in section 7.5.3.

Some utilities about the relation between the internal (expected) object counts kept in this class and those in the various **AllocCounter** classes:

⟨ NodeFactoryTester Support Implementation 7.5.3.7 ⟩ ≡

```
void NodeFactoryTester::snapshotCounts( )
  {
  SPnAlloc = TestStatePoint::nAlloc( );
  SPnFree = TestStatePoint::nFree( );
  TPnAlloc = TestTimePoint::nAlloc( );
  TPnFree = TestTimePoint::nFree( );
  NDnAlloc = TestNode::nAlloc( );
  NDnAllocBase = NDnAlloc;
  NDnFree = TestNode::nFree( );
  }

void NodeFactoryTester::checkCounts( ) const
  {
  BOOST_CHECK_EQUAL(SPnAlloc, TestStatePoint::nAlloc( ));
  BOOST_CHECK_EQUAL(SPnFree, TestStatePoint::nFree( ));
  BOOST_CHECK_EQUAL(TPnAlloc, TestTimePoint::nAlloc( ));
  BOOST_CHECK_EQUAL(TPnFree, TestTimePoint::nFree( ));
  BOOST_CHECK_EQUAL(NDnAlloc, TestNode::nAlloc( ));
  BOOST_CHECK_EQUAL(NDnFree, TestNode::nFree( ));
  }
```

This code is used in section 7.5.3.

8 Benchmarks

8.1 Single CPU Benchmarks

Here are results of running `make -f Makefile.full time`. That executes `src/test/profile.cc`, which makes 3 evaluations of the canonical dataset. For each configuration I made several (N) runs; the table gives the mean and range of those runs. This time excludes time reading the data in and covers 3 evaluations of the likelihood. The other columns give the environment in which the test was run, including the date the test was run and the name of the system it was run on. Above each block of tests is a note giving the subversion revision of the code and other salient features.

Yes, `-O2` is faster than `-O3` on OS-X Power PC with gcc 3.3.

Mean	Range	N	Chip	compiler	flags	OS	date	name
baseline, rev 816								
4.88	4.83–4.94	6	PPC	gcc 3.3	O3	OS-X 10.3	2007-04-07	statcluster
4.64	4.62–4.66	5	PPC	gcc 3.3	O2	OS-X 10.3	2007-04-07	statcluster
3.20	3.19–3.20	5	P4 3G	gcc 4.1.2	O3	Linux 2.6.18	2007-04-07	corn
AbstractPathGenerator, rev 817								
5.17	5.12–5.22	6	PPC	gcc 3.3	O3	OS-X 10.3	2007-04-07	statcluster
4.89	4.83–4.93	5	PPC	gcc 3.3	O2	OS-X 10.3	2007-04-07	statcluster
3.89	3.85–3.91	5	P4 3G	gcc 4.1.2	O2	Linux 2.6.18	2007-04-07	corn
3.40	3.36–3.44	5	P4 3G	gcc 4.1.2	O3	Linux 2.6.18	2007-04-07	corn
corn was under load for 2 above								
3.26	3.21–3.28	5	P4 3G	gcc 4.1.2	O3	Linux 2.6.18	2007-04-07	corn
remove virtual from nextTimePoint, v 818								
3.21	3.19–3.23	5	P4 3G	gcc 4.1.2	O3	Linux 2.6.18	2007-04-07	corn
inline nextTimePoint, v 819								
3.17	3.16–3.18	5	P4 3G	gcc 4.1.2	O3	Linux 2.6.18	2007-04-07	corn

9 Overall System

"mspath.C" 9 ≡
⟨ basic types 3 ⟩
⟨ TimePoint.h 4.17 ⟩
⟨ TimePoint.cc 5.16 ⟩⟨ TimeSteps.h 4.16 ⟩
⟨ Node.h 4.13 ⟩
⟨ NodeFactory.h 4.12 ⟩
⟨ Path.h 4.11 ⟩
⟨ Path.cc 5.9 ⟩
⟨ Environment.h 4.18 ⟩
⟨ Environment.cc 5.17 ⟩
⟨ ScratchPad.h 4.19.1 ⟩
⟨ ScratchData.h 4.19.2 ⟩
⟨ ScratchDataProducer.h 4.19.3 ⟩
⟨ Evaluator.h 4.21 ⟩
⟨ Evaluator.cc 5.13 ⟩
⟨ Recorder.h 4.20 ⟩
⟨ SimpleRecorder.h 4.20.1 ⟩
⟨ SimpleRecorder.cc 5.12 ⟩
⟨ EvaluatorRecorder.h 4.20.2 ⟩
⟨ Model.h 4.5 ⟩
⟨ Model.cc 5.5 ⟩
⟨ Data.h 4.22 ⟩
⟨ Covariates.h 4.10 ⟩
⟨ Covariates.cc 5.15 ⟩
⟨ Data.cc 5.14 ⟩
⟨ mspath.h 6.4.1 ⟩
⟨ mspath.cc 6.4.8 ⟩
⟨ mspathR.h 6.2.1 ⟩
⟨ mspathR.cc 6.2.4 ⟩
⟨ LinearProduct.h 4.8 ⟩
⟨ LinearProduct.cc 5.7 ⟩
⟨ Coefficients.h 4.9 ⟩
⟨ Coefficients.cc 5.8 ⟩⟨ Specification.h 4.7 ⟩
⟨ Specification.cc 5.6 ⟩
⟨ AbstractTimeStepsGenerator.h 4.15 ⟩
⟨ AbstractTimeStepsGenerator.cc 5.10.1 ⟩
⟨ FixedTimeStepsGenerator.h 4.15.1 ⟩
⟨ FixedTimeStepsGenerator.cc 5.10.2.1 ⟩
⟨ CompressedTimeStepsGenerator.h 4.15.2 ⟩
⟨ CompressedTimeStepsGenerator.cc 5.10.3 ⟩
⟨ TimeStepsGenerator.h 4.15.3 ⟩
⟨ TimeStepsGenerator.cc 5.10.4 ⟩
⟨ StateTimeClassifier.h 4.4 ⟩
⟨ StateTimeClassifier.cc 5.4 ⟩
⟨ SuccessorGenerator.h 4.3 ⟩
⟨ SuccessorGenerator.cc 5.3 ⟩⟨ PathGenerator.h 4.2 ⟩
⟨ PathGenerator.cc 5.2 ⟩
⟨ Manager.h 4.1 ⟩
⟨ Manager.cc 5.1 ⟩

⟨ MSPathError.h 4.23 ⟩
⟨ HistoryComputer.h 4.14 ⟩
⟨ PrimitiveHistoryComputer.h 4.14.1 ⟩
⟨ PrimitiveHistoryComputer.cc 5.11 ⟩
⟨ CompositeHistoryComputer.h 4.14.2 ⟩
⟨ CompositeHistoryComputer.cc 5.11.0.1 ⟩
⟨ ModelBuilder.h 4.24.1 ⟩
⟨ ModelBuilder.cc 6.5 ⟩
⟨ Testing Files 7.2 ⟩

10 INDEX

⟨ 2D Subset of 1D 3.2.1 ⟩ Used in section 3.2.
⟨ AbstractCovariates Interface 4.10.1 ⟩ Used in section 4.10.
⟨ AbstractDataIterator 4.22.2.1 ⟩ Used in section 4.22.2.
⟨ AbstractPathGenerator interface 4.2.1 ⟩ Used in section 4.2.
⟨ AbstractSpecification basic behaviors 4.7.1.2 ⟩ Used in section 4.7.1.
⟨ AbstractSpecification c'tor 4.7.1.1 ⟩ Used in section 4.7.1.
⟨ AbstractSpecification data 4.7.1.5 ⟩ Used in section 4.7.1.
⟨ AbstractSpecification interface 4.7.1 ⟩ Used in section 4.7.
⟨ AbstractSpecification protected accessors 4.7.1.4 ⟩ Used in section 4.7.1.
⟨ AbstractSpecification queries 4.7.1.3 ⟩ Used in section 4.7.1.
⟨ AbstractSpecification::operator¡¡ 5.6.0.5 ⟩ Used in section 5.6.0.4.
⟨ AbstractTimeStepsGenerator.cc 5.10.1 ⟩ Used in section 9.
⟨ AbstractTimeStepsGenerator.h 4.15 ⟩ Used in section 9.
⟨ AllocCounter Actions 7.3.1.3 ⟩ Used in section 7.3.1.
⟨ AllocCounter Constructors 7.3.1.1 ⟩ Used in section 7.3.1.
⟨ AllocCounter Count Accessors 7.3.1.2 ⟩ Used in section 7.3.1.
⟨ AllocCounter Data 7.3.1.4 ⟩ Used in section 7.3.1.
⟨ AllocCounter Instance Operations 7.3.1.5 ⟩ Used in section 7.3.1.
⟨ AllocCounter Printing 7.3.1.6 ⟩ Used in section 7.3.1.
⟨ AllocCounter Static Initialization 7.3.1.7 ⟩ Used in section 7.3.1.
⟨ AllocCounter.h 7.3.1 ⟩ Used in section 7.2.
⟨ BadInitialProbs 4.23.0.4 ⟩ Used in section 4.23.
⟨ Coefficients.cc 5.8 ⟩ Used in section 9.
⟨ Coefficients.h 4.9 ⟩ Used in section 9.
⟨ CompositeHistoryComputer Interface 4.14.2.1 ⟩ Used in section 4.14.2.
⟨ CompositeHistoryComputer.cc 5.11.0.1 ⟩ Used in section 9.
⟨ CompositeHistoryComputer.h 4.14.2 ⟩ Used in section 9.
⟨ CompressedTimeStepsGenerator helpers 4.15.2.1 ⟩ Used in section 4.15.2.
⟨ CompressedTimeStepsGenerator.cc 5.10.3 ⟩ Used in section 9.
⟨ CompressedTimeStepsGenerator.h 4.15.2 ⟩ Used in section 9.
⟨ CompressedTimeStepsGenerator::lastObs 5.10.3.1 ⟩ Used in section 5.10.3.
⟨ ConstantLinearProduct Implementation 5.7.1 ⟩ Used in section 5.7.
⟨ ConstantLinearProduct definition 4.8.1 ⟩ Used in section 4.8.
⟨ Covariates.cc 5.15 ⟩ Used in section 9.
⟨ Covariates.h 4.10 ⟩ Used in section 9.
⟨ Data Accessors 4.22.1.1 ⟩ Used in section 4.22.
⟨ Data Constructors 4.22.1 ⟩ Used in section 4.22.
⟨ Data Data 4.22.1.2 ⟩ Used in section 4.22.
⟨ Data Iteration 4.22.2 ⟩ Used in section 4.22.
⟨ Data Iterator Code 5.14.0.1 ⟩ Used in section 5.14.
⟨ Data.cc 5.14 ⟩ Used in section 9.
⟨ Data.h 4.22 ⟩ Used in section 9.
⟨ DataIteratorErrors 4.23.0.8 ⟩ Used in section 4.23.
⟨ DataIterators 4.22.2.2 ⟩ Used in section 4.22.2.
⟨ DataLinearProduct Implementation 5.7.2 ⟩ Used in section 5.7.
⟨ DataLinearProduct definition 4.8.2 ⟩ Used in section 4.8.
⟨ DataModelInconsistency 4.23.0.1 ⟩ Used in section 4.23.
⟨ Debug Tools 3.3 ⟩ Used in section 3.
⟨ DuplicatePath Error 7.4.5.2 ⟩ Used in section 7.4.5.
⟨ DuplicateTerm 4.23.0.5 ⟩ Used in section 4.23.
⟨ Environment Accessors 4.18.0.3 ⟩ Used in section 4.18.
⟨ Environment Actions 4.18.0.8 ⟩ Used in section 4.18.

⟨ Environment Constructors 4.18.0.2 ⟩ Used in section 4.18.
⟨ Environment Data 4.18.0.9 ⟩ Used in section 4.18.
⟨ Environment Iterator Accessors 4.18.0.6 ⟩ Used in section 4.18.0.3.
⟨ Environment Node-Specific Accessors (explicit) 4.18.0.5 ⟩ Used in section 4.18.0.3.
⟨ Environment Node-Specific Accessors (implicit) 4.18.0.4 ⟩ Used in section 4.18.0.3.
⟨ Environment Randomness 4.18.0.10 ⟩ Used in section 4.18.
⟨ Environment Setters 4.18.0.7 ⟩ Used in section 4.18.
⟨ Environment Typedefs 4.18.0.1 ⟩ Used in section 4.18.
⟨ Environment.cc 5.17 ⟩ Used in section 9.
⟨ Environment.h 4.18 ⟩ Used in section 9.
⟨ Evaluator Accessors 4.20.2.1 ⟩ Used in section 4.20.2.
⟨ Evaluator Actions 4.20.2.2 ⟩ Used in section 4.20.2.
⟨ Evaluator Data 4.20.2.3 ⟩ Used in section 4.20.2.
⟨ Evaluator Protected Accessors 4.20.2.4 ⟩ Used in section 4.20.2.
⟨ Evaluator.cc 5.13 ⟩ Used in section 9.
⟨ Evaluator.h 4.21 ⟩ Used in section 9.
⟨ EvaluatorRecorder.h 4.20.2 ⟩ Used in section 9.
⟨ FixedTimeStepsGenerator time manipulation 5.10.2 ⟩ Used in section 5.10.2.1.
⟨ FixedTimeStepsGenerator.cc 5.10.2.1 ⟩ Used in section 9.
⟨ FixedTimeStepsGenerator.h 4.15.1 ⟩ Used in section 9.
⟨ Fundamental Types 3.1 ⟩ Used in section 3.
⟨ HistoryComputer Data 4.14.0.4 ⟩ Used in section 4.14.
⟨ HistoryComputer Parsing 4.14.0.2 ⟩ Used in section 4.14.
⟨ HistoryComputer Setup Post-Parsing 4.14.0.3 ⟩ Used in section 4.14.
⟨ HistoryComputer c'tors 4.14.0.1 ⟩ Used in section 4.14.
⟨ HistoryComputer printing 4.14.0.5 ⟩ Used in section 4.14.
⟨ HistoryComputer.h 4.14 ⟩ Used in section 9.
⟨ Illegal State Change Error 7.4.5.3 ⟩ Used in section 7.4.5.
⟨ InconsistentModel 4.23.0.2 ⟩ Used in section 4.23.
⟨ InterceptCoefficients Definition 4.9.1 ⟩ Used in section 4.9.
⟨ InterceptCoefficients output 5.8.0.2 ⟩ Used in section 5.8.
⟨ LegalMoves Implementation 7.5.2.6 ⟩ Used in section 7.5.2.
⟨ LegalMoves constructor 7.5.2.7 ⟩ Used in section 7.5.2.6.
⟨ LegalMoves::setState 7.5.2.8 ⟩ Used in section 7.5.2.6.
⟨ LinearProduct.cc 5.7 ⟩ Used in section 9.
⟨ LinearProduct.h 4.8 ⟩ Used in section 9.
⟨ LnHistoryComputer Interface 4.14.2.2 ⟩ Used in section 4.14.2.
⟨ MSPathError.h 4.23 ⟩ Used in section 9.
⟨ Manager Actions 4.1.0.3 ⟩ Used in section 4.1.
⟨ Manager Constructors 4.1.0.2 ⟩ Used in section 4.1.
⟨ Manager Data 4.1.0.4 ⟩ Used in section 4.1.
⟨ Manager Helpers 4.1.0.6 ⟩ Used in section 4.1.
⟨ Manager Protected Accessors 4.1.0.5 ⟩ Used in section 4.1.
⟨ Manager Simulation 4.1.0.7 ⟩ Used in section 4.1.
⟨ Manager Typedefs 4.1.0.1 ⟩ Used in section 4.1.
⟨ Manager.cc 5.1 ⟩ Used in section 9.
⟨ Manager.h 4.1 ⟩ Used in section 9.
⟨ Manager::getResults 5.1.0.5 ⟩ Used in section 5.1.
⟨ Manager::go 5.1.0.1 ⟩ Used in section 5.1.
⟨ Manager::mainOperation 5.1.0.4 ⟩ Used in section 5.1.
⟨ Manager::setupCount 5.1.0.2 ⟩ Used in section 5.1.
⟨ Manager::setupLikelihood 5.1.0.3 ⟩ Used in section 5.1.

⟨ Manager::simulate 5.1.0.6 ⟩ Used in section 5.1.
⟨ Matrix Output 5.18.0.2 ⟩ Used in section 5.18.
⟨ MatrixCovariates Implementation 5.15.1 ⟩ Used in section 5.15.
⟨ MatrixCovariates Interface 4.10.2 ⟩ Used in section 4.10.
⟨ MatrixCovariates::Memento Implementation 5.15.1.1 ⟩ Used in section 5.15.1.
⟨ Model Accessors 4.5.0.4 ⟩ Used in section 4.5.
⟨ Model Constructors 4.5.0.2 ⟩ Used in section 4.5.
⟨ Model Data 4.5.0.3 ⟩ Used in section 4.5.
⟨ Model Operation 4.5.0.5 ⟩ Used in section 4.5.
⟨ Model Printing 4.5.0.8 ⟩ Used in section 4.5.
⟨ Model Queries 4.5.0.6 ⟩ Used in section 4.5.
⟨ Model Simulation 4.5.0.7 ⟩ Used in section 4.5.
⟨ Model Types 4.5.0.1 ⟩ Used in section 4.5.
⟨ Model.cc 5.5 ⟩ Used in section 9.
⟨ Model.h 4.5 ⟩ Used in section 9.
⟨ Model::evaluate 5.5.0.2 ⟩ Used in section 5.5.
⟨ Model::fillModelData 5.5.0.1 ⟩ Used in section 5.5.
⟨ Model::operator¡¡ 5.5.0.4 ⟩ Used in section 5.5.
⟨ Model::simulateObservation 5.5.0.6 ⟩ Used in section 5.5.
⟨ Model::simulatePath 5.5.0.5 ⟩ Used in section 5.5.
⟨ Model::validate 5.5.0.3 ⟩ Used in section 5.5.
⟨ ModelBuilder Data 4.24.1.6 ⟩ Used in section 4.24.1.
⟨ ModelBuilder History Implementation 6.5.7 ⟩ Used in section 6.5.
⟨ ModelBuilder c'tors 4.24.1.7 ⟩ Used in section 4.24.1.
⟨ ModelBuilder stage2 main loop 6.5.12 ⟩ Used in section 6.5.11.
⟨ ModelBuilder.cc 6.5 ⟩ Used in section 9.
⟨ ModelBuilder.h 4.24.1 ⟩ Used in section 9.
⟨ ModelBuilder::allComputers implementation 6.5.8 ⟩ Used in section 6.5.7.
⟨ ModelBuilder::fillparvec implementation 6.5.6 ⟩ Used in section 6.5.
⟨ ModelBuilder::fillparvec interface 4.24.1.5 ⟩ Used in section 4.24.1.
⟨ ModelBuilder::makeHistory interfaces 4.24.1.8 ⟩ Used in section 4.24.1.
⟨ ModelBuilder::makeHistoryIndirection implementation 6.5.13 ⟩ Used in section 6.5.
⟨ ModelBuilder::makeHistoryIndirection interface 4.24.1.10 ⟩ Used in section 4.24.1.
⟨ ModelBuilder::makeHistoryStage1 implementation 6.5.10 ⟩ Used in section 6.5.7.
⟨ ModelBuilder::makeHistoryStage2 implementation 6.5.11 ⟩ Used in section 6.5.7.
⟨ ModelBuilder::makeInitial implementation 6.5.2 ⟩ Used in section 6.5.
⟨ ModelBuilder::makeInitial interface 4.24.1.11 ⟩ Used in section 4.24.1.
⟨ ModelBuilder::makeModel arguments 4.24.1.2 ⟩ Used in sections 4.24.1.1 and 6.5.1.
⟨ ModelBuilder::makeModel implementation 6.5.1 ⟩ Used in section 6.5.
⟨ ModelBuilder::makeModel interface 4.24.1.1 ⟩ Used in section 4.24.1.
⟨ ModelBuilder::makeRequests implementation 6.5.9 ⟩ Used in section 6.5.7.
⟨ ModelBuilder::makeSimpleSpecification implementation 6.5.5 ⟩ Used in section 6.5.
⟨ ModelBuilder::makeSimpleSpecification interface 4.24.1.4 ⟩ Used in section 4.24.1.
⟨ ModelBuilder::makeSlope implementation 6.5.4 ⟩ Used in section 6.5.3.
⟨ ModelBuilder::makeSpecification implementation 6.5.3 ⟩ Used in section 6.5.
⟨ ModelBuilder::makeSpecification interface 4.24.1.3 ⟩ Used in section 4.24.1.
⟨ Node Accessors 4.13.0.3 ⟩ Used in section 4.13.
⟨ Node Actions 4.13.0.6 ⟩ Used in section 4.13.
⟨ Node Constructors 4.13.0.2 ⟩ Used in section 4.13.
⟨ Node Data 4.13.0.5 ⟩ Used in section 4.13.
⟨ Node Friend Accessors 4.13.0.8 ⟩ Used in section 4.13.
⟨ Node Operators 4.13.0.7 ⟩ Used in section 4.13.

⟨ Node Tests 4.13.0.4 ⟩ Used in section 4.13.
⟨ Node Typedefs 4.13.0.1 ⟩ Used in section 4.13.
⟨ Node.h 4.13 ⟩ Used in section 9.
⟨ NodeFactory Constructors 4.12.0.1 ⟩ Used in section 4.12.
⟨ NodeFactory Data 4.12.0.5 ⟩ Used in section 4.12.
⟨ NodeFactory Node Creation 4.12.0.2 ⟩ Used in section 4.12.
⟨ NodeFactory Node Destruction 4.12.0.3 ⟩ Used in section 4.12.
⟨ NodeFactory friends 4.12.0.4 ⟩ Used in section 4.12.
⟨ NodeFactory.h 4.12 ⟩ Used in section 9.
⟨ NodeFactoryTester Actions 7.4.4.3 ⟩ Used in section 7.4.4.
⟨ NodeFactoryTester Constructor Implementation 7.5.3.5 ⟩ Used in section 7.5.3.
⟨ NodeFactoryTester Constructors 7.4.4.2 ⟩ Used in section 7.4.4.
⟨ NodeFactoryTester Data 7.4.4.7 ⟩ Used in section 7.4.4.
⟨ NodeFactoryTester Destroy Down 7.5.3.3 ⟩ Used in section 7.5.3.1.
⟨ NodeFactoryTester Destructor Implementation 7.5.3.6 ⟩ Used in section 7.5.3.
⟨ NodeFactoryTester Expected Values 7.4.4.5 ⟩ Used in section 7.4.4.
⟨ NodeFactoryTester Generate Up 7.5.3.2 ⟩ Used in section 7.5.3.1.
⟨ NodeFactoryTester Internal Accessors 7.4.4.6 ⟩ Used in section 7.4.4.
⟨ NodeFactoryTester Internal Types 7.4.4.4 ⟩ Used in section 7.4.4.
⟨ NodeFactoryTester Public Types 7.4.4.1 ⟩ Used in section 7.4.4.
⟨ NodeFactoryTester Support Implementation 7.5.3.7 ⟩ Used in section 7.5.3.
⟨ NodeFactoryTester Support Interface 7.4.4.8 ⟩ Used in section 7.4.4.
⟨ NodeFactoryTester Testing 7.5.3.1 ⟩ Used in section 7.5.3.
⟨ NodeFactoryTester Validate 7.5.3.4 ⟩ Used in section 7.5.3.
⟨ NodeFactoryTester.cc 7.5.3 ⟩ Used in section 7.2.
⟨ NodeFactoryTester.h 7.4.4 ⟩ Used in section 7.2.
⟨ Old C I don't think I'll need 6.4.6 ⟩ Used in section 6.4.1.
⟨ OneInitialState 4.23.0.3 ⟩ Used in section 4.23.
⟨ Path Generator Inner Loop Body 5.2.0.3 ⟩ Used in section 5.2.0.2.
⟨ Path.cc 5.9 ⟩ Used in section 9.
⟨ Path.h 4.11 ⟩ Used in section 9.
⟨ PathCovariates Implementation 5.15.2 ⟩ Used in section 5.15.
⟨ PathCovariates Interface 4.10.3 ⟩ Used in section 4.10.
⟨ PathDependentLinearProduct Implementation 5.7.3 ⟩ Used in section 5.7.
⟨ PathDependentLinearProduct definition 4.8.3 ⟩ Used in section 4.8.
⟨ PathGenerator interface 4.2.2 ⟩ Used in section 4.2.
⟨ PathGenerator state changes 5.2.0.4 ⟩ Used in section 5.2.
⟨ PathGenerator.cc 5.2 ⟩ Used in section 9.
⟨ PathGenerator.h 4.2 ⟩ Used in section 9.
⟨ PathGenerator::nextBranch 5.2.0.2 ⟩ Used in section 5.2.
⟨ PathGenerator::startTree 5.2.0.1 ⟩ Used in section 5.2.
⟨ PathSet Declaration 7.4.2.7 ⟩ Used in section 7.4.2.
⟨ PathSet Implementation 7.5.2.4 ⟩ Used in section 7.5.2.
⟨ PathSet free functions 7.4.2.8 ⟩ Used in section 7.4.2.
⟨ PickyStateTimeClassifier Interface 4.4.2 ⟩ Used in section 4.4.
⟨ PickyStateTimeClassifier::c'tor 5.4.0.2 ⟩ Used in section 5.4.
⟨ PickyStateTimeClassifier::startTree 5.4.0.6 ⟩ Used in section 5.4.
⟨ PrimitiveHistoryComputer Interface 4.14.1.1 ⟩ Used in section 4.14.1.
⟨ PrimitiveHistoryComputer.cc 5.11 ⟩ Used in section 9.
⟨ PrimitiveHistoryComputer.h 4.14.1 ⟩ Used in section 9.
⟨ RandomPathGenerator implementation 5.2.0.5 ⟩ Used in section 5.2.
⟨ RandomPathGenerator interface 4.2.3 ⟩ Used in section 4.2.

⟨ RandomPathGenerator internal helpers 4.2.3.2 ⟩ Used in section 4.2.3.
⟨ RandomPathGenerator state changes 5.2.0.6 ⟩ Used in section 5.2.0.5.
⟨ RandomPathGenerator types 4.2.3.1 ⟩ Used in section 4.2.3.
⟨ RandomPathGenerator::addLastPoint 5.2.0.9 ⟩ Used in section 5.2.0.5.
⟨ RandomPathGenerator::nextStep 5.2.0.8 ⟩ Used in section 5.2.0.5.
⟨ RandomPathGenerator::recordObservation 5.2.0.10 ⟩ Used in section 5.2.0.5.
⟨ RandomPathGenerator::startTree 5.2.0.7 ⟩ Used in section 5.2.0.5.
⟨ Recorder Accessors 4.20.0.3 ⟩ Used in section 4.20.
⟨ Recorder Actions 4.20.0.4 ⟩ Used in section 4.20.
⟨ Recorder Constructors 4.20.0.2 ⟩ Used in section 4.20.
⟨ Recorder Data 4.20.0.5 ⟩ Used in section 4.20.
⟨ Recorder Typedefs 4.20.0.1 ⟩ Used in section 4.20.
⟨ Recorder.h 4.20 ⟩ Used in section 9.
⟨ ScratchData.h 4.19.2 ⟩ Used in section 9.
⟨ ScratchDataProducer.h 4.19.3 ⟩ Used in section 9.
⟨ ScratchPad.h 4.19.1 ⟩ Used in section 9.
⟨ Simple Matrices and Vectors 3.2 ⟩ Used in section 3.
⟨ Simple Matrix 3.2.0.2 ⟩ Used in section 3.2.
⟨ Simple Vector 3.2.0.1 ⟩ Used in section 3.2.
⟨ SimpleRecorder Accessors 4.20.1.2 ⟩ Used in section 4.20.1.
⟨ SimpleRecorder Action Helpers 4.20.1.4 ⟩ Used in section 4.20.1.3.
⟨ SimpleRecorder Actions 4.20.1.3 ⟩ Used in section 4.20.1.
⟨ SimpleRecorder Constructors 4.20.1.1 ⟩ Used in section 4.20.1.
⟨ SimpleRecorder Output 4.20.1.6 ⟩ Used in section 4.20.1.
⟨ SimpleRecorder data 4.20.1.5 ⟩ Used in section 4.20.1.
⟨ SimpleRecorder.cc 5.12 ⟩ Used in section 9.
⟨ SimpleRecorder.h 4.20.1 ⟩ Used in section 9.
⟨ SimpleSpecification basic behaviors 4.7.2.2 ⟩ Used in section 4.7.2.
⟨ SimpleSpecification constructor implementation 5.6.0.1 ⟩ Used in section 5.6.
⟨ SimpleSpecification constructors 4.7.2.1 ⟩ Used in section 4.7.2.
⟨ SimpleSpecification data 4.7.2.4 ⟩ Used in section 4.7.2.
⟨ SimpleSpecification interface 4.7.2 ⟩ Used in section 4.7.
⟨ SimpleSpecification mementos 4.7.2.5 ⟩ Used in section 4.7.2.
⟨ SimpleSpecification protected accessors 4.7.2.3 ⟩ Used in section 4.7.2.
⟨ SimpleSpecification::operator¡¡ 5.6.0.7 ⟩ Used in section 5.6.0.4.
⟨ SlopeCoefficients Definition 4.9.2 ⟩ Used in section 4.9.
⟨ SlopeCoefficients multiply 5.8.0.1 ⟩ Used in section 5.8.
⟨ SlopeCoefficients output 5.8.0.3 ⟩ Used in section 5.8.
⟨ Specification Basic Behavior 4.7.3.2 ⟩ Used in section 4.7.3.
⟨ Specification Constructors 4.7.3.1 ⟩ Used in section 4.7.3.
⟨ Specification Data 4.7.3.6 ⟩ Used in section 4.7.3.
⟨ Specification Protected Accessors 4.7.3.3 ⟩ Used in section 4.7.3.
⟨ Specification Protected Computation 4.7.3.4 ⟩ Used in section 4.7.3.
⟨ Specification Scratch Data 4.7.3.5 ⟩ Used in section 4.7.3.
⟨ Specification interface 4.7.3 ⟩ Used in section 4.7.
⟨ Specification.cc 5.6 ⟩ Used in section 9.
⟨ Specification.h 4.7 ⟩ Used in section 9.
⟨ Specification::computeResult 5.6.0.3 ⟩ Used in section 5.6.
⟨ Specification::evaluate 5.6.0.2 ⟩ Used in section 5.6.
⟨ Specification::operator¡¡ 5.6.0.6 ⟩ Used in section 5.6.0.4.
⟨ StatePoint 3.1.0.1 ⟩ Used in section 3.
⟨ StateTimeClassifier Interface 4.4.1 ⟩ Used in section 4.4.

⟨ StateTimeClassifier.cc 5.4 ⟩ Used in section 9.
⟨ StateTimeClassifier.h 4.4 ⟩ Used in section 9.
⟨ StateTimeClassifier::c'tor 5.4.0.1 ⟩ Used in section 5.4.
⟨ StateTimeClassifier::isOK 5.4.0.4 ⟩ Used in section 5.4.
⟨ StateTimeClassifier::isTerminal 5.4.0.3 ⟩ Used in section 5.4.
⟨ StateTimeClassifier::startTree 5.4.0.5 ⟩ Used in section 5.4.
⟨ Subset Data Iterator Code 5.14.0.2 ⟩ Used in section 5.14.
⟨ SubsetDataIterator 4.22.2.3 ⟩ Used in section 4.22.2.
⟨ SuccessorGenerator.cc 5.3 ⟩ Used in section 9.
⟨ SuccessorGenerator.h 4.3 ⟩ Used in section 9.
⟨ SumLinearProducts Definition 4.8.4 ⟩ Used in section 4.8.
⟨ SumLinearProducts Implementation 5.7.4 ⟩ Used in section 5.7.
⟨ SumLinearProducts::Memento Definition 4.8.4.1 ⟩ Used in section 4.8.4.
⟨ SumLinearProducts::State Definition 4.8.4.2 ⟩ Used in section 4.8.4.
⟨ TangledDependencies 4.23.0.7 ⟩ Used in section 4.23.
⟨ TestCovariates.h 7.4.3 ⟩ Used in section 7.2.
⟨ TestError.h 7.4.5 ⟩ Used in section 7.2.
⟨ TestManager Accessors 7.4.1.3 ⟩ Used in section 7.4.1.
⟨ TestManager Constructors 7.4.1.1 ⟩ Used in section 7.4.1.
⟨ TestManager Data 7.4.1.2 ⟩ Used in section 7.4.1.
⟨ TestManager Helpers 7.4.1.4 ⟩ Used in section 7.4.1.
⟨ TestManager States 7.4.1.5 ⟩ Used in section 7.4.1.
⟨ TestManager.cc 7.5.1 ⟩ Used in section 7.2.
⟨ TestManager.h 7.4.1 ⟩ Used in section 7.2.
⟨ TestManager::doInitialProcessing 7.5.1.2 ⟩ Used in section 7.5.1.
⟨ TestManager::generateExpectedPaths 7.5.1.4 ⟩ Used in section 7.5.1.
⟨ TestManager::generateTimeSteps 7.5.1.3 ⟩ Used in section 7.5.1.
⟨ TestManager::go 7.5.1.1 ⟩ Used in section 7.5.1.
⟨ TestRecorder Accessors 7.4.2.3 ⟩ Used in section 7.4.2.
⟨ TestRecorder Actions 7.4.2.4 ⟩ Used in section 7.4.2.
⟨ TestRecorder Constructors 7.4.2.1 ⟩ Used in section 7.4.2.
⟨ TestRecorder Data 7.4.2.5 ⟩ Used in section 7.4.2.
⟨ TestRecorder Most State Changes 7.5.2.1 ⟩ Used in section 7.5.2.
⟨ TestRecorder Setup Declaration 7.4.2.2 ⟩ Used in section 7.4.2.
⟨ TestRecorder Setup Implementation 7.5.2.5 ⟩ Used in section 7.5.2.
⟨ TestRecorder State 7.4.2.9 ⟩ Used in section 7.4.2.
⟨ TestRecorder state enum 7.4.2.10 ⟩ Used in section 7.4.2.9.
⟨ TestRecorder.cc 7.5.2 ⟩ Used in section 7.2.
⟨ TestRecorder.h 7.4.2 ⟩ Used in section 7.2.
⟨ TestRecorder::LegalMoves 7.4.2.11 ⟩ Used in section 7.4.2.9.
⟨ TestRecorder::clear 7.5.2.3 ⟩ Used in section 7.5.2.
⟨ TestRecorder::dumpSets 7.5.2.9 ⟩ Used in section 7.5.2.
⟨ TestRecorder::goodPath 7.5.2.2 ⟩ Used in section 7.5.2.
⟨ Testing Files 7.2 ⟩ Used in section 9.
⟨ TimeInPreviousStatesComputer Interface 4.14.1.3 ⟩ Used in section 4.14.1.
⟨ TimeInStateComputer Interface 4.14.1.2 ⟩ Used in section 4.14.1.
⟨ TimePoint Accessors 4.17.0.3 ⟩ Used in section 4.17.
⟨ TimePoint Constructors 4.17.0.2 ⟩ Used in section 4.17.
⟨ TimePoint Typedefs 4.17.0.1 ⟩ Used in section 4.17.
⟨ TimePoint binary ops 4.17.0.5 ⟩ Used in section 4.17.
⟨ TimePoint data 4.17.0.4 ⟩ Used in section 4.17.
⟨ TimePoint streaming 4.17.0.6 ⟩ Used in section 4.17.

⟨ TimePoint.cc 5.16 ⟩ Used in section 9.
⟨ TimePoint.h 4.17 ⟩ Used in section 9.
⟨ TimeSinceOriginComputer Interface 4.14.1.4 ⟩ Used in section 4.14.1.
⟨ TimeSteps.h 4.16 ⟩ Used in section 9.
⟨ TimeStepsGenerator.cc 5.10.4 ⟩ Used in section 9.
⟨ TimeStepsGenerator.h 4.15.3 ⟩ Used in section 9.
⟨ Top-Level Testing Errors 7.4.5.1 ⟩ Used in section 7.4.5.
⟨ UnknownTerm 4.23.0.6 ⟩ Used in section 4.23.
⟨ Vector Output 5.18.0.1 ⟩ Used in section 5.18.
⟨ C Entry Helpers 6.4.5 ⟩ Used in section 6.4.1.
⟨ C Entry Point 6.4.3 ⟩ Used in section 6.4.1.
⟨ C Matrix Typedefs 6.4.2 ⟩ Used in section 6.4.1.
⟨ **R** ModelBuilder helper 6.2.10 ⟩ Used in section 6.2.4.
⟨ **R** compute body 6.2.7 ⟩ Used in section 6.2.4.
⟨ **R** finalizer body 6.2.8 ⟩ Used in section 6.2.4.
⟨ **R** setParams body 6.2.6 ⟩ Used in section 6.2.4.
⟨ **R** simulation `list` setup 6.2.13 ⟩ Used in section 6.2.12.
⟨ **R** simulation body 6.2.12 ⟩ Used in section 6.2.4.
⟨ **R** simulation get individual results 6.2.14 ⟩ Used in section 6.2.12.
⟨ **R** simulation put results in frame 6.2.15 ⟩ Used in section 6.2.12.
⟨ **R** string conversion 6.2.11 ⟩ Used in section 6.2.10.
⟨ **R** subsetting bodies 6.2.9 ⟩ Used in section 6.2.4.
⟨ all Specification streaming operators 5.6.0.4 ⟩ Used in section 5.6.
⟨ basic types 3 ⟩ Used in section 9.
⟨ basic.cc 5.18 ⟩ Used in section 3.
⟨ boost compiler support 3.4 ⟩ Used in section 3.
⟨ get initial state 5.1.0.7 ⟩ Used in section 5.1.0.6.
⟨ makeHistoryStage1 helper class 4.24.1.9 ⟩ Used in section 4.24.1.8.
⟨ makeManager args 6.2.2 ⟩ Used in sections 6.2.1, 6.2.5, 6.2.6, and 6.2.10.
⟨ makeManager body 6.2.5 ⟩ Used in section 6.2.4.
⟨ makeManager params 6.2.3 ⟩ Used in sections 6.2.5 and 6.2.6.
⟨ mspath.cc 6.4.8 ⟩ Used in section 9.
⟨ mspath.h 6.4.1 ⟩ Used in section 9.
⟨ mspathCEntry args 6.4.4 ⟩ Used in sections 6.4.3 and 6.4.9.
⟨ mspathCEntry function 6.4.9 ⟩ Used in section 6.4.8.
⟨ mspathR.cc 6.2.4 ⟩ Used in section 9.
⟨ mspathR.h 6.2.1 ⟩ Used in section 9.
⟨ simulation loop 5.1.0.8 ⟩ Used in section 5.1.0.6.

COMMAND LINE: `"fweave mspath.web"`.
WEB FILE: `"mspath.web"`.
CHANGE FILE: `(none)`.
GLOBAL LANGUAGE: C++.